Investigating the Dynamics of Canonical Flux Tubes

Jens von der Linden

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Setthivoine You, Chair

Brian Nelson

Uri Shumlak

Program Authorized to Offer Degree:
Aeronautics & Astronautics

University of Washington

Abstract

Investigating the Dynamics of Canonical Flux Tubes

Jens von der Linden

Chair of the Supervisory Committee:
Assistant Professor Setthivoine You
William E. Boeing Department of Aeronautics & Astronautics

Observations indicate that the dynamics of plasmas in our cosmos, the heliosphere, and terrestrial experiments can involve conversions between magnetic and kinetic energies over a wide range of plasma scales, such as in reconnection and dynamos. Canonical flux tubes present the distinct advantage of reconciling all plasma regimes, e.g. single particle, kinetic, two-fluid, and magnetohydrodynamics (MHD), with the topological concept of helicity: twists, writhes, and linkages. This thesis presents the first application of the theory of canonical helicity transport to design a laboratory canonical flux tube experiment, develop methods to measure a large volumetric dataset of magnetic field and ion flow, reconstruct the 3D dynamics of canonical flux tubes, and determine the canonical helicity evolution. Newcomb's variational ideal MHD stability analysis is used to identify the possibility of a lengthening current-carrying magnetic flux tube undergoing a sausage-kink instability cascade. The instability cascade may couple to smaller scales at which conversions between species helicities are expected to occur. Experiment control and high-throughput field-programmable gate array (FPGA) based digitizers provide the means to measure the large datasets. Ion and electron canonical flux tubes are visualized from a dataset of Mach, triple, and $\dot{B}$ probe measurements at over $10,000$ spatial locations of a gyrating kinked plasma column. The flux tubes co-gyrate with the peak density and electron temperature in and out of a measurement subvolume. The electron and ion flux tubes twist with opposite handedness and the ion canonical flux tube writhes around the electron

canonical flux tube. Videos of the canonical flux tube reconstructions are available as supplemental material in ProQuest Dissertations & Theses Global. The cross helicity between the magnetic and ion flow vorticity flux tubes dominates the ion canonical helicity and is anticorrelated with the magnetic helicity. The methods developed in this thesis can be applied to other laboratory experiments to improve the understanding of canonical helicity transport; which could help identify how destabilizing magnetic twist is converted to stabilizing shear flows in astrophysical jets and could aid in developing methods for driving shear flow transport barriers in fusion devices.

TABLE OF CONTENTS

Page

List of Figures vi

List of Tables xiii

Chapter 1: Introduction 1
1.1 Plasma physics: a multiscale problem 1
1.1.1 Hierarchy of plasma scales 1
1.2 Context: astrophysical, heliospheric, and laboratory plasmas 4
1.2.1 Astrophysical jets 4
1.2.2 Coronal loops in the heliosphere 6
1.2.3 Topological change in fusion energy concepts 7
1.3 Generalized fluxes and their helicity provide insight across scales 8

Chapter 2: Canonical flux tubes and their helicity 9
2.1 Introduction 9
2.2 Canonical flux tubes 9
2.2.1 Definitions 9
2.2.2 Frozen-in canonical flux condition 11
2.2.3 Theoretical and experimental studies 13
2.3 Canonical helicity 20
2.3.1 Relative canonical helicity 21
2.3.2 Canonical helicity transport 23
2.4 Conclusion 25

Chapter 3: Sausage instabilities on top of kinking lengthening current-carrying magnetic flux tubes 27
3.1 Background 28

3.2 Theory . . . 30
3.3 Numerics . . . 40
3.4 Discussion . . . 45
3.5 Conclusion . . . 48

Chapter 4: Experiment control and high-throughput FPGA based digitizers for canonical helicity measurements . . . 50
4.1 Challenge of measuring canonical helicity . . . 50
4.2 Mochi.Labjet: Canonical flux tubes with boundary conditions relevant to astrophysical jets . . . 51
4.3 Mochi Control: Experiment control & data storage . . . 53
4.3.1 Mochi Control user interface . . . 56
4.3.2 Mochi Control architecture . . . 58
4.4 High-throughput FPGA based digitizers . . . 62
4.5 Conclusion . . . 64

Chapter 5: Measurements of canonical flux tubes and their helicity . . . 65
5.1 Overview of RSX experiment . . . 66
5.1.1 A gyrating kinked canonical flux tube . . . 66
5.2 Diagnostics . . . 67
5.2.1 $\dot{B}$ probes . . . 69
5.2.2 Triple probes . . . 69
5.2.3 Mach probes . . . 70
5.3 Step 1: Prepare data . . . 71
5.3.1 Identifying a coherent shot subset . . . 71
5.3.2 Conditional sampling of measurements across shots . . . 73
5.3.3 Interpolating the measurements to a common rectilinear grid . . . 73
5.3.4 Constraining the under determined velocity . . . 74
5.4 Step 2: Reconstruct and interpret canonical flux tubes . . . 77
5.4.1 Gyration path . . . 77
5.4.2 Canonical Flux Tubes . . . 81
5.5 Step 3: Determine vector potential and reference fields . . . 88
5.5.1 Comparing vector fields . . . 89
5.5.2 Reference fields: Solving Laplace equation with fast Fourier methods . . . 90

5.5.3 Two-component vector potential 92
5.6 Step 4: Reconstruct and interpret canonical helicity 94
5.7 Conclusion 98

Chapter 6: Conclusion and outlook 99

Bibliography 102

Appendix A: Fluxtube stability code 116
A.1 Dependencies 116
A.2 Setup 117
A.3 Important files 117
A.4 Run 117
A.5 newcomb.py description 118
A.6 Source code 118
A.6.1 init_database.py 118
A.6.2 skin_core_scanner.py 119
A.6.3 paper_figures.py 124
A.6.4 call_provenance.py 139
A.6.5 equil_solver.py 139
A.6.6 newcomb.py 173
A.6.7 newcomb_init.py 183
A.6.8 newcomb_f.py 186
A.6.9 newcomb_g.py 189
A.6.10 find_singularities 195
A.6.11 singularity_frobenius.py 198
A.6.12 external_stability.py 203
A.6.13 lambda_k_plotting.py 206
A.6.14 analytic_condition.py 217
A.6.15 diagnostic_plots.py 223
A.6.16 test_equil_solver.py 226
A.6.17 test_newcomb.py 227

Appendix B: MochiControl & MochiFPGAControl code 231
B.1 MochiControl 231
B.1.1 Requirements 232
B.1.2 Important Files 233
B.1.3 Setup 233
B.1.4 Adapting the code to new hardware 234
B.1.5 Reference 234
B.1.6 Acknowledgments 235
B.1.7 Contact 235
B.2 MochiFPGAControl 235
B.2.1 Requirements 235
B.2.2 Important Files 236
B.2.3 Setup 237
B.2.4 Reference 237
B.2.5 Improvements 237
B.2.6 Acknowledgments 238
B.2.7 Contact 238

Appendix C: Mochi.PSU1 239
C.1 Overview figures 240
C.2 Drawings 242
C.2.1 Capacitor Bank Assembly 242
C.2.2 Plates 244
C.2.3 Ignitron cover parts 251
C.2.4 Panels 258

Appendix D: Canonical flux tube and helicity reconstruction code 270
D.1 Dependencies 270
D.2 RSX data and analysis scripts 271
D.3 Directory structure 271
D.4 Important files 271
D.5 Reference 272
D.6 Contact 272

D.7 Main Scripts 272
D.7.1 write_measurements_to_unstructured_grid.py 272
D.7.2 fit_field_null.py 275
D.7.3 interpolate_measurements.py 280
D.7.4 calculate_dependent_quantities.py 284
D.7.5 calculate_helicity.py 292
D.7.6 vis_canonical_fluxtubes.py 298

LIST OF FIGURES

Figure Number Page

1.1 Characteristic plasma length scales in a protostellar jet (purple), coronal loop (red), and the Mochi.Labjet laboratory experiment (black). All lengths are normalized to the respective scale length L. For the protostellar jet $T_e = T_i = 1\ eV$ [7], $n_0 = 10^{12}\ m^{-3}$ [7], $m_i = m_p$, $B = 10^{-5}\ T$ [8], $Z = 1$, $L = 10^{14}\ m$ [7] were chosen. For the coronal $T_e = T_i = 100eV$ [9], $n_0 = 10^{15}\ m^{-3}$ [9], $m_i = m_p$, $B = 10^{-3}\ T$ [4], $Z = 1$, $L = 10^8\ m$ [9] were chosen. For Mochi.Labjet $T_i = 1\ eV$, $T_i = 5\ eV$, $n_0 = 10^{22}\ m^{-3}$, $m_i = m_p$, $B = 0.1\ T$, $Z = 1$, $L = 1\ m$ were chosen based on the initial experiment design parameters. Z is the ionization number ($q_i = Ze$) and $m_p = 1.7 \cdot 10^{-27}\ kg$ is the mass of a proton. 5

2.1 Artist's interpretation of canonical flux tube. When density is uniform, a canonical flux tube (grey) is the weighted sum of a magnetic flux tube (red) and a flow vorticity flux tube (blue). 10

2.2 Five distinct flux tubes of quantities constrained by a choice of dipole vacuum magnetic field $\bar{B}$, uniform current $\bar{j}$, and electron velocity $\bar{u_e}$ parallel to $\bar{B}$ of magnitude $|\bar{u_e}|$. a) Current-carrying magnetic flux tube ψ_{jet}. b) Electron and ion flow flux tubes ψ_e and ψ_i. c) Electron and ion flow vorticity flux tubes $\mathscr{F}_e$ and $\mathscr{F}_i$. d) Electron and ion canonical momentum flux tubes $\mathscr{P}_e$ and $\mathscr{P}_i$. e) Electron and ion canonical flux tubes Ψ_e and Ψ_i. Bar denotes dimensionless quantities. Reproduced from [16] by permission of the AAS. 13

2.3 Left) Bulged ion canonical vorticity flux tube Ψ_i (dark gray) for a diffuse core current jet with low current ($\bar{\lambda} = 0.175$) and a large electron velocity ($\bar{u}_e = 40$). Bar denotes dimensionless quantities. Reproduced from [16] by permission of the AAS. Right) M2-9 Butterfly nebula (NASA HST STIS/CCD – MIRVIS), a bipolar planetary nebula with distinctive bulging that resembles the ion canonical flux tube on the left. By ESA/Hubble, CC BY 4.0 [27]. 14

2.4 Two counter-helicity spheromaks merge. The initial magnetic helicities are experimentally varied and characterized by the eigenvalue λ_0. During the merging process magnetic reconnection converts magnetic energy into ion heating, ion flows, and magnetic activity. When the ion skin depth λ_i over the system length L is large the magnitude of the shear flows becomes large. The ion heating and ion flows form an enthalpy difference across the flux tubes. If the initial magnetic helicity is below a threshold, the helicity is annihilated and the final configuration is an FRC with negligible magnetic helicity and $\lambda \approx 0$. If the initial magnetic helicity is above a threshold, low mode number magnetic activity restores magnetic helicity and the final configuration is a spheromak with finite magnetic helicity and $\lambda \neq 0$. Image credit: Setthivoine You 15

2.5 Threshold of initial eigenvalue $\lambda_0/\lambda_{Taylor}$ as a function of the size parameter S^* for forming a spheromak or a FRC from counter-helicity merging. Reprinted from [28]. Copyright 2005 IOP Publishing. 16

2.6 Gauge-dependence of helicity. An integral over the gray volume cannot count the number of linkages of the two flux tubes external to the grey volume. Adapted from [24]. Image credit: Jaclyn Lake. 22

3.1 a) Sausage $m = 0$ instability. Increased B_{θ_2} exerts an increased pinch force. Magnetic pressure and tension from compressed and bent magnetic field resist the perturbation. b) Kink $m = 1$ instability. The magnetic pressure gradient in B_θ from the inboard side to the outboard side of the perturbation accelerates the perturbation. Magnetic tension from bent axial magnetic field resists the perturbation. 28

3.2 Three idealized flux tubes. 2D plots show axial current density and pressure profiles. Axial magnetic field is uniform for all cases. 3D figures show magnetic field (red) in the core, skin of the flux tube and in vacuum. a) The skin current case used by Tayler and Kruskal-Shafranov. b) Diffuse current case used by Newcomb. c) Core-skin current case used in this derivation. 31

3.3 Analytical $\bar{k}$ - $\bar{\lambda}$ stability spaces parameterized with core current fraction ϵ and rigidness δ. a) Stability spaces for $\epsilon = 0.1$ and $\delta = 0.1$. White region is stable, gray is unstable, hatched region is kink ($m = 1$) unstable, and crosshatched region is unstable to both kink and sausage ($m = 0$) modes. b) Kink stability space dependence on ϵ with $\delta = 0$. Dark red region is unstable. c) Sausage stability space dependence on ϵ with $\delta = 0.7$. Dark green region is unstable. d) Sausage stability boundary dependence on δ with $\epsilon = 0.2$. Dark green region is unstable. 38

3.4 Radial profiles of $\bar{j}_z$, $\bar{B_\theta}$, and $\bar{p}$ for a) $\epsilon_{eff} = 0.7$, b) $\epsilon_{eff} = 0.5$, and c) $\epsilon_{eff} = 0.1$ used for the calculations shown in fig. 3.5. Dashed vertical lines demarcate (from left to right): core, transition, skin, and transition regions. 39

3.5 Numerical $\bar{k} - \bar{\lambda}$ stability spaces with relative growth rates. Normalized kink $\delta W_{m=1}$ contours for (a) $\epsilon_{eff} = 0.7$, (b) $\epsilon_{eff} = 0.5$, and (c) $\epsilon_{eff} = 0.1$; Normalized sausage $\delta W_{m=0}$ contours for (d) $\epsilon_{eff} = 0.7$, (e) $\epsilon_{eff} = 0.5$, and (f) $\epsilon_{eff} = 0.1$. The cross hatched regions indicate Suydam unstable regions in the $m = 1$ plots and regions with internal instabilities in the $m = 0$ plots. [Associated dataset available at http://dx.doi.org/10.5281/zenodo.230611.][43] . . . 43

3.6 The ratio of $\delta W_{m=0}/\delta W_{m=1}$ for (a) $\epsilon_{eff} = 0.7$, (b) $\epsilon_{eff} = 0.5$, and (c) $\epsilon_{eff} = 0.1$. In the white space either at least one of the potential energies is positive or undetermined. [Associated dataset available at http://dx.doi.org/10.5281/zenodo.230611.][43] 44

3.7 Critical $\bar{\lambda}$ value for a given size parameter S^* above which the instability will couple to microscopic scales. The hatched region starts at the minimum sausage unstable $\bar{\lambda}$ for an $\epsilon_{eff} = 0.7$ current profile. The dashed line denotes the Tayler criterion $\bar{\lambda} = 2\sqrt{2}$. 48

4.1 Three-electrode planar plasma gun configuration: showing magnetic field (red), electric field (black), gas valves (grey), outer electrode (orange), middle electrode (gold), inner electrode (gray), current (green), and flows (blue). The gas injection slits in the electrode are azimuthally symmetric. a) Vacuum $\vec{B}$ and $\vec{E}$ fields are imposed by a solenoid and the applied electrode potentials, respectively. b) Initial breakdown occurs along arched magnetic field. c) MHD pumping has collimated the magnetic field. Core and skin currents are driven by the applied electrode potentials. d) MHD pumping drives axial flows. The azimuthally symmetric gas injection slits allow the plasma to freely rotate in response to the radial electric field. 52

4.2 Experiment control: Operators control the experiment with a PC application using LabVIEW DAQmx and LabVIEW FPGA module to program the PXI Cards. The pulse generators / counters and digital outputs generate fast pulses and DC signals. These signals are converted to optical outputs, travel through optical isolation fibers, and are converted back to electrical signals at the experiment. 120 analog inputs digitize diagnostic signals. Control of the 6 pulse generators in the PXI-6133 is not yet implemented. 55

4.3 GUI of experiment control software: a) Timing sequence panel, b) PSU/DIO control panel, c) Global settings panel, d) slow digitizer panel, e) fast digitizer panel, and f) storage control panel. 56

4.4 Experiment control software: The initialize block loads default settings and creates a queue. The queue is passed to two loops running in parallel. The producer loop handles GUI events by creating jobs and enqueuing them. The consumer dequeues jobs. Jobs are processed in the corresponding state of the state machine with calls to DAQmx and NISync Vis. If the state is part of a multi-state sequence, the state may enqueue further jobs. The shutdown event closes the queue and both loops and calls the shutdown block. 58

4.5 Shot sequence in the state machine: Operators pressing the *FIRE* button trigger the shot sequence. Each state is processed in order. DIO are digital output tasks controlling the DC receivers. After *Clean Up* the state machine returns to the *Idle* state. 59

4.6 FPGA block diagram: The FPGA code consists of four single-cycle-timed loops (SCTL): 1) Initialize and register manipulation loop (40 MHz), 2) Acquire loop (50 MHz), 3) Write to DRAM (100 MHz), and 4) Readout DRAM (100 MHz). . 61

5.1 Left) CCD image of emission from kinked canonical flux tube. Right) Artist's interpretation of gyrating canonical flux tube. Axial flow and vacuum magnetic field are directed towards the anode. The current flows in the opposite direction. Measurement planes and interpolation volume are shown in purple. 66

5.2 Coherent gyration. a) Gun current (dashed green) and azimuthal magnetic field (red) measured 5 cm from gyration center by the stationary fiducial probe during a typical shot (#15253). After the current exceeds the Kruskal-Shafranov limit, the magnetic field develops a modulation corresponding to the magnetic flux tube gyration. b) Frequency distribution across all shots of the modulation in the fiducial probe signal. Frequencies are determined by the peak of a Fourier transform. A Gaussian with a constant offset is fit to the distribution. c) Distribution of modulation amplitude in the fiducial probe signal just before the crowbar across all shots. 67

5.3 Plasma length and time scales, and maximum achieved spatial and temporal resolution of internal probes (cross-hatched). The resolution is set by the Nyquist frequency of the digitizers 10 MHz and the Nyquist length of the smallest achieved spatial resolution (0.006 m). Plasma scales are calculated with magnetic field $B = 0.02\ T$, number density $n = 3 \cdot 10^{19}\ m^{-3}$, electron temperature $T_e = 10\ eV$, ion temperature $T_i = 1\ eV$, and flux tube length $L = 0.5\ m$ [65]. 68

5.4 Left) Normalized power spectrum density of azimuthal magnetic field measured 5 cm from gyration axis by fiducial probe before the crowbar, averaged across 10 shots with significant gyration (#15256−15262 and #15267−15269), and 10 shots without gyration (#15305, #15306, #15336, #15352, #15408, #15408, #15410, #15412, #15414, #15416, and #15418). Right) Normalized spectral power distribution for all 2,369 shots obtained by integrating the red area in the left plot. 72

5.5 Points at which measurements were taken in the coherent shot set with $\dot{B}$, triple, and Mach probes. The (purple hatched) region is the union of measurement spaces which is used to reconstruct canonical flux tubes and their helicity. . . . 75

5.6 Linear barycentric interpolation: for interpolation on a 2D grid the vertices of a triangle are weighted by the z values. Each triangle has a uniform slope in the x and y directions ($\Delta z/\Delta x$ and $\Delta z/\Delta y$). The RSX data is on a 3D grid, so tetrahedrons are used. 76

5.7 Left) Field null iteration scheme. $\vec{B}$ and $|\vec{B}|$ at time $t = 8.5\ \mu s$ in the $z = 25$ cm plane. A field line (successive steps from light gray to black) is integrated in both directions starting from the peak of $|\vec{B}|$, a circle is least squares fitted to the field line, and a new integration starting point is determined by moving 10% of the radius towards the circle center (successive steps from light gray to black). Right) Field nulls over one gyration period of 17 μs (white to black). Error bars (light gray) are determined by repeating the field null iteration scheme with successively stronger filtered data, averaging across shots. The field null gyration lies on a circle (dashed red). Circles with 2 cm radius are plotted around every 25$^{\text{th}}$ field null. 2 cm is the radius at which current and density drop of by $1/e$ [65]. Light gray region is the space in which both measurements of B_x and B_y were made. Dark gray region is the joint measurement space of all diagnostics used for flux tube reconstruction and helicity integrals. 78

5.8 Gyrating electron canonical flux tubes. $n\vec{B}$ ($\vec{\Omega}_e$) field lines trace two electron canonical flux tubes over one gyration period of 17 μs in the joint measurement volume. The field lines are launched from circles of 1 mm (orange) and 5 mm (red) radii centered at the null of B_x and B_y in the $z = 25$ cm plane. A single field line (dark red) on the outer flux tube helps visualize the twist. Contours of j_z are plotted in the $z = 25$ cm plane, the colorbar units are $\frac{A}{m^2}$. The field null is inside the joint measurement from $t = 4.1\ \mu s$ to $t = 12\ \mu s$. 81

5.9 Stationary electron canonical flux tubes. $n\vec{B}$ ($\vec{\Omega}_e$) field lines tracing out two electron canonical flux tubes over one gyration period of 17 μs. The field lines are launched from circles of 1 mm (orange) and 5 mm (red) radii centered at $x = 0$ and $y = 0$. A single field line (dark red) on the outer flux tube helps to visualize the twist. Contours of j_z are plotted in the $z = 25\ cm$ plane, the colorbar units are $\frac{A}{m^2}$. 82

5.10 Isosurfaces of n normalized by the peak value in each of the x-y measurement plane over one gyration period of 17 μs. The last isosurface encloses the top 10% of n. For comparison, electron canonical flux tubes and j_z from fig. 5.8 are plotted. 83

5.11 Isosurfaces of T_e normalized by the peak value in each x-y measurement plane over one gyration period of 17 μs. The last isosurface encloses the top 10% of T_e. For comparison, electron canonical flux tubes and j_z from fig. 5.8 are plotted. 84

5.12 Gyrating ion canonical flux tubes. $\vec{\Omega}_i$ field lines trace out two ion canonical flux tubes over one gyration period of 17 μs. The field lines are launched from circles of 1 mm (black) and 5 mm (grey) radii centered at the null of B_x and B_y in the $z = 25\ cm$ plane. For comparison $\vec{B}$ flux tubes and j_z from fig. 5.8 are plotted. 85

5.13 Schematics of twists and writhe. The positive axial direction is from left to right. Left) A right-handed twist (black) and a left-handed twist (red). Right) The black tube writhes with right-handed sense around the red tube. 86

5.14 Taylor plot comparing known vacuum magnetic field (black star) with gradient of scalar potential (red triangles) obtained for 4 grid resolutions with the discrete cosine transform method for solving the Laplace problem. The known vacuum field is in the upper right quadrant of a x-y coordinate system with the conductor at its center. The known magnetic field is used as Neumann boundary condition for the Laplace solver. 91

5.15 Left) Test problem setup for the two-component vector potential solver. The known test fields are the magnetic fields from a gyrating current-carrying cylindrical conductor at several angles. Right) Taylor plot comparing known magnetic fields from gyrating current-carrying conductor at various angles with curl of vector potential solutions obtained with the two-component vector potential method (squares). 93

5.16 Magnetic (red), cross (green), flow (blue), and total ion canonical (black) helicity in the joint measurement volume identified in figs. 5.7 and 5.5. a) Gauge-dependent helicities with real units. b) Relative helicities with real units. c) Boxcar (1.02 μs width) filtered relative helicities, normalized to the maximum relative cross helicity. Dark gray shading marks the time the field null is inside the joint measurement volume. Light gray shading marks the time at which the field null is inside the region with both B_x and B_y measurements shown in fig. 5.7. Dashed orange shows the magnetic helicity of a simplified model problem. The model problem consists of a cylindrical conductor with 0.02 m radius and uniform current density $j = 140\ kA/m^2$ with an exponential density cloud, $n(r) = 1.3 \cdot 10^{19} e^{r/(0.02\,m)}\ m^{-3}$, where r is the radius. The conductor is gyrating along the red dashed circle fitted to the field null path in fig. 5.7. All helicities are the mean determined from running the analysis nine times with successively increasing the standard deviation of the Gaussian kernel filtering of the interpolated measurements. The standard deviation of the runs is plotted as an envelope around the helicity curves, however, the helicity calculations are insensitive to this filtering and the envelope is only partially visible. 95

C.1 Mochi.PSU1: Two capacitor banks in an 80-20 frame enclosed with ABS plastic panels. 240

C.2 One bank of PSU1: ABS panels have been removed. The acrylic tube around the ignitron coaxial housing is visible. 241

C.3 Exploded view of capacitor bank assembly. 242

LIST OF TABLES

Table Number Page

1.1 Select plasma length and time scales. B is the magnetic field, T_σ is the species temperature, n_0 is, assuming quasi-neutrality, the particle number density, q_σ is the species charge, ϵ_0 is the vacuum permittivity, k_B is Boltzmann's constant, c is the speed of light, k_D is the drift wave number, and Λ is the plasma parameter which is the number of particles in a Debye sphere. 2

ACKNOWLEDGMENTS

This thesis describes the methods and results of my work as a graduate student; however, it does not capture the human nature of this endeavor. I am indebted to Professor Setthivoine You for his patience and insight throughout this work. He invited me to participate in the great opportunity of building a new plasma physics laboratory. It is amazing to consider how far we have come from an empty room to regularly generating plasma jets traveling at a hundred thousand miles per hour, with temperatures of a hundred thousand degrees, and hundreds of kiloamps of current. I will always cherish the discussions on engineering, physics, and the totally absurd I had with my fellow graduate students while we were building the experiment: Alexander Card, Evan Carroll, Eric Sander Lavine, Iman Datta, Manuel Azuara-Rosales, Morgan Quinley, Keon Vereen, and Yu Kamikawa. I thank the Department of Energy SCGSR program for funding my stay at Lawrence Livermore National Laboratory (LLNL) where I reconstructed canonical flux tubes from the RSX dataset. I am grateful to Dr. Jason Sears for mentoring me during my year at LLNL. I thank Dr. Alan Glasser for introducing me to his DCON code and discussions on MHD stability analysis. The faculty of the Aeronautics & Astronautics department, especially: Prof. Antonio Ferrante, Prof. Brian Nelson, Prof. Tom Jarboe, Prof. Raymond Golingo, and Prof. Uri Shumlak taught me how to think, design, and question like a plasma physicist. The machinists at the Physics, Mechanical Engineering, and Aeronautics & Astronautics machine shops, especially Ron Musgrave taught me how to work safely and helped me design and build components for our experiment. I also would like to thank my graduate student friends, the Pacific Science Center, and Software Carpentry for much fun and time away from research. I have immense respect for my father, Walter, for how he cared for me after the loss of his wife, my mother, Barbara. I thank him for his continuing support and

encouragement to this day. And most of all I thank my best friend and wife, Taralyn. Thanks for supporting me, for your artistic eye and input on my figures, for reading over this thesis, and for your patience! I look forward to our next adventures with your growing family of four siblings and their spouses. Thanks to them too for their patience and support. I promise to make up for all the birthdays and holidays missed.

DEDICATION

In loving memory of my mother, Barbara Sue von der Linden, and in deep gratitude to my father, Walter von der Linden.

Chapter 1

INTRODUCTION

1.1 Plasma physics: a multiscale problem

A plasma is a quasi-neutral gas of electrically charged and neutral particles that exhibits collective behavior [1]. Each kind of charged particle in the gas of a specific charge and mass is designated a plasma species σ, e.g., electrons e and ions i. It has been estimated that 99% of the visible matter in the universe is plasma [2]. The study of plasma physics provides the basis for the dynamics of space and astrophysical systems from the solar corona to accretion disks surrounding black holes. On earth, plasmas are used in many applications ranging from fluorescent lights to industrial material processing. Fusion energy research endeavors to harness energy resulting from the fusion of low atomic number ions such as deuterium and tritium. A crucial difference between plasma and conventional fluids, solids, and gases is the importance of collective effects. Plasma particles interact simultaneously with a large number of particles over long distances through electromagnetic forces. In principle, one could determine the behavior of a plasma by integrating the equations of motions of individual particles and evolving electromagnetic fields according to Maxwell's equations. However, the sheer number of particles ($10^{19} - 10^{22}$ m^{-3} for the experiments discussed in chapters 4 and 5) combined with the wide range of spatial and temporal scales at which the particles interact make this approach practically impossible [3]. Particle collisions and long range electromagnetic fields influence plasma behavior on a hierarchy of temporal and spatial scales [4, 5].

1.1.1 Hierarchy of plasma scales

Table 1.1 shows select characteristic plasma lengths and frequencies in their approximate hierarchical ordering. The exact ordering depends on the parameters of the plasma of interest. Ref.

Debye length	λ_D	$\sqrt{\frac{\epsilon_0 k_B T_e}{n_0 q_e^2}}$
electron skin depth	λ_e	$\frac{c}{\omega_{pe}} = c\sqrt{\frac{\epsilon_0 m_e}{n_0 q_e^2}}$
thermal ion Larmor radius	r_{Li}	$\frac{v_{th}}{\omega_{ci}} = \frac{\sqrt{m_i k_B T_i}}{q_i B}$
ion skin depth	λ_i	$\frac{c}{\omega_{pi}} = c\sqrt{\frac{\epsilon_0 m_i}{n_0 q_i^2}}$
electron mean free path	λ_{mfp}	$\frac{v_{th}}{\nu_{ei}} = \frac{12\pi^{3/2}\epsilon_0^2 (k_B T_e)^2}{\sqrt{2} n_0 q_i^2 q_e^2 ln\Lambda}$
system scale length	L	-

electron plasma frequency	ω_{pe}	$\sqrt{\frac{n_o q_e^2}{\epsilon_0 m_e}}$
ion cyclotron frequency	ω_{ci}	$\frac{q_i B}{m_i}$
ion plasma frequency	ω_{pi}	$\sqrt{\frac{n_0 q_i^2}{\epsilon_0 m_i}}$
Alfvén transit frequency	ω_A	$\pi \frac{B}{L\sqrt{\mu_0 n_o m_i}}$
drift wave frequency	ω_D	$\frac{k_y dn_0/dx k_B T_e}{n_0 q_e B_z}$
electron-ion collision frequency	ν_{ei}	$\frac{\sqrt{2} n_0 q_i^2 q_e^2 ln\Lambda}{12\pi^{3/2}\sqrt{m_e}\epsilon_0^2 (k_B T_e)^{3/2}}$
electron-ion energy equilibration frequency	ν_{Eei}	$\approx \frac{m_e}{m_i}\nu_{ei}$

Table 1.1: Select plasma length and time scales. B is the magnetic field, T_σ is the species temperature, n_0 is, assuming quasi-neutrality, the particle number density, q_σ is the species charge, ϵ_0 is the vacuum permittivity, k_B is Boltzmann's constant, c is the speed of light, k_D is the drift wave number, and Λ is the plasma parameter which is the number of particles in a Debye sphere.

[5] surveys the hierarchy of time scales. Here, an overview of the hierarchy of length scales in a plasma will be given. In the absence of external forces, plasmas are microscopically neutral. The Debye length λ_D is the scale at which charge is shielded. At the Debye length λ_D, the thermal particle energy, which causes charge separation, and the opposing electrostatic energy balance [1]. The stability of the macroscopic neutrality of a plasma is described by the electron skin depth λ_e. When an external perturbation displaces a charge from its equilibrium position, resulting electric fields will give rise to electron motion which restores charge neutrality [1]. The characteristic frequency of oscillation for the electrons in a plasma is the electron plasma frequency ω_{pe}. The electron skin depth λ_e is the wavelength of electromagnetic waves of this frequency $c = \lambda_e \omega_{pe}$, where c is the speed of light. λ_e depends only on the particle density n_0. In plasmas with magnetic fields the thermal ion cyclotron frequency ω_{ci} is the frequency associated with electrostatic (Bernstein) and electromagnetic cyclotron waves, which can be used to heat a plasma [5]. The thermal ion Larmor radius r_{Li} is the cyclotron radius for thermal ions. Magnetic fields constrain perpendicular motion of charged particles. When the Larmor radius is much shorter than the mean free path of a particle species, it acts as an effective

mean free path in fluid closure schemes, which allow the plasma species to be treated as fluids on scales larger than the Larmor radius. This separates kinetic models which track the evolution of velocity distribution functions for each species from fluid models where the distribution function is assumed to be Maxwellian, and only the evolution of the moments describing the Maxwellian need to be tracked. A magnetic field may also restrict the motion of electrons, allowing heavier ions to respond to a displacement. The characteristic oscillation frequency of ions is the ion plasma frequency ω_{pi}. The ion skin depth or ion inertial length λ_i is the wavelength of electromagnetic waves of this frequency. Beyond this length scale the difference in ion and electron inertia becomes negligible, allowing the ions to move together with the electrons and for models to treat the species as a single conducting fluid (magnetohydrodynamics, MHD). λ_i is a factor of the charge ratio times the square root of the ion-to-electron mass ratio ($q_e/q_i\sqrt{m_i/m_e}$) larger than λ_e. Collisions between particles of a like and unlike species lead to transfer of momentum (directed velocity) and energy (thermal velocity) [6]. The collision frequencies for momentum and energy transfer in and between species have their own hierarchical ordering in terms of powers of the electron-to-ion mass ratio m_e/m_i. The electron mean free path λ_{mfp} is the length scale at which momentum is transferred between electrons and ions and is of the same order of magnitude as the length scale at which momentum is equilibrated among electrons. The transfer of directed velocity between electrons and ions inhibits the flow of current and is responsible for resistivity. When the resistivity is small the plasma can be treated as perfectly conducting (ideal MHD). In ideal MHD plasmas, the magnetic field and plasma density oscillate together and waves propagate at the Alfvén velocity v_a. The characteristic time of the ideal MHD model is given by the transit time of an Alfvénic wave through the system scale length L.

In typical plasmas, several orders of magnitude separate the Debye length from the electron mean free path. One challenge for a plasma physicist is to identify the relevant scales of the problem of interest and build an appropriate model involving kinetic, two-fluid, or MHD approximations. This task is complicated by the tendency of phenomena on one scale to couple to disparate scales. Drift waves in plasmas with density gradients provide a prominent exam-

ple. Although the characteristic frequency of the drift wave instability is usually several orders of magnitude larger than the ion cyclotron frequency, the it can couple kinetic effects to slow MHD timescales [5, 6].

1.2 Context: astrophysical, heliospheric, and laboratory plasmas

Fig. 1.1 provides examples of characteristic length scales in three distinct plasmas: an astrophysical jet from a star forming region, a coronal loop, and the Mochi.Labjet laboratory experiment.

1.2.1 Astrophysical jets

Astrophysical jets are collimated, high-speed outflows from rotating accretion disks surrounding massive objects. Jet lengths range from less than a light year (10^{16} m) in young stellar objects (YSO) to megaparsecs (10^{22} m) in active galactic nuclei (AGN) [7]. Jets are collimated and very straight; their aspect ratio can be in the order of 10:1 to 1000:1. Outflow velocities vary from $\sim$100 $km\ s^{-1}$ in protostellar jets to relativistic speeds in AGN jets. Observations of signatures of helical magnetic fields [10, 11] with plasma parameters listed in fig. 1.1 indicate that the jets can be described with MHD models. This raises several questions: Why are jets so long and thin, often without apparent signs of large scale instabilities that are common in laboratory plasma experiments? When jets do form irregularities such as bulges, knots, and wiggles, what causes them and can they provide insights into activities in the source of the jets? Observations of azimuthal rotation [12, 13] and axial shear flow [14] suggest that shear flows may stabilize MHD instabilities as has been observed in laboratory experiments [15]. It has been hypothesized that destabilizing helical magnetic twist may be converted to stabilizing helical shear flows at length scales of ion inertial lengths or ion Larmor radii [16]. In fig. 1.1 the characteristic plasma scales in a protostellar jet vary from the Debye length λ_D to the electron mean free path λ_{mfp} by six orders of magnitude; the jet length is 16 orders of magnitude longer than λ_D. A mechanism must therefore be identified that could couple the MHD scales

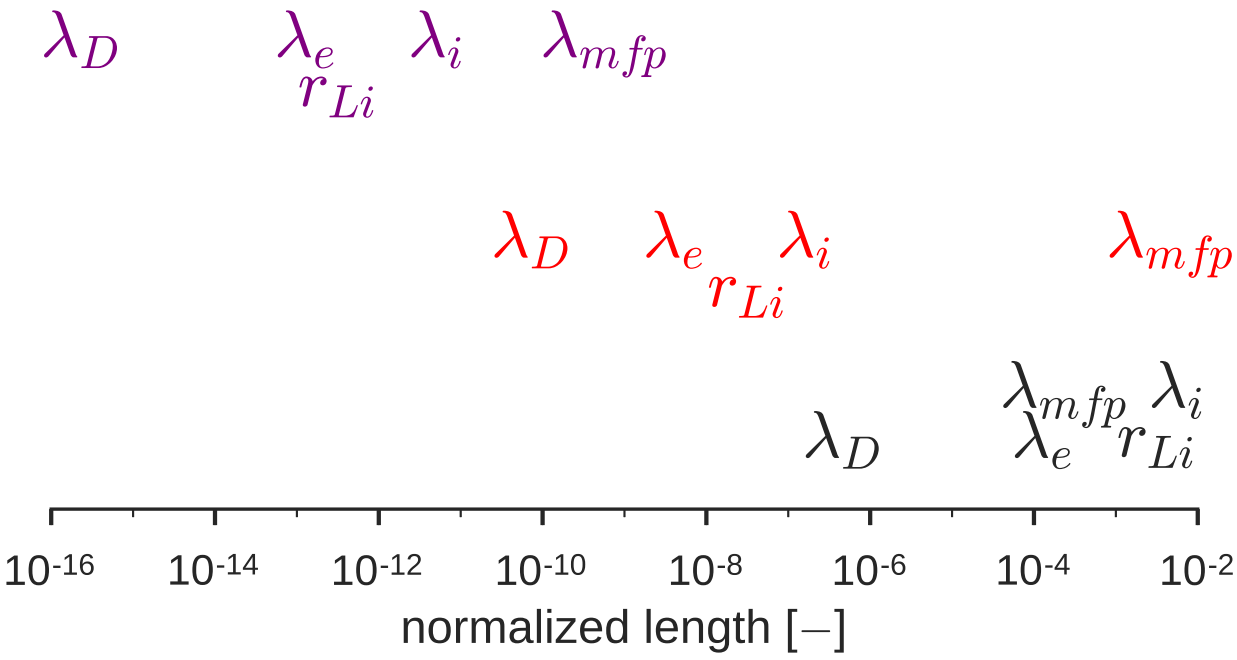


Figure 1.1: Characteristic plasma length scales in a protostellar jet (purple), coronal loop (red), and the Mochi.Labjet laboratory experiment (black). All lengths are normalized to the respective scale length L. For the protostellar jet $T_e = T_i = 1\ eV$ [7], $n_0 = 10^{12}\ m^{-3}$ [7], $m_i = m_p$, $B = 10^{-5}\ T$ [8], $Z = 1$, $L = 10^{14}\ m$ [7] were chosen. For the coronal $T_e = T_i = 100eV$ [9], $n_0 = 10^{15}\ m^{-3}$ [9], $m_i = m_p$, $B = 10^{-3}\ T$ [4], $Z = 1$, $L = 10^{8}\ m$ [9] were chosen. For Mochi.Labjet $T_i = 1\ eV$, $T_i = 5\ eV$, $n_0 = 10^{22}\ m^{-3}$, $m_i = m_p$, $B = 0.1\ T$, $Z = 1$, $L = 1\ m$ were chosen based on the initial experiment design parameters. Z is the ionization number ($q_i = Ze$) and $m_p = 1.7 \cdot 10^{-27}\ kg$ is the mass of a proton.

to the two-fluid λ_i and kinetic r_{Li} scales.

1.2.2 Coronal loops in the heliosphere

Coronal loops are arched structures formed when twisted magnetic flux rises above the photosphere with their footpoints anchored to the photosphere. The loops may become further twisted due to footpoint motion. Coronal loops vary in length from 10^6 m to 10^9 m. Plasma flows along the coronal loops with velocities between 5 $km\ s^{-1}$ and 30 $km\ s^{-1}$, with densities ranging from 10^{15} m^{-3} to 10^{17} m^{-3} [17]. Solar flares and coronal mass ejections may be associated with coronal loop interactions. A solar flare is a rapid, local release of electromagnetic energy in form of x-ray and gamma ray radiation. Coronal mass ejections (CME) are often associated with flares and are large releases of plasma matter threaded with magnetic fields. Coronal loops can produce collimated jets with lengths of $7,000-40,000$ km, lifetimes of minutes and velocities of $10-1000$ $km\ s^{-1}$. The coronal plasma with temperatures over a million Kelvin is much hotter than the photosphere plasma below the corona with temperatures of only 6000 Kelvin [18]. Magnetic reconnection may release magnetic energy injected by coronal foot point motion into heat, electromagnetic waves, and plasma particle kinetic energy. Models of reconnection that rely on resistive reconnection on MHD scales (Sweet-Parker, Petschek) do not predict reconnection fast enough to reconcile with observations of solar flares [19]. Thus, solar magnetic reconnection is a prominent example of interactions between different plasma scales. A large global region is thought to feed magnetic flux to small localized regions where two-fluid and kinetic effects allow fast breaking of field lines and conversion of magnetic energy to thermal and kinetic energy [20]. Fig. 1.1 illustrates the wide separation of scales. The ion Larmor radius r_{Li} and the ion inertial length λ_i are eight and seven orders of magnitude smaller than the length of the coronal loop, respectively.

1.2.3 Topological change in fusion energy concepts

Plasma jets and loops can be created in the laboratory; one approach to do this uses planar plasma guns. In these experiments, breakdown occurs along a magnetic dipole field linking concentric electrodes. Gas puffed in through discrete holes in the electrodes forms dense plasma arches with a low density background. The dense plasma arches merge, because they carry parallel currents, forming a central current-carrying jet. The magnetic dipole field is flared and current is driven by an electrode potential difference along the flux tube fulfilling the conditions for MHD pumping and collimation of the flux tube [21]. A pressure gradient and $\vec{J} \times \vec{B}$ forces accelerate plasma axially to Alfvénic velocities. The flow carries magnetic flux and stagnates at the end of the jet where the increased magnetic flux collimates the jet [22]. The dense part of flux tube lengthens continuously and collimates throughout the shot ($\sim 10-100$ μs duration), while the current of the power supply ramps up. If the aspect ratio times the the ratio of current of axial magnetic field exceeds a certain value, the Kruskal-Shafranov limit, the plasma jet may kink. Ref. [23] observed Rayleigh-Taylor instabilities forming on top of the kinking jet plasmas. The acceleration of the kink acts as an effective gravity for Rayleigh-Taylor instabilities to develop. The Rayleigh-Taylor instabilities may create small enough spatial scales to generate regions where two-fluid or kinetic physics might allow faster reconnection and topological change to occur which result in the formation of toroidal configurations of interest for fusion energy concepts. Toroidal plasma configurations, called spheromaks, have been observed to form from linear pinches and jets ejected from plasma guns [24]. Another toroidal configuration, called spherical torus, has been formed from flux tubes with current driven by washer guns [25]. Laboratory experiments are diverse; in fig. 1.1 the Mochi.Labjet experiment at the University of Washington serves as an example to illustrate the characteristic plasma scales. Seven orders of magnitude separate the jet length from the Debye length λ_D. The ion Larmor radius r_{Li} and ion inertial length λ_i are both within three orders of magnitude of the jet length. The low temperature and high densities shift the electron mean free path λ_{mfp} below the thermal ion Larmor radius r_{Li}.

1.3 Generalized fluxes and their helicity provide insight across scales

The above examples show that there are a wide range of length scales in typical plasmas and several indications for coupling between different scales. These phenomena can be modeled by subdividing plasmas into regions where different physics apply. Some models of reconnection treat the global scale with MHD, while microscopic reconnection points are treated with two-fluid or kinetic models [19]. Difficulties arise at the boundaries of these regions, where two-fluid physics or the non-Maxwellian nature of kinetic velocity distributions have to be reconciled with MHD. This thesis will focus on a complementary approach by studying the evolution of generalized fluxes and their helicity. For example, in ideal MHD the plasma density is frozen-in to the magnetic flux, i.e. particle excursions off flux surfaces can be neglected. This means it is sufficient to track the evolution of magnetic flux tubes. The magnetic flux tubes evolve subject to the topological constraint of helicity conservation, their twist, writhe, and interlinking is conserved. At scales where non-ideal effects, such as ion inertia, kinetic distribution functions, and finite Larmor radius effects become important the frozen-in magnetic flux condition is generalized to the frozen-in canonical flux condition [16]. Each plasma species is frozen-in to a canonical flux related to its canonical momentum. Canonical momentum includes both fluid flow and magnetic terms, so canonical helicity can be expressed as the sum of helicity of fluid flow and helicity of magnetic fields. Conversions between magnetic and kinetic energy can be viewed topologically as conversions between magnetic and fluid flow twist while conserving the overall twistedness of canonical flux tubes of all species. Chapter 2 will define canonical flux tubes and their helicity and review recent theoretical and experimental studies, with a focus on the spatial scales at which the different helicities are conserved and when conversions can occur. Chapter 3 will identify an instability cascade during which conversions between the fluid, cross, and magnetic terms of helicity can occur. Chapter 4 will describe the development of experimental control schemes and diagnostics to aid in measuring the 3D evolution of canonical flux and helicity. Chapter 5 will reconstruct canonical flux tubes and their helicity from measurements of a laboratory experiment.

Chapter 2

CANONICAL FLUX TUBES AND THEIR HELICITY

2.1 Introduction

This chapter first defines canonical flux tubes and reviews the canonical Ohm's equation which shows that there is a one-to-one relationship between a plasma species and its canonical flux tubes. Next, a review of experimental observations focuses on the dynamics of canonical flux tubes and relaxation subject to the constraint of canonical helicity: the twist, writhe, and interlinking of canonical flux tubes. The observations are interpreted with the recent theory of canonical helicity transport and dimensional analysis of helicity injection and decay. The overview of canonical helicity includes a new discussion of the effect of density gradients on helicity volume integrals.

2.2 Canonical flux tubes

2.2.1 Definitions

Each species in a plasma has a canonical momentum

$$\vec{P}_\sigma = \rho_\sigma \vec{u}_\sigma + \rho_{c\sigma} \vec{A}, \tag{2.1}$$

which is the weighted sum of the species velocity $\vec{u}_\sigma$ and the magnetic vector potential $\vec{A}$. The weights are the mass function ρ_σ and the charge function $\rho_{c\sigma}$. The choice of the functions depends on the plasma model [26]. For single particles, the functions reduce to the particle species mass and charge $\rho_\sigma \to m_\sigma$ and $\rho_{c\sigma} \to q_\sigma$. In kinetic models which track the velocity distribution function f_σ for the particles of each species, the appropriate choice is $\rho_\sigma \to f_\sigma m_\sigma$ and $\rho_{c\sigma} \to f_\sigma q_\sigma$. In fluid models which track the number density n_σ, velocity $\vec{u}_\sigma$ and energy / temperature T_σ of a fluid species, the functions reduce to $\rho_\sigma \to n_\sigma m_\sigma$ and $\rho_{c\sigma} \to n_\sigma q_\sigma$. This

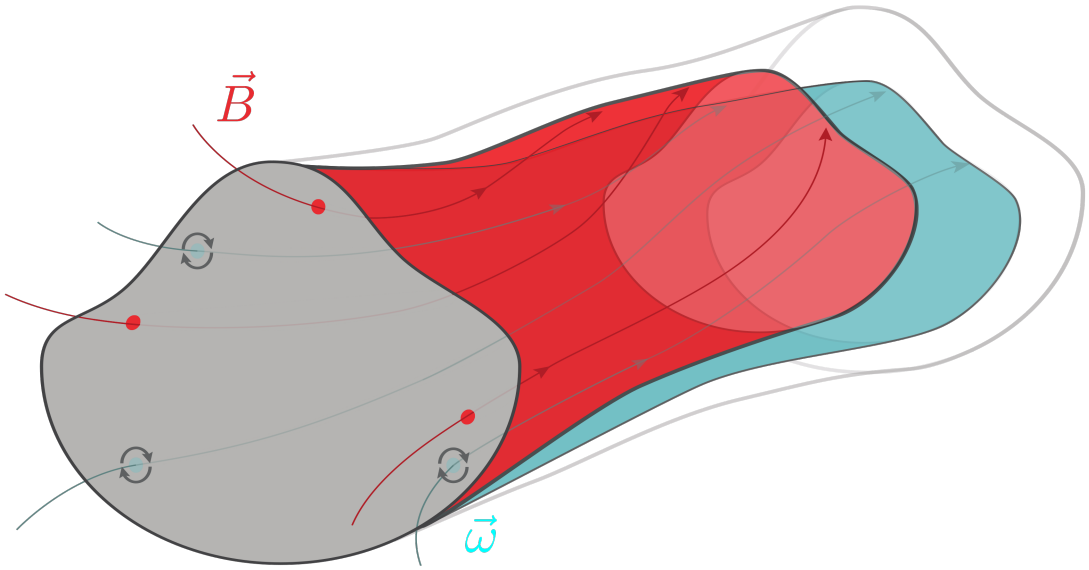


Figure 2.1: Artist's interpretation of canonical flux tube. When density is uniform, a canonical flux tube (grey) is the weighted sum of a magnetic flux tube (red) and a flow vorticity flux tube (blue).

thesis will focus on two-fluid systems with an ion ($\sigma \to i$) and electron ($\sigma \to e$) species. The curl of canonical momentum is called canonical circulation $\vec{\Omega}_\sigma = \nabla \times \vec{P}_\sigma$,

$$\vec{\Omega}_\sigma = n_\sigma m_\sigma \vec{\omega}_\sigma + n_\sigma q_\sigma \vec{B} + \frac{\nabla n_\sigma}{n_\sigma} \times \vec{P}_\sigma, \tag{2.2}$$

where $\vec{\omega}_\sigma$ is the flow vorticity, $\nabla \times \vec{u}_\sigma = \vec{\omega}_\sigma$, and $\vec{B}$ is the magnetic field. Canonical circulation is divergence-free, meaning a cross-sectional area traced out by field lines integrated from an initial closed arbitrary area will enclose a constant amount of canonical flux. The canonical flux Ψ_σ is the surface integral of canonical circulation

$$\Psi_\sigma = \int_S \vec{\Omega}_\sigma \cdot d\vec{S}, \tag{2.3}$$

where S is the surface and $\vec{S}$ the elementary surface normal. For the barotropic case of no density variation, canonical flux tubes can be written as a weighted sum of a magnetic flux tube $\psi = \int B \cdot d\vec{S}$ and a flow vorticity flux tube $\mathscr{F} = \int \omega_\sigma \cdot d\vec{S}$ (fig. 2.1),

$$\Psi_\sigma = n_\sigma m_\sigma \mathscr{F}_\sigma + n_\sigma q_\sigma \psi. \tag{2.4}$$

In the limit of negligible electron mass (inertia) the electron canonical flux tube becomes a magnetic flux tube ($\Psi_e \to \psi$).

2.2.2 Frozen-in canonical flux condition

Ref. [26] shows that the same set of canonical field equations in terms of canonical momentum and circulation can be derived from a Lagrangian-Hamiltonian formulation of relativistic single particle equations, the kinetic Vlasov equation, and the two-fluid evolution equations. This means that for all plasma scales the canonical fields evolve according to Maxwell's equations. The equation of motion for each model can be re-expressed as the canonical Ohm's law

$$\vec{\Sigma}_\sigma + \vec{u}_\sigma \times \vec{\Omega}_\sigma = -\vec{R}_\sigma. \tag{2.5}$$

The canonical force-field $\vec{\Sigma}_\sigma$ represents the conservative and inductive forces, $\vec{u}_\sigma \times \vec{\Omega}_\sigma$ represents the forces that do no work, e.g. Coriolis and Lorentz forces, and $\vec{R}_\sigma$ represents the

dissipative forces. Analogous to the electric field the canonical force field is the sum of a potential gradient and an inductive term $\vec{\Sigma}_\sigma = -\nabla h_\sigma - \partial \vec{P}/\partial t$. For the two-fluid case the enthalpy h_σ,

$$h_\sigma = n_\sigma q_\sigma \phi + \frac{1}{2} n_\sigma m_\sigma \vec{u}_\sigma^2 + \mathscr{P}, \tag{2.6}$$

combines the conservative electrostatic potential ϕ and scalar pressure $\mathscr{P}$, and the kinetic energy. Taking the curl of Ohm's law (2.5), in the absence of friction, results in the canonical induction equation [16], a generalization of the ideal MHD induction equation to include non-negligible inertia,

$$\frac{\partial \vec{\Omega}_\sigma}{\partial t} - \nabla \times (\vec{u}_\sigma \times \vec{\Omega}_\sigma) = 0. \tag{2.7}$$

This shows that in the absence of friction the canonical circulation is frozen-in to the species motion at all scales. There is a one-to-one relationship between the canonical flux tubes and the species. A consequence of the frozen-in condition appears under equilibrium conditions. Setting the time-varying and frictional terms in eq. (2.5) to zero, dotting with $\vec{\Omega}$, and assuming a steady-state equilibrium, yields

$$\nabla h_\sigma \cdot \vec{\Omega}_\sigma = 0. \tag{2.8}$$

For an equilibrium, gradients of enthalpy are always perpendicular to canonical circulation. When no potentials are applied, the pressure gradient is perpendicular to $\vec{\Omega}$. In the barotropic case where the pressure is purely density dependent, the condition reduces to

$$\nabla n_\sigma \cdot \vec{\Omega}_\sigma = 0. \tag{2.9}$$

Gradients of a species density are perpendicular to a species canonical circulation. For a single fluid MHD plasma without vorticity this condition reduces to the familiar flux surface condition

$$\nabla n_\sigma \cdot \left(n_\sigma q_\sigma \vec{B} + \frac{\nabla n_\sigma}{n_\sigma} \times \vec{P_\sigma} \right) = \nabla n_\sigma \cdot \vec{B} = 0. \tag{2.10}$$

Optical emission visible in images of a uniform temperature plasma is roughly proportional to the square of the density. The intensity profiles visible in these emission images are signatures of the magnetic, or more generally, the canonical flux [16].

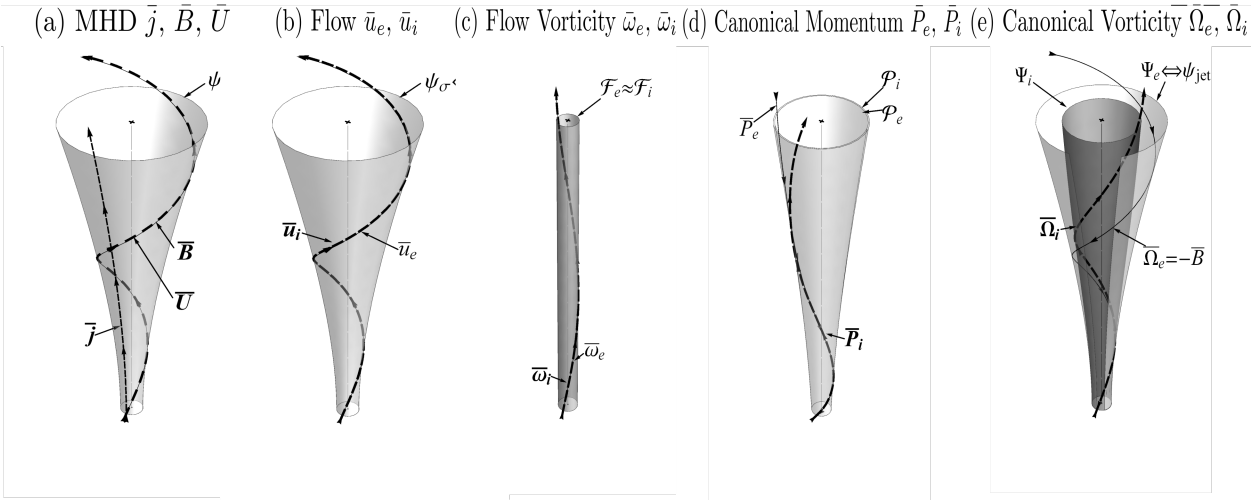
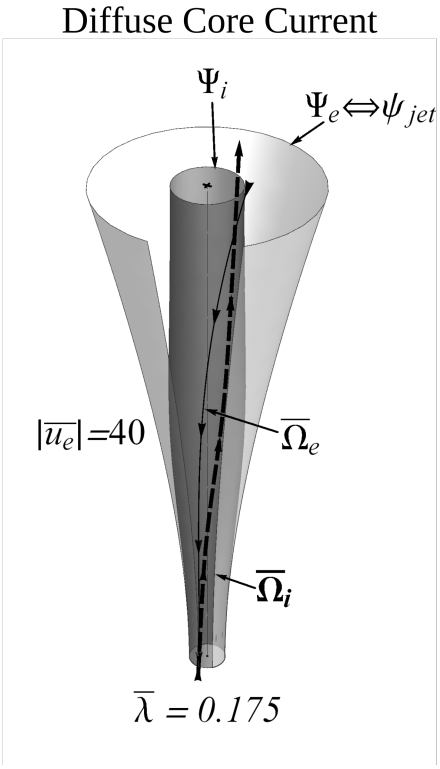


Figure 2.2: Five distinct flux tubes of quantities constrained by a choice of dipole vacuum magnetic field $\bar{B}$, uniform current $\bar{j}$, and electron velocity $\bar{u_e}$ parallel to $\bar{B}$ of magnitude $|\bar{u_e}|$. a) Current-carrying magnetic flux tube ψ_{jet}. b) Electron and ion flow flux tubes ψ_e and ψ_i. c) Electron and ion flow vorticity flux tubes $\mathscr{F}_e$ and $\mathscr{F}_i$. d) Electron and ion canonical momentum flux tubes $\mathscr{P}_e$ and $\mathscr{P}_i$. e) Electron and ion canonical flux tubes Ψ_e and Ψ_i. Bar denotes dimensionless quantities. Reproduced from [16] by permission of the AAS.

2.2.3 Theoretical and experimental studies

Several laboratory experiments generate linear canonical flux tubes and study their dynamics with a focus on magnetic flux tubes. Plasma discharges along arched magnetic field lines connecting two electrodes create current-carrying arched magnetic flux tubes with geometries similar to coronal loops [29, 30]. Other experiments have a flared magnetic field, e.g. a dipole field connects a circular electrode with an outer annular electrode. A flared field is a field with radial and axial components, the radial component varies along the central axis. MHD



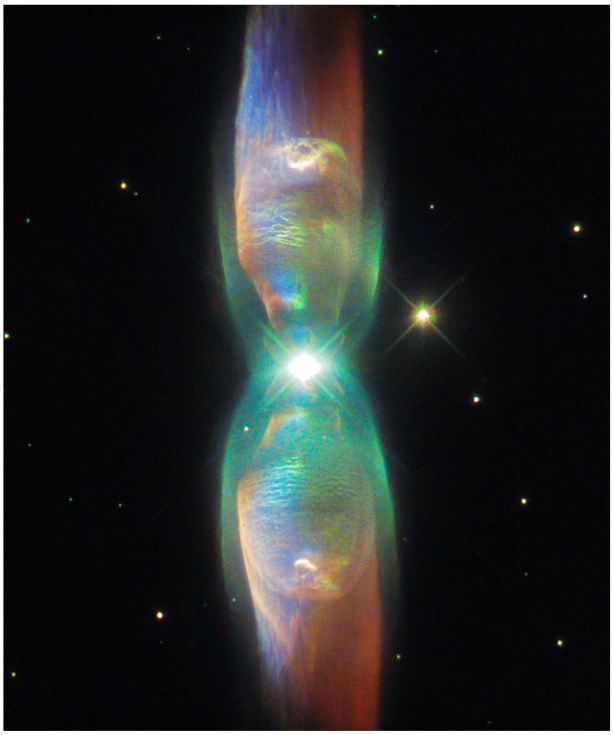


Figure 2.3: Left) Bulged ion canonical vorticity flux tube Ψ_i (dark gray) for a diffuse core current jet with low current ($\bar{\lambda} = 0.175$) and a large electron velocity ($\bar{u}_e = 40$). Bar denotes dimensionless quantities. Reproduced from [16] by permission of the AAS. Right) M2-9 Butterfly nebula (NASA HST STIS/CCD – MIRVIS), a bipolar planetary nebula with distinctive bulging that resembles the ion canonical flux tube on the left. By ESA/Hubble, CC BY 4.0 [27].

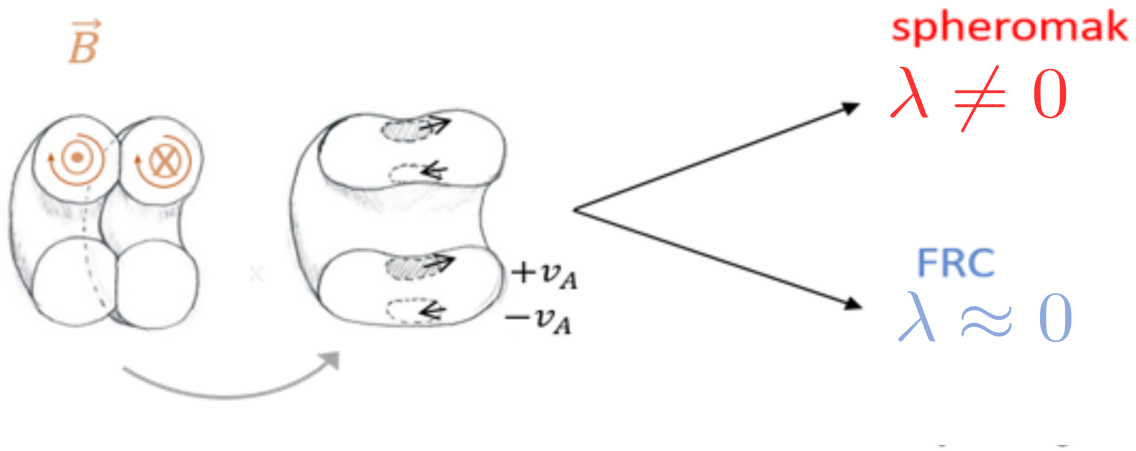


Figure 2.4: Two counter-helicity spheromaks merge. The initial magnetic helicities are experimentally varied and characterized by the eigenvalue λ_0. During the merging process magnetic reconnection converts magnetic energy into ion heating, ion flows, and magnetic activity. When the ion skin depth λ_i over the system length L is large the magnitude of the shear flows becomes large. The ion heating and ion flows form an enthalpy difference across the flux tubes. If the initial magnetic helicity is below a threshold, the helicity is annihilated and the final configuration is an FRC with negligible magnetic helicity and $\lambda \approx 0$. If the initial magnetic helicity is above a threshold, low mode number magnetic activity restores magnetic helicity and the final configuration is a spheromak with finite magnetic helicity and $\lambda \neq 0$. Image credit: Setthivoine You

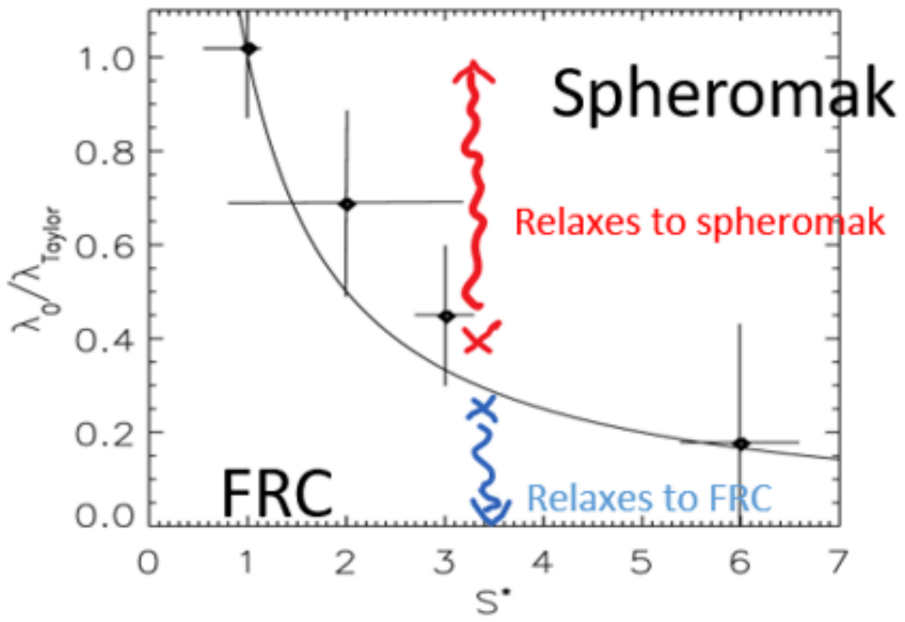


Figure 2.5: Threshold of initial eigenvalue $\lambda_0/\lambda_{Taylor}$ as a function of the size parameter S^* for forming a spheromak or a FRC from counter-helicity merging. Reprinted from [28]. Copyright 2005 IOP Publishing.

pumping will pump density and magnetic flux along the arched flux tubes, collimating them into lengthening current-carrying magnetic flux tubes [21, 22]. Recently, an analytical study visualized the topology of current-carrying canonical flux tubes in a flared magnetic field [16]. The study assumes a barotropic two-fluid with massless electrons. Fig. 2.2 shows equivalent flux tubes of a system with an imposed flared vacuum magnetic field in the axial and radial directions and a uniform current parallel to the vacuum magnetic field. Fig. 2.2 (a) shows the current-carrying flared magnetic flux tube $\psi = \int \vec{B} \cdot d\vec{S}$ with twisted magnetic field lines due to the superposition of the flared dipole field and the dynamic magnetic field generated by the uniform current j. The center-of-mass flow $\vec{u}$ is frozen into the magnetic flux surface. The center-of-mas flow $\vec{u}$ is equivalent to the ion flow $\vec{u}_i$ if the electron mass is assumed to be negligible. Fig. 2.2 (b) shows the ion and electron flow tubes $\psi_\sigma = \int \vec{u}_\sigma \cdot d\vec{S}$. The electron flow is parallel to the magnetic field since the electrons are assumed to be massless. The ion flow is fully constrained by the imposed current $\vec{j}$ and the electron speed ($\vec{j} = n_0 (q_i \vec{u}_i - q_e \vec{u}_e)$). Both species flow tubes overlap and the species flows remain on the flow flux tube. Fig. 2.2 (c) shows the species flow vorticity flux tubes $\mathscr{F} = \int \vec{\omega}_\sigma \cdot d\vec{S}$. Since there is no azimuthal current the azimuthal species velocities and axial flow vorticity are identical. The axial flow vorticity dominates the radial flow vorticity at moderate to large currents, collimating the vorticity flux tubes. Fig. 2.2 (d) shows the canonical momentum tubes $\mathscr{P}_\sigma = \int \vec{P}_\sigma \cdot d\vec{S}$. These flux surfaces are distinct for each species; while the electron canonical momentum is identical to the magnetic vector potential surface, the ion flow momentum pulls the ion canonical momentum surface towards the ion velocity surfaces. Fig. 2.2 (e) shows flux tubes of canonical circulation: the canonical flux tubes. The electron canonical flux tube overlaps with the magnetic flux tube. The ion canonical flux tube collimates due to the contribution of the axial flow vorticity. When a diffuse current profile with a low ratio of current to vacuum magnetic flux and high electron velocity is imposed the canonical flux tube can bulge as shown in fig. 2.3. This resembles the bulging of emission observed in astrophysical jets launched from planetary nebula. It is important to remember that these visualizations were based on a kinematic model. The canonical quantities were determined directly by the imposed current, vacuum magnetic

field, electron flow velocity, and ignoring the self-consistent dynamics of canonical equations of motion (which would require a full simulation). This explains why the ion flows seem to violate the canonical frozen-in condition by following the magnetic flux surface instead of the ion canonical circulation flux surface. Nevertheless, canonical flux tubes offer a topological view of, in this case, two-fluid physics and indicate that two-fluid effects may contribute to collimation and bulge formation. The detailed dynamics and the reason the two-fluid terms of the equation of motion should be significant at the long scales of an astrophysical jet (see fig. 1.1) remain to be identified.

As is illustrated in the canonical flux tube visualization, driving a current along a canonical flux tube twists up the magnetic field and flows, twisting the canonical flux tube. A current is generated by applying a potential difference across the ends of the canonical flux tube. More generally applying an enthalpy difference will lead to canonical flux tubes twisting, writhing, and interlinking. This twist, writhe, and interlinking of canonical flux tubes is called canonical helicity. In many systems, it has been observed that the helicity is well conserved compared to the system energy leading to a constrained relaxation towards a minimum energy state for a given helicity [31]. The minimum energy states for plasmas with a finite magnetic helicity (twisted magnetic flux tubes) have been widely studied. In a simply connected vacuum chamber the minimum energy state is called a spheromak, in a toroidal vacuum chamber it is called a reversed-field configuration (RFP) [24]. Spheromaks and RFPs have been observed in many experiments, see [32, 33]. In RFPs relaxation occurs in sawtooth events which are discrete in time. The internal magnetic field profile evolution in RFPs during these sawtooth crashes has been reconstructed from equilibrium models using measured pressure and edge magnetic fields. The decay rate of magnetic energy is 2-4 times as large as the decay rate of magnetic helicity [34].

However, there are also experimental observations that indicate a fast conversion of twisted magnetic flux tubes into twisted flow vorticity flux tubes. Attempts to generalize relaxation theory to plasmas with canonical helicity [35] have identified the field-reverse configuration as a possible minimum energy state for plasmas with finite fluid helicity (twisted flow vorticity flux

tubes) but no (negligible) magnetic helicity. Both spheromaks and FRCs are compact toroids (see fig. 2.4). It is useful to define a coordinate system consisting of toroidal and poloidal directions. A spheromak has both toroidal and poloidal magnetic fields, while an FRC only has poloidal magnetic field with no (or a weak) toroidal field. At the TS-4 experiment [36] two spheromaks with opposite magnetic helicity but no flows merge and may form either a single spheromak with finite magnetic helicity or a field-reversed configuration (FRC) with negligible magnetic helicity and finite fluid helicity (fig. 2.4). The formation of either a spheromak or FRC is a bifurcation characterized by a sharp threshold depending on the initial helicity in the two merging spheromaks [28]. The initial helicity of the spheromaks can be characterized by the eigenvalue λ which quantifies the amount of current parallel to the magnetic field. If the magnetic helicity in the spheromaks is exactly equal and opposite $\lambda = 0$, otherwise λ will be a finite value. The minimum λ eigenvalue necessary to form a spheromak is called λ_{Taylor}. Fig. 2.5 shows the threshold λ/λ_{Taylor} to be exceeded by the merging spheromaks for a spheromak to form, against the size parameter S^*, which is the ratio of the ion Larmor radius r_{Li} over the system length L. S^* is varied by using different gases, i.e. varying the ion mass to change r_{Li} while keeping L constant. If the initial λ/λ_{Taylor} eigenvalue is below (above) the threshold a FRC (a spheromak) is formed. Ref. [28] observed that initially the magnetic helicity (λ) decreases in all merging experiments through magnetic reconnection. The outflows from the magnetic reconnection cause a 'slingshot' effect which generates magnetic activity, ion shear flows, and ion heating [37, 38, 39]. The magnetic activity cascades from high (Fourier) mode numbers down to low mode numbers. Magnetic activity at low mode numbers has been observed at other experiments to amplify poloidal magnetic flux and may cause the increase in magnetic helicity for spheromak end states. However, for low values of the size parameter the shear flows become large and their ratio with the Alfvén velocity exceeds a value that has been shown to stabilize low mode number instabilities in current-carrying plasma columns with no axial magnetic fields (Z-pinches) [15, 40]. This shear flow stabilization may prevent the restoration of magnetic helicity and lead to the formation of an FRC.

Ref. [28] explains in a qualitative matter the mechanisms responsible for the bifurcation;

however, to understand the observed threshold quantitative models are needed of: 1) the partitioning of the magnetic energy into magnetic activity, ion heating, and ion flows, 2) the strength of the flux amplification through low mode magnetic activity, 3) the threshold of azimuthal shear flow stabilization (in a toroidal geometry), and 4) possible nonlinear feedback between this three mechanisms. The theory of canonical helicity can provide a complementary explanation, sidestepping the detailed mechanisms and instead focusing on the transfer of electron to ion canonical helicity. The shear flow and ion heating occurring during the spheromak merging at the TS-4 experiment represent an enthalpy difference being applied to a canonical flux tube. To understand under what conditions the enthalpy leads to injection of electron (magnetic) helicity and the formation of a spheromak or the injection of ion (fluid flow) helicity the canonical helicity transport equations must be derived and analyzed.

2.3 Canonical helicity

Helicity is the volume integral of vector potential of a flux quantity dotted with the flux quantity. For canonical flux tubes the canonical helicity is defined as

$$K_\sigma = \int \vec{P}_\sigma \cdot \vec{\Omega}_\sigma \, dV. \tag{2.11}$$

Substituting the two-fluid limits of expressions (2.1) and (2.2) yields

$$K_\sigma = \int n_\sigma^2 \left[m_\sigma^2 \vec{u}_\sigma \cdot \vec{\omega}_\sigma + q_\sigma^2 \vec{A} \cdot \vec{B} + m_\sigma q_\sigma \left(\vec{u}_\sigma \cdot \vec{B} + \vec{A} \cdot \vec{\omega}_\sigma \right) \right] + \vec{P}_\sigma \cdot \left(\frac{\nabla n_\sigma}{n_\sigma} \times \vec{P}_\sigma \right) dV. \tag{2.12}$$

Since a cross product dotted with one of its factors is always zero the last term can be dropped. While canonical flux tubes depend on density gradients, their canonical helicity is independent of any density gradients but is locally weighted by the square of the number density n_σ^2. Applying vector identities and Gauss's theorem a surface integral can be extracted from the volume integral

$$K_\sigma = \int n_\sigma^2 \left[m_\sigma^2 \vec{u}_\sigma \cdot \vec{\omega}_\sigma + q_\sigma^2 \vec{A} \cdot \vec{B} + 2 m_\sigma q_\sigma \vec{u}_\sigma \cdot \vec{B} \right] dV + \int n_\sigma^2 \vec{A} \times \vec{u}_\sigma \cdot d\vec{S}. \tag{2.13}$$

For an isolated system where neither the flow vorticity nor the magnetic field penetrate the integration volume, the three vectors $\vec{A}$, $\vec{u}_\sigma$, and $d\vec{S}$ are coplanar so the last term of the equation

above is zero. The canonical helicity can be expressed as the sum of three helicities

$$K_\sigma = \mathscr{H}_\sigma + \mathscr{X}_\sigma + \mathscr{K}_\sigma. \tag{2.14}$$

The three helicites are the fluid helicity

$$\mathscr{H}_\sigma = m_\sigma^2 \int n_\sigma^2 \vec{u}_\sigma \cdot \vec{\omega}_\sigma \, dV, \tag{2.15}$$

the cross helicity

$$\mathscr{X}_\sigma = 2 m_\sigma q_\sigma \int n_\sigma^2 \vec{u}_\sigma \cdot \vec{B} \, dV, \tag{2.16}$$

and the magnetic helicity

$$\mathscr{K}_\sigma = q_\sigma^2 \int n_\sigma^2 \vec{A} \cdot \vec{B} \, dV. \tag{2.17}$$

Canonical helicity can thus be understood as the sum of helicity between flux tubes of flow vorticity $\int \vec{\omega}_\sigma \cdot \vec{S}$, helicity between flux tubes of magnetic field $\int \vec{B} \cdot \vec{S}$, and helicity due to interlinking between the flow vorticity and magnetic flux tubes. While many references define the fluid, cross, and magnetic helicities without the mass and charge factors, this thesis will include them to make the units uniform for all helicities. In the limit of negligible electron mass, electron canonical flux tubes reduce to magnetic flux tubes. The ion canonical flux tubes are a weighted sum of flow vorticity and magnetic flux tubes. The spheromak discussed previously has only magnetic helicity and no fluid helicity, so its electron and ion helicities are equal if the ions are singly ionized. The FRC has no (negligible) magnetic helicity and significant fluid helicity, so its electron helicity is negligible but its ion helicity is significant.

2.3.1 Relative canonical helicity

The expressions for canonical helicity presented so far are only unambiguous for isolated systems where neither the flow vorticity $\vec{\omega}_\sigma$ nor the magnetic field $\vec{B}$ penetrate the integration boundary. When current is driven along magnetic flux tubes by a static potential difference applied to electrodes, the magnetic flux tubes necessarily intersect the electrode. If flux tubes penetrate the integration volume there is an ambiguity in their helicity since the linkages outside the volume cannot be measured [24] (fig. 2.6). Moreover, ref. [39] notes that each time

Figure 2.6: Gauge-dependence of helicity. An integral over the gray volume cannot count the number of linkages of the two flux tubes external to the grey volume. Adapted from [24]. Image credit: Jaclyn Lake.

species flux tubes intercept, as would be the case in the spheromak merging experiments there is a gauge dependent ambiguity since species integration volumes are intersecting. To alleviate this problem, ref. [39] introduces a gauge-invariant relative canonical helicity

$$K_{\sigma rel} = \int \vec{P}_{\sigma-} \cdot \vec{\Omega}_{\sigma+} \, dV, \tag{2.18}$$

where the plus and minus subscripts offset a vector $\vec{X}$ by a reference field $\vec{X}_{pm} = \vec{X} \pm \vec{X}_{ref}$. The canonical reference field must satisfy the conditions [16]

$$\begin{aligned} \nabla \times \vec{\Omega}_{\sigma ref}\big|_V &= 0, \\ \vec{\Omega}_{\sigma ref} \cdot d\vec{S} &= -\vec{\Omega}_{\sigma} \cdot d\vec{S}, \text{and} \\ \nabla \times \vec{P}_{\sigma ref} &= \vec{\Omega}_{\sigma ref}. \end{aligned} \tag{2.19}$$

This means that the normal component of the reference canonical circulation must be equal and opposite to the canonical circulation at the integration boundary. The three helicity expression eq. (2.14) of canonical helicity can be written in relative form

$$K_{\sigma rel} = \mathscr{H}_{\sigma rel} + \mathscr{X}_{\sigma rel} + \mathscr{K}_{\sigma rel}. \tag{2.20}$$

The three helicities are the relative fluid helicity

$$\mathscr{H}_{\sigma rel} = m_\sigma^2 \int n_\sigma^2 \vec{u}_{\sigma-} \cdot \vec{\omega}_{\sigma+} \, dV, \tag{2.21}$$

the relative cross helicity

$$\mathscr{X}_{\sigma rel} = m_\sigma q_\sigma \int n_\sigma^2 \left(\vec{u}_{\sigma-} \cdot \vec{B}_+ + \vec{u}_{\sigma+} \cdot \vec{B}_- \right) dV, \tag{2.22}$$

and the relative magnetic helicity

$$\mathscr{K}_{\sigma rel} = q_\sigma^2 \int n_\sigma^2 \vec{A}_- \cdot \vec{B}_+ \, dV. \tag{2.23}$$

2.3.2 Canonical helicity transport

The canonical helicity evolution equation can be obtained by combining the derivative of eq. (2.18) with the canonical Ohm's law eq. (2.5) and induction equation eq. (2.7)

$$\frac{dK_{\sigma rel}}{dt} = -\int \left(\vec{\Sigma}_{\sigma+} \cdot \vec{\Omega}_{\sigma-} + \vec{\Sigma}_{\sigma-} \cdot \vec{\Omega}_{\sigma+} \right) dV - \int h_{\sigma-} \vec{\Omega}_{\sigma+} \cdot d\vec{S} - \int \vec{P}_{\sigma-} \times \frac{\partial \vec{P}_{\sigma+}}{\partial t} \cdot d\vec{S} + \int \left(\vec{P}_{\sigma-} \cdot \vec{\Omega}_{\sigma+} \right) \vec{v}_\sigma \cdot d\vec{S}. \tag{2.24}$$

The integral terms represent a dissipative term, a battery term, an inductive term, and a Leibniz term due to the boundary motion $\vec{v}_\sigma$ of the system, respectively. The battery term injects helicity when an enthalpy difference is applied to a canonical flux tube, e.g. when a potential is applied at the ends of a canonical flux tube or, in the case of spheromak merging, when kinetic energy and pressure differences are created by reconnecting canonical flux tubes. To understand when helicity is preferentially injected as magnetic or fluid helicity it is helpful to look at the battery term in isolation. In the limit of constant density, the ion battery term can be rewritten to differentiate between injection into the flow vorticity flux tube and the magnetic flux tube

$$\dot{K}_{irel} = m_i \Delta h_{i-} \int \vec{\omega}_{i+} \cdot d\vec{S} + q_i \Delta h_{i-} \int \vec{B}_+ \cdot d\vec{S} \tag{2.25}$$

where Δh_{i-} is the relative enthalpy difference across the ion canonical flux tube. The ratio of helicity injection into the magnetic and flow vorticity flux tubes can be defined as the helicity injection threshold [39]

$$\bar{K}_{thr} \equiv \frac{|m_i \Delta h_i \int \vec{\omega}_i \cdot d\vec{S}|}{|q_i \Delta h_i \int \vec{B} \cdot d\vec{S}|} \sim \frac{m_i u_{ith}}{q_i L B} \sim \frac{r_{Li}}{L} \sim \frac{1}{S^*}. \tag{2.26}$$

Applying dimensional analysis to the helicity injection threshold recovers the same size parameter S^* dependence as was observed in the TS-4 experiment [1].

To understand when a species helicity dissipates slower than the system energy, i.e. is a suitable relaxation constraint, the species helicity evolution has to be compared to the energy evolution. Ref. [26] derives an energy H evolution equation from the same Lagrangian-Hamiltonian formation that was used to derive the canonical evolution equations, with a dissipative term $dH/dt = -\int \vec{\Sigma}_\sigma \cdot \vec{u}_\sigma \, dV$. In a fixed, isolated system, with no intersecting species flux tubes, all terms except for frictional dissipation can be ignored and no reference fields are needed

$$\frac{\Delta K_\sigma}{\Delta H_\sigma} = 2 \frac{\int \vec{R}_\sigma \cdot \vec{\Omega}_\sigma \, dV}{\int \vec{R}_\sigma \cdot \vec{u}_\sigma \, dV} \frac{H_{\sigma 0}}{K_{\sigma 0}} \sim 2 \frac{\vec{\Omega}_\sigma}{\vec{u}_\sigma}. \tag{2.27}$$

[1] Strictly the dimensional analysis can not specifically identify the ion Larmor radius r_{Li}. The analysis identifies a dependence on the square root of the ion mass $\sqrt{m_i}$, since $u_{ith} \sim 1/\sqrt{m_i}$. The ion skin depth also has a dependence on $\sqrt{m_i}$.

Applying dimensional analysis to the factor yields

$$\frac{\Delta K_\sigma}{\Delta H_\sigma} = 2\frac{\vec{\Omega}_\sigma}{\vec{u}_{\sigma 0}} \sim 2\frac{\frac{n_0 m_\sigma u_{\sigma 0}}{L} + q_\sigma B_0}{u_0} \tag{2.28}$$

where $u_{\sigma 0}$ and B_0 are the characteristic species flow and magnetic field, respectively. A characteristic Larmor radius r_L can be identified by pulling out the mass density and then a factor of $q_\sigma B$

$$\frac{\Delta K_\sigma}{\Delta H_\sigma} \sim \frac{n_0 m_\sigma}{\frac{L m_\sigma u_{\sigma 0}}{2(u_{\sigma 0} m_\sigma + q_\sigma B_0 L)}} \sim \frac{n_0 m_\sigma}{\frac{\frac{L m_\sigma u_{\sigma 0}}{q_\sigma B_0}}{2\left(\frac{u_{\sigma 0} m_\sigma}{q_\sigma B_0} + L\right)}} \sim \frac{\rho_0}{\frac{L r_L}{2(r_L + L)}} \sim \frac{\rho_0}{\frac{1}{2\left(\frac{1}{L} + \frac{1}{r_L}\right)}}. \tag{2.29}$$

The ratio of the rate of change in species helicity over energy scales with a mass density gradient. Identifying L as a characteristic scale of circulation L_c, the scale length $L_H = 1/(1/L_c + 1/r_{Li})$ is a hybrid length of a characteristic length of flow circulation L_c and a characteristic Larmor radius r_L. When the mass density gradient scale is below the hybrid length L_H then the species helicity will decay faster than energy. Ref. [39] shows that while one species helicity may decay the other will increase so that the sum of the ion and electron helicities will be conserved.

2.4 Conclusion

Recent theoretical work and experimental observations suggest that canonical flux tubes and their helicity may provide a complimentary point of view, valid across all scales of plasma physics. The theory of canonical helicity transport recovers the plasma size parameter S^* scaling dependence which is observed as the bifurcation threshold in compact toroid merging experiments. Density gradients on the scale of the species Larmor radius r_L for magnetically dominated plasmas have been identified as the condition under which species helicity decay and conversion to the other species may occur. Analytical models have explored the kinematic evolution of canonical flux tubes. This thesis aims to connect these previous results with experimental physics by 1) identifying dynamics in current-carrying magnetic flux tubes that may create small-scale structures at which the density gradients could meet conditions for species

helicity decay and conversion, 2) developing experimental and numerical techniques to facilitate the measurement of large scale datasets needed to reconstruct canonical flux tubes and their helicity from measurements, and 3) reconstructing canonical flux tubes and their helicity from measurements of a linear plasma column.

Chapter 3

SAUSAGE INSTABILITIES ON TOP OF KINKING LENGTHENING CURRENT-CARRYING MAGNETIC FLUX TUBES

This chapter will theoretically explore the possibility of sausage instabilities developing on top of a kink instability in lengthening current-carrying magnetic flux tubes. Recent laboratory experiments suggest that hierarchies of instabilities, such as kink and Rayleigh-Taylor, could create small scale structure and density gradients at which conversions of electron (magnetic) and ion (fluid) canonical helicity may occur. Sausage instabilities can also provide this coupling mechanism between disparate scales. Flux tube experiments can be classified by the flux tube's evolution in a configuration space described by a normalized inverse aspect-ratio $\bar{k}$ and current-to-magnetic flux ratio $\bar{\lambda}$. A lengthening current-carrying magnetic flux tube traverses this $\bar{k}$ - $\bar{\lambda}$ space and crosses stability boundaries. We derive a single general criterion for the onset of the sausage and kink instabilities in idealized magnetic flux tubes with core and skin currents. The criterion indicates a dependence of the stability boundaries on current profiles and shows overlapping kink and sausage unstable regions in the $\bar{k}$ - $\bar{\lambda}$ space with two free parameters. Numerical investigation of the stability criterion reduces the number of free parameters to a single one that describes the current profile, and confirms the overlapping sausage and kink unstable regions in $\bar{k}$ - $\bar{\lambda}$ space. A lengthening, ideal current-carrying magnetic flux tube can therefore become sausage unstable after it becomes kink unstable.

The primary part of this chapter is reproduced from von der Linden, J., & You, S. (2017). Sausage instabilities on top of kinking lengthening current-carrying magnetic flux tubes. Physics of Plasmas, 24(5), 052105 [41], with the permission of AIP Publishing.

The code used for the analysis and to generate figs. 3.3, 3.4, 3.5, and 3.6 in this chapter has been published on Zenondo at

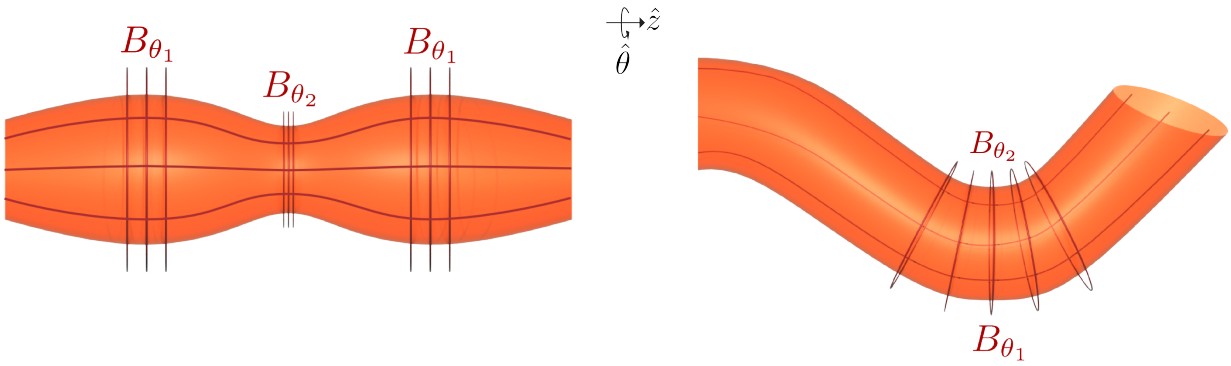


Figure 3.1: a) Sausage $m = 0$ instability. Increased B_{θ_2} exerts an increased pinch force. Magnetic pressure and tension from compressed and bent magnetic field resist the perturbation. b) Kink $m = 1$ instability. The magnetic pressure gradient in B_θ from the inboard side to the outboard side of the perturbation accelerates the perturbation. Magnetic tension from bent axial magnetic field resists the perturbation.

https://doi.org/10.5281/zenodo.230489 and is included in appendix A [42].

The dataset underlying fig. 3.5 and fig. 3.6 has been published on Zenodo at https://doi.org/10.5281/zenodo.230611 [43].

3.1 Background

Magnetic flux tubes are defined as a bundle of magnetic field lines that run through a closed contour [44]. When current flows along a magnetic flux tube, several current-driven instabilities may develop (fig. 3.1), which are generally well understood with the ideal magnetohydrodynamic (MHD) behavior of current-carrying flux tubes [45], where for mathematical convenience, the column is modeled by an infinitely long cylinder and classified into one of two ideal analytical models: a skin screw pinch (with an axial magnetic field and an electrical

current flowing only on an infinitely thin boundary layer separating the plasma from vacuum) [46], or a diffuse pinch (with an axial magnetic field, no skin current, but some distribution of current inside the plasma) [47]. In all cases, classical instability onset conditions are retrieved from the Energy Principle [48] on Fourier perturbed states of the idealized current-carrying magnetic flux tube: i.e. starting from a static equilibrium like a ball at rest on a hill or valley, if the perturbed potential energy is negative, the system would be unstable to that particular mode. Indications that the coupling of different types of instabilities may play a role in astrophysical, heliospheric, and laboratory phenomena have motivated a renewed interest in revisiting and extending classical instability criteria. MHD instabilities may be responsible for structure formation in astrophysical jets [49] and understanding these structures may provide insight into activity at the source of the jets. Kink and sausage instabilities have been observed in coronal loops and could trigger large energy releases and heating of the solar corona [18]. Instabilities appear to be necessary for the formation of toroidal plasma configurations from initially open flux tubes, such as spheromaks [50, 51] and spherical tori [25]. Recent experiments have identified an instability cascade, where a macroscopic instability triggers a microscopic instability, creating small scale structures which directly couple the disparate plasma scales [52]: the acceleration of a growing kink on the scale length of the flux tube acts as an effective gravity, leading into a Rayleigh-Taylor instability. The Rayleigh-Taylor instability forms structures on the scale of ion inertial lengths, enabling fast reconnection [23]. Two neighboring current-carrying magnetic flux tubes may kink on system scale lengths, attract each other, merge, reconnect on ion inertial lengths, and launch ion-cyclotron waves [18]. Spheromak merging experiments observe annihilation of magnetic helicity followed by the formation of a Field-Reversed Configuration (FRC) with ion flows, when the scale lengths approach ion Larmor radii [28, 39]. Partially-toroidal flux tubes in a solar loop laboratory experiment erupt when they are unstable to both the torus and kink instabilities, but fail to erupt when they are unstable to only one of the instabilities [29]. In toroidal fusion devices the merging of discrete temperature flux tubes has been associated with global sawtooth events [53]. To identify possible couplings between MHD instabilities, it is necessary to examine current-carrying magnetic

flux tube stability beyond the conditions used to derive classical stability criteria. Finite-length screw pinches with one or both ends tied down have kink instability criteria modified from classical Kruskal-Shafranov values [54], explaining some experimental results [55]. Sheared axial plasma flow stabilizes the Z-pinch to kink and sausage instabilities [15], an effect subsequently observed in the laboratory [40]. In many flux tube experiments the flux tube is bound between two electrodes and remains at a fixed length while the current ramps up. In contrast, a flared current-carrying magnetic flux tube generated from one boundary, gives rise to strong axial plasma flows before collimating into a filamentary cylinder. This collimation occurs continuously as the flows convect magnetic flux into the flared end of the flux tube, resulting in a lengthening of the collimated flux tube while the current ramps up. This behavior was described theoretically [21] and observed experimentally [22]. The same flux tube has then been observed to become kink unstable, detach from the source electrodes and form a spheromak [50]. The detachment of the flux tube may be initiated by a sausage-like instability, but comparisons between measurements and theory currently rely on stability criteria derived for skin screw-pinch models of infinite length, even though the experiment's flux tube has a distributed current and lengthens during the shot duration. This paper presents a single general stability criterion for the onset of ideal MHD instabilities in a lengthening current-carrying magnetic flux tube with both a diffuse internal and sharp skin current and comparisons to numerical calculations.

3.2 Theory

Here, the general, minimized form of the perturbed potential energy of the system is subdivided into 3 terms: the first (internal) term is simplified with Newcomb's analysis [47] of internal stability; the second (skin) term is further simplified by considering Bellan's analysis [21] of a self-collimating flared flux tube; the third (vacuum) term is simplified by assuming the wall is at infinity. Differing from standard treatments, none of the terms are subsequently set to zero in the analytical treatment (fig. 3.2).

The starting point is the general, minimized, 1D perturbed potential energy of the current-

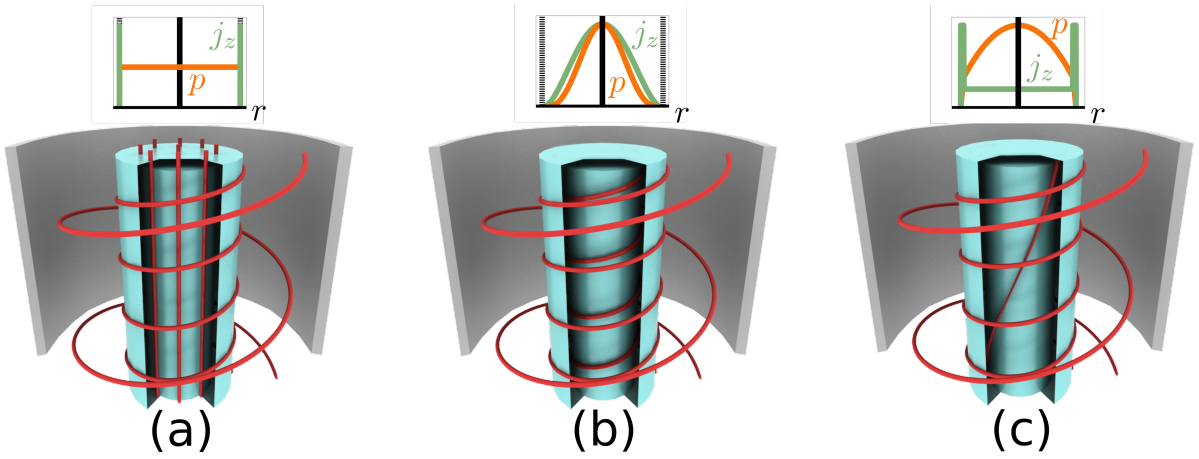


Figure 3.2: Three idealized flux tubes. 2D plots show axial current density and pressure profiles. Axial magnetic field is uniform for all cases. 3D figures show magnetic field (red) in the core, skin of the flux tube and in vacuum. a) The skin current case used by Tayler and Kruskal-Shafranov. b) Diffuse current case used by Newcomb. c) Core-skin current case used in this derivation.

carrying magnetic flux tube $\delta W(\xi)$ which depends only on the radial displacement $\xi(r)$ of the flux surface at radial position r. This real 1D scalar expression is reduced from $\delta W(\vec{\xi})$, a complex expression of the infinitesimal 3D displacement $\vec{\xi} = \xi(r)\hat{r} + \eta(\phi)\hat{\phi} + \zeta(z)\hat{z}$, with classical assumptions: the idealized cylindrically-symmetric flux tube with periodicity length L, radius a, is surrounded by vacuum, no walls (wall at radius $r = b \to \infty$), and subdivided into the internal (plasma) volume between $0 \leq r \leq p$ and a thin shell (interface or skin) between $p \leq r \leq v$, where p is the 'plasma' side of a and v is the 'vacuum' side of a. The plasma pressure is isotropic and inviscid. The plasma is incompressible, $\nabla \cdot \vec{\xi} = 0$, which is the most unstable worst case scenario and allows reduction of the 3D $\vec{\xi}$ to a 1D ξ. This assumption also removes pressure-driven instabilities from consideration, leaving only the current-driven contributions. The fundamental instabilities are the sausage ($m = 0$, $n \geq 1$) and kink ($|m| \geq 1$, $n \geq 1$) modes, where m is the azimuthal mode number and n the longitudinal mode number of the Fourier perturbation on the cylinder. The perturbations are small and linear, to ignore

higher order terms, and adiabatic, so the perturbed plasma pressure can be simply expressed in terms of the equilibrium pressure and the displacement ξ. The longest (worst-case, $n = 1$) perturbation wave length is assumed so that $k = 2\pi/L$. The system is described by ideal MHD so the perturbed magnetic field can be expressed simply in terms of the plasma displacement and the equilibrium magnetic field. The system is assumed initially to be in static equilibrium, so equilibrium quantities are not time dependent and integration constants can be put to zero.

The linearly perturbed potential energy δW is thus integrated by parts into contributions δW_{pl} from the plasma, δW_i from the interface and δW_v from the vacuum [45]. The δW_{pl} term represents the available free energy of the plasma, with contributions from the internal vacuum magnetic field, the plasma pressure and internal currents (δW_{B_1}, δW_{p_1}, δW_{J_0} respectively). The δW_i term represents the work required to move the boundary between the plasma and the vacuum. The δW_v represents the energy transferred to the external vacuum magnetic fields. From the Energy Principle, the flux tube is stable if the total perturbed potential energy $\delta W = \delta W_{pl} + \delta W_i + \delta W_v \geq 0$ and unstable otherwise. The general, minimized form of the perturbed potential energy of the system is thus subdivided into three terms and, differing from standard treatments, none of the terms are subsequently set to zero in our analytical treatment. We simplify the first (internal) term with Newcomb's analysis [47] of internal stability, the second (skin) term with Bellan's analysis [21] of a self-collimating flared flux tube, and the third (vacuum) term with the assumption the wall is at infinity. We now look at each term individually. The incompressible plasma's perturbed potential energy δW_{pl} is given by [47]

$$\delta W_{pl} = \frac{\pi L}{2\mu_0}\left(\int_0^p \left[f\left(\frac{\partial \xi}{\partial r}\right)^2 + g\xi^2\right] dr + \left[h\xi^2\right]_0^p\right) \tag{3.1}$$

where $f = r(krB_z + mB_\phi)/(k^2r^2 + m^2)$, $h = (k^2r^2B_z^2 - m^2B_\phi^2)/(k^2r^2 + m^2)$ and g is also a function of k, r, B_z, B_ϕ that will soon be eliminated, and m is the azimuthal mode number. The first term of eq. (3.1) can be minimized if ξ is the solution to the associated Euler-Lagrangian equation

$$\frac{\partial}{\partial r}\left[f\frac{\partial \xi}{\partial r}\right] - g\xi = 0. \tag{3.2}$$

Eq. (3.2) has a singular point in the flux tube whenever $f = 0$, i.e. wherever $q = m$, having defined $q = 2\pi r B_z/(L B_\phi)$, corresponding to rational mode surfaces $q = m/n$ with $n = 1$. Solutions ξ thus only exist in regions between two singular points (called an 'interval'). It is sufficient to only consider stability to the $m = 0, \pm 1$ modes because if a column is stable to both, it is also stable to higher order modes [47]. So the 'no singular point' condition becomes $q \neq 0, \pm 1$. Newcomb [47] proves extensively the conditions for the existence of solutions and the conditions for internal stability ($\delta W_{pl} \geq 0$) of a collimated magnetic flux tube: a collimated magnetic flux tube is stable if and only if (a) the Suydam criterion [56] is satisfied for $r > 0$ (but doesn't have to be at $r = 0$); (b) the Euler-Lagrangian solutions satisfy the 'small' conditions on the left-hand side of an interval, namely

$$\xi(0) = 0 \quad \text{if} \quad m \neq \pm 1 \quad \text{and} \quad \frac{\partial \xi}{\partial r} = 0 \quad \text{if} \quad m = \pm 1 \tag{3.3}$$

and (c) the Euler-Lagrangian solution doesn't vanish twice in any interval. The existence of singularities and solutions are dependent on the experimental conditions, so numerical codes are generally employed to solve eq. (3.2). But we proceed analytically for now.

The lengthening flux tube is initially stable, therefore the three conditions (a), (b), and (c) are fulfilled. Early in the discharge, the collimated flux tube is short (L small) and stable (small B_ϕ), so $q \approx r B_z/(L B_\phi) >> 1$. As the collimated flux tube lengthens and current increases, q approaches one from above and is always non-zero. Hence, just before the onset of instability, there are no singular points inside the flux tube and we can consider the whole plasma cross-section as a unique 'interval' where the three Newcomb stability conditions are fulfilled, in particular, conditions (b) and (c). So, except at $r = 0$, the solution $\xi(r) \neq 0$ and the Euler-Lagrangian eq. (3.2) is satisfied, giving $g = \partial/\partial r[f\,\partial\xi/\partial r]/\xi$, which can be substituted into eq. (3.1) to give

$$\delta W_{pl} = \frac{\pi L}{2\mu_0}\left(\left[f\,\xi\frac{\partial \xi}{\partial r}\right]_0^p + \left[h\xi^2\right]_0^p\right). \tag{3.4}$$

Strictly, the lower bound is at $r_c < a$ with $r_c \to 0$, but those terms vanish since, at the flux

tube center, the solutions all satisfy the Newcomb small-value conditions $\xi(0) = 0$ or $\xi'(0) = 0$ where $'$ denotes $\partial/\partial r$. At the plasma edge $r = p$, we take $\xi(p) = \xi(a) \equiv \xi_a$, $\xi'(p) \approx \xi_a\ \delta/a$ where δ represents a 'rigidness' of the displacement of the boundary and $f(p) = f(a)$. The plasma term eq. (3.4), thus becomes

$$\delta W_{pl} = \frac{\pi L}{2\mu_0}\delta\frac{f(a)}{a}\xi_a^2 + \frac{\pi L}{2\mu_0}h(a)\xi_a^2. \tag{3.5}$$

For the interface term δW_i, the standard derivation [57] of the perturbed potential energy gives

$$\delta W_i = \frac{\pi a L}{2}\xi_a^2\left(\left[\frac{\partial}{\partial r}\frac{B^2}{2\mu_0}\right]_v - \left[\frac{\partial}{\partial r}\left(P + \frac{B^2}{2\mu_0}\right)\right]_p\right) \tag{3.6}$$

where $B^2 = B_\phi^2 + B_z^2$ and the plasma pressure P are the equilibrium values. This term only exists if a surface skin current exists and represents the work required to move the plasma boundary. Since we assumed the starting point to be a static, axisymmetric equilibrium, then the pinch force $-B_\phi/(\mu_0 r)$ balances the total pressure gradient $\partial/\partial r[P + B^2/(2\mu_0)]$. The 'jump' in the current profile from the core plasma current to the skin current value is represented by $B_\phi(p)$ being different to $B_\phi(v)$ giving

$$\delta W_i = \frac{\pi L}{2\mu_0}\xi_a^2\left(B_{\phi p}^2 - B_{\phi v}^2\right). \tag{3.7}$$

The classical skin current model assumes no plasma current inside the flux tube, i.e. $B_{\phi p} = 0$, but we keep both terms here. The vacuum contribution δW_v is given by the usual expression [57] (with wall radius $b \to \infty$)

$$\delta W_v = -\frac{\pi L}{2\mu_0}(kaB_{zv} + mB_{\phi v})^2\frac{\xi_a^2}{ka}\frac{K_m(|ka|)}{K_m'(|ka|)} \tag{3.8}$$

where K_m is the modified Bessel function of the second kind of order m. Reassembling eqs. (3.5), (3.7), and (3.8) gives the total perturbed potential energy as

$$\begin{aligned}\delta W = \quad & \frac{\pi L}{2\mu_0}\delta\frac{f(a)}{a}\xi_a^2 + \frac{\pi L}{2\mu_0}h(a)\xi_a^2 \\ & + \frac{\pi L}{2\mu_0}\xi_a^2(B_{\phi p}^2 - B_{\phi v}^2) \\ & - \frac{\pi L}{2\mu_0}(kaB_{zv} + mB_{\phi v})^2\frac{\xi_a^2}{ka}\frac{K_m(|ka|)}{K_m'(|ka|)},\end{aligned} \tag{3.9}$$

which is compared to zero to determine the onset of stability. After dividing by $\pi L/(2\mu_0)\xi_a^2$ and defining $\bar{k} \equiv ka$ and $m \to -m$, for convenience since for the kink the most unstable modes will be perpendicular to the magnetic field ($kB_\theta + mB_z = 0$), the flux tube is stable if

$$\begin{aligned}\frac{(\delta+1)\bar{k}^2B_{zp}^2 + (\delta-1)m^2B_\phi^2 + 2\delta m\bar{k}B_{\phi p}B_{zp}}{\bar{k}^2+m^2} + B_{\phi p}^2 - B_{\phi a}^2 \\ - \frac{(mB_{\phi v} - \bar{k}B_{zv})^2}{\bar{k}}\frac{K_m(|\bar{k}|)}{K_m'(|\bar{k}|)} > 0.\end{aligned} \tag{3.10}$$

This is normally as far as we can go since B_{zp}, B_{za}, B_{zv} and $B_{\phi p}$, $B_{\phi a}$, $B_{\phi v}$ are generally independent parameters that have to be determined experimentally. To continue, we remember $\lambda = \frac{\mu_0 I}{\psi}$ with $\psi = B_z\pi r^2$, so Ampere's law $B_\phi = \mu_0 I/(2\pi r)$ can be rewritten as

$$\lambda_p = \frac{2B_{\phi p}}{aB_{zp}} \quad \text{and} \quad \lambda_v = \frac{2B_{\phi v}}{aB_{zv}} \tag{3.11}$$

for the plasma side and the vacuum side respectively. Previous experimental results [22] showed our starting point (collimated magnetic flux tube) results from MHD pumping in a flared flux tube [21]. The MHD pumping theory points out that once collimated, the magnetic flux tube must have uniform axial magnetic field across the interface, i.e. $B_{zp} = B_{za} = B_{zv}$, where the plasma current is entirely axial. Therefore, if the flux tube is fully collimated at the onset of instability, after defining dimensionless $\bar{\lambda} \equiv \lambda_v a$ and $\lambda_p a \equiv \epsilon\bar{\lambda}$, the stability condition eq. (3.10) reduces to

$$\frac{[2\bar{k}-m\epsilon\bar{\lambda}][(\delta+1)2\bar{k}-(\delta-1)m\epsilon\bar{\lambda}]}{\bar{k}^2+m^2} + (\epsilon^2-1)\bar{\lambda}^2 - \frac{(m\bar{\lambda}-2\bar{k})^2}{\bar{k}}\frac{K_m(|\bar{k}|)}{K'_m(|\bar{k}|)} \quad > 0 \tag{3.12}$$

with the conditions $B_{\phi v} = B_{\phi a} \neq 0$ and $\bar{\lambda} \neq 0$. If the left-hand side of (3.12) is negative for $m = 0$ then the flux tube is sausage unstable and if it is negative for $m = 1$ it is kink unstable. The Kruskal-Shafranov $m = 1$ kink condition can be retrieved from (3.12) by bearing in mind it is usually derived for a long, thin column ($k \to 0$) with a skin current and no internal currents ($\epsilon = 0$). For small argument, $K_1(x) \to x^{-1}$ and $K'_1(x) \to -x^{-2}$, which simplifies (3.12) to the Kruskal-Shafranov condition $\bar{k} > \bar{\lambda}/2$ for $\delta \to 0$. Remembering that $\bar{k} = 2\pi a/L$ and $\bar{\lambda} = \mu_0 I a/\psi$ where $\psi = B_z \pi a^2$ and $\mu_0 I = B_\phi/(2\pi a)$ gives the well-known $2\pi a B_z/(L B_\phi)$ in cylindrical geometry. The Tayler criterion [58] for sausage $m = 0$ stability in a skin screw-pinch can also be retrieved from (3.12) by taking the same limits. One obtains $\bar{\lambda} > 2\sqrt{2}$ for $\delta \to 1$, equivalent to the more familiar $B_z^2 > B_\phi^2/2$.

The stability condition (3.12) thus maps out a stability space (fig. 3.3). The kink instability region is hatched, to the right of the curve that approaches the classical Kruskal-Shafranov condition (dotted line $\bar{k} = \bar{\lambda}/2$). The region that is both sausage and kink unstable is cross-hatched, to the right of the curve that falls on the $\bar{k} = 0$ axis away from the origin. A short, low current magnetic flux tube ($\bar{k} \to$ large, $\bar{\lambda} \to 0$) thus starts out stable at the top left of the space (white region). As the column lengthens and current increases ($\bar{k} \to 0$, $\bar{\lambda} \to$ large), the column's ($\bar{k}$; $\bar{\lambda}$) point travels to the bottom-right of the space, crossing the $m = 1$ threshold to be kink unstable (hatched region). Eventually, the column may cross the $m = 0$ threshold and becomes sausage unstable (cross hatched). The parameter $\epsilon = \lambda_p a/\bar{\lambda}$ represents the ratio of the internal current to the skin current, and characterizes the 'hollowness' of the current profile across the flux tube, e.g., if $\epsilon = 0$ the plasma column has no internal current, only a skin current. The parameter $\delta = a\xi'(a)/\xi(a)$, represents the 'abruptness' or 'rigidness' of the displacement of the plasma boundary with respect to the internal plasma. If $\delta = 0$, the plasma cross-section moves rigidly radially without compressing, as in a kink instability. If $\delta = 1$ then

the boundary layer displacement is greater than the internal layers' displacement, as in the compressing sausage instability. Eq. (3.12) thus demonstrates three things: 1) a dependence of both the kink and sausage instability boundaries on the current profile ϵ; 2) a significant region that is *both* kink and sausage unstable; and 3) a dependence of the instability boundaries on the value of the rigidness parameter δ.

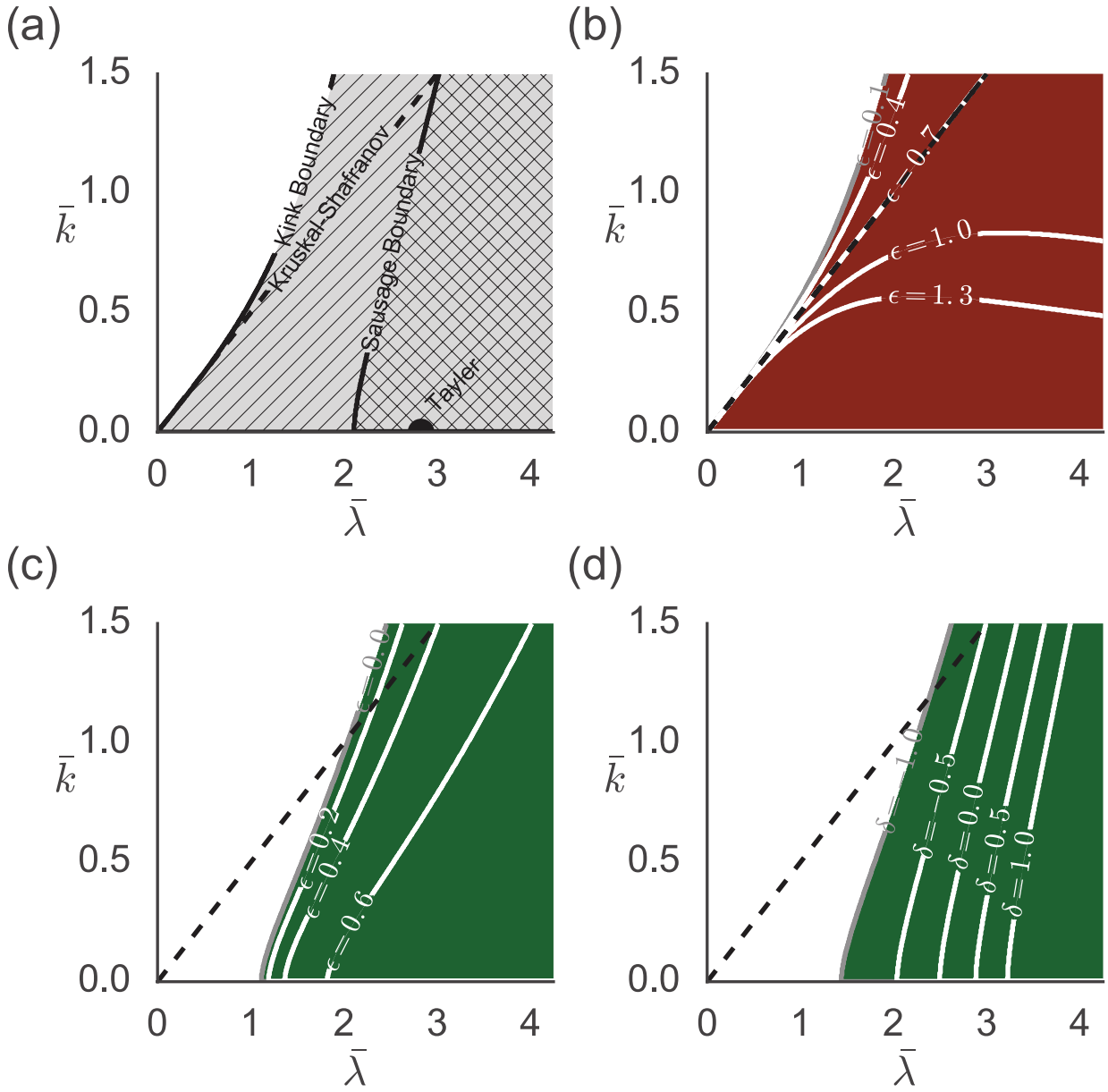


Figure 3.3: Analytical $\bar{k}$ - $\bar{\lambda}$ stability spaces parameterized with core current fraction ϵ and rigidness δ. a) Stability spaces for $\epsilon = 0.1$ and $\delta = 0.1$. White region is stable, gray is unstable, hatched region is kink ($m = 1$) unstable, and crosshatched region is unstable to both kink and sausage ($m = 0$) modes. b) Kink stability space dependence on ϵ with $\delta = 0$. Dark red region is unstable. c) Sausage stability space dependence on ϵ with $\delta = 0.7$. Dark green region is unstable. d) Sausage stability boundary dependence on δ with $\epsilon = 0.2$. Dark green region is unstable.

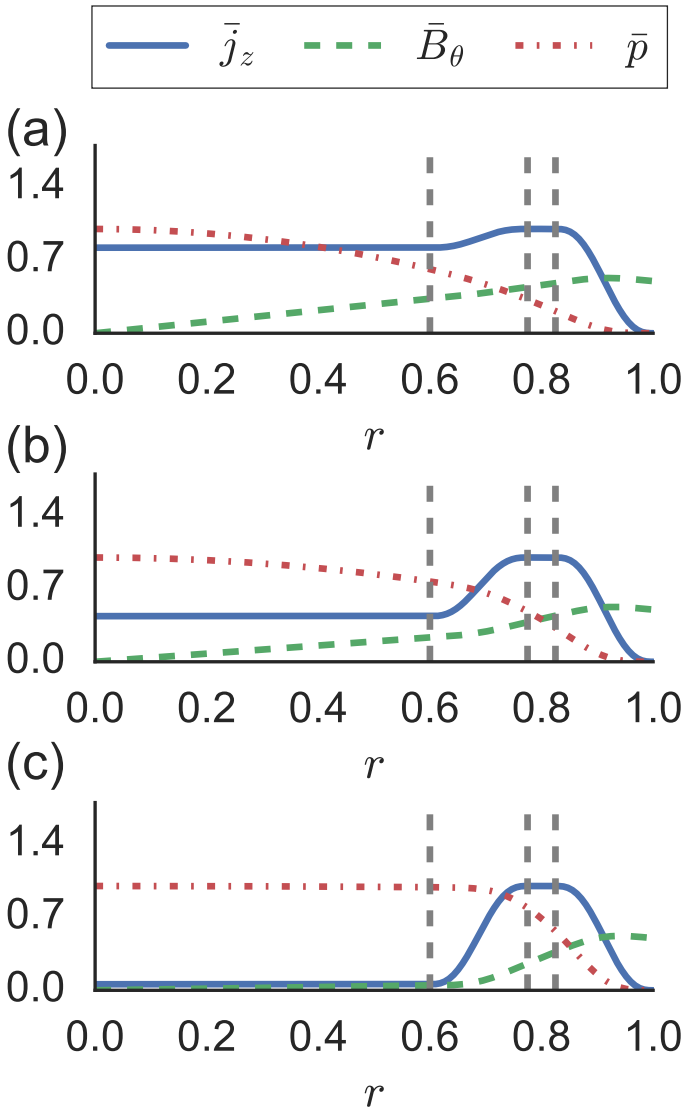


Figure 3.4: Radial profiles of $\bar{j}_z$, $\bar{B_\theta}$, and $\bar{p}$ for a) $\epsilon_{eff} = 0.7$, b) $\epsilon_{eff} = 0.5$, and c) $\epsilon_{eff} = 0.1$ used for the calculations shown in fig. 3.5. Dashed vertical lines demarcate (from left to right): core, transition, skin, and transition regions.

3.3 Numerics

Although we cannot solve analytically for δ, its value is uniquely determined for given periodicity numbers (m and $\bar{k}$), axial magnetic field ($\bar{\lambda}$), and current profile (ϵ) by the Euler-Lagrange equation (eq. 3.2). Numerical integration of eq. (3.2) will thus constrain our value of δ. Eq. (3.2) is a second order ordinary differential equation that can have several singularities. For numerical integration, the skin current region can no longer be assumed to be infinitesimal, so we construct profiles with a skin region of finite thickness. To continue, the equilibrium quantities will be expressed as dimensionless quantities (denoted by an over bar). The dimensionless axial current density $\bar{j}_z$ is described by a step function (fig. 3.4); in the core, the current density is constant, rises smoothly to the skin current and drops off smoothly to zero at the plasma edge. The smoothly varying transition regions are described by fourth order polynomials, continuous with the core and skin regions up to the second derivative. Since $\bar{j}_z$ smoothly varies to zero, the parameter $\epsilon = 0$. The current profile is described by the fraction of core-to-total current $\epsilon_{eff} = I_{core}/I_{total}$, where I_{core} is the current in the constant current core region and I_{total} is the total current driven along the flux tube. The azimuthal magnetic field $\bar{B}_\theta$ is determined by Ampere's law. The axial magnetic field $\bar{B}_z$ is taken to be constant across the flux tube. The pressure gradient $\bar{p}'$ is balanced by $\bar{j}_z\bar{B}_\theta$. Given the above assumptions, all profiles are uniquely determined by the dimensionless numbers ϵ_{eff}, $\bar{k}$, and $\bar{\lambda}$. A fourth dependent dimensionless number is the ratio of plasma pressure to magnetic pressure at the axis β_0.

Since the Euler-Lagrange equation may have several singularities the adaptive solver LSODA from ODEPACK is used. The solver is accessed through wrappers in the Python scipy.integrate library [59]. The Euler-Lagrange equation is re-expressed as a system of two first order ODEs:

$$\frac{d\bar{\xi}}{d\bar{r}} = \bar{\xi}' \quad \text{and} \quad \frac{d}{d\bar{r}}(\bar{f}\bar{\xi}') = \bar{g}\bar{\xi} \tag{3.13}$$

LSODA is an adaptive solver, where the stepsize is modified to achieve the desired accuracy. So a request for the value of the derivatives can be made at any given r. To accommodate these requests, the profiles are stored as cubic splines created with FITPACK routines accessed through Python wrappers in scipy.interpolate. Each interval, bounded by the singularities, can

be integrated separately using the small solution eq. (3.3) as initial value to the right of the singularity. Since this study is only concerned with external stability, it will suffice to test the equilibrium for Suydam stability at the singularities and integrate only the last interval.

To determine the initial value to the left of the singularity, we use the Frobenius power series method for regular singular points [60]. The Frobenius power series expansion gives the relationship between $\bar{\xi}$ and $\bar{\xi}'$ close to the singularity at $\bar{r} = 0$, for $m = 0$: ($\bar{\xi} \sim C\bar{r}^1$ and $\bar{\xi}' \sim C\bar{r}^0$), and for $m \neq 0$: ($\bar{\xi} \sim C\bar{r}^{|m-1|}$ and $\bar{\xi}' \sim C|m-1|\bar{r}^{|m-2|}$). Since the constant C can not be determined, the magnitude of $\bar{\xi}$ is arbitrary and only the ratio $\bar{\xi}'/\bar{\xi}$ has meaning. At the non-geometric ($\bar{r} \neq 0$) singularities, the Frobenius method gives a quadratic indicial equation

$$n_{\pm} = -\frac{1}{2} \pm \sqrt{\frac{1}{4} + \frac{\beta}{\alpha}} \tag{3.14}$$

with

$$\alpha = \frac{\bar{r}_s \bar{B}_\theta^2 \bar{B}_z^2}{\bar{B}^2} \left(\frac{\bar{q}'}{\bar{q}} \right)^2 \quad \text{and} \quad \beta = \frac{2\bar{B}_\theta^2 \beta_0}{\bar{B}^2} \frac{d\bar{p}}{d\bar{r}} \tag{3.15}$$

where $\bar{r}_s$ is the radial position of the singularity. The initial values for $\bar{\xi}$ to the right of the singularity are given by $|\bar{r} - \bar{r}_s|^{n_\pm}$. The solution with the larger n is the 'small' solution [47]. The small solution may diverge at $\bar{r}_s$ but will be well behaved away from the singularity. If $\frac{\beta}{\alpha} < -\frac{1}{4}$ the exponents n will be complex, resulting in rapid oscillations through $\bar{\xi} = 0$ around the singularity. The condition for complex exponents is equivalent to the Suydam instability [60]. These initial values are used to integrate $\bar{\xi}$ for the given magnetic field profiles in the last interval of the Euler-Lagrange equation. The values of δ and $\bar{\xi}_a$ are then substituted into eq. (3.12) with ϵ set to zero, to determine the δW of each mode.

The perturbed potential energy δW is calculated at 50 × 50 points in the $\bar{k}$ - $\bar{\lambda}$ space, and then interpolated throughout the space [42]. The results of numerical calculations of the $\bar{k}$ - $\bar{\lambda}$ stability space are shown in fig. 3.5 for three values of the current fraction ϵ_{eff}: 0.7, 0.5, and 0.1. In contrast to the analytical approach in fig. 3.3, the numerical integration shows that the rigidness parameter δ is not a constant but varies with $\bar{k}$ and $\bar{\lambda}$ although the quantitative behavior is similar. The magnitude of δW is arbitrary in the variational approach, however we can compare relative magnitudes (all δW values plotted are normalized). The kink unstable

region matches the Kruskal-Shafranov condition for the long-thin regime, low $\bar{k}$ (fig. 3.5 a, b, c). In regions of high $\bar{\lambda}$, β_0 becomes large and the β term in eq. (3.14) dominates. Since the pressure gradient for the chosen current profiles is always negative, any singularity due to the safety factor q crossing a rational mode will result in infinite oscillations of the displacement ξ, i.e. be Suydam unstable. Eq. (3.2) is the $\rho\omega^2 \to 0$ case of the general cylindrical eigenvalue problem, where ρ is the plasma density, ω is the mode frequency, and ω^2 the eigenvalue. The solutions to the eigenvalue problem are Sturmian, their zero-crossings decrease with decreasing values of the eigenvalue ω^2 [60]. This means that Suydam-unstable profiles are likely unstable to the external kinks, as decreasing zero-crossings allow the eigenfunction ξ to match the vacuum boundary conditions. To express this likely external kink instability, without solving the full eigenvalue problem, the Suydam-unstable regions are cross-hatched in fig. 3.5 a, b, c. The stability boundaries of both the sausage mode and kink modes depend on the core-to-total current fraction ϵ_{eff} (fig. 3.5 d, e, f). As in the analytical results, there is a significant region which is unstable to both the kink and sausage modes. For high ϵ and high $\bar{\lambda}$ the solution to the Euler-Lagrange equation indicates external stability, however, ξ does cross zero in these cases, suggesting internal instability. Again, taking the Sturmian property of the general eigenvalue problem into account, these zero crossings will likely allow the eigenfunction ξ to match vacuum boundary conditions at lower ω^2. To express this likely sausage instability the internally unstable region is cross-hatched in fig. 3.5 d. While the variational approach can not determine the linear growth rates, the ratio of the potential energies $\delta W_{m=0}/\delta W_{m=-1}$, determines which mode will have the faster growth rate. In the region where both sausage and kink instabilities occur, the sausage instability has a larger linear growth rate for higher $\bar{k}$ values (shorter, fatter tubes) and this boundary shifts downward as the ϵ_{eff} decreases (fig. 3.6 a, b, c). This suggests that a sausage could develop rapidly on top of a slowly growing kink.

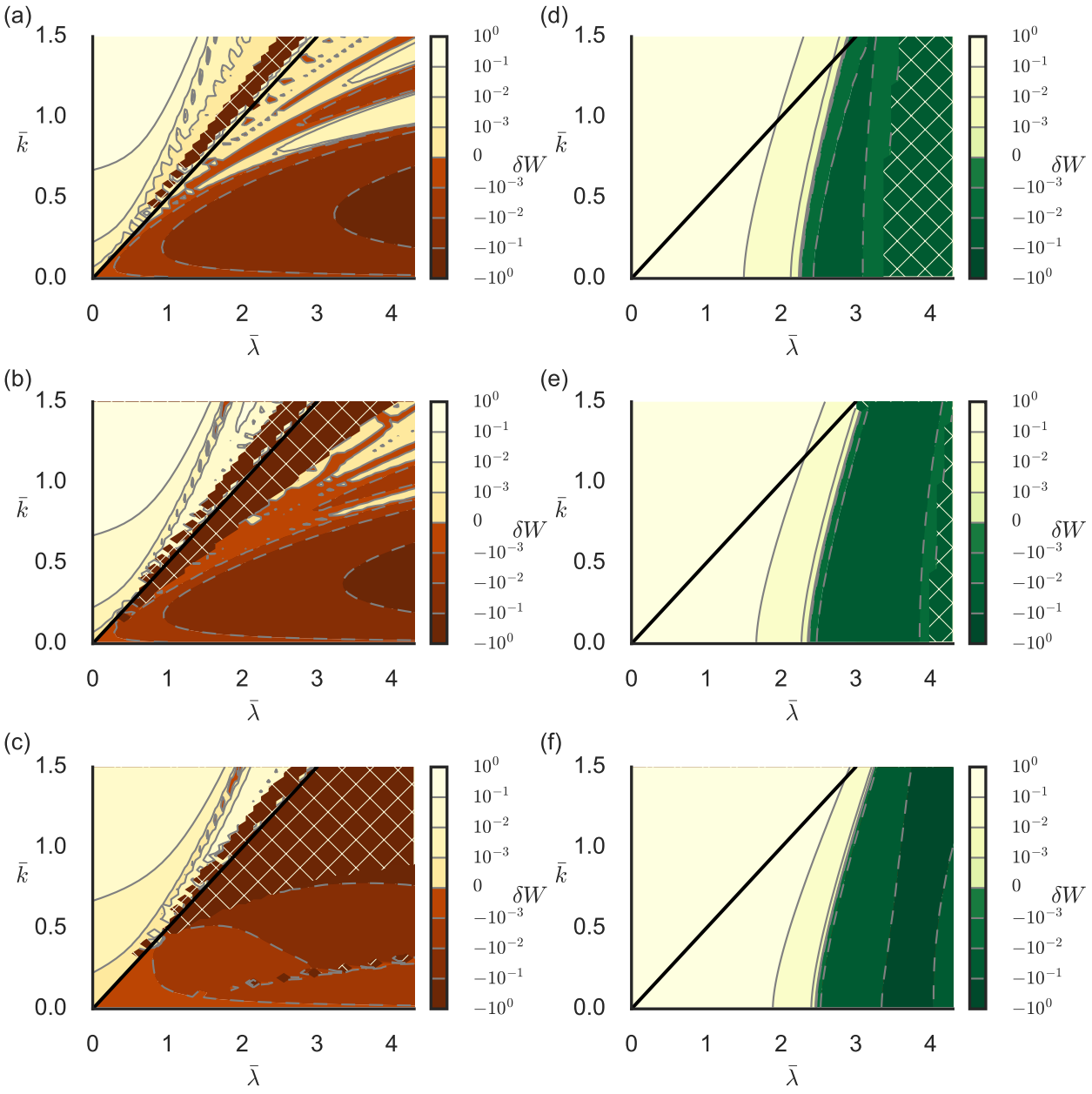


Figure 3.5: Numerical $\bar{k}-\bar{\lambda}$ stability spaces with relative growth rates. Normalized kink $\delta W_{m=1}$ contours for (a) $\epsilon_{eff} = 0.7$, (b) $\epsilon_{eff} = 0.5$, and (c) $\epsilon_{eff} = 0.1$; Normalized sausage $\delta W_{m=0}$ contours for (d) $\epsilon_{eff} = 0.7$, (e) $\epsilon_{eff} = 0.5$, and (f) $\epsilon_{eff} = 0.1$. The cross hatched regions indicate Suydam unstable regions in the $m = 1$ plots and regions with internal instabilities in the $m = 0$ plots. [Associated dataset available at http://dx.doi.org/10.5281/zenodo.230611.][43]

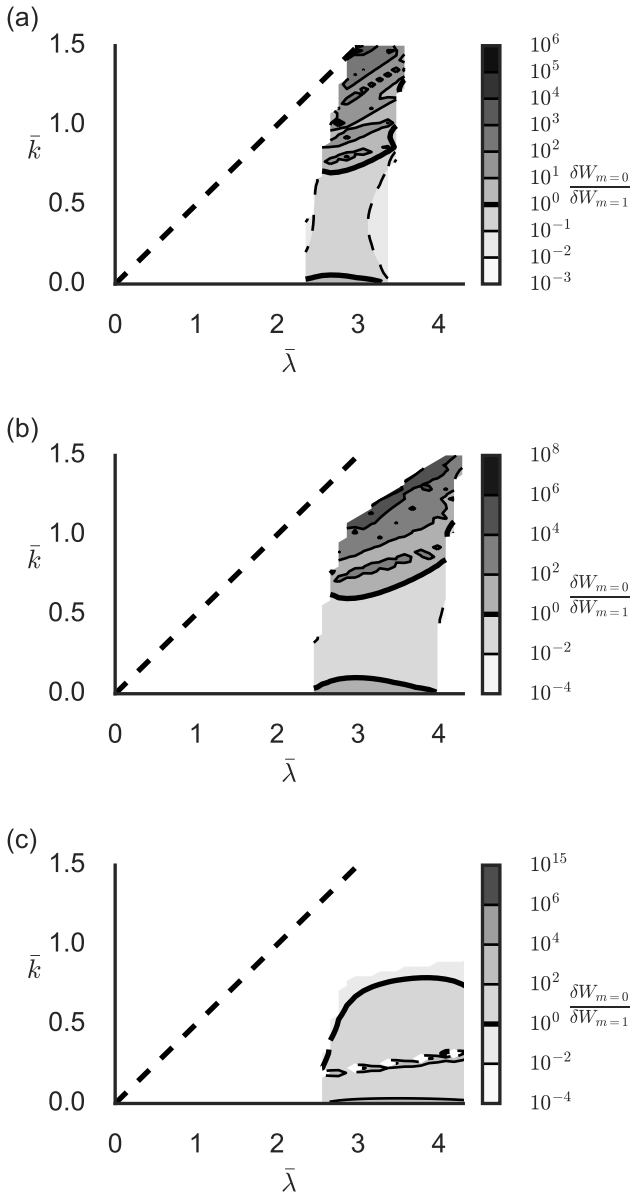


Figure 3.6: The ratio of $\delta W_{m=0}/\delta W_{m=1}$ for (a) $\epsilon_{eff} = 0.7$, (b) $\epsilon_{eff} = 0.5$, and (c) $\epsilon_{eff} = 0.1$. In the white space either at least one of the potential energies is positive or undetermined. [Associated dataset available at http://dx.doi.org/10.5281/zenodo.230611.][43]

3.4 Discussion

The analytical and numerical stability conditions present evidence for the possibility of a sausage instability developing on top and after a kink instability in a lengthening current-carrying magnetic flux tube, particularly in the general case of a flux tube with internal *and* skin currents. A sausage instability in a screw pinch plasma may be unfamiliar because classical analyses consider diffuse or skin currents separately, however, evidence for sausage instabilities have been observed in coronal loops [61], and predicted in magnetospheric current sheets [62] and cylindrical liner compression experiments with axial fields [63]. We note that in ideal MHD a linear configuration without magnetic field reversal is never sausage unstable without also being unstable to the kink. Sausage instabilities are commonly observed in Z-pinches [64], which are always unstable to the kink. Likewise, for current-carrying magnetic flux tubes the sausage unstable regions are inside the kink unstable regions of the $\bar{k}$ - $\bar{\lambda}$ space. In magnetic flux tubes with helical fields, kinks have been observed to redistribute the current path, and amplify the axial field [50]. The increased axial field could stabilize both the kink and sausage mode. A sausage instability is thus most likely to occur if the magnetic flux tube can quickly evolve to the upper right quadrant of the $\bar{k}$ - $\bar{\lambda}$ configuration space where the sausage mode growth rates dominate. The linear growth rates will be on the order of the Alfvén transit time, however, nonlinear effects may lead to saturation of the kink instability as is observed in other experiments [65]. A sausage instability has been observed forming on top of a kinking current-carrying liquid mercury column [66].

Another crucial feature of the $\bar{k}$ - $\bar{\lambda}$ stability space is the dependence of both the kink and sausage instability on the current distribution ϵ. This effect is not observed in linear plasma experiments driven by washer guns [55, 67] since the current ramp-up occurs on a much longer time scale than the lengthening of the flux tube. Washer-gun driven flux tubes are confined to low $\bar{k}$ when appreciable $\bar{\lambda}$ values are reached (bottom right of $\bar{k}$ - $\bar{\lambda}$ space). In the long-thin (low $\bar{k}$) regime the kink instability boundary always matches the classical Kruskal-Shafranov condition (assuming ideal periodic boundaries). Plasma jets produced by planar plasma gun

experiments, however, evolve in $\bar{k}$ and $\bar{\lambda}$ on the same time scale. In these plasma jets, rapid pinching of the plasma from the gun followed by detachment have been observed at high $\bar{\lambda}$, which may be an indication for a sausage instability [50]. Solar loop stability experiments have noted the effect of finite aspect ratios (large $\bar{k}$) [29].

In regard to experiments it is important to understand what implications a kink-sausage instability might have, and if there are any unique signatures. In Z-pinch discharges, sausage instabilities are associated with beams of of high energy ions [64, 68]. The ions could be accelerated by a strong electric field generated from the increased resistivity in the pinched region of the sausage mode. While the magnetic field in a Z-pinch is purely azimuthal, in a current-carrying magnetic flux tube the field is helical with shear across the magnetic flux surfaces. The compression of these sheared magnetic flux surfaces would provide a favorable magnetic topology for reconnection. An interesting question is whether a kink sausage instability could couple the MHD system to microscopic scales of ion inertial lengths or ion Larmor radii where two-fluid and kinetic effects become important and can lead to fast reconnection [23]. The assumptions of MHD are invalid when the ratio of drift over Alfvén velocity is close to or exceeds unity. At high drift velocities kinetic effects such as wave-particle interactions and the decoupling of perpendicular ion and electron motion described by the Hall term could dominate. To understand when this may occur it is helpful to express the ratio of the average drift velocity $v_d = I_z/(\pi a^2 n_0 q)$ and the Alfvén velocity $v_A = B/\sqrt{\mu_0 n_0 m_i}$ in terms of the configuration space parameters, where n_0 is the number density, q is the charge, m_i the ion mass, and μ_0 the magnetic permeability. Starting with the velocity definitions

$$\frac{v_d}{v_A} = \frac{I_z}{\pi a^2 n_0 q} \frac{\sqrt{\mu_0 n_0 m_i}}{\sqrt{B_{\theta v}^2 + B_{zv}^2}} \tag{3.16}$$

we can cancel a power of the azimuthal magnetic field after replacing the current with Ampère's law $I_z = 2\pi a B_\theta/\mu_0$

$$\frac{v_d}{v_A} = \frac{\sqrt{m_i}}{\sqrt{\mu_0 n_0 q^2 a}} \frac{2}{\sqrt{1 + \frac{B_{zv}^2}{B_{\theta v}^2}}}. \tag{3.17}$$

Substituting the magnetic field ratio $\bar{\lambda}_v$ eq. (3.11) and the ratio of the characteristic length

over the ion skin depth λ_i, also called the size parameter $S^* = a/\lambda_i = a\sqrt{\mu_o n_0 q_i^2/m_i}$, choosing the radius a as the characteristic length and assuming singly ionized plasmas, we obtain

$$\frac{v_d}{v_A} = \frac{1}{S^*}\frac{2}{\sqrt{1+\frac{4}{\bar{\lambda}_v^2}}}. \tag{3.18}$$

The critical value of $\bar{\lambda}_v$ at which the velocity ratio becomes unity is given by

$$\bar{\lambda}_{vcrit} = \frac{2S^*}{\sqrt{4-S^{*2}}}. \tag{3.19}$$

This condition has a singularity at $S^* = 2$ with a relative shallow slope before S^* 1.5, indicating that for low S^* a sausage instability could couple to microscopic scales (fig. 3.7). Low values of S^* can be achieved with plasmas with higher ion masses, lower densities and by reaching the top right corner of the $\bar{k}$ - $\bar{\lambda}$ configurations space where the flux tube radius is still large. Sausage instabilities can still occur when MHD dominates, at high S^*, without coupling to kinetic and two-fluid scales.

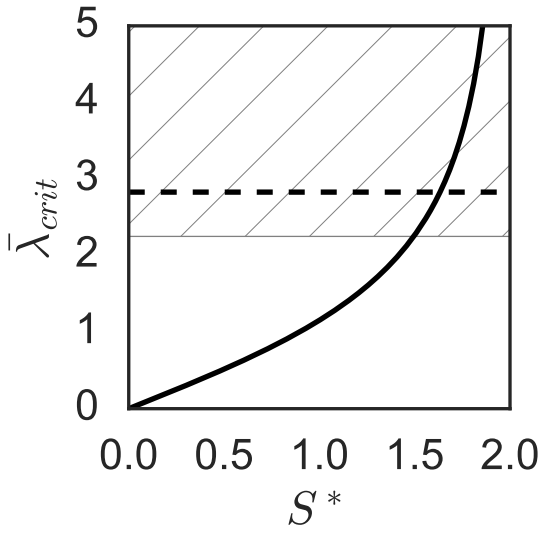


Figure 3.7: Critical $\bar{\lambda}$ value for a given size parameter S^* above which the instability will couple to microscopic scales. The hatched region starts at the minimum sausage unstable $\bar{\lambda}$ for an $\epsilon_{eff} = 0.7$ current profile. The dashed line denotes the Tayler criterion $\bar{\lambda} = 2\sqrt{2}$.

3.5 Conclusion

This work presents a general stability condition for current-carrying magnetic flux tubes with a wide range of aspect ratios, current-to-magnetic flux ratios, and current profiles. The analytical and numerical results point to sausage instabilities on top of kinking flux tubes as a possible cascade of instabilities. Further numerical and experimental studies are in progress to verify the $\bar{k}$ - $\bar{\lambda}$ configuration space and determine growth rates of the kink and sausage instabilities. A new triple electrode planar plasma gun, Mochi.Labjet, is designed to generate magnetic flux tubes with discrete core and skin currents evolving over a wide range of the $\bar{k}$ - $\bar{\lambda}$ stability space. Preliminary results indicate that the magnetic flux tube life time is several Alfvén transit times while the peak rate of increase of $\bar{\lambda}$ is at least an order of magnitude faster than the Alfvén transit time and four times faster than the rate of increase of $\bar{k}$. This indicates the

sausage unstable region of the $\bar{k}$ - $\bar{\lambda}$ configuration space should be accessible. The kink-sausage instability cascade could couple to two-fluid and kinetic regimes when the size parameter S^* is small. This can be achieved with high ion masses, e.g. argon or krypton plasmas. High spatial resolution magnetic probe arrays will identify the cascade of MHD instabilities.

If the kink-sausage instability cascade couples to two-fluid or kinetic scales, species canonical helicity conversions could occur. Experimental techniques are needed to explore the $\bar{k}$ - $\bar{\lambda}$ space, to reliably recreate the instability cascade, and to take large volumetric datasets needed to calculate helicity volume integrals.

Chapter 4

EXPERIMENT CONTROL AND HIGH-THROUGHPUT FPGA BASED DIGITIZERS FOR CANONICAL HELICITY MEASUREMENTS

4.1 Challenge of measuring canonical helicity

Several methods have been proposed to determine helicity of a vector field [69]. The most straightforward method is to measure the canonical quantities in a finite volume and integrate the volume integral eq. (2.18). An alternative method is to examine the topology of discrete flux tubes reconstructed from the measurements and determine their twist [70]. A third approach relies on determining the helicity flux on the surfaces of the volume of interest by measuring the injection terms in the helicity transport equation (2.24) [71]. The helicity flux approach does not require measurements inside the volume of interest but is only able to determine the total canonical helicity injected; species conversion occurring inside the volume cannot be captured. The finite volume and twist number methods require measurements of the canonical quantities throughout the finite volume. For a laboratory, linear canonical flux tube experiment evolving in a cube of 1 m length, $\sim$10 cm width, and height with a 1 cm ion inertial length $\sim 80,000$ measurement points would be needed to resolve the ion inertial length over the whole volume (using Nyquist length of 0.5 cm). At each location, multiple measurements over multiple experiment shots are needed to sample the statistical distribution of the measured quantities. Hence, in order to gather the measurements necessary to determine canonical helicity, it is advantageous to develop 1) experiment control to create reproducible shots and 2) high-throughput digitization in order to take as many measurements as possible per shot.

This chapter will introduce the Mochi.Labjet experiment and its experiment control and high-throughput FPGA (field-programmable gate array) based digitizers.

The experiment control LabVIEW application, the MDSplus data storage model, and the high-throughput FPGA code has been published on Zenodo under https://doi.org/10.5281/zenodo.495643, https://doi.org/10.5281/zenodo.495631, and https://doi.org/10.5281/zenodo.495787 [72, 73, 74]; appendix B provides additional information about setup and dependencies of this software.

4.2 Mochi.Labjet: Canonical flux tubes with boundary conditions relevant to astrophysical jets

Mochi.Labjet is a planar plasma gun experiment, designed to drive current profiles, and axial and azimuthal shear flows in a canonical flux tube. The Mochi.LabJet gun consists of three planar electrodes. The inner and middle electrodes can be biased with ignitron-triggered capacitor banks independently between 0 and $-10\ KV$ with respect to the grounded outer electrode. A dipole magnetic field (up to $1\ T$) connects the inner electrode with the outer electrode and the middle electrode with the outer electrode (fig. 4.1 a). Gas is injected through azimuthally symmetric slits in the electrodes (fig. 4.1 b). Breakdown occurs along the arched magnetic field lines connecting the electrodes. The arched plasma "Bundt cake" merges into a central collimated canonical flux tube. An axial $\vec{j} \times \vec{B}$ force drives an Alfvénic plasma flow. The flowing plasma convects frozen-in magnetic flux to the flared region, increasing magnetic flux in that region and collimating the flux tube [21, 22]. The differentially biased electrodes are intended to drive discrete core and skin currents in the canonical flux tube (fig. 4.1 c). Ref. [21] shows that the axial velocity of the plasmas depends on the flaring of the magnetic field and the current, thus the current profile imposes an axial shear flow. The radial electric field between electrodes combined with the bias field is equivalent to an azimuthal rotation in ideal MHD, $\vec{E} = -\vec{u} \times \vec{B}$, mimicking the azimuthal rotation in an astrophysical jet-accretion disk system (fig. 4.1 d). The plasma gun is mounted to a spherical vacuum chamber with $1.4\ m$ diameter. Roughing pumps and a cryopump provide a vacuum of down to $10^{-7} Torr$. The high vacuum enables a large density gradient between plasma and background which results in sharp boundaries in plasma emission in camera images. Ref. [75] describes the triple elec-

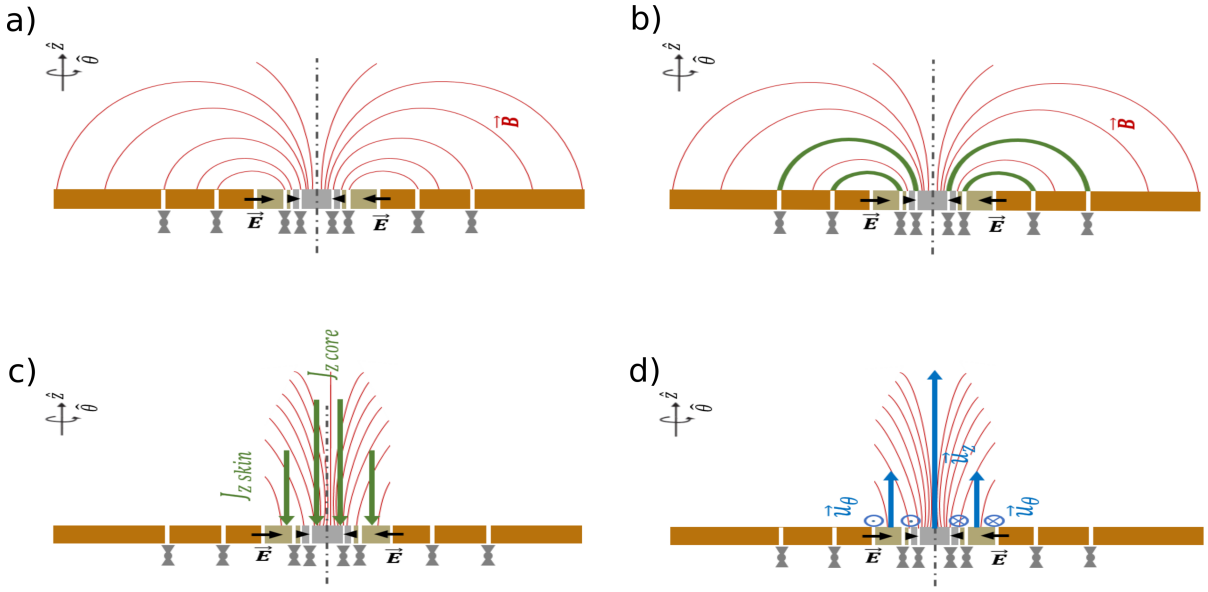


Figure 4.1: Three-electrode planar plasma gun configuration: showing magnetic field (red), electric field (black), gas valves (grey), outer electrode (orange), middle electrode (gold), inner electrode (gray), current (green), and flows (blue). The gas injection slits in the electrode are azimuthally symmetric. a) Vacuum $\vec{B}$ and $\vec{E}$ fields are imposed by a solenoid and the applied electrode potentials, respectively. b) Initial breakdown occurs along arched magnetic field. c) MHD pumping has collimated the magnetic field. Core and skin currents are driven by the applied electrode potentials. d) MHD pumping drives axial flows. The azimuthally symmetric gas injection slits allow the plasma to freely rotate in response to the radial electric field.

trode plasma gun design in detail. There are three gun power supplies (PSU). Mochi.PSU1 is described in appendix C. Mochi.PSU2 and Mochi.PSU3 are described in ref. [75].

4.3 Mochi Control: Experiment control & data storage

Pulsed power plasma experiments, such as Mochi.Labjet, share several features in common. These experiments achieve plasma breakdown by injecting neutral gas and applying large voltages from pulsed power supplies with short *ns* - *ms* discharge durations. Magnetized plasma experiments require pulsing current through external field coils several hundreds of milliseconds before breakdown. This time allows the magnetic field to soak in through the conducting electrodes and vacuum chamber. To achieve breakdown, the gas injection valves, magnetic field coil, and electrode power supplies need to be triggered with accurately timed electrical pulses. In contrast, other components such as high voltage relays for charging and dump circuits need to be energized for durations of up to several minutes, for which switching within human timescales is acceptable. Charging power supplies generally can be charged to a specified voltage by setting the charging time according to their *RC* time constant. In summary, to control a pulsed power experiment, two types of signals are required: 1) Fast input pulses with 100 μs or shorter durations and starting times specified within 100 *ns*. 2) Non-varying DC input signals with durations of up to several minutes and with start times specified within seconds or manually operated on a graphical user interface.

Mochi Lab uses Mochi Control, a flexible National Instruments (NI) LabVIEW experiment control application[1], based on the producer-consumer design pattern, to allow operators real-time control over all experimental settings, including 16 timing channels with a dynamic timing map, 120 digitizer channels, and a MDSplus settings save and load interface. Fig. 4.2 gives an overview of the experiment control system. Operators control the experiment from a PC running Mochi Control connected through fiber optic cables to two NI chassis. The application programs NI cards housed in the chassis. The synchronizing, timing, and multi-function cards generate the control signals. Transmitter circuits convert the signals to optical pulses, carried by fiber optic cables. Custom built receiver circuits convert the signals back to electrical inputs for the respective experiment components. Ref. [75] describes the circuits in detail. The

[1]Developed in collaboration with Woodruff Scientific Inc.

operators control the non-varying DC control signals by switching 88 digital outputs on the PXI-6602 and PXI-6133 cards. For the fast input pulses the operators pre-select the timings. The experiment control application transmits the desired experiment timings to the PXI-6602 cards which use 16 on-board clocks / pulse generators to generate optically isolated trigger pulses for the gas puff valves, oscilloscopes, and bias coil and plasma gun capacitor banks. Voltages from diagnostics are acquired by 24x 14 bit, 2.5 MS/s analog inputs on the PXI-6133 and 96x 12 bit, 50 MS/s, AC coupled analog inputs on the NI-5752 digitizers. Measurements from diagnostics electrically connected to the experiment such as high voltage probes are isolated with custom fiber optic transmitter-receiver circuits [76], whereas measurements from diagnostics without electrical contact to the experiment, such as $\dot{B}$ probes and Rogowski coils, are directly digitized. The fiber optic cables between operator PC and chassis, and between chassis and experiment provide electrical isolation for operator safety, for protection of the NI equipment, and to avoid ground loops.

A hierarchical MDSplus tree [77] stores all experiment settings such as capacitor bank voltages and discharge times together with all data acquired through National Instruments PXI-6133 and NI-5752 digitizer cards, Tektronix TDS-2024B oscilloscopes, and a Princeton Instruments PI-MAX 3 fast framing camera. Operators can load previous shot settings from the MDSplus tree and manipulate the settings. The experiment control application accesses the storage through the MDSplus LabVIEW interface [78], which leverages object oriented and multi-threaded features of MDSplus, allowing the code to simultaneously translate LabVIEW data types and MDSplus data types, write and read to and from multiple MDSplus tree nodes. For each experimental shot MDSplus creates a new tree structure based on a model tree [73]. This tree can store numbers, strings, data arrays, and 2D data/time conglomerations called signals. Experimental data from diagnostics are stored together with experimental settings to facilitate interpretation and shot reproducibility. Using jScope, the stored data arrays from any previous shot can be accessed in graphical format.

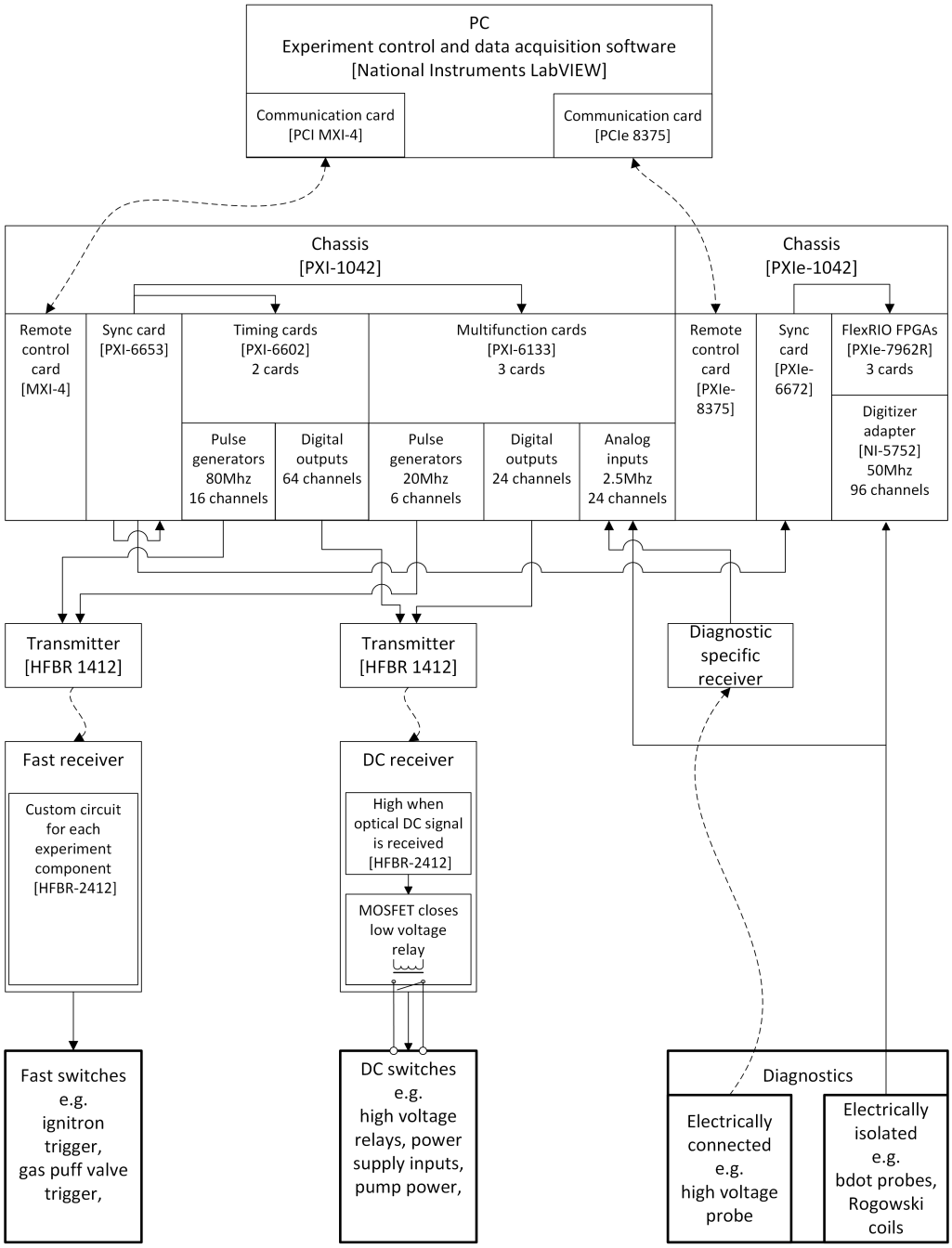


Figure 4.2: Experiment control: Operators control the experiment with a PC application using LabVIEW DAQmx and LabVIEW FPGA module to program the PXI Cards. The pulse generators / counters and digital outputs generate fast pulses and DC signals. These signals are converted to optical outputs, travel through optical isolation fibers, and are converted back to electrical signals at the experiment. 120 analog inputs digitize diagnostic signals. Control of the 6 pulse generators in the PXI-6133 is not yet implemented.

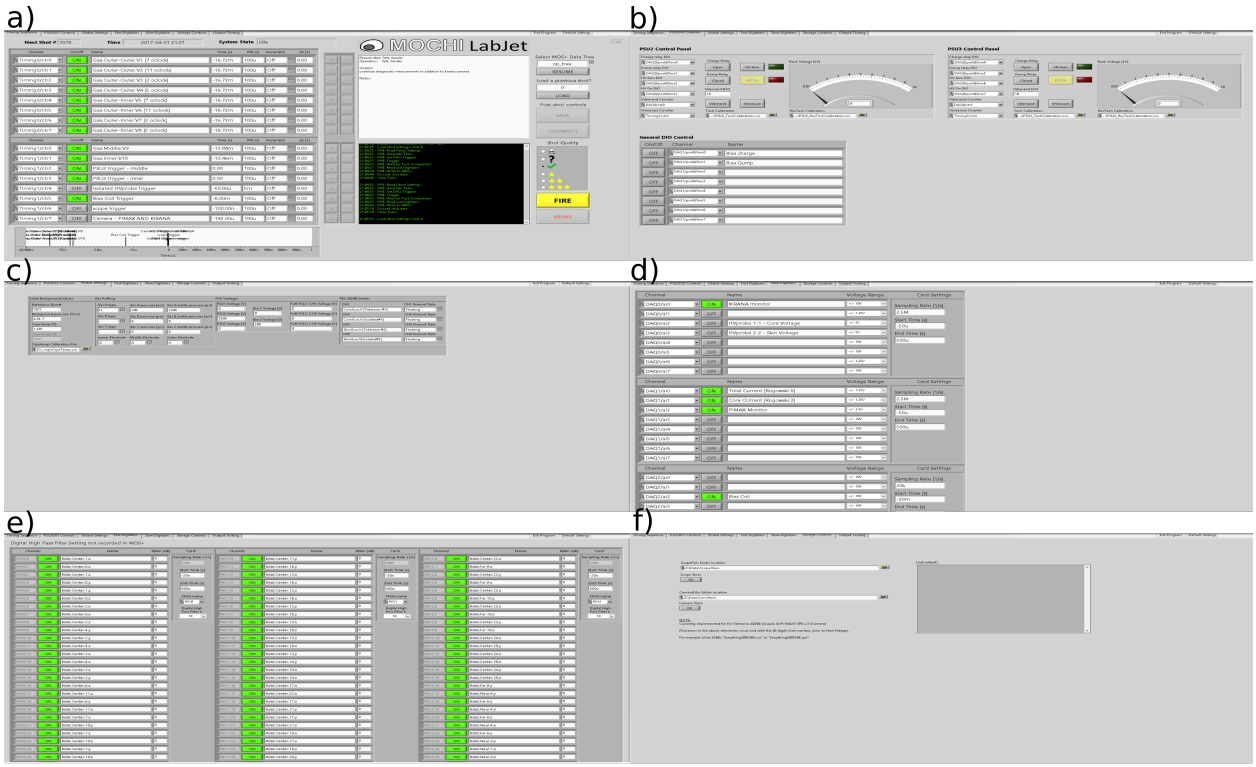


Figure 4.3: GUI of experiment control software: a) Timing sequence panel, b) PSU/DIO control panel, c) Global settings panel, d) slow digitizer panel, e) fast digitizer panel, and f) storage control panel.

4.3.1 Mochi Control user interface

Using the front-panel interface shown in fig. 4.3, operators can adjust experimental settings such as component trigger timings, data acquisition start times, durations, and sampling rates, charge the capacitor banks, and store data acquired on oscilloscopes and fast framing cameras. The *timing sequence* panel (fig. 4.3 a) serves as the main control panel. Operators can use the *resume* and *load* buttons to load all settings from a previous shot stored in MDSplus. In addition, operators can set the trigger time for each pulse generator / counter and specify a name that will be stored in MDSplus. The *FIRE* button initiates a shot sequence during which status updates are displayed in the black terminal. After a shot has been completed, operators

can rate the quality of the shot and record additional notes in the white *shot notes* window. The shot can either be saved to MDSplus with the *save* button or discarded with the *overwrite* button. With the *PSU/DIO control* panel (fig. 4.3 b) operators can control the charging of capacitor banks, specifying and switching the digital outputs. In the *global settings* panel (fig. 4.3 c) operators can store experiment settings not directly controlled by the application, such as the type of gas injected, the background reference shot, the chamber pressure, the charging voltage of the power supplies and to which diagnostics the oscilloscope channels are connected. Under the *slow digitizers* panel (fig. 4.3 d) operators can select which of the 2.5 MS/s analog inputs should record data, name the inputs, set input range, and acquisition times. The FPGA digitizers read the settings from the *fast digitizers* panel (fig. 4.3 e) the channels can be named, the acquisition time, and an internal digital high pass filter can be set. The digital high pass filter removes a resonance that occurs around 50 kHz in the ADC chips of the NI-5752. Ref. [79] shows the frequency response curves for each filter setting. A good default value is K10. The oscilloscope and the PI-MAX3 fast framing camera are not controlled by the experiment control application but through software provided by their respective vendors. At the end of the day the operators can use the *storage controls* panel (fig. 4.3 f) to load the data files of the Oscilloscope and fast framing camera to the MDSplus tree.

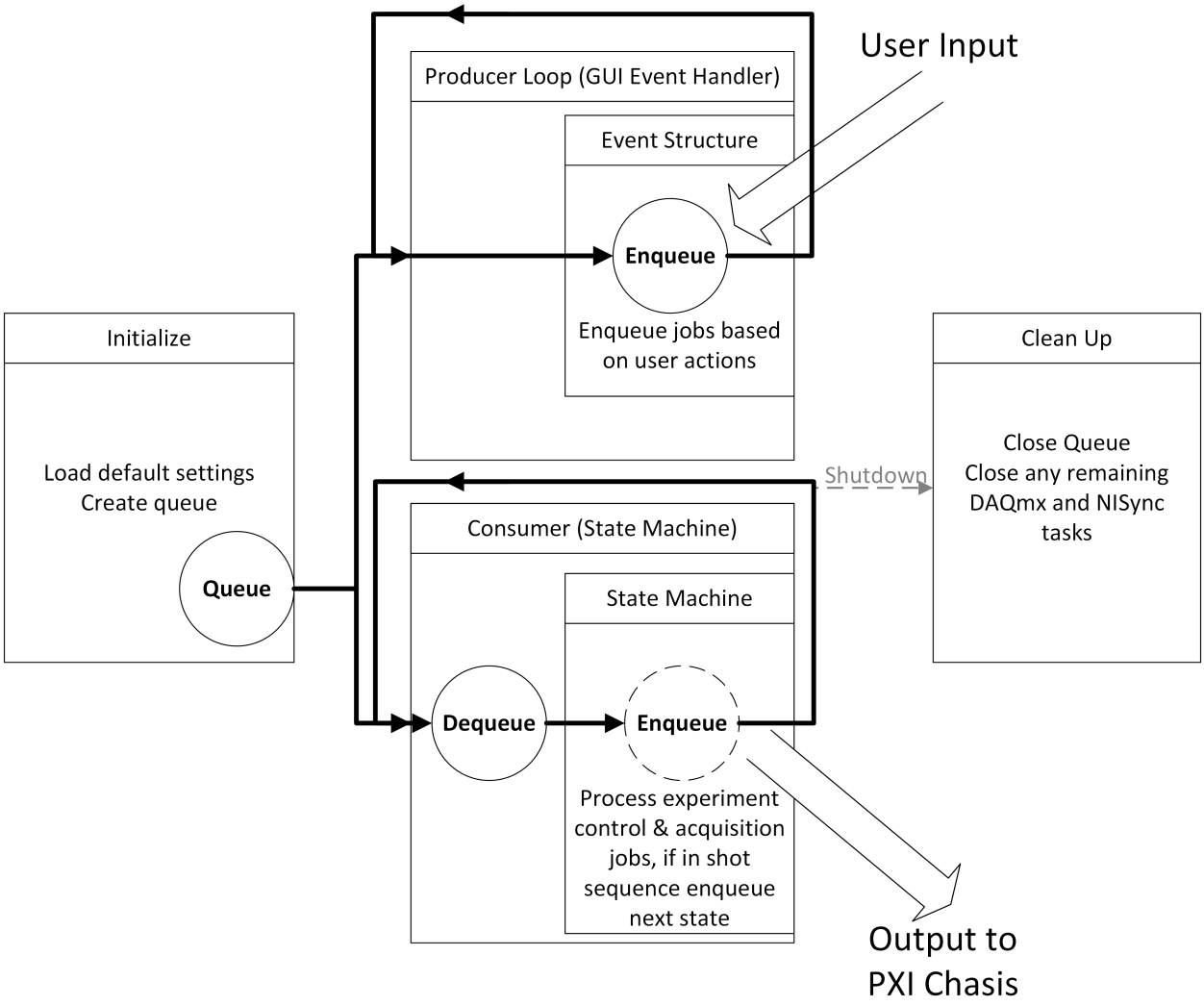


Figure 4.4: Experiment control software: The initialize block loads default settings and creates a queue. The queue is passed to two loops running in parallel. The producer loop handles GUI events by creating jobs and enqueuing them. The consumer dequeues jobs. Jobs are processed in the corresponding state of the state machine with calls to DAQmx and NISync Vis. If the state is part of a multi-state sequence, the state may enqueue further jobs. The shutdown event closes the queue and both loops and calls the shutdown block.

4.3.2 Mochi Control architecture

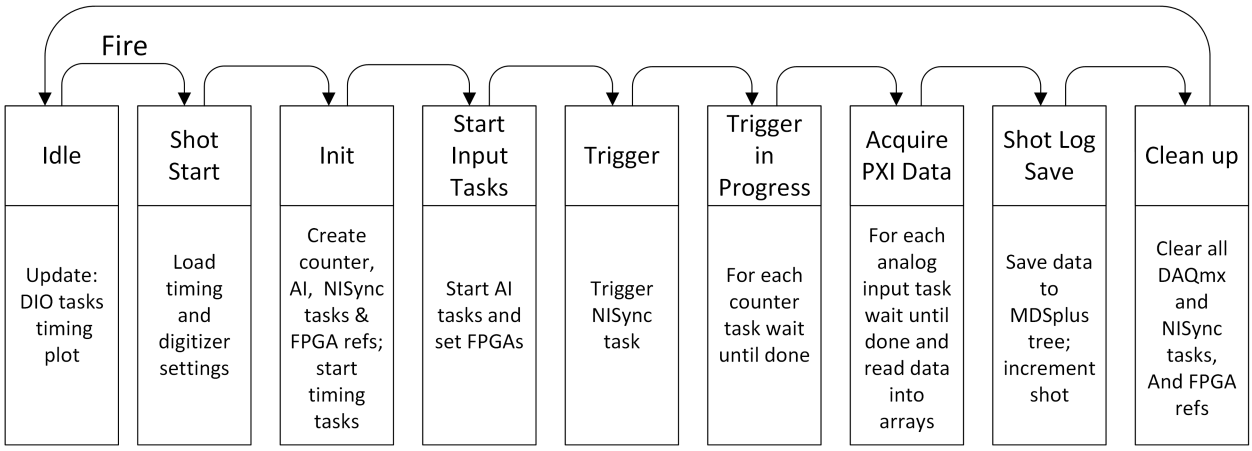


Figure 4.5: Shot sequence in the state machine: Operators pressing the *FIRE* button trigger the shot sequence. Each state is processed in order. DIO are digital output tasks controlling the DC receivers. After *Clean Up* the state machine returns to the *Idle* state.

The Mochi Control LabVIEW virtual instrument (VI) consists of 4 blocks. At startup, the initialize block loads the default settings into the front panel, and clears any existing NI tasks on the computer. A message queue is created that facilitates communication between the parallel threads of the application. The producer and consumer loops separate user interface interactions and experiment control in two parallel threads preventing user input from being lost and the application from becoming unresponsive. The producer loop is designed to listen with an event structure for user generated events, interpret the events as jobs, and place the jobs into the queue. The consumer loop dequeues these jobs and processes them in the order they were received (fig. 4.4). The consumer loop uses the DAQmx library and a custom built FPGA host library to communicate with the digital outputs, analog inputs, and counters / pulse generators. The DAQmx library abstracts the interactions with the NI hardware to tasks. DC signals and can be switched on and off on demand with “DIO write” tasks while fast pulses and data acquisitions are only initialized during a shot, once the *FIRE* button is pressed (Fig. 4.5). At that instant, all fast timing and digitizer settings are loaded from the front panel GUI. These

settings are used to create the appropriate DAQmx pulse generation tasks for the fast timing, and DAQmx analog input tasks and FPGA references for the data acquisition. Next all pulse generation, FPGA references and analog input tasks are set to be triggered, by a NI Sync task. NI-Sync is a library of VIs for setting up the synchronization cards. For each synchronization card one NI-Sync task is created. To synchronize the two chassis, one synchronization card acts as a leader and the other as a follower. The leader and follower synchronization modules need to start sending trigger signals to the cards in each chassis at the same time. The leader synchronization module sends out a trigger pulse to the follower synchronization module through a 1 m coaxial cable, to account for any cable propagation delay the leader module also sends out a trigger pulse along a 1 m coaxial cable that reconnects to the leader module. Once the respective pulse arrives, the synchronization cards send out the trigger pulses to other cards. The fire sequence waits for all tasks to complete and reads in the data from the analog input tasks, then proceeds to store the data arrays and experimental settings as discussed in the following section. After storing, the code runs its cleaning task, preparing for another shot. When the consumer loop receives input from the storage control panel to store the oscilloscope and camera data into the MDSplus tree, the producer loop calls Python scripts to read in the oscilloscope files (csv) and camera image files (spe). The csv files are read with io functions in the numerical Python (numpy) package [80]. The spe files are read with the tsphot Python package [81]. When a shutdown event is received by the producer loop, a shutdown job is enqueued and the producer terminates. Upon dequeuing a shutdown event, the consumer loop terminates itself and the clean up block is initiated which closes the queue and all remaining DAQmx, NIsync, and FPGA references.

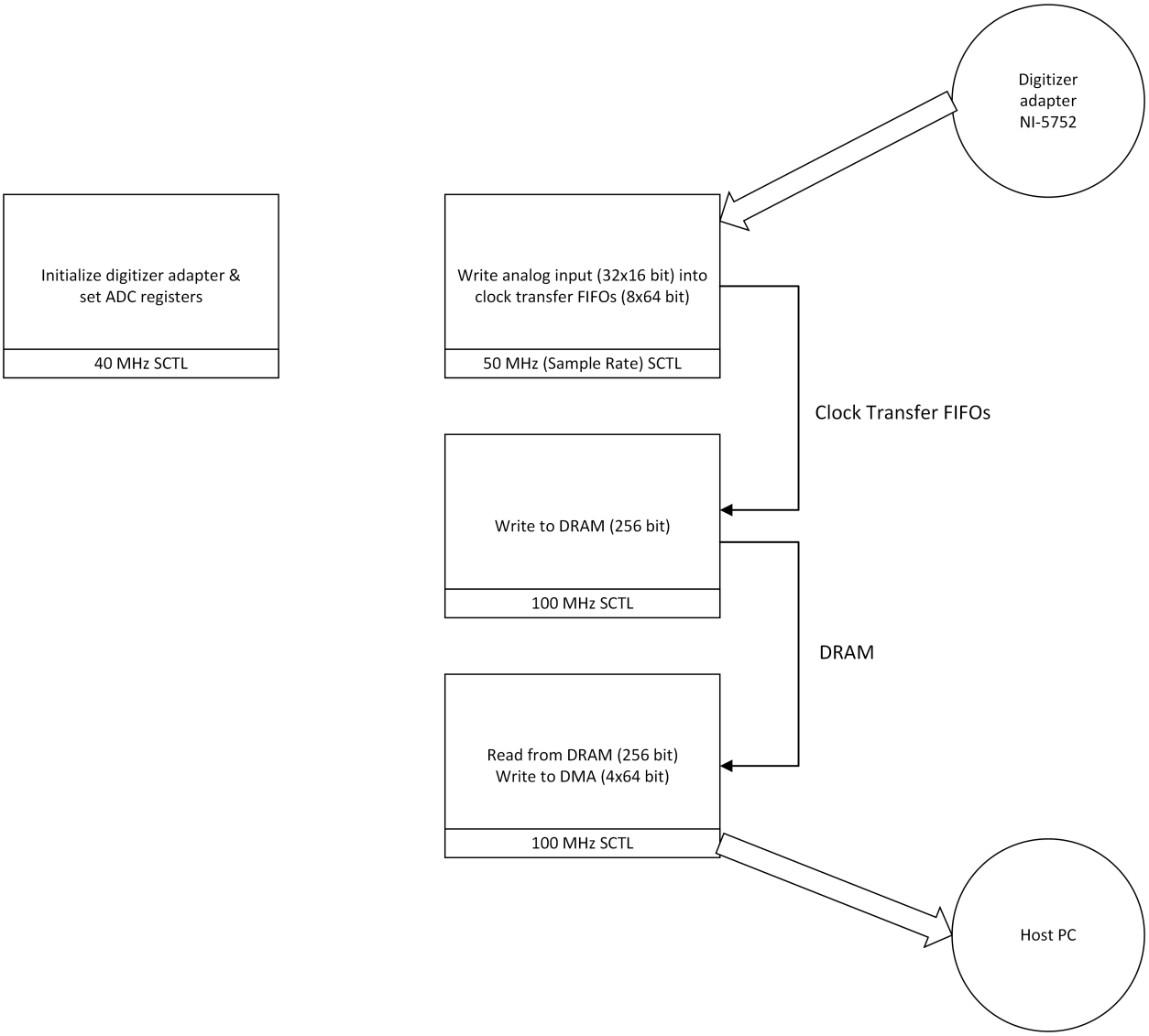


Figure 4.6: FPGA block diagram: The FPGA code consists of four single-cycle-timed loops (SCTL): 1) Initialize and register manipulation loop (40 MHz), 2) Acquire loop (50 MHz), 3) Write to DRAM (100 MHz), and 4) Readout DRAM (100 MHz).

4.4 High-throughput FPGA based digitizers

Data acquisition consists of high frequency analog to digital conversions of an input voltage and the rapid storage of these data samples. FPGAs are configurable integrated circuits consisting of logic and storage elements. The wires or interconnects between the elements can be programmed, rearranged to achieve different functionality. As circuits FPGAs are inherently suited for parallel transfer of digitization samples from many channels to storage. The PXI-6133 multi-function card is FPGA based, however the FPGA is not configurable by end users. The National Instruments FlexRIO product line separates the FPGA and the analog-to-digital conversion (ADC) into two separate modules which are purchased separately. The digitizer adapter modules can be swapped out to accommodate different digitization needs. Mochi Lab purchased three PXIe-7962 FPGAs and three NI-5752 digitizer adapters. Each NI-5752 adapter includes four Texas Instruments AFE5801 ADC chips which simultaneously sample 32 channels at 50 MS/s with 12 bit. The channels are AC coupled with a 3 dB point at 16 kHz. By default, an anti-alias filter internal to the ADC chips operates with a 3 dB frequency of 14 MHz. Custom breakout boards have been designed to breakout VHDCI inputs of the NI-5752 to double shielded twisted pair CAT-7 cables. The CAT-7 cables combine the inductive pickup rejection of twisted pair cables with capacitive pickup shielding. The breakout boards have load balanced switchable $0-76$ dB attenuators. The load balancing enhances the capacitive and inductive pickup common mode rejection of differential digitization [82].

A clip provided by National Instruments for the PXIe-7962 FPGAs reads out the samples as 16 bit words, resulting in a data throughput of 3.2 GB/s. A custom FPGA code developed and optimized with the LabVIEW FPGA module achieves a throughput to the on-board DRAM of $\sim$3 GB/s, limiting the acquisition time to 800 μs when the overflow data fills the on-board block memory. After each experiment shot the experiment control application reads out and stores the data into the MDSplus tree.

The FPGA code is optimized for high-throughput [83]. As shown in fig. 4.6 the code consists of four single-cycle-timed loops running in parallel. Single-cycle-timed loops execute at every

tick of a specified clock. The loop running at 40 MHz initializes the NI-5752 adapter clip and then processes any input from the host PC to modify the ADC registers. The digital high pass filters mentioned in sec. 4.3.1 are set with these ADC registers. The acquire loop runs at the sampling rate of 50 MHz and transfers thirty-two 16 bit samples from the analog inputs to first-in first-out (FIFO) arrays implemented in block RAM. FIFOs are necessary to transfer data between loops with different clocks, from the data acquisition clock (50 MHz) to the DRAM read-write clock (100 MHz). In the DRAM write loop the samples are transferred from the FIFOs to the DRAM. Write operations to the DRAM can fail and take more than one cycle to complete, so it is necessary to use handshaking patterns. Each sample is stored in a feedback node, that will store the sample between loop cycles, and the next sample is only read from the FIFO if the write operation to the DRAM has succeeded. The maximum throughput to the DRAM is $\sim$3 GB/s, which is below the 3.2 GB/s data throughput from the analog input. With the FIFOs acting as a buffer, more than 800 μs of data can be stored before the FIFOs overflow. The last loop is the read DRAM loop where the data is read from the DRAM and written to a direct memory access (DMA) to the control PC. This readout process is only initiated after the acquisition to maximize the write throughput to the DRAM. A read from DRAM always takes at least two clock cycles to complete. In the first clock cycle an address is requested and in the second cycle the corresponding data is received. At least one cycle will pass before the data is available. As the DMA throughput to the control PC is slower than the DRAM throughput, handshaking is necessary to ensure that no data is lost. The size of the packets written to the FIFOs and DRAM, 8x 64 bit and 256 bit, respectively, have been carefully chosen to maximize throughput. The samples have to be leaved together to form these packet sizes which necessitates a deleaving step on the control PC before the data can be stored in the MDSplus tree.

With further work, it is possible to increase the sampling time. The samples could be bit-packed, removing the 4 bits that contain no data from each sample and placing parts of two samples into 16 bit words which would reduce the data inflow below the DRAM throughput limit. However, the number of channels that can be bitpacked may be limited by the logic

resources available on the PXIe-7962.

4.5 Conclusion

This chapter develops the experimental techniques to gather the large volumetric datasets needed to calculate canonical helicity integrals. The experiment control application Mochi Control provides operators with real-time control over all experimental settings, and a MDSplus settings save and load interface. Mochi Control facilities creating reproducible shot conditions to gather measurements of coherent plasma phenomena over many shots. High-throughput FPGA digitizers have been developed to provide measurements on 96x 12 bit channels at 50 MHz. Data analysis techniques are needed to assemble measurements from many shots, identify coherent plasma behavior, and calculate the necessary reference fields to determine gauge-independent relative canonical helicity.

Chapter 5

MEASUREMENTS OF CANONICAL FLUX TUBES AND THEIR HELICITY

This chapter reconstructs canonical flux tubes and their helicity from measurements of magnetic field and ion flow of gyrating plasma columns. The Mochi.LabJet experiment is currently in a campaign to acquire these vector fields. To prepare for the Mochi.Labjet data, the analysis methods to reconstruct canonical flux tubes and their relative canonical helicity will be developed with a volumetric dataset of $\dot{B}$, triple probe, and Mach probe measurements from the Reconnection Scaling Experiment (RSX). The data analysis can be divided into four steps:

1. Prepare data: Since the measurements are taken over $2,369$ shots a coherent shot subset with common canonical flux tube motion must be identified. The measurements are conditionally sampled using the common relative time base of the canonical flux tube gyration and then are interpolated to a rectilinear grid.

2. Reconstruct and interpret canonical flux tubes: Line integrals of the interpolated vector measurements trace out canonical flux tubes in RSX.

3. Determine vector potential and reference fields: Fast Fourier methods solve the Laplace equation for reference circulation fields and an appropriate gauge choice simplifies the calculation of vector potentials needed to determine relative canonical helicity.

4. Reconstruct and interpret canonical helicity: The time evolution of canonical helicity is determined inside a 3D subvolume of the RSX experiment.

The canonical flux tube and helicity reconstruction code [84] has been published on Zenodo under https://doi.org/10.5281/zenodo.58119 and is described in appendix D.

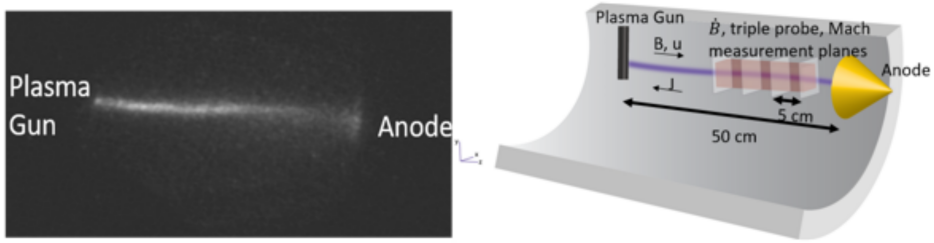


Figure 5.1: Left) CCD image of emission from kinked canonical flux tube. Right) Artist's interpretation of gyrating canonical flux tube. Axial flow and vacuum magnetic field are directed towards the anode. The current flows in the opposite direction. Measurement planes and interpolation volume are shown in purple.

5.1 Overview of RSX experiment

5.1.1 A gyrating kinked canonical flux tube

RSX could generate up to four current-carrying canonical flux tubes of radius $a = 2-3\ cm$ in a cylindrical vacuum chamber of 0.2 m radius between negatively biased plasma washer guns and a translatable conical anode [85] (fig. 5.1). This study analyses a dataset of shots with a single canonical flux tube and a 22° half-angle conical anode fixed at a distance of $L = 0.52\ m$ from the washer gun [65]. Current flowing through external coils provided, for the duration of the discharge, a constant axial guide field of $B_z = 0.02\ T$. The lifetime of a single plasma column in the RSX experiment comprises three distinct phases (fig. 5.2). 1) The flux rope current ramps up. 2) Once the current exceeds the Kruskal-Shafranov limit, the flux rope kinks, displaces helically, and gyrates. 3) At 2.1 ms the current supply circuit is crowbarred and the current decays. Ohmic heating through the current achieves electron temperatures of $10-15$ eV and ion temperatures of $1eV$. The washer plasma gun carefully controls the gas injection achieving an ion to neutral particle ratio of 10 with an ion particle density of $1-3\cdot 10^{19}\ m^{-3}$. All shots used hydrogen as gas. The plasma gun injects axial flow ($\leq 2.5\cdot 10^4\ m/s$), directed

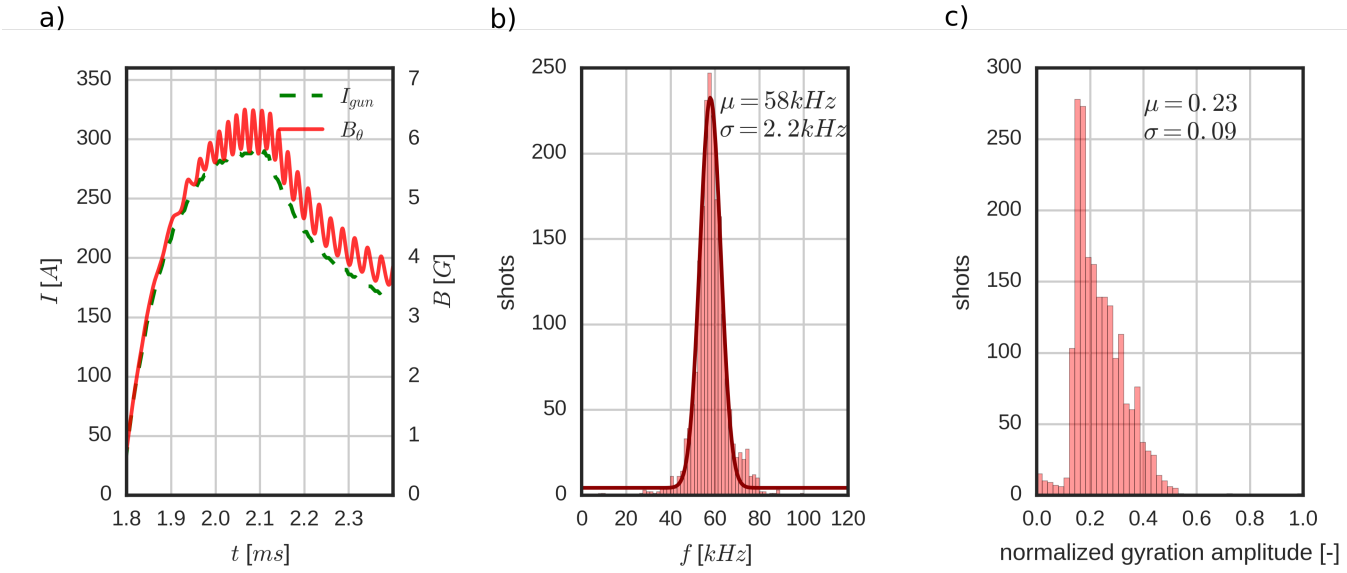


Figure 5.2: Coherent gyration. a) Gun current (dashed green) and azimuthal magnetic field (red) measured 5 *cm* from gyration center by the stationary fiducial probe during a typical shot (#15253). After the current exceeds the Kruskal-Shafranov limit, the magnetic field develops a modulation corresponding to the magnetic flux tube gyration. b) Frequency distribution across all shots of the modulation in the fiducial probe signal. Frequencies are determined by the peak of a Fourier transform. A Gaussian with a constant offset is fit to the distribution. c) Distribution of modulation amplitude in the fiducial probe signal just before the crowbar across all shots.

towards the anode, into the flux rope. This axial flow makes the helical displacement of the kink appear to be gyrating for stationary observers, such as internal probes.

5.2 Diagnostics

Internal Mach, magnetic, and triple probes measured ion velocity, change in magnetic field, density, temperature, and electric potential, respectively. During each shot, all probes measured the quantity of interest locally at one discrete point, except for the $\dot{B}$ probes which have $6-20$

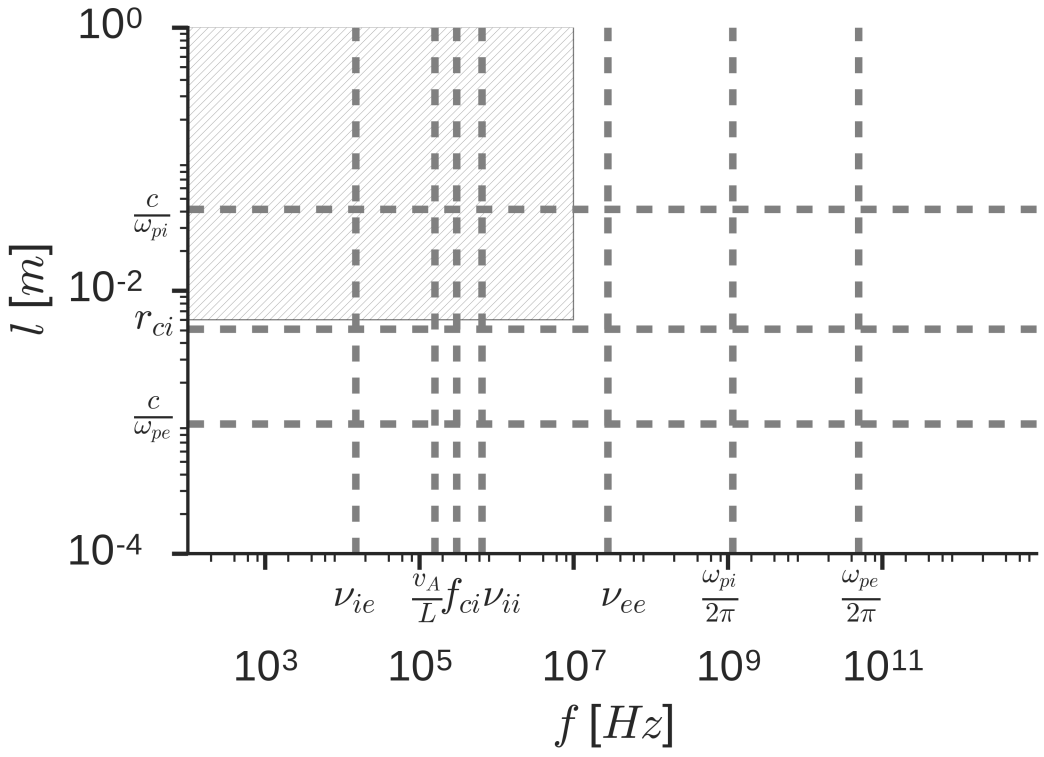


Figure 5.3: Plasma length and time scales, and maximum achieved spatial and temporal resolution of internal probes (cross-hatched). The resolution is set by the Nyquist frequency of the digitizers 10 MHz and the Nyquist length of the smallest achieved spatial resolution (0.006 m). Plasma scales are calculated with magnetic field $B = 0.02\ T$, number density $n = 3 \cdot 10^{19}$ m^{-3}, electron temperature $T_e = 10\ eV$, ion temperature $T_i = 1\ eV$, and flux tube length $L = 0.5$ m [65].

sensors. The probes are positioned between shots through flat plane ports [86] to cover x-y cut planes at four axial locations: 24.9 cm, 30.2 cm, 35.7 cm, and 41.6 cm. In the remainder of this chapter the planes will be referred to by their first two significant digits. The spatial accuracy of the probe positioner in the x-y cut planes is 0.5 mm, in this data set a resolution of

3 mm was achieved. The digitization rate is 20 MHz. Fig. 5.3 shows the spatial and temporal resolution of internal probes together with plasma scales. The ion inertial length is resolved but not the ion plasma frequency. The ion gyrofrequency is resolved; the ion gyroradius is within the same order of magnitude as the spatial resolution.

This study uses the results of the data analysis conducted in ref. [65] as a starting point. The magnetic field, density, electron temperature, and ion Mach number are treated as the 'raw' data. Although the raw probe data was not analyzed for this study a brief description of the probes will be given to help understand sources of uncertainty originating from the instruments and the raw data analysis.

5.2.1 $\dot{B}$ probes

The $3D$ magnetic field vectors are measured with $\dot{B}$ probes. The $\dot{B}$ probes contain multiple $(6-20)$ inductor chips [87] aligned in three perpendicular directions and covered by an electrostatic shield inside a pyrex tube. Following Faraday's law, the change in magnetic field perpendicular to the area A traced out by the inductor chip loops is proportional to the voltage V induced in each chip $dB/dt = -V/(NA)$, where N is the number of loops. The effective NA, taking into account misalignment, is determined through calibration with a known magnetic field applied in the x, y, and z directions of the lab frame [87, 88]. The response of the $\dot{B}$ probes rolls off for MHz frequencies. dB/dt is integrated with the trapezoidal method. The instrument uncertainty is expected to be smaller than the shot-to-shot variation in the magnetic field.

5.2.2 Triple probes

In a plasma where the electron velocity distribution is Maxwellian, the current density (particle flux) impinging on a conducting probe inserted in the plasma will be $\propto e^{q_e V/T_e}$ between the ion saturation current density $j_{isat} = q_e n\sqrt{m_e/k_B T_e}$ and the electron saturation current density $j_{esat} = j_{isat}\sqrt{m_i/m_e}$ [89]. Triple probes determine three points on the I-V curve, so the

electron temperature and density can be determined. Triple probes have a probe head with three electrodes; two are floating with a constant bias potential ΔV between them and one is floating freely. The negative electrode of the pair will draw the ion saturation current $I_1 = I_{isat}$ at a voltage V_1 while the positive electrode will draw an equal and opposite current $I_2 = -I_{isat}$ at voltage $V_2 = V_1 + \Delta V$. The floating electrode will be at the plasma floating potential $V_3 = V_f$ and draw no current $I_3 = 0$. Subtracting the ion saturation current I_{isat} from the current of the positive electrode and the floating electrode current and taking their ratio, yields

$$\frac{I_2 - I_{isat}}{I_3 - I_{isat}} = 2 = e^{\frac{q_e \Delta V}{k_B T_e}}, \tag{5.1}$$

allowing the electron temperature to be determined. The electron density can be determined from the I_{isat} current.

Both the electron mean free path ($\lambda_{mfp} \approx 2\ \mu m$) and the electron collision frequency ($\nu_{ee} \approx 30\ MHz$) are smaller than the probe and faster than the digitization frequency, respectively, making it a reasonable assumption that the electrons are Maxwellian. The electrode spacing of the triple probe has to be below the temperature and density gradient scale lengths for the three probes to sample the 'same' plasma. Instrument uncertainty stems from imperfections in the electrode surface area, both in the physical geometry of the surface area and from phenomena which increase the effective area such as secondary electron emission, photoemission of electrons, and negative ions [89].

5.2.3 Mach probes

Mach probes have two conducting faces on opposite sides of a probe head. The electrodes are biased to collect the ion saturation current. When there is no ion drift velocity, the ion saturation current on both electrode faces is identical. A non-zero ion drift increases the ion saturation current on the upstream side I_{up} and decreases the current on the downstream side I_{down} [90]. The ratio of currents through the electrode surface is proportional to the exponential raised to the power of the Mach number [91]. Taking the logarithm the Mach number M can

be expressed as

$$M = M_c \ln\left(\frac{I_{up}}{I_{down}}\right), \tag{5.2}$$

where M_c is a model dependent constant. The ion velocity is obtained by multiplying the Mach number with the acoustic speed $c_s = \gamma\sqrt{k_B T_e/m_i}$, where γ is the ratio of specific heats of the plasma and taken to be unity. Models for M_c depend on the magnetization of the ions characterized by the ratio of the ion Larmor radius over the size of the electrode r_{Li}/a [92]. In RSX the ratio is $0.5r_{Li}$, a weakly magnetized regime. The RSX team uses $M_c = 0.45$, however, no independent measurement exists to verify this value. Error bars of a factor of two should be assumed, based on the variation of M_c values reviewed in ref. [93]. It may be possible to constrain the range of M_c values by comparing with the $\vec{E} \times \vec{B}$ term of ideal Ohm's law [89], or time of flight measurements [91], however, this has not been attempted. Additional uncertainty comes from differences in the size of the electrodes on each side of the probe, resulting in differences in the upstream and downstream currents even in the case of no drift flow.

5.3 Step 1: Prepare data

This study aims to reconstruct the 3D plasma behavior from single point measurements taken from 2,369 experimental shots. Studies drawing on data across all shots require preparing and structuring the data, so that relevant shots can easily be assembled and coherent features across shots can be differentiated from shot variability.

5.3.1 Identifying a coherent shot subset

During RSX operation, raw digitizer traces were stored in a MDSplus database, and shot settings were recorded in several spreadsheets and handwritten lab books. For this study, an SQL database is created and populated with all shot settings and MDSplus shot numbers to facilitate selecting subsets of shots meeting specified conditions. To ensure that the analysis takes account of shot variability, several shot heuristics are stored in the database that quantify

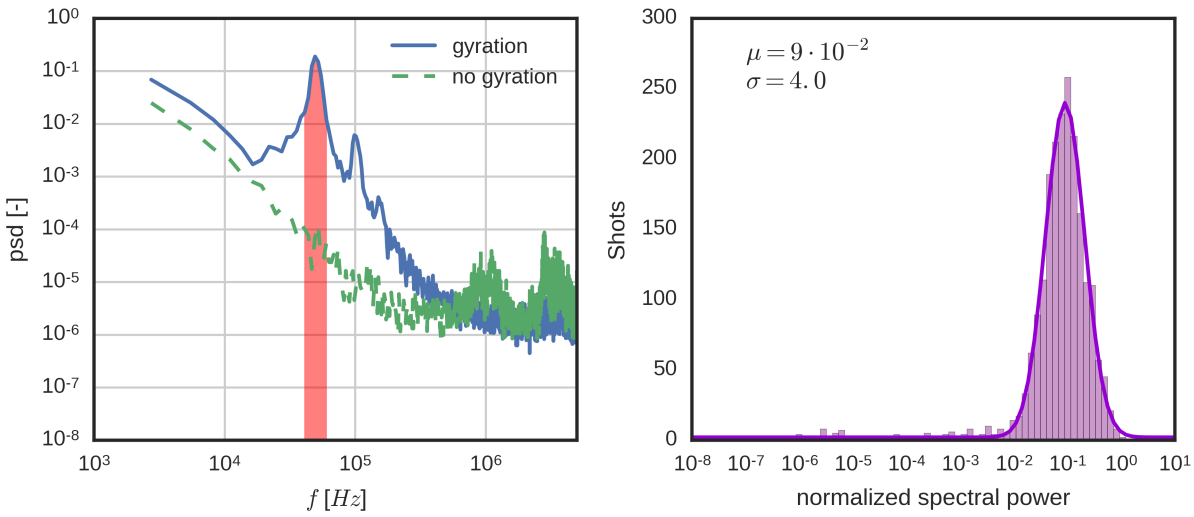


Figure 5.4: Left) Normalized power spectrum density of azimuthal magnetic field measured 5 cm from gyration axis by fiducial probe before the crowbar, averaged across 10 shots with significant gyration (#15256 – 15262 and #15267 – 15269), and 10 shots without gyration (#15305, #15306, #15336, #15352, #15408, #15408, #15410, #15412, #15414, #15416, and #15418). Right) Normalized spectral power distribution for all 2,369 shots obtained by integrating the red area in the left plot.

the coherence. The plasma behavior across shots can be differentiated into coherent behavior, dependent on applied currents and magnetic fields being consistent across shots, and into behavior with intermittent and chaotic effects that vary from shot-to-shot. The gyration appears as a modulation to the azimuthal magnetic field B_θ measured by a stationary fiducial probe 5 cm from the gyration axis (fig. 5.2). The standard deviation in the observed frequencies and amplitudes of the modulation are 8% and 40% of the mean values, respectively, which makes the gyration remarkably coherent across shots. The onset of gyration varies between shots and can occur after the crowbar. The power supply acts as a current source only before the crowbar. After the crowbar the current varies with the plasma impedance which has a large

shot-to-shot variation. Hence, only shots where significant gyration has developed before the crowbar are considered to be coherent. The spectral power density of the gyration frequency can be determined from the square of the discrete-time Fourier transform of the fiducial signal. Histograms of the spectral power density before the crowbar in all shots reveal a log Gaussian distribution with a non-Gaussian tail towards low densities (fig. 5.4). Only shots in which the spectral power density in a time window before crowbarring is within the Gaussian part of the distribution are considered for this study to eliminate shots that have no or too low gyration.

5.3.2 Conditional sampling of measurements across shots

Among the gyrating kink shots, there are variations of 100 μs in onset time. The variations in onset time can be taken into account by conditional sampling [94]: time shifting all probe traces so that the gyration oscillation in the fiducial traces are phase aligned. Start thresholds in the quadrature amplitude and slope in the sinusoidal modulation of the fiducial probe signal time align all shots in the relative time base of one gyration phase of 17 μs with 250 time steps of 68 ns.

5.3.3 Interpolating the measurements to a common rectilinear grid

Fig. 5.5 shows the measurement locations for each diagnostic on all four planes. The union of the measurement locations for all diagnostics is cross-hatched and is used as the joint measurement volume for reconstructing canonical flux tubes and integrating canonical helicity. The triple probe measurements in the $z = 30\ cm$ plane are dropped from further analysis, as they were qualitatively different from the measurements in the other planes; there are no localized peaks in temperature and density. The measurement locations follow a regular grid, but there are several gaps and irregularities. To simplify the data analysis, all measurement quantities are interpolated onto a common rectilinear grid with linear barycentric interpolation using a Delaunay triangulation of the measurement points of each quantity (fig. 5.6). The Delaunay triangulation is performed with the scientific Python (scipy) wrapper around the Qhull code

[95, 96]. Most locations only have a single measurement. To quantify the uncertainty in the helicity integrations the analysis is repeated several times; each time applying a Gaussian filter with successively larger standard deviations to the interpolated measurements. This averages the measurements over different shots at the cost of a coarser grid.

5.3.4 Constraining the under determined velocity

Mach probe measurements were only taken in the y direction on the $z = 0.42\ m$ plane and in the z direction on the $z = 0.36\ m$ and $z = 0.42\ m$ planes. The assumption of rigid rotation constrains the Mach numbers in the x direction. The Mach numbers in the x direction are determined by shifting the Mach numbers in the y direction by a quarter gyration period. For this study only the Mach measurements in the $z = 0.42\ m$ plane are used, which means that the axial variation of the ion velocity stems from the axial variation of the electron temperature.

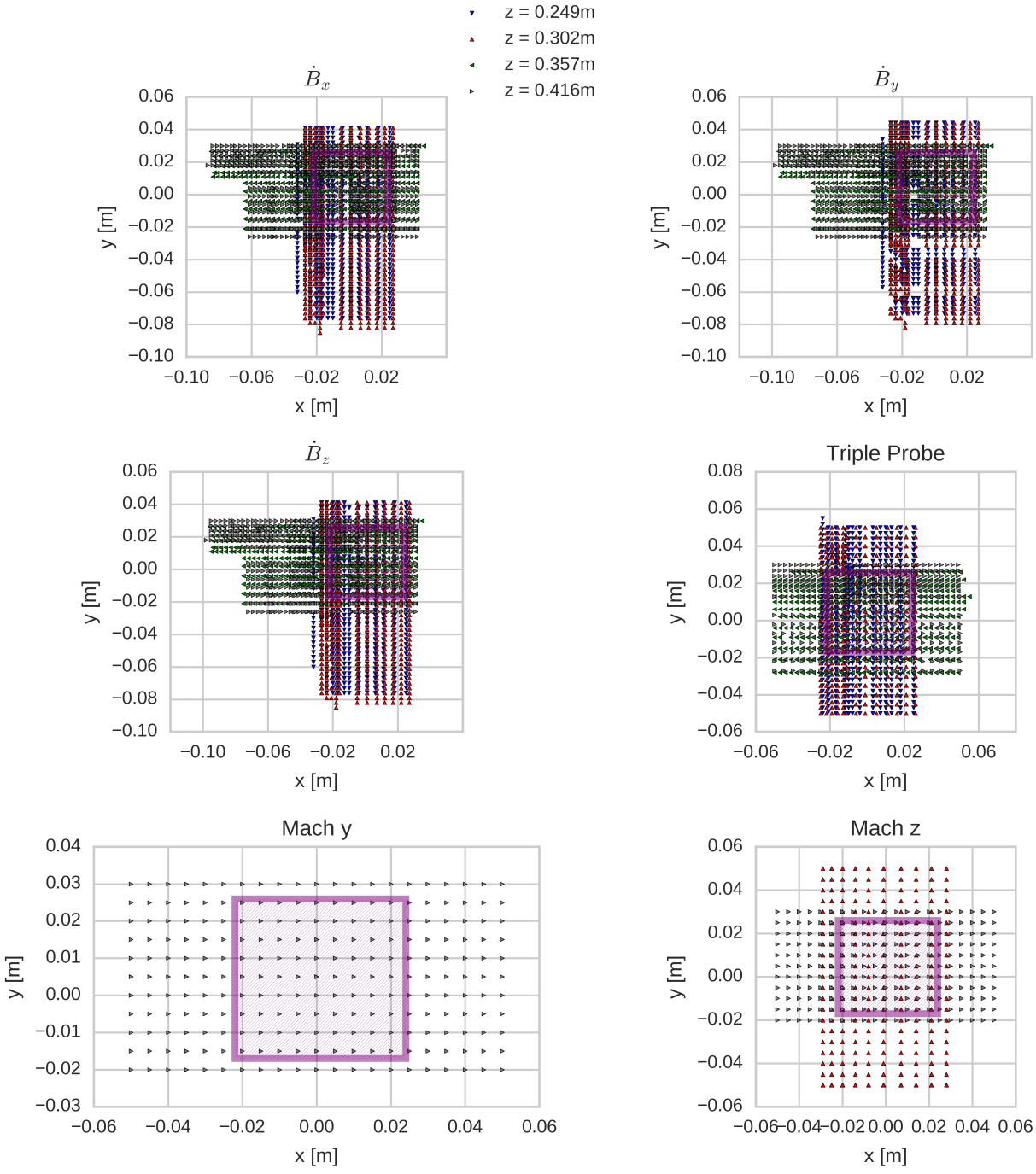


Figure 5.5: Points at which measurements were taken in the coherent shot set with $\dot{B}$, triple, and Mach probes. The (purple hatched) region is the union of measurement spaces which is used to reconstruct canonical flux tubes and their helicity.

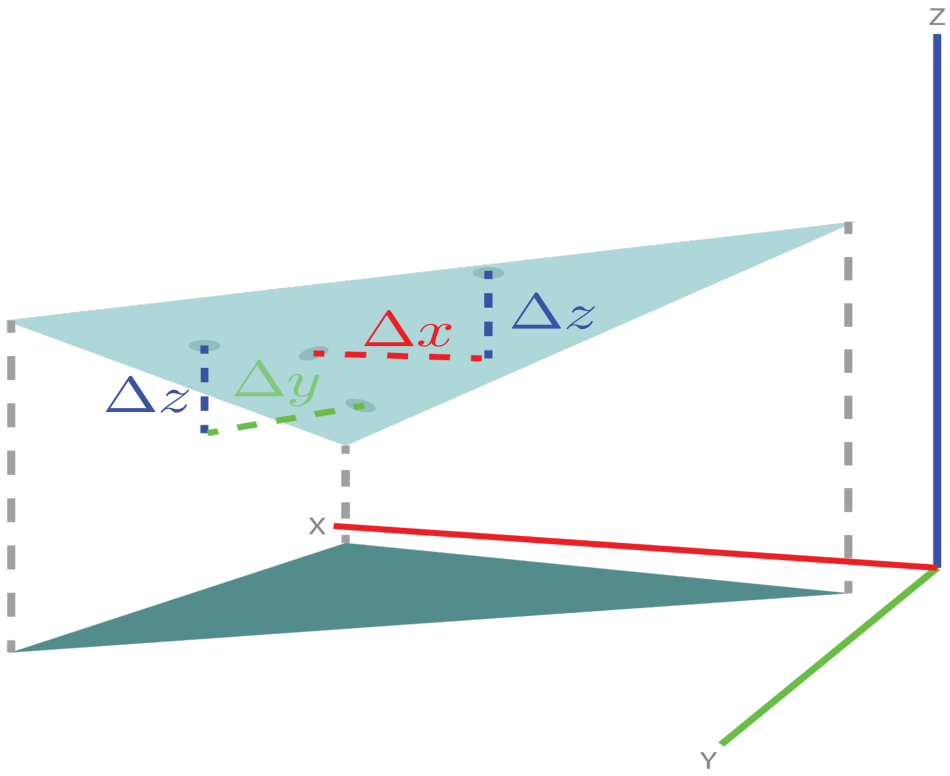


Figure 5.6: Linear barycentric interpolation: for interpolation on a 2D grid the vertices of a triangle are weighted by the z values. Each triangle has a uniform slope in the x and y directions ($\Delta z/\Delta x$ and $\Delta z/\Delta y$). The RSX data is on a 3D grid, so tetrahedrons are used.

5.4 Step 2: Reconstruct and interpret canonical flux tubes

If a vector field is divergence-less, field lines launched from the boundary of any 2D surface will trace out a tube of constant flux. To visualize canonical flux tubes the appropriate vector field definition and launching surfaces must be chosen. Species canonical flux is defined by eq. (2.3). However, the goal of the flux tube visualization is to develop intuition for canonical helicity. Since chapter 2 showed that canonical helicity is independent of the density gradient term in the canonical circulation (eq. (2.2)), the density gradient term will be dropped, reducing the canonical flux to eq. (2.4). In RSX the difference between current and ion flow vectors is small, so that the electron canonical flux tubes reduce to magnetic flux tubes. Three field quantities are needed to calculate the ion and electron canonical circulation: the density, the magnetic field, and the ion flow vorticity. The density and magnetic field have already been determined; the ion flow vorticity is determined by constructing the curl of ion flow from the slopes of the barycentric simplices. This study is interested in the dynamics of the gyrating current-carrying canonical flux tubes, so a launching surface should follow the gyration path.

5.4.1 Gyration path

The field null of the x and y components of the magnetic field has been identified as a tracer of the rotation. The x and y components of the magnetic field are shown in fig. 5.7 (left) together with their magnitudes. The field null is recognizable as the center around which the magnetic vectors circulate close to the peak of the magnetic field magnitude. However, since the magnetic field is asymmetric this is not the maxima of the curl of the magnetic field. An iterative scheme of fitting circles to integrated field lines of the x and y magnetic field and moving towards the circle center, determines the field null position for each time step. The starting point for the iteration scheme is the maximum of the x-y magnetic field magnitude. At each iteration, the field line integration launch point is moved towards the center of the least squares fitted circle by 10% of the radius. The iteration scheme continues until the fitted radius falls below 1 mm or the launch point falls outside of the common x-y magnetic field

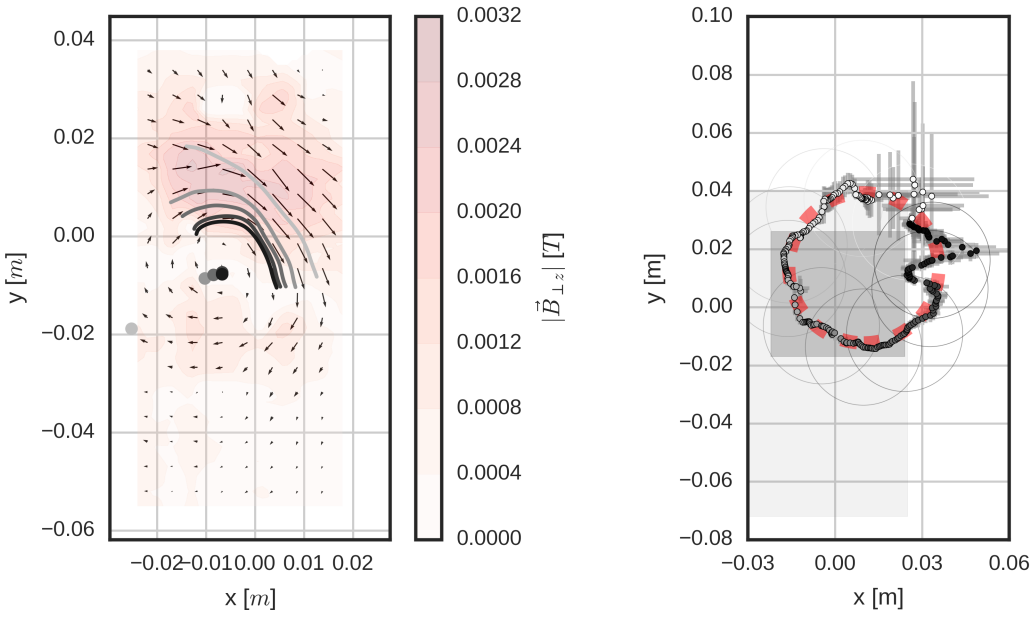


Figure 5.7: Left) Field null iteration scheme. $\vec{B}$ and $|\vec{B}|$ at time $t = 8.5\ \mu s$ in the $z = 25\ cm$ plane. A field line (successive steps from light gray to black) is integrated in both directions starting from the peak of $|\vec{B}|$, a circle is least squares fitted to the field line, and a new integration starting point is determined by moving 10% of the radius towards the circle center (successive steps from light gray to black). Right) Field nulls over one gyration period of 17 μs (white to black). Error bars (light gray) are determined by repeating the field null iteration scheme with successively stronger filtered data, averaging across shots. The field null gyration lies on a circle (dashed red). Circles with 2 cm radius are plotted around every 25$^{\text{th}}$ field null. 2 cm is the radius at which current and density drop of by $1/e$ [65]. Light gray region is the space in which both measurements of B_x and B_y were made. Dark gray region is the joint measurement space of all diagnostics used for flux tube reconstruction and helicity integrals.

measurement surface. Fig. 5.7 shows the gyration path of the field null in the $z = 0.25\ cm$ plane. The points and error bars are the averages and standard deviations of nine runs, the field null finding algorithm, one with unfiltered magnetic field interpolations and the others

with filtered magnetic field interpolation, where the standard deviation of the Gaussian filter kernel increases from 0.5 mm to 4.5 mm. The kernel is truncated at three times the standard deviation. The error bars account for both uncertainty due to the shot-to-shot variation of the measurements and the sensitivity of the fitting algorithm. The error is largest when the field null lies outside of the B_x and B_y measurement space, i.e. when the field null position has to be extrapolated from far away measurements. The field null positions with error bars lie on a circle (dashed red) centered at $x = 1\ cm$ and $y = 1.3\ cm$, and radius 2.5 cm.

5.4.2 *Canonical Flux Tubes*

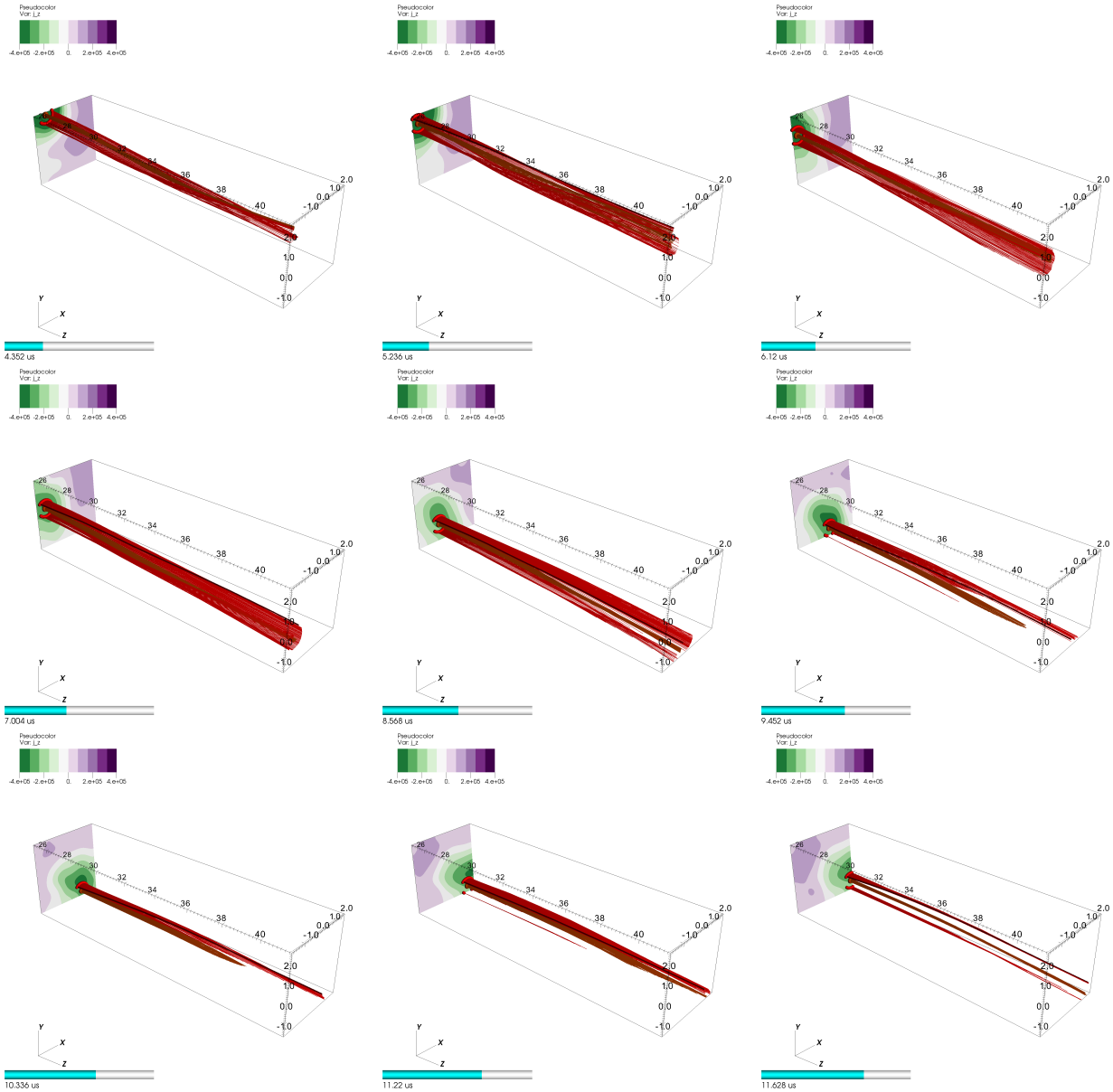


Figure 5.8: Gyrating electron canonical flux tubes. $n\vec{B}$ ($\vec{\Omega}_e$) field lines trace two electron canonical flux tubes over one gyration period of 17 μs in the joint measurement volume. The field lines are launched from circles of 1 mm (orange) and 5 mm (red) radii centered at the null of B_x and B_y in the $z = 25$ cm plane. A single field line (dark red) on the outer flux tube helps visualize the twist. Contours of j_z are plotted in the $z = 25$ cm plane, the colorbar units are $\frac{A}{m^2}$. The field null is inside the joint measurement from $t = 4.1$ μs to $t = 12$ μs.

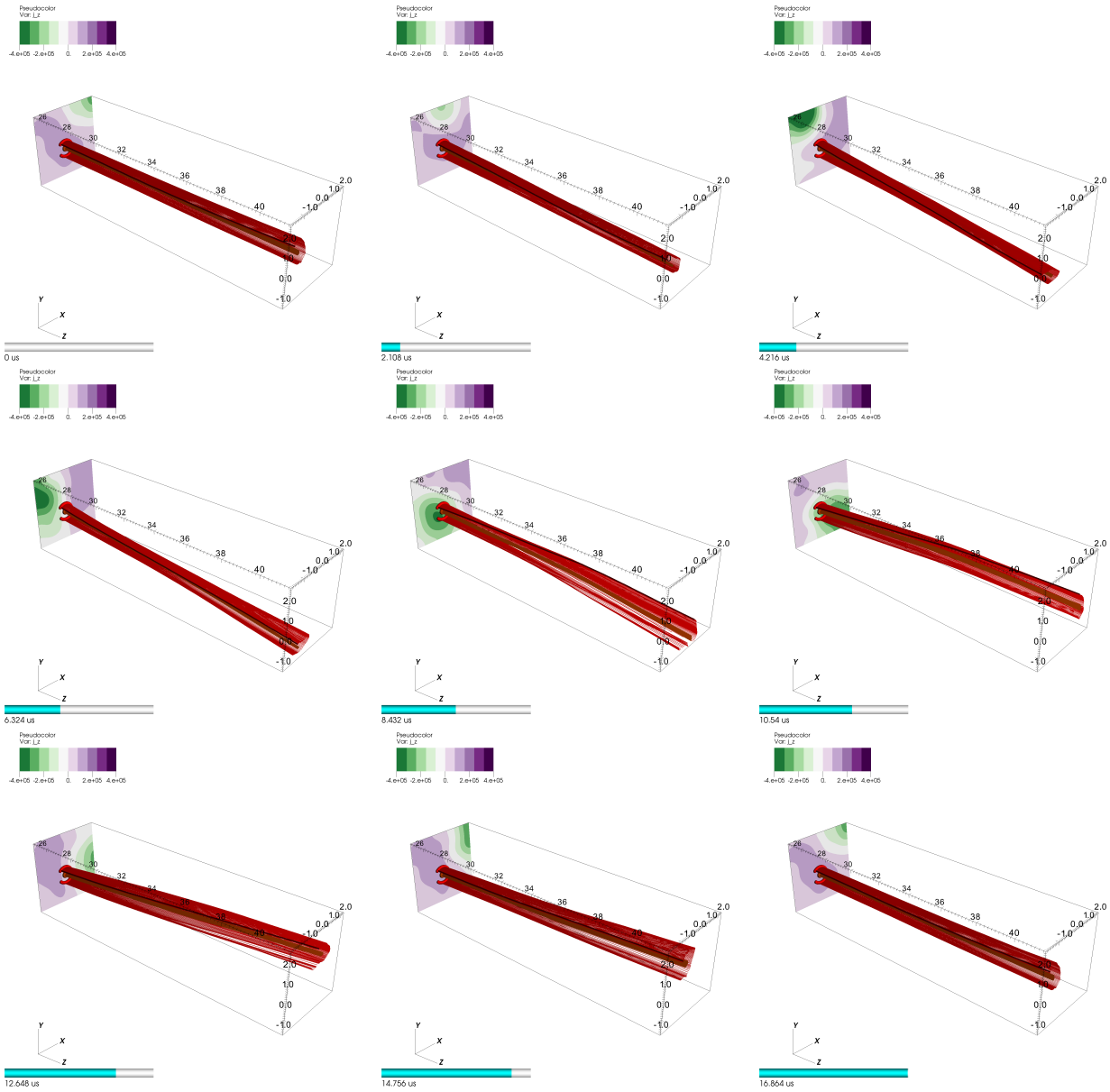


Figure 5.9: Stationary electron canonical flux tubes. $n\vec{B}$ ($\vec{\Omega}_e$) field lines tracing out two electron canonical flux tubes over one gyration period of 17 μs. The field lines are launched from circles of 1 mm (orange) and 5 mm (red) radii centered at $x = 0$ and $y = 0$. A single field line (dark red) on the outer flux tube helps to visualize the twist. Contours of j_z are plotted in the $z = 25$ cm plane, the colorbar units are $\frac{A}{m^2}$.

.

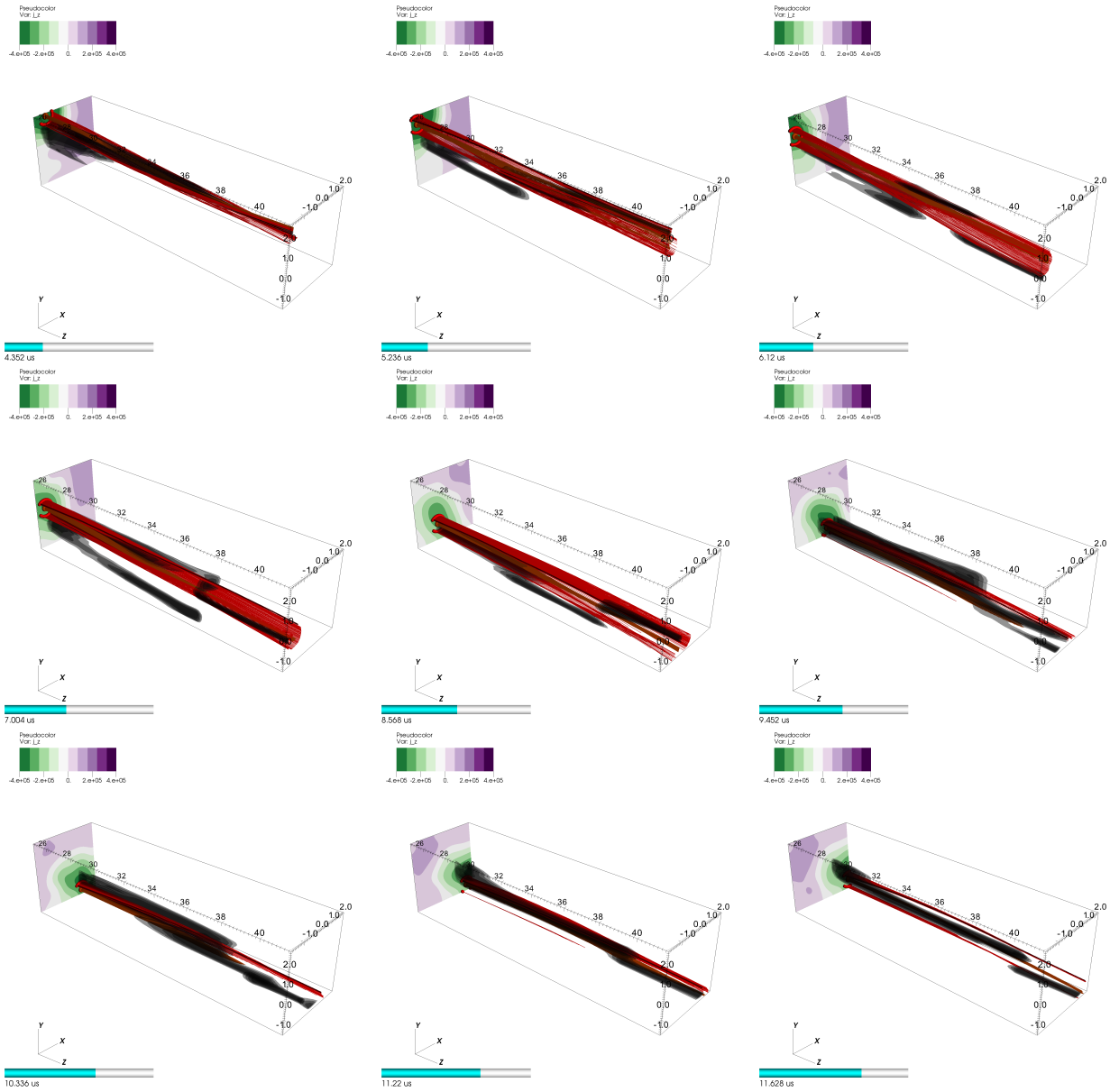


Figure 5.10: Isosurfaces of n normalized by the peak value in each of the x-y measurement plane over one gyration period of 17 μs. The last isosurface encloses the top 10% of n. For comparison, electron canonical flux tubes and j_z from fig. 5.8 are plotted.

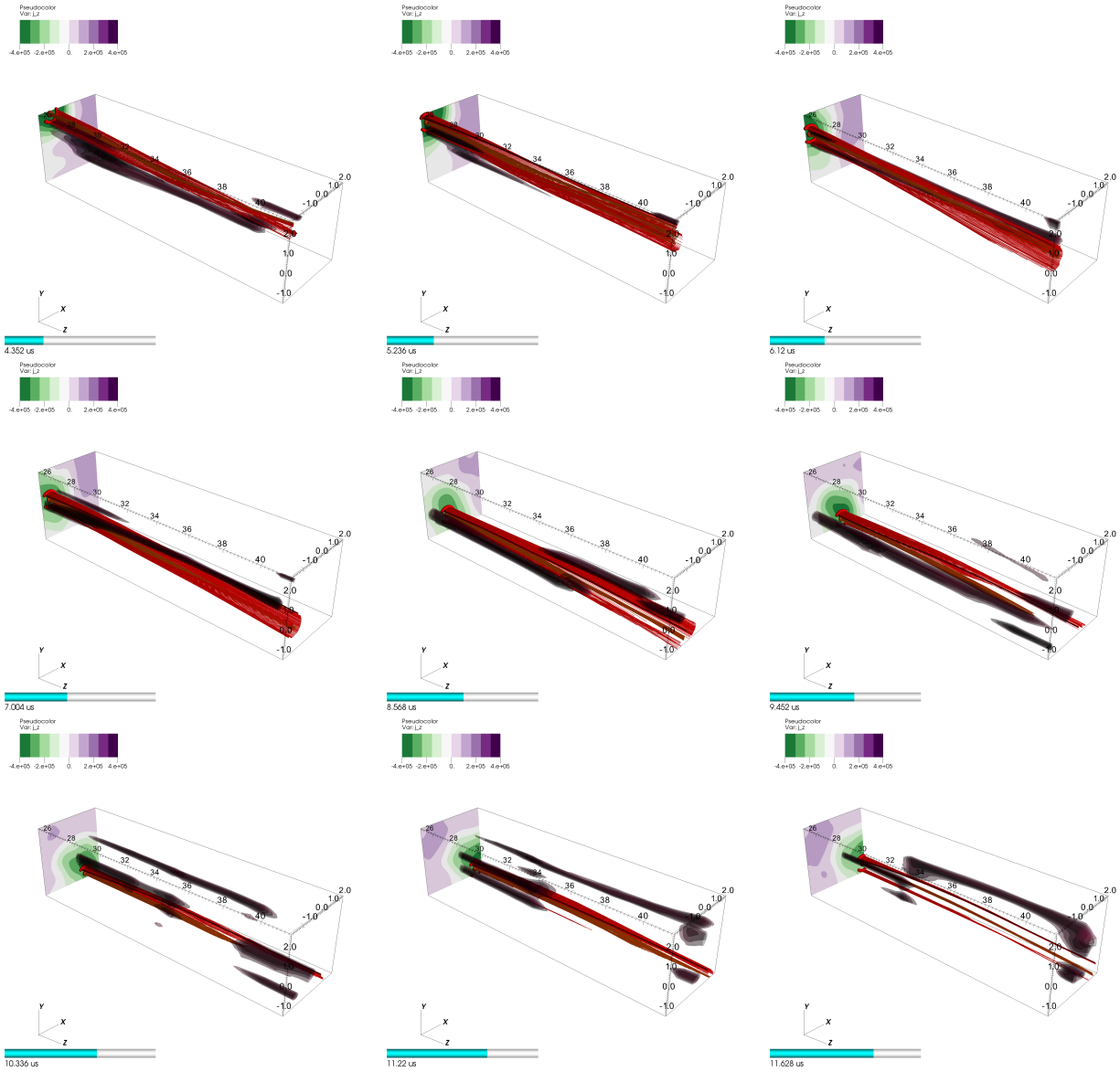


Figure 5.11: Isosurfaces of T_e normalized by the peak value in each x-y measurement plane over one gyration period of 17 μs. The last isosurface encloses the top 10% of T_e. For comparison, electron canonical flux tubes and j_z from fig. 5.8 are plotted.

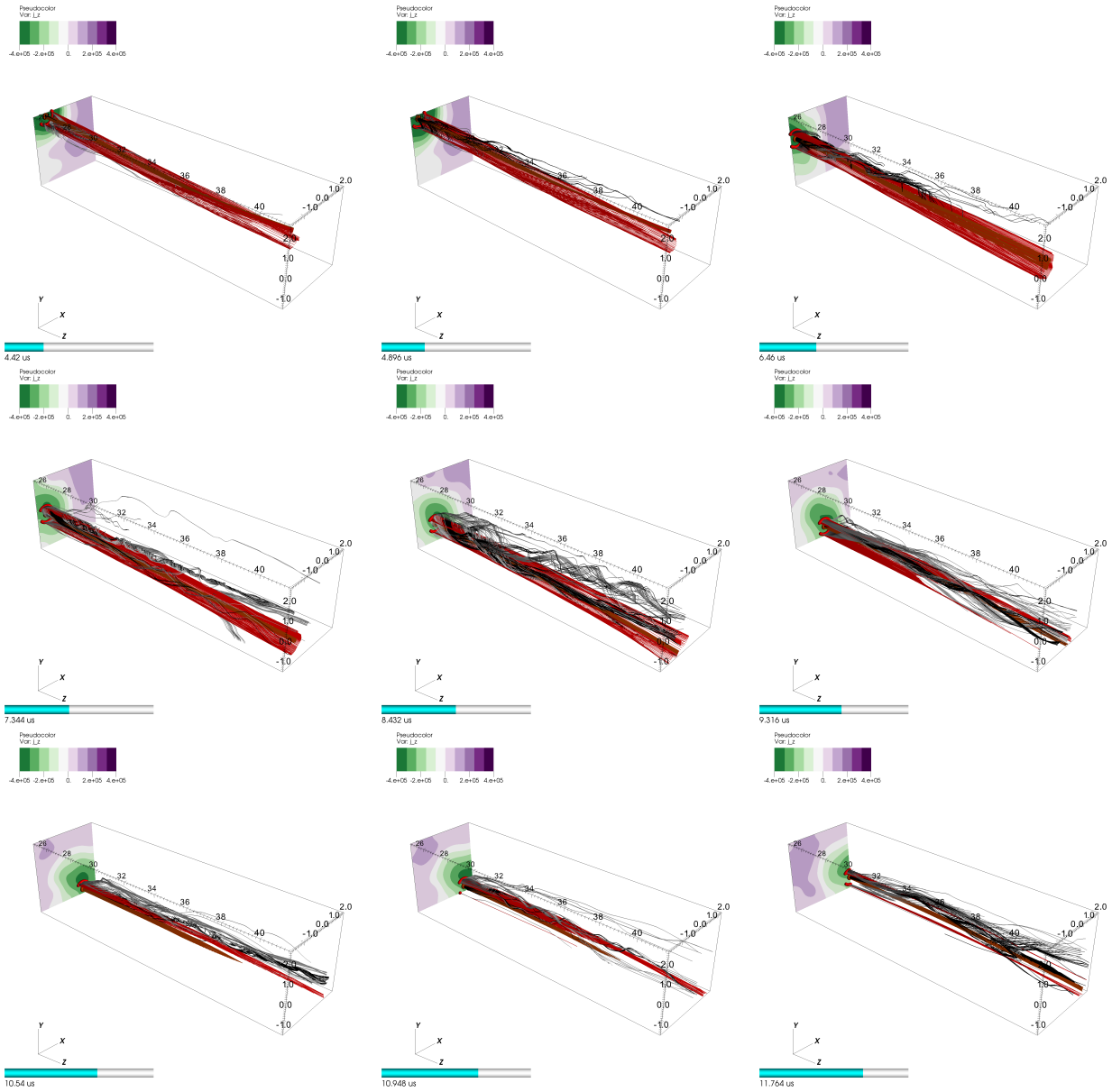


Figure 5.12: Gyrating ion canonical flux tubes. $\vec{\Omega}_i$ field lines trace out two ion canonical flux tubes over one gyration period of 17 μs. The field lines are launched from circles of 1 mm (black) and 5 mm (grey) radii centered at the null of B_x and B_y in the $z = 25$ cm plane. For comparison $\vec{B}$ flux tubes and j_z from fig. 5.8 are plotted.

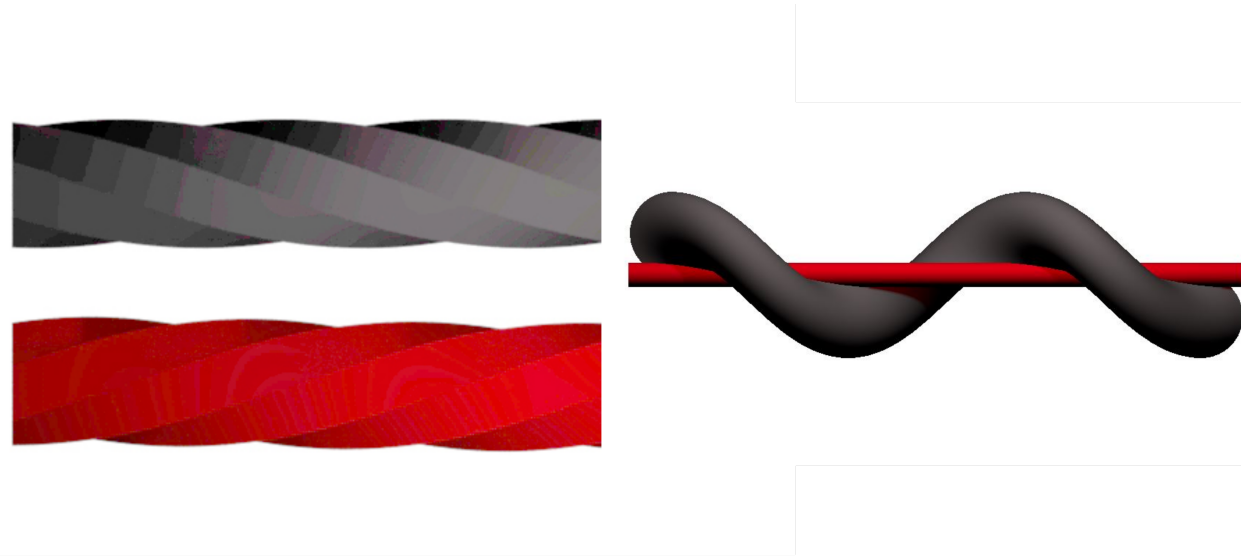


Figure 5.13: Schematics of twists and writhe. The positive axial direction is from left to right. Left) A right-handed twist (black) and a left-handed twist (red). Right) The black tube writhes with right-handed sense around the red tube.

Fig. 5.8 shows the evolution of two electron canonical flux tube during one gyration period. Field lines are launched from circles centered at the field nulls of B_x and B_y with 1 mm (orange) and 5 mm (red) radii. The electron canonical flux tube shape is dominated by the axial bias magnetic field which is an order of magnitude larger than the magnetic field due to internal currents. There is a reverse current that gyrates into the joint measurement volume as the field null is leaving the volume. This reverse current was identified as an eddy current in ref. [65], it is localized to $z \approx 25{-}35\ cm$. A single field line is plotted in dark red on the surface of the outer flux tube. The single field line raps in a clockwise direction around the flux tube, indicating that canonical electron helicity is negative. The twist of the electron canonical flux and helicity in the measurement volume switches sign as the field null leaves the volume and the reverse current is centered in the volume as can be seen by following the single field line plotted on top of the stationary outer flux tube in fig. 5.9 (compare fig. 5.13). The field null tends to

be slightly below and to the left of the centroid of axial current j_z. If the plasma column were axisymmetric the field null would lie on the current centroid, however, since the column has kinked the current profile has a 3D structure, it is neither radially or azimuthally symmetric. Figs. 5.10 and 5.11 show that the position of the field null and the electron canonical flux tube are correlated with the peaks of density and electron temperature. These figures plot the gyrating electron canonical flux tube with isosurfaces of the top 10% of the maximum density and electron temperature values, respectively. The density and temperature were normalized in each plane prior to plotting so that the isosurfaces would be continuous. The peak density overlaps with field null, or at times, is below and to the right of the field null 'leading' the field null gyration. Since there is no extrapolation of the density the peak must be inside the joint measurement volume, even when the field null leaves the volume. The co-motion of the density and electron canonical flux tubes is evidence of the MHD frozen-in flux condition. The peak electron temperature overlaps with the field null, or at times, is above and to the left of the field null 'lagging' the field null gyration. Both density and electron temperature isosurfaces follow the axial structure of the electron canonical flux tubes bending downward at higher z. The leading and lagging of the peak density and temperature may be due to positional uncertainty in the probes or may be the result of the plasma not being in equilibrium. Ref. [65] found a 25% discrepancy in the magnitudes of $\vec{J} \times \vec{B}$ and ∇p terms of the MHD force balance and suggested Coriolis forces from the gyration flows may be responsible for the discrepancy. The Alfvén transit time along the gyration path length ($2\pi \cdot 2.5\ cm$) is $\sim 2\ \mu s$, using the axial magnetic field $0.02\ T$ and peak density $3 \cdot 10^{19} m^{-3}$. $2\ \mu s$ is almost an order of magnitude smaller than the gyration period. This suggests that the system should be in a gyrating pseudo-equilibrium. However, ref. [65] notes that the pseudo-equilibrium would be irreducibly 3D as the current and magnetic field profiles are radially asymmetric and the reverse current disappears in the $z = 42\ cm$ plane. In Fig. 5.12 the electron canonical flux tubes are plotted together with ion canonical flux tubes. The ion canonical flux tubes vary rapidly because the ion vorticity has a larger amount of error due to amplification of noise when taking the derivatives required for determining the curl of ion flow. The noise makes the ion canonical flux vary rapidly and

without strong tubular structure. Since the ion canonical vorticity is calculated from a curl it is practically divergence-less. The divergence of the ion canonical circulation is on the order of $10^{-16}\ kg\ s^{-1}\ m^{-1}$ (without multiplying by density n), which is double-precision floating point accuracy. Hence, the field lines should trace cross-sections of constant flux, however it is the cross-sections themselves that are varying rapidly. The divergence of the magnetic field varies between 0.1 T/m, when the field null is inside the measurement volume, and 0.01 T/m, when the field null is outside the measurement volume. The divergence stems from interpolation and instrument error in the magnetic probes. This means that the field lines do not strictly trace out cross-sections of constant flux, however since the axial vacuum magnetic field dominates the dynamic magnetic field by an order of magnitude, a tubular structure is preserved. The ion canonical flux tubes show significant right-handed twist and at some time steps (7.344 μs, 8.432 μs, 9.316 μs) writhe around the magnetic flux tubes (compare fig. 5.13). The right-handed twist and writhes indicate that the total ion canonical helicity is positive and exceeds the magnetic helicity which must be due to cross or fluid helicity. These plots represent the first reconstruction of canonical flux tubes from experimental measurements.

5.5 Step 3: Determine vector potential and reference fields

Helicity is the volume integral of vector potential of the flux quantity of interest dotted with its circulation. Relative canonical helicity has to be used since intersections of ion and electron canonical flux tubes and the open system boundaries will add a gauge dependence to the non-relative forms of canonical helicity. The relative canonical helicity integrals require choosing suitable reference circulation fields and determining the reference vector potential fields. This section will discuss methods for determining reference fields $\vec{B}_{ref}$ and $\vec{\omega}_{iref}$ and vector potentials $\vec{A}$, $\vec{A}_{ref}$, and $\vec{u}_{iref}$.

5.5.1 Comparing vector fields

To test the methods developed in this section their solutions will be compared to known vector potentials and reference fields, however, vector fields are non-trivial to compare. Two scalar fields can be similar in certain regions and differ in others. In addition to regional differences, two vectors fields can have similar directions but different magnitudes and vice versa. To evaluate the reference field and vector potential calculation methods it is useful to quantify what effect they have on the helicity integrals. Ref. [97] developed the Taylor diagram that plots the scalar field characteristics: Pearson correlation coefficient, root mean squared field difference, and standard deviations of the scalar fields combined into a single polar diagram using a cosine relationship. Ref. [98] extended the Taylor diagram to vector fields introducing new characteristics: vector similarity coefficient, root mean square vector difference, and root mean square length. The vector similarity coefficient can be understood as a normalized helicity density sum between two vector fields and will be called the latter going forward. The normalized helicity density sum between two vectors $\vec{A}$ and $\vec{B}$ is defined as

$$\tilde{K} = \frac{\sum_{i=1}^{N} \vec{A}_i \cdot \vec{B}_i}{\sqrt{\sum_{i=1}^{N} |A_i|^2} \sqrt{\sum_{i=1}^{N} |B_i|^2}}, \tag{5.3}$$

where the sums are over all N spatial elements i. The normalized helicity density sum $\tilde{K}$ measures the pattern similarity of vector fields and depends on both their length and directions. $\tilde{K}$ does not change when either vector field is multiplied by a constant. The normalized helicity density sum varies between -1 and 1. A positive (negative) $\tilde{K}$ indicates that the angles between the vector fields are generally smaller (larger) than 90°. When the two vector fields are equal (equal and opposite), $\tilde{K}$ is 1 (-1). The root mean square vector difference $RMSVD$ between two vector fields $\vec{A}$ and $\vec{B}$ is defined as

$$RMSVD = \sqrt{\frac{1}{N} \sum_{i=1}^{N} |\vec{A}_i - \vec{B}_i|^2}. \tag{5.4}$$

$RMSVD$ quantifies the average difference in lengths of the two vector fields. The root mean square vector length L_A of a vector field $\vec{A}$ is defined as

$$L_A = \sqrt{\frac{1}{N}\sum_{i=1}^{N} |\vec{A}_i|^2}. \tag{5.5}$$

The normalized helicity density sum $\tilde{K}$, the root mean square vector difference $RMSVD$, and the root mean square lengths L_A and L_B can be expressed in a law of cosines

$$RMSVD^2 = L_A^2 + L_B^2 - 2L_A L_B \tilde{K}, \tag{5.6}$$

where $\tilde{K}$ acts as the angle between the L_A and L_B sides of a triangle and RMSVD is opposite to $\tilde{K}$. The three quantities can be plotted in a polar plot where $\tilde{K}$ is the azimuthal angle, L is the radius, and RMSVD is the distance between two points. Testing numerical techniques by plotting their solutions together with known solutions helps quantify the effect the numerical error has on helicity calculations. The effect on helicity can be separated between the effect from differences in the normalized helicity density sum $\tilde{K}$ and from differences in the root mean squared vector difference $RMSVD$.

5.5.2 Reference fields: Solving Laplace equation with fast Fourier methods

The conditions on the reference fields for canonical helicity (eq. 2.19) setup a Laplace problem with Neumann boundary conditions. The Laplace problem can be solved for a scalar potential ϕ of which the gradient is the reference circulation field. The Laplace problem is defined as

$$\begin{aligned} \nabla^2\phi &= 0 \\ \frac{\partial\phi}{\partial\hat{n}}|_S &= -\vec{\Omega}_\sigma \cdot \hat{n}|_S, \end{aligned} \tag{5.7}$$

where $\hat{n}$ is the surface normal. Since all quantities have been interpolated on a regular rectilinear grid the problem can be expressed as a finite difference formulation and solved with fast Fourier methods [99, 100]. On a 3D grid with indices j_1, j_2, and j_3, varying from 0 to J_n the Laplace equation can be expressed in finite differences

$$\phi_{j_1+1,j_2,j_3} + \phi_{j_1-1,j_2,j_3} + \phi_{j_1,j_2-1,j_3} + \phi_{j_1,j_2+1,j_3} + \phi_{j_1,j_2,j_3-1} + \phi_{j_1,j_2,j_3+1} + 6\phi_{j_1,j_2,j_3} = 0, \tag{5.8}$$

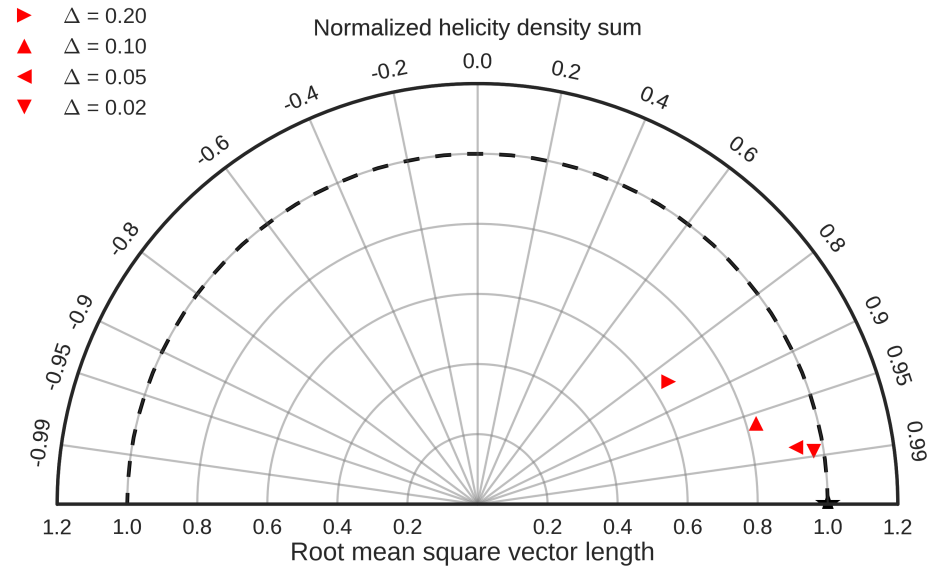


Figure 5.14: Taylor plot comparing known vacuum magnetic field (black star) with gradient of scalar potential (red triangles) obtained for 4 grid resolutions with the discrete cosine transform method for solving the Laplace problem. The known vacuum field is in the upper right quadrant of a x-y coordinate system with the conductor at its center. The known magnetic field is used as Neumann boundary condition for the Laplace solver.

with boundary condition on the lower x side

$$\frac{\phi_{-1,j_2,j_3} + \phi_{1,j_2,j_3}}{2\Delta_x} = -\vec{\Omega}_\sigma \cdot \hat{x}|_s, \tag{5.9}$$

where Δ_x is the spatial discretization in the x direction. The boundary conditions on the other edges are analogous. Discrete cosine transforms (DCT) use multiples of half-period cosines as basis functions. These functions have the convenient property that they impose homogeneous Neumann conditions $\partial\tilde{\phi}/\partial n = 0$. The inhomogeneous Neumann conditions can be treated by adding $2\Delta - \vec{\Omega}_\sigma \cdot \hat{x}|_S$ to the left hand side of eq. (5.8) at the edges [99]. The algorithm of the Laplace solver is 1) apply the DCT to the modified left hand side of eq. (5.8) ($2\Delta(-\vec{\Omega}_\sigma \cdot \hat{x}|_S)$), 2) solve for $\tilde{\phi}$ by dividing the transformed left hand side by the right hand side, and 3) apply the inverse DCT to obtain ϕ [101]. There are multiple possible definitions of the DCT. This

study chooses what is commonly referred to as the DCT-1

$$\tilde{f}_k = f_0 + (-1)^k f_{n-1} + 2\sum_{j=1}^{J-2} f_j cos(\pi jk/(J-1)). \tag{5.10}$$

The inverse DCT is eq. (5.10) multiplied by $1/(2(J-1))$. For 3D grids the DCT has to be applied three times for each dimension. This study uses the scientific Python (scipy) implementation of the DCT [95, 102]. The divisor in step 2) can be expressed as

$$\lambda = -\left(\frac{\pi^3(k_1+1)(k_2+1)(k_3+1)}{K_1K_2K_3}\right)^2. \tag{5.11}$$

Fig. 5.14 compares three Laplace solutions for a magnetic vacuum field outside a current-carrying conductor. The normalized helicity density sum and the ratio of root mean square lengths of the known solution and the gradient of the computed scalar potential approach 0.95 and 1, respectively, with increasing resolution.

5.5.3 Two-component vector potential

The choice of a convenient gauge can simplify the calculation of the vector potential. Several coronal loop studies and simulations have identified setting the axial field component to zero $A_z = 0$ as part of a convenient gauge choice [103, 69]. The curl of $\vec{A}$ becomes

$$\nabla \times \vec{A} = -\frac{\partial A_y}{\partial z}\hat{x} + \frac{\partial A_x}{\partial z}\hat{y} + \left(\frac{\partial A_y}{\partial x} - \frac{\partial A_x}{\partial y}\right)\hat{z}. \tag{5.12}$$

For a region of interest bounded by $[x_1 : x_2, y_1 : y_2, z_1 : z_2]$ the vector potential can be expressed as integrals along z

$$A_x(x,y,z) = A_{0x}(x,y) + \int_{z_1}^{z} B_y(x,y,z')\,dz' \tag{5.13}$$

and

$$A_y(x,y,z) = A_{0y}(x,y) - \int_{z_1}^{z} B_x(x,y,z')\,dz'. \tag{5.14}$$

The choice of A_{0x} and A_{0y} is constrained by the third term of eq. 5.12, one valid choice is

$$A_{0x}(x,y) = -\frac{1}{2}\int_{y_1}^{y} B_z(x,y',z=z_1)\,dy' \tag{5.15}$$

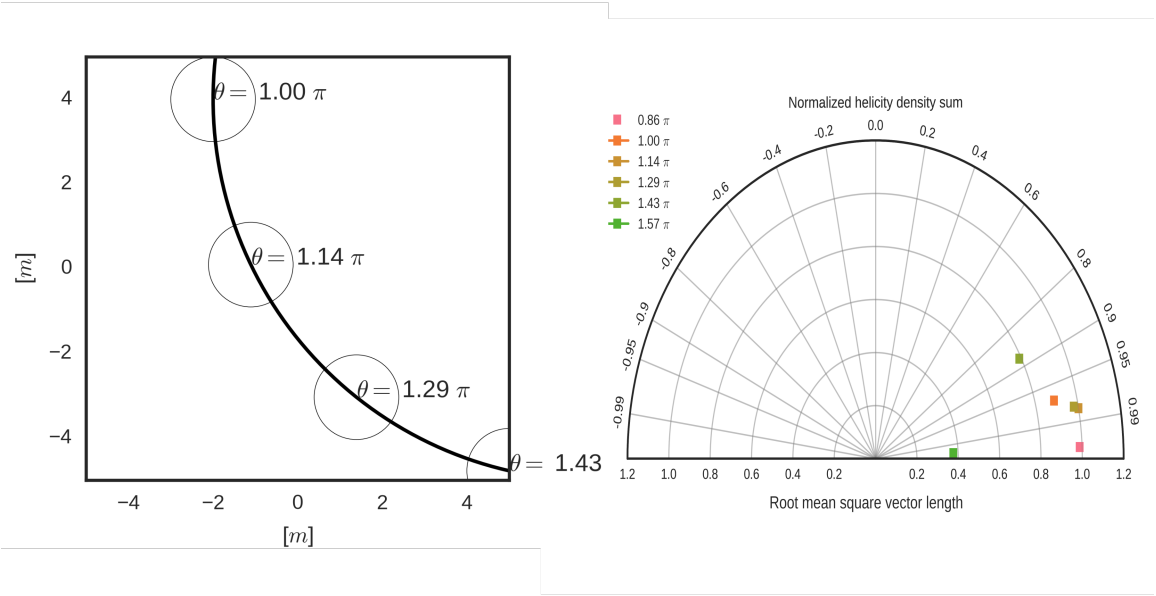


Figure 5.15: Left) Test problem setup for the two-component vector potential solver. The known test fields are the magnetic fields from a gyrating current-carrying cylindrical conductor at several angles. Right) Taylor plot comparing known magnetic fields from gyrating current-carrying conductor at various angles with curl of vector potential solutions obtained with the two-component vector potential method (squares).

and

$$A_{0y}(x,y) = \frac{1}{2}\int_{x_1}^{x} B_z(x', y, z = z_1)\, dx'. \tag{5.16}$$

The eqs. (5.15), (5.16), and $A_z = 0$ define the gauge. Fig. 5.15 shows that this method of determining the vector field performs well for the test problem of a uniform current-carrying conductor, which is gyrating in the x-y plane. The normalized helicity density sum of the curl of the calculated vector potentials and the known magnetic field are above 0.85. The two-component vector potential method performs well even when the current-carrying conductor is at the boundary, i.e. the boundary conditions have strong non-periodicity, which is in contrast to a method based on inverting the curl in (periodic) Fourier space, proposed in ref. [92].

5.6 Step 4: Reconstruct and interpret canonical helicity

Now that all necessary quantities are known, the dot products of the three-helicity expression of relative ion canonical helicity (eq. 2.20) are integrated with the trapezoidal method. Fig. 5.16 shows the evolution of magnetic (red), cross (green), fluid (blue) helicity, and their sum, which is the total ion canonical helicity (black), in the joint measurement volume during one gyration period. The gauge-dependent magnetic helicity is positive when the field null is outside the measurement volume, decreases as the field null enters the measurement volume, reaches its peak magnitude and increases again to a positive value as the field null exits the measurement volume. The gauge-dependent cross helicity is always positive and is anticorrelated with the gauge-dependent magnetic helicity, although the gauge-dependent cross helicity starts increasing $\sim 2\ \mu s$ after the the magnitude of the gauge-dependent magnetic helicity starts increasing. The signs of helicities agree with the handedness of the twist of the electron and ion canonical flux tubes in figs. 5.8, 5.9, and 5.12. The density weighting of helicity increases the magnitude of the helicities when the field null, which co-gyrates with the peak density is inside the joint measurement volume. The strong twist and writhing visible on the ion canonical flux tubes must be due to the cross helicity as it is an order of magnitude larger than the fluid helicity. The relative canonical helicities have smaller magnitudes than their gauge-dependent counterparts. They exhibit increased noise due to the differentiation step in determining the relative helicity, which is taking the gradient of the reference potential field. The magnitude of the relative fluid helicity barely exceeds the noise, it may be negligible. The ratio of peaks of the relative magnetic helicity and relative cross helicity is 0.12 which agrees roughly with the following dimensional analysis: Using ion mass $m_0 = 1.7 \cdot 10^{-27}\ kg$, the elementary charge $q_0 = 1.6 \cdot 10^{-19}\ C$, the perpendicular magnetic field $B_0 = 0.002\ T$, the radial current and density e-folding length $l_0 = 0.02\ m$, and the peak ion velocity $u_0 = 2.5 \cdot 10^4\ m/s$, the ratio of magnetic helicity over cross helicity becomes

$$\frac{\mathscr{K}_{irel}}{\mathscr{X}_{irel}} = \frac{q_i^2 \int n\left(\vec{A}_- \cdot \vec{B}_+\right) dV}{q_i m_i \int n\left(\vec{u}_{i-} \cdot \vec{B}_+ + \vec{B}_- \cdot \vec{u}_{i+}\right) dV} \approx \frac{q_0 B_0 l_0}{2 m_0 u_0} \approx 0.08. \tag{5.17}$$

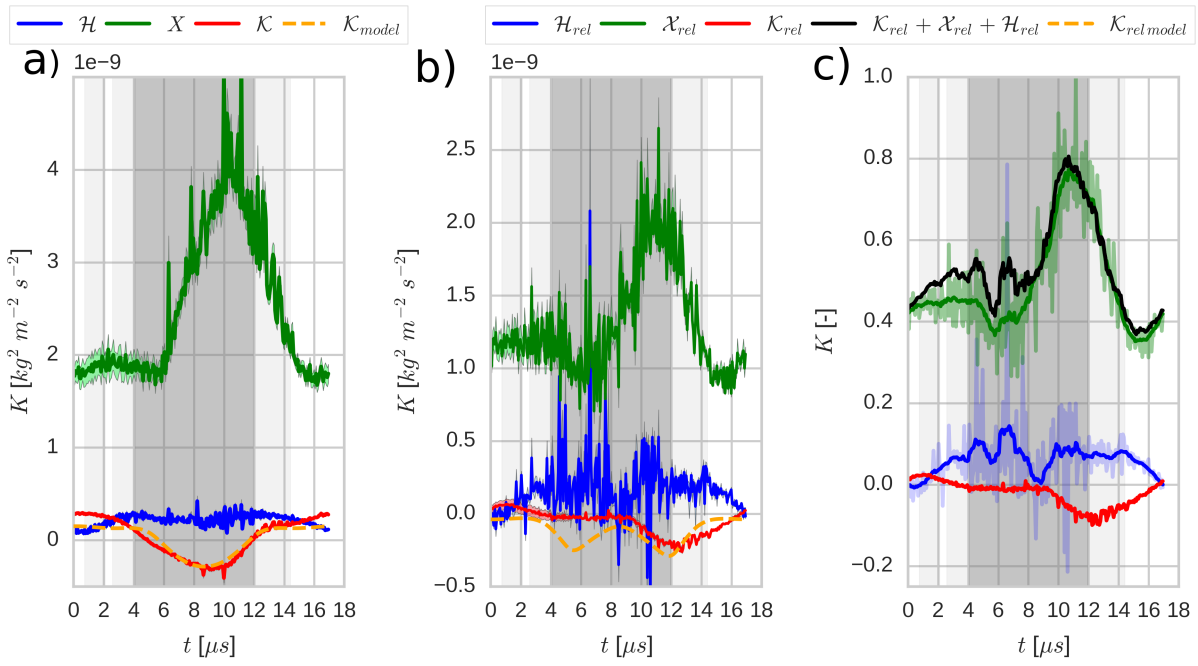


Figure 5.16: Magnetic (red), cross (green), flow (blue), and total ion canonical (black) helicity in the joint measurement volume identified in figs. 5.7 and 5.5. a) Gauge-dependent helicities with real units. b) Relative helicities with real units. c) Boxcar (1.02 μs width) filtered relative helicities, normalized to the maximum relative cross helicity. Dark gray shading marks the time the field null is inside the joint measurement volume. Light gray shading marks the time at which the field null is inside the region with both B_x and B_y measurements shown in fig. 5.7. Dashed orange shows the magnetic helicity of a simplified model problem. The model problem consists of a cylindrical conductor with 0.02 m radius and uniform current density $j = 140\ kA/m^2$ with an exponential density cloud, $n(r) = 1.3 \cdot 10^{19} e^{r/(0.02\,m)}\ m^{-3}$, where r is the radius. The conductor is gyrating along the red dashed circle fitted to the field null path in fig. 5.7. All helicities are the mean determined from running the analysis nine times with successively increasing the standard deviation of the Gaussian kernel filtering of the interpolated measurements. The standard deviation of the runs is plotted as an envelope around the helicity curves, however, the helicity calculations are insensitive to this filtering and the envelope is only partially visible.

The increases in the magnitudes of the relative magnetic and cross helicities are better aligned in time than the increases of their gauge-dependent counterparts. To understand the time offset between the field null entering the joint measurement volume and the increase in helicity magnitudes, it is helpful to reduce the problem to the approximation of a current-carrying cylindrical conductor, parallel to the z-axis, gyrating along the circle fit to the field null gyration in fig. 5.7 through vacuum. Choosing the e-folding length of the current and density (0.02 m) as the radius of the conductor, a uniform current density of $j = 140 kA/m^2$ (compare current density in fig. 5.8) and an exponential density cloud given by $n = 1.3 \cdot 10^{19} e^{r/(0.02\,m)}\ m^{-3}$, where r is the radius, yields a gauge-dependent magnetic helicity (dashed orange) consistent with the gauge-dependent magnetic helicity calculated from the RSX data. However, there is a discrepancy in the relative magnetic helicities calculated from gyrating conductor and the RSX magnetic fields. While the magnitude of relative magnetic helicity of the model problem has two peaks, the magnitude of helicity calculated from the RSX magnetic fields only has a single peak, which closely follows the second peak of the model. Looking at the canonical flux tube evolution (fig. 5.8), the discrepancy may be due to the reverse eddy current and the offset between the current-carrying canonical flux tube and the peak density. From $t = 4\ \mu s$ when the field null enters the joint measurement volume to about $t = 7\ \mu s$ the reverse current encompasses more than half the measurement volume with a peak of about 1/3 of the magnitude of the negative axial current. The reverse current is in the $+z$ direction, generating a counterclockwise azimuthal field which together with the positive axial magnetic bias field gives a right-handed (positive) helicity. This positive helicity counteracts the negative helicity from the main (negative) axial current. The axial bending of the canonical flux tubes and current path may also have an effect on the helicity. It should be possible to identify the contributing factors by expanding the simple current-carrying conductor model to include more features of the RSX current-carrying canonical flux tubes.

Since most measurement locations in fig. 5.5 only have a single measurement, it is important to test if the helicity calculations are sensitive to the shot-to-shot variation. The helicities plotted are the averages of nine runs of the data analysis described in this chapter. One run

starts with unfiltered interpolated values of B, n, T_e, and Mach numbers, the other runs start with filtered interpolations, where the standard deviation of the Gaussian filter kernel increases from 0.5 mm to 4.5 mm. The Gaussian kernel is truncated at three times the standard deviation. This averages across measurements at the cost of a coarser grid. The standard deviations of these runs are plotted as envelopes around the helicity curves. However, the helicities are not sensitive to the filtering of the interpolations. To reduce the shot variability in gyration amplitude it may be possible to average measurements over several periods of gyration, or repeat the helicity analysis with a limited subset of shots with a more limited spectral power distribution. A significant source of noise and uncertainty in the cross and fluid helicities is the curl of the linearly interpolated velocity. The noise could be reduced by fitting higher order functions to the measurements. Since the measurements are on a mostly rectilinear grid with only a few gaps, linear interpolation could be applied selectively to the gaps, followed by tri-cubic interpolation across the whole rectilinear grid. In addition, it may be possible to take advantage of cross-correlation between the four measurement quantities with a form of Gaussian process regression [104].

Since the RSX plasma motion becomes exceptionally coherent once the kink has saturated, this study has focused on calculating the helicity evolution during this time period. The variations in the helicities during the saturated kink seem to be explained by different sections of the current distribution gyrating in and out of the joint measurement volume. However, since the current structure in RSX is on the scale of the ion inertial length, there may be a conversion between destabilizing magnetic helicity and cross helicity as the kink develops and saturates. Indeed, a kinetic simulation of a periodic current-carrying magnetic flux tube with similar parameters to RSX develops forces and dynamics on kinetic scales and identifies possible reconnection sites [105]. If the helicities could be reconstructed in the current ramp-up phase it may be possible to observe a reduction in magnetic helicity together with an increase in cross helicity during onset of the kink. Cross-spectral data analysis techniques [106] could identify coherent mode structures between the fiducial and translatable probes at specific frequencies [107].

5.7 Conclusion

Canonical flux tubes and relative canonical helicity have been reconstructed from the Mach, triple, and $\dot{B}$ probe measurements from 2,369 shots of kinked, gyrating canonical flux tubes. The electron and ion canonical flux tubes co-gyrate together with peak density and temperature. The electron and ion canonical flux tubes twist in left-handed and right-handed sense, respectively; in addition, the ion canonical flux tube writhes around the electron canonical flux tube. The signs of the twist agrees with the signs of the calculated relative magnetic and cross helicities. The magnetic and cross helicities are anticorrelated and increase and decrease in magnitude as the flux tubes gyrate in and out of the joint measurement volume. An offset in the increase of the relative magnetic and cross helicity magnitudes and the entry of the field null into the joint measurement volume may be due to the reverse eddy current and the helical shape of the canonical flux tube. While minimum energy states resulting from magnetic helicity conservation have been widely explored, there have been few direct experimental calculations without symmetry assumptions due to lack of 3D measurements [34, 108]. Researchers at the LAPD experiment at UCLA have calculated magnetic and cross helicity of gyrating and merging plasma columns, however, not in a gauge-independent relative form [109]. This work is the first calculation of relative canonical helicity in a dynamic system.

Chapter 6

CONCLUSION AND OUTLOOK

Observations indicate that the dynamics of plasmas in our cosmos, the heliosphere, and terrestrial experiments can involve coupling and conversions between magnetic and kinetic energies, such as in reconnection and dynamo behavior. Recent theoretical work [39] provides a geometrical picture of these dynamics as conversions between electron (magnetic) and ion canonical (magnetic, cross, and fluid) helicity, while conserving their sum. Experimental validation of the theory of canonical helicity transport could extend the quasi-static theory of magnetic helicity relaxation to phenomena with fast reconnection and multiscale dynamics, such as coronal eruptions, sawteeth in reverse field pinches and tokamaks, and could elucidate how destabilizing magnetic twist converts to shear flow in astrophysical jets and compact toroid merging. Sheared flows have been observed to be stabilizing in Z-pinches [40] and are thought to play a role in transport barriers in H-mode tokamaks [110]. Understanding canonical helicity injection may lead to methods for driving shear flow in fusion experiments analogous to how coaxial [111] and inductive [112] magnetic helicity injection drives current. To improve the understanding of canonical helicity transport, the theory needs to be tested and compared to experimental observations.

This thesis presents the first application of the theory of canonical helicity transport to design a laboratory canonical flux tube experiment and develops methods to measure a large volumetric dataset, to reconstruct the 3D dynamics of canonical flux tubes, and to interpret the canonical helicity evolution. A sausage-kink instability cascade has been identified in an idealized lengthening current-carrying magnetic flux tube. If the flux tube's current-to-magnetic flux ratio can be increased faster than the inverse aspect ratio decreases, and the ratio of the flux tube radius over the ion skin depth is kept below two, the instability cascade may couple

the MHD system to smaller scales at which conversions between species helicities are expected to occur. Experiment control and high-throughput FPGA based digitizers for the Mochi.Labjet experiment provide the means to measure the large volumetric datasets needed to calculate helicity volume integrals. Ion and electron canonical flux tubes and their helicity have been visualized and calculated from a volumetric dataset of Mach, triple, and $\dot{B}$ probe measurements at over $10,000$ spatial locations of a gyrating kinked plasma column in a subvolume of the RSX experiment. The flux tubes are launched from the field null of the perpendicular magnetic field and co-gyrate with the peak density and electron temperature on a circular path in and out of the subvolume. The electron and ion flux tubes twist with opposite handedness and the ion canonical flux tube writhes around the electron canonical flux tube. The cross helicity between the magnetic and ion flow vorticity flux tubes dominates the ion canonical helicity and is anticorrelated with the magnetic helicity.

The Mochi.Labjet experiment described in this thesis is currently exploring the $\bar{k}$ - $\bar{\lambda}$ stability space of current-carrying magnetic flux tubes. A high-density 3D magnetic probe array and vector tomography based on ion Doppler spectroscopy measurements will provide the data necessary to calculate canonical helicity and investigate possible conversion during an instability cascade.

The data analysis techniques developed in this thesis can be applied to other experiments with potential for interactions between magnetic, cross, and fluid helicities. The decreasing cost of high-speed digitizers with high-channel count and high speed cameras will make the collection of large sets of electrical and spectroscopic data feasible on a variety of experiments.

For example, the Magnetic Reconnection Experiment (MRX) team observed an arched magnetic flux tube self-organize into a lower energy state while undergoing a torus instability; a decrease in toroidal current and associated poloidal field generates a stabilizing increase in the toroidal field, conserving magnetic helicity [29]. However, the relaxation process can also be interrupted by a kink instability with a violent conversion of magnetic energy into kinetic energy, mimicking a coronal loop eruption. While MRX focuses on measurements in a 2D plane its successor experiment FLARE aims to record 3D magnetic field and ion flow measurements

[113].

One configuration of the Large Plasma Device (LAPD) generates multiple plasma columns, which rotate around each other and periodically bounce and collide [114]. In the experiment the magnetic helicity and the magnetic field line squashing factor oscillate, with peaks of magnetic helicity corresponding to onsets of magnetic field squashing factor growth [115].

Reversed-field pinches (RFP) are relaxed minimum energy states constrained by magnetic helicity. An RFP exhibits a periodic sawtooth cycle characterized by a slow peaking of the current profile followed by a sudden global relaxation of current and density gradients, representing a discrete relaxation event in time. Magnetic helicity evolution has been measured indirectly during sawtooth crashes at the Madison Symmetric Torus (MST) by using a neural network to infer internal magnetic profiles from other measured plasma quantities [34]. Magnetic helicity varies slower than magnetic energy as MHD theories predict, however, the magnetic helicity decay is larger than expected. Simulations, though unable to reach the Lundquist numbers of lab plasmas, find that magnetic helicity is degraded when two-fluid effects are included [116]. Insertable Mach and magnetic probes can survive in the edge plasma and acquire measurements of sawteeth events at the $q = 0$ surface. The presence of ion flows on inertial length scales [117] suggests the possibility of conversion of magnetic helicity into cross helicity. Calculating the canonical helicity could reveal if part of the magnetic helicity is converted into cross helicity.

Applying the analysis techniques developed in this thesis to 3D measurements of magnetic fields and ion flows at these experiments could probe if the magnetic helicity is converted to cross or fluid helicity and if helicity injection and dissipation terms can be identified and quantified.

BIBLIOGRAPHY

[1] J. A. Bittencourt. *Fundamentals of Plasma Physics*. New York: Springer-Verlag, 2004. DOI: 10.1007/978-1-4757-4030-1.

[2] D. A. Gurnett and A. Bhattacharjee. *Introduction to Plasma Physics*. Cambridge, UK: Cambridge University Press, 2005. DOI: 10.1017/cbo9780511809125.

[3] Richard Dendy, ed. *Plasma Physics: An Introductory Course*. Cambridge, UK: Cambridge University Press, 1994. ISBN: 0521433096.

[4] Toshiki Tajima and Kazunari Shibata. *Plasma Astrophysics (Frontiers in Physics)*. Boulder, CO: Westview Press, 2002. ISBN: 0813339960.

[5] Toshiki Tajima. *Computational Plasma Physics: With Applications to Fusion and Astrophysics (Frontiers in Physics, Vol. 72)*. Redwood City, CA: Addison-Wesley, 1989. ISBN: 0201164116.

[6] P. M. Bellan. *Fundamentals of Plasma Physics*. Cambridge, UK: Cambridge University Press, 2006. ISBN: 978-0-521-52800-9. DOI: 10.1017/cbo9780511807183.

[7] Micheal D. Smith. *Astrophysical Jets and Beams*. Cambridge, UK: Cambridge University Press, 2012.

[8] C. Goddi, G. Surcis, L. Moscadelli, H. Imai, W. H. T. Vlemmings, H. J. van Langevelde, and A. Sanna. "Measuring magnetic fields from water masers in the synchrotron protostellar jet in W3(H2O)". In: *Astronomy & Astrophysics* 597 (2016), A43. DOI: 10.1051/0004-6361/201629321.

[9] G. R. Gupta, Durgesh Tripathi, and Helen E. Mason. "Spectroscopic observations of a coronal loop: basic physical plasma parameters along the full loop length". In: *The Astrophysical Journal* 800.2 (2015), p. 140. DOI: 10.1088/0004-637x/800/2/140.

[10] C. Carrasco-Gonzalez, L. F. Rodriguez, G. Anglada, J. Marti, J. M. Torrelles, and M. Osorio. “A Magnetized Jet from a Massive Protostar”. In: *Science* 330.6008 (2010), pp. 1209–1212. DOI: 10.1126/science.1195589.

[11] J.-F. Donati, I. D. Howarth, M. M. Jardine, P. Petit, C. Catala, J. D. Landstreet, J.-C. Bouret, E. Alecian, J. R. Barnes, T. Forveille, F. Paletou, and N. Manset. “The surprising magnetic topology of Sco: fossil remnant or dynamo output?” In: *Monthly Notices of the Royal Astronomical Society* 370.2 (2006), pp. 629–644. DOI: 10.1111/j.1365-2966.2006.10558.x.

[12] D. Coffey, F. Bacciotti, J. Woitas, T.P. Ray, and J. EislÃűffel. “Rotation of Jets From T-Tauri Stars: New Clues From HST/STIS Observations”. In: *Astrophysics and Space Science* 292.1-4 (2004), pp. 553–558. DOI: 10.1023/b:astr.0000045062.71787.34.

[13] Avery E. Broderick and Abraham Loeb. “Signatures of relativistic helical motion in the rotation measures of active galactic nucleus jets”. In: *The Astrophysical Journal* 703.2 (2009), pp. L104–L108. DOI: 10.1088/0004-637x/703/2/l104.

[14] John Bally, Steve Heathcote, Bo Reipurth, Jon Morse, Patrick Hartigan, and Richard Schwartz. “Hubble Space Telescope Observations of Proper Motions in Herbig-Haro Objects 1 and 2”. In: *The Astronomical Journal* 123.5 (2002), pp. 2627–2657. DOI: 10.1086/339837.

[15] U. Shumlak and C. Hartman. “Sheared Flow Stabilization of the m=1 Kink Mode in Z Pinches”. In: *Phys. Rev. Lett.* 75.18 (1995), pp. 3285–3288. DOI: 10.1103/PhysRevLett.75.3285.

[16] Eric Sander Lavine and Setthivoine You. “The topology of canonical flux tubes in flared jet geometry”. In: *The Astrophysical Journal* 835.1 (2017), p. 89. DOI: 10.3847/1538-4357/835/1/89.

[17] E. Priest. *Magnetohydrodynamics of the Sun*. Cambridge. UK: Cambridge University Press, 2014. ISBN: 9780521854719.

[18] M. R. Brown, P. K. Browning, M. E. Dieckmann, I. Furno, and T. P. Intrator. "Microphysics of Cosmic Plasmas: Hierarchies of Plasma Instabilities from MHD to Kinetic". In: *Space Sci Rev* 178.2-4 (2013), pp. 357–383. DOI: 10.1007/s11214-013-0005-7.

[19] M. Yamada, R. Kulsrud, and H. Ji. "Magnetic reconnection". In: *Reviews of Modern Physics* 82.1 (2010), pp. 603–664. DOI: 10.1103/revmodphys.82.603.

[20] Ellen G. Zweibel and Masaaki Yamada. "Magnetic Reconnection in Astrophysical and Laboratory Plasmas". In: *Annual Review of Astronomy and Astrophysics* 47.1 (2009), pp. 291–332. DOI: 10.1146/annurev-astro-082708-101726.

[21] P. M. Bellan. "Why current-carrying magnetic flux tubes gobble up plasma and become thin as a result". In: *Phys. Plasmas* 10.5 (2003), pp. 1999–1999. DOI: 10.1063/1.1558275.

[22] S. You, G. Yun, and P. Bellan. "Dynamic and Stagnating Plasma Flow Leading to Magnetic-Flux-Tube Collimation". In: *Phys. Rev. Lett.* 95.4 (2005), pp. 2–5. DOI: 10.1103/PhysRevLett.95.045002.

[23] Auna L. Moser and Paul M. Bellan. "Magnetic reconnection from a multiscale instability cascade". In: *Nat.* 482.7385 (2012), pp. 379–381. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature10827.

[24] P. M Bellan. *Spheromaks A practical application of magnetohydrodynamic dynamos and plasma self-organization*. 1st ed. London, UK: Imperial College Press, 2000. ISBN: 1-86094-141-9.

[25] J. B. O'Bryan and C. R. Sovinec. "Simulated flux-rope evolution during non-inductive startup in Pegasus". In: *Plasma Phys. Controll. Fusion* 56.6 (2014), pp. 064005–064005. DOI: 10.1088/0741-3335/56/6/064005.

[26] S. You. “A field theory approach to the evolution of canonical helicity and energy”. In: *Physics of Plasmas* 23.7 (2016), p. 072108. DOI: 10.1063/1.4956465.

[27] ESA/Hubble & NASA Acknowledgement: Judy Schmidt. *The Twin Jet Nebula*. 2015.

[28] E Kawamori, Y Murata, K Umeda, D Hirota, T Ogawa, T Sumikawa, T Iwama, K Ishii, T Kado, T Itagaki, M Katsurai, A Balandin, and Y Ono. “Ion kinetic effect on bifurcated relaxation to a field-reversed configuration in TS-4 CT experiment”. In: *Nucl. Fusion* 45.8 (2005), pp. 843–848. DOI: 10.1088/0029-5515/45/8/010.

[29] Clayton E. Myers, Masaaki Yamada, Hantao Ji, Jongsoo Yoo, William Fox, Jonathan Jara-Almonte, Antonia Savcheva, and Edward E. DeLuca. “A dynamic magnetic tension force as the cause of failed solar eruptions”. In: *Nature* 528.7583 (2015), pp. 526–529. DOI: 10.1038/nature16188.

[30] J. F. Hansen, S. K. P. Tripathi, and P. M. Bellan. “Co- and counter-helicity interaction between two adjacent laboratory prominences”. In: *Physics of Plasmas* 11.6 (2004), pp. 3177–3185. DOI: 10.1063/1.1724831.

[31] Michael R. Brown. “Experimental evidence of rapid relaxation to large-scale structures in turbulent fluids: selective decay and maximal entropy”. In: *Journal of Plasma Physics* 57.1 (1997), pp. 203–229. DOI: 10.1017/S0022377896005211.

[32] T R Jarboe. “Review of spheromak research”. In: *Plasma Physics and Controlled Fusion* 36.6 (1994), pp. 945–990. DOI: 10.1088/0741-3335/36/6/002.

[33] J.S. Sarff, A.F. Almagri, J.K. Anderson, M. Borchardt, W. Cappechi, D. Carmody, K. Caspary, B.E. Chapman, D.J. Den Hartog, J. Duff, S. Eilerman, A. Falkowski, C.B. Forest, M. Galante, J.A. Goetz, D.J. Holly, J. Koliner, S. Kumar, J.D. Lee, D. Liu, K.J. McCollam, M. McGarry, V.V. Mirnov, L. Morton, S. Munaretto, M.D. Nornberg, P.D. Nonn, S.P. Oliva, E. Parke, M.J. Pueschel, J.A. Reusch, J. Sauppe, A. Seltzman, C.R. Sovinec, D. Stone, D. Theucks, M. Thomas, J. Triana, P.W. Terry, J. Waksman, G.C. Whelan, D.L. Brower, W.X. Ding, L. Lin, D.R. Demers, P. Fimognari, J. Titus, F. Auriemma, S.

Cappello, P. Franz, P. Innocente, R. Lorenzini, E. Martines, B. Momo, P. Piovesan, M. Puiatti, M. Spolaore, D. Terranova, P. Zanca, V.I. Davydenko, P. Deichuli, A.A. Ivanov, S. Polosatkin, N.V. Stupishin, D. Spong, D. Craig, H. Stephens, R.W. Harvey, M. Cianciosa, J.D. Hanson, B.N. Breizman, M. Li, and L.J. Zheng. "Overview of results from the MST reversed field pinch experiment". In: *Nuclear Fusion* 55.10 (2015), p. 104006. DOI: 10.1088/0029-5515/55/10/104006.

[34] H. Ji, S. C. Prager, and J. S. Sarff. "Conservation of Magnetic Helicity during Plasma Relaxation". In: *Phys. Rev. Lett.* 74.15 (1995), pp. 2945–2948. DOI: 10.1103/PhysRevLett.74.2945.

[35] Loren C Steinhauer, Hideaki Yamada, and Akio Ishida. "Two-fluid flowing equilibria of compact plasmas". In: *Physics of Plasmas* 8.9 (2001), pp. 4053–4061. DOI: 10.1063/1.1388034.

[36] Mayuko Tsuruda, Yasushi Ono, and Makoto Katsurai. "Comparative experiments of co- and counter-helicity merging in the compact toroid merging device TS-4". In: *Electrical Engineering in Japan* 145.4 (2003), pp. 1–11. DOI: 10.1002/eej.10207.

[37] Y. Ono, M. Yamada, T. Akao, T. Tajima, and R. Matsumoto. "Ion Acceleration and Direct Ion Heating in Three-Component Magnetic Reconnection". In: *Physical Review Letters* 76.18 (1996), pp. 3328–3331. DOI: 10.1103/physrevlett.76.3328.

[38] P. M. Bellan. "Collisionless reconnection using Alfvén wave radiation resistance". In: *Physics of Plasmas* 5.9 (1998), pp. 3081–3088. DOI: 10.1063/1.873034.

[39] S. You. "The transport of relative canonical helicity". In: *Phys. Plasmas* 19.9 (2012), p. 092107. DOI: 10.1063/1.4752215.

[40] U. Shumlak, B. A. Nelson, R. P. Golingo, S. L. Jackson, E. A. Crawford, and D. J. Den Hartog. "Sheared flow stabilization experiments in the ZaP flow Z pinch". In: *Phys. Plasmas* 10.5 (2003), pp. 1683–1683. DOI: 10.1063/1.1558294.

[41] Jens von der Linden and Setthivoine You. “Sausage instabilities on top of kinking lengthening current-carrying magnetic flux tubes”. In: *Physics of Plasmas* 24.5 (2017), p. 052105. DOI: 10.1063/1.4981231.

[42] Jens von der Linden and Setthivoine You. *Sausage Instabilities on Top of Kinking Lengthening Current-Carrying Magnetic Flux Tubes - Code used for analysis and to generate figures 1-4*. 2017. DOI: 10.5281/zenodo.230489.

[43] Jens von der Linden and Setthivoine You. *Sausage Instabilities on Top of Kinking Lengthening Current-Carrying Magnetic Flux Tubes - Data underlying figures 3 & 4*. 2017. DOI: 10.5281/zenodo.230611.

[44] Margarita Ryutova. *Physics of Magnetic Flux Tubes*. Berlin Heidelberg, Germany: Springer, 2015. DOI: 10.1007/978-3-662-45243-1.

[45] J. Freidberg. *Ideal MHD*. Cambridge, UK: Cambridge University Press, 2014.

[46] M. Kruskal and J. L. Tuck. “The Instability of a Pinched Fluid with a Longitudinal Magnetic Field”. In: *Proc. R. Soc. A* 245.1241 (1958), pp. 222–237. DOI: 10.1098/rspa.1958.0079.

[47] William A Newcomb. “Hydromagnetic stability of a diffuse linear pinch”. In: *Ann. Phys.* 10.2 (1960), pp. 232–267. ISSN: 0003-4916. DOI: 10.1016/0003-4916(60)90023-3.

[48] I. B. Bernstein, E. A. Frieman, M. D. Kruskal, and R. M. Kulsrud. “An Energy Principle for Hydromagnetic Stability Problems”. In: *Proc. R. Soc. A* 244.1236 (1958), pp. 17–40. DOI: 10.1098/rspa.1958.0023.

[49] G. Benford. “The electrodynamic ’snake’ at the Galactic Centre”. In: *Mon. Not. R. Astron. Soc.* 285.3 (1997), pp. 573–579. DOI: 10.1093/mnras/285.3.573.

[50] S. Hsu and P. Bellan. “Experimental Identification of the Kink Instability as a Poloidal Flux Amplification Mechanism for Coaxial Gun Spheromak Formation”. In: *Phys. Rev. Lett.* 90.21 (2003), pp. 1–4. DOI: 10.1103/PhysRevLett.90.215002.

[51] B. I. Cohen, C. A. Romero-TalamÃąs, D. D. Ryutov, E. B. Hooper, L. L. LoDestro, H. S. McLean, T. L. Stewart, and R. D. Wood. “The role of the n_ϕ=1 column mode in spheromak formation”. In: *Phys. Plasmas* 16.4 (2009), pp. 042501–042501. DOI: 10.1063/1.3097909.

[52] Philippa Browning and Alex Lazarian. “Notes on Magnetohydrodynamics of Magnetic Reconnection in Turbulent Media”. In: *Space Sci. Rev.* 178.2-4 (2013), pp. 325–355. ISSN: 0038-6308, 1572-9672. DOI: 10.1007/s11214-013-0022-6.

[53] G. S. Yun, H. K. Park, W. Lee, M. J. Choi, G. H. Choe, S. Park, Y. S. Bae, K. D. Lee, S. W. Yoon, Y. M. Jeon, C. W. Domier, N. C. Luhmann, B. Tobias, A. J. H. DonnÃĺ, and KSTAR Team. “Appearance and Dynamics of Helical Flux Tubes under Electron Cyclotron Resonance Heating in the Core of KSTAR Plasmas”. In: *Phys. Rev. Lett.* 109.14 (2012), p. 145003. DOI: 10.1103/PhysRevLett.109.145003.

[54] D. D. Ryutov, R. H. Cohen, and L. D. Pearlstein. “Stability of a finite-length screw pinch revisited”. In: *Phys. Plasmas* 11.10 (2004), pp. 4740–4740. DOI: 10.1063/1.1781624.

[55] I. Furno, T. P. Intrator, G. Lapenta, L. Dorf, S. Abbate, and D. D. Ryutov. “Effects of boundary conditions and flow on the kink instability in a cylindrical plasma column”. In: *Phys. Plasmas* 14.2 (2007), pp. 022103–022103. DOI: 10.1063/1.2435306.

[56] Bergen Suydam. “Stability of a Linear Pinch”. In: *Proc. Second U. N. Internatl. Conf. Peac. Use Atomic Energy*. Geneva, 1958.

[57] Paul M Bellan. *Fundamentals of Plasma Physics*. Cambridge, UK: Cambridge University Press, 2008. ISBN: 978-0-521-52800-9.

[58] R J Tayler. “The Influence of an Axial Magnetic Field on the Stability of a Constricted Gas Discharge”. In: *Proceedings of the Physical Society. Section B* 70.11 (1957), pp. 1049–1063. DOI: 10.1088/0370-1301/70/11/305.

[59] Travis E. Oliphant. “Python for Scientific Computing”. In: *Comput. Sci. Eng.* 9.3 (2007), pp. 10–20. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.58.

[60] J. P. Goedbloed and S Poedts. *Principles of Magnetohydrodynamics: With Applications to Laboratory and Astrophysical Plasmas*. Cambridge, UK: Cambridge University Press, 2004.

[61] A. K. Srivastava, R. ErdÃľlyi, Durgesh Tripathi, V. Fedun, N. C. Joshi, and P. Kayshap. "Observational Evidence of Sausage-Pinch Instability in Solar Corona By SDO/AIA". In: *Astrophys. J.* 765.2 (2013), pp. L42–L42. DOI: 10.1088/2041-8205/765/2/L42.

[62] L. M. Zelenyi, H. V. Malova, A. V. Artemyev, V. Yu Popov, and A. A. Petrukovich. "Thin current sheets in collisionless plasma: Equilibrium structure, plasma instabilities, and particle acceleration". In: *Plasma Phys. Rep.* 37.2 (2011), pp. 118–160. ISSN: 1063-780X. DOI: 10.1134/S1063780X1102005X.

[63] M. R. Weis, P. Zhang, Y. Y. Lau, P. F. Schmit, K. J. Peterson, M. Hess, and R. M. Gilgenbach. "Coupling of sausage, kink, and magneto-Rayleigh-Taylor instabilities in a cylindrical liner". In: *Phys. Plasmas* 22.3 (2015), p. 032706. ISSN: 1070-664X, 1089-7674. DOI: 10.1063/1.4915520.

[64] Oscar A. Anderson, William R. Baker, Stirling A. Colgate, John Ise, and Robert V. Pyle. "Neutron Production in Linear Deuterium Pinches". In: *Phys. Rev.* 110.6 (1958), pp. 1375–1387. DOI: 10.1103/physrev.110.1375.

[65] J. Sears, Y. Feng, T. P. Intrator, T. E. Weber, and H. O. Swan. "Flux rope dynamics in three dimensions". In: *Plasma Phys. Control. Fusion* 56.9 (2014), p. 095022. ISSN: 0741-3335. DOI: 10.1088/0741-3335/56/9/095022.

[66] B. Dattner, B. Lehnert, and S. Lundquist. "Liquid Conductor Model of Instabilities in a Pinched Discharge". In: *Proc. Second U. N. Internatl. Conf. Peac. Use Atomic Energy*. Geneva, 1958.

[67] W. Bergerson, C. Forest, G. Fiksel, D. Hannum, R. Kendrick, J. Sarff, and S. Stambler. "Onset and Saturation of the Kink Instability in a Current-Carrying Line-Tied Plasma".

In: *Phys. Rev. Lett.* 96.1 (2006), pp. 015004–015004. DOI: 10.1103/PhysRevLett.96.015004.

[68] A. Schmidt, V. Tang, and D. Welch. "Fully Kinetic Simulations of Dense Plasma Focus Z-Pinch Devices". In: *Phys. Rev. Lett.* 109.20 (2012), p. 205003. DOI: 10.1103/physrevlett.109.205003.

[69] Gherardo Valori, Etienne Pariat, Sergey Anfinogentov, Feng Chen, Manolis K. Georgoulis, Yang Guo, Yang Liu, Kostas Moraitis, Julia K. Thalmann, and Shangbin Yang. "Magnetic Helicity Estimations in Models and Observations of the Solar Magnetic Field. Part I: Finite Volume Methods". In: *Space Science Reviews* 201.1-4 (2016), pp. 147–200. DOI: 10.1007/s11214-016-0299-3.

[70] Y. Guo, M. D. Ding, X. Cheng, J. Zhao, and E. Pariat. "Twist accumulation and topology structure of a solar magnetic flux rope". In: *The Astrophysical Journal* 779.2 (2013), p. 157. DOI: 10.1088/0004-637x/779/2/157.

[71] E. Pariat, P. Démoulin, and M. A. Berger. "Photospheric flux density of magnetic helicity". In: *Astronomy and Astrophysics* 439.3 (2005), pp. 1191–1203. DOI: 10.1051/0004-6361:20052663.

[72] Jens von der Linden, Alexander Card, and Setthivoine You. *MochiControl: LabVIEW experiment control with MDSplus data storage*. 2017. DOI: 10.5281/zenodo.495643.

[73] Alexander Card. *MochiModel: MDSplus tree for MochiLab & Mochi.Labjet*. 2017. DOI: 10.5281/zenodo.495631.

[74] Jens von der Linden. *MochiFPGAcontrol: High-throughput FPGA code for NI-5752 digitizers*. 2017. DOI: 10.5281/zenodo.495787.

[75] Evan Carroll. "Driving flows in laboratory astrophysical plasma jets : The Mochi.LabJet experiment". MA thesis. University of Washington, 2016.

[76] Vernon H Chaplin and Paul M Bellan. “Fast Ignitron Trigger Circuit Using Insulated Gate Bipolar Transistors”. In: *IEEE Transactions on Plasma Science* 41.4 (2013), pp. 975–979. DOI: 10.1109/TPS.2013.2249113.

[77] J. A. Stillerman, T. W. Fredian, K.A. Klare, and G. Manduchi. “MDSplus data acquisition system”. In: *Review of Scientific Instruments* 68.1 (1997), pp. 939–942. DOI: 10.1063/1.1147719.

[78] G. Manduchi, E. De Marchi, and A. Mandelli. “A new LabVIEW interface for MDSplus”. In: *Fusion Engineering and Design* 88.6-8 (2013), pp. 1196–1199. DOI: 10.1016/j.fusengdes.2012.12.013.

[79] *8-Channel Variable-Gain Amplifier (VGA) With Octal High-Speed ADC*. AFE5801. Texas Instruments. 2010.

[80] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/mcse.2011.37.

[81] Samuel Harrold. *ccd-utexas/tsphot: Time series photometry using astropy*. https://github.com/ccd-utexas/tsphot. 2015.

[82] Henry W. Ott. *Electromagnetic Compatibility Engineering*. John Wiley & Sons, Inc., 2009. DOI: 10.1002/9780470508510.

[83] *NI LabVIEW High-Performance FPGA Developer’s Guide*. National Instruments. 2014.

[84] Jens von der Linden. *Investigating the dynamics of canonical flux tubes - Code*. 2017. DOI: 10.5281/zenodo.581197.

[85] I. Furno, T. Intrator, E. Torbert, C. Carey, M. D. Cash, J. K. Campbell, W. J. Fienup, C. A. Werley, G. A. Wurden, and G. Fiksel. “Reconnection scaling experiment: A new device for three-dimensional magnetic reconnection studies”. In: *Review of Scientific Instruments* 74.4 (2003), pp. 2324–2331. DOI: 10.1063/1.1544051.

[86] T. Intrator, X. Sun, L. Dorf, I. Furno, and G. Lapenta. “A three dimensional probe positionera)”. In: *Review of Scientific Instruments* 79.10 (2008), 10F129. ISSN: 0034-6748, 1089-7623. DOI: 10.1063/1.2956746.

[87] C. A. Romero-Talamás, P. M. Bellan, and S. C. Hsu. “Multielement magnetic probe using commercial chip inductors”. In: *Review of Scientific Instruments* 75.8 (2004), pp. 2664–2667. DOI: 10.1063/1.1771483.

[88] Auna L Moser. “Dynamics of magnetically driven plasma jets: An instability of an instability, gas cloud impacts, shocks, and other deformations”. PhD thesis. California Institute of Technology, 2012.

[89] I. H. Hutchinson. *Principles of Plasma Diagnostics*. Cambridge University Press, 11, 2005. ISBN: 052167574X.

[90] Kyu-Sun Chung. “Why Is the Mach Probe Formula Expressed as $R = J_{up}/J_{dn} = exp[KM_{\infty}]$?” In: *Japanese Journal of Applied Physics* 45.10A (2006), pp. 7914–7916. ISSN: 0021-4922, 1347-4065. DOI: 10.1143/JJAP.45.7914.

[91] X. Zhang, D. Dandurand, T. Gray, M. R. Brown, and V. S. Lukin. “Calibrated cylindrical Mach probe in a plasma wind tunnel”. In: *Review of Scientific Instruments* 82.3 (2011), p. 033510. ISSN: 0034-6748, 1089-7623. DOI: 10.1063/1.3559550.

[92] Eric Eugene Lawrence. “Colliding Magnetic Flux Ropes and Quasi-Separatrix Layers in a Laboratory Plasma”. PhD thesis. University of California, Los Angeles, 2010.

[93] Kyu-Sun Chung. “Mach probes”. In: *Plasma Sources Sci. Technol.* 21.6 (2012), p. 063001. ISSN: 0963-0252. DOI: 10.1088/0963-0252/21/6/063001.

[94] I. Furno, C. Theiler, A. Fasoli, and B. Labit. “Three-Dimensional Imaging of a Radially Propagating Plasma Blob Using Conditional Sampling Techniques”. In: *IEEE Transactions on Plasma Science* 39.11 (2011), pp. 3018–3019. ISSN: 0093-3813. DOI: 10.1109/TPS.2011.2162254.

[95] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–.

[96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The quickhull algorithm for convex hulls". In: *ACM Transactions on Mathematical Software* 22.4 (1996), pp. 469–483. DOI: 10.1145/235815.235821.

[97] Karl E. Taylor. "Summarizing multiple aspects of model performance in a single diagram". In: *Journal of Geophysical Research: Atmospheres* 106.D7 (2001), pp. 7183–7192. DOI: 10.1029/2000jd900719.

[98] Zhongfeng Xu, Zhaolu Hou, Ying Han, and Weidong Guo. "A diagram for evaluating multiple aspects of model performance in simulating vector fields". In: *Geoscientific Model Development* 9.12 (2016), pp. 4365–4380. DOI: 10.5194/gmd-9-4365-2016.

[99] William H. Press. *Numerical Recepies*. Cambridge University Press, 2007. ISBN: 0521880688.

[100] Roger W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis Ltd, 1, 1988. 540 pp. ISBN: 0852743920.

[101] V. Fuka. "PoisFFT – A free parallel fast Poisson solver". In: *Applied Mathematics and Computation* 267 (2015), pp. 356–364. DOI: 10.1016/j.amc.2015.03.011.

[102] J. Makhoul. "A fast cosine transform in one and two dimensions". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28.1 (1980), pp. 27–34. DOI: 10.1109/tassp.1980.1163351.

[103] C. Richard DeVore. "Magnetic Helicity Generation by Solar Differential Rotation". In: *The Astrophysical Journal* 539.2 (2000), pp. 944–953. DOI: 10.1086/309274.

[104] M. A. Chilenski, M. Greenwald, Y. Marzouk, N. T. Howard, A. E. White, J. E. Rice, and J. R. Walk. "Improved profile fitting and quantification of uncertainty in experimental measurements of impurity transport coefficients using Gaussian process regression". In: *Nucl. Fusion* 55.2 (2015), p. 023012. ISSN: 0029-5515. DOI: 10.1088/0029-5515/55/2/023012.

[105] A. L. Restante, S. Markidis, G. Lapenta, and T. Intrator. “Geometrical investigation of the kinetic evolution of the magnetic field in a periodic flux rope”. In: *Physics of Plasmas* 20.8 (2013), p. 082501. DOI: 10.1063/1.4817167.

[106] D.E. Smith, E.J. Powers, and G.S. Caldwell. “Fast-Fourier-Transform Spectral-Analysis Techniques as a Plasma Fluctuation Diagnostic Tool”. In: *IEEE Transactions on Plasma Science* 2.4 (1974), pp. 261–272. ISSN: 0093-3813. DOI: 10.1109/TPS.1974.4316849.

[107] S. K. P. Tripathi, B. Van Compernolle, W. Gekelman, P. Pribyl, and W. Heidbrink. “Excitation of shear Alfvén waves by a spiraling ion beam in a large magnetoplasma”. In: *Physical Review E* 91.1 (2015). DOI: 10.1103/physreve.91.013109.

[108] E. Pariat, G. Valori, P. Démoulin, and K. Dalmasse. “Testing magnetic helicity conservation in a solar-like active event”. In: *Astronomy & Astrophysics* 580 (2015), A128. DOI: 10.1051/0004-6361/201525811.

[109] T. Dehaas. “Helicity Transformation under the Collision and Merging of Magnetic Flux Ropes”. In: *APS Meeting Abstracts*. 2016. DOI: 10.1103/BAPS.2016.DPP.DI2.1.

[110] K. H. Burrell. “Effects of $E \times B$ velocity shear and magnetic shear on turbulence and transport in magnetic confinement devices”. In: *Physics of Plasmas (1994-present)* 4.5 (1997), pp. 1499–1518. ISSN: 1070-664X, 1089-7674. DOI: 10.1063/1.872367.

[111] R. Raman, T. R. Jarboe, B. A. Nelson, V. A. Izzo, R. G. O’Neill, A. J. Redd, and R. J. Smith. “Demonstration of Plasma Startup by Coaxial Helicity Injection”. In: *Physical Review Letters* 90.7 (2003). DOI: 10.1103/physrevlett.90.075005.

[112] B. S. Victor, T. R. Jarboe, A. C. Hossack, D. a. Ennis, B. A. Nelson, R. J. Smith, C. Akcay, C. J. Hansen, G. J. Marklin, N. K. Hicks, and J. S. Wrobel. “Evidence for Separatrix Formation and Sustainment with Steady Inductive Helicity Injection”. In: *Physical Review Letters* 107.16 (2011), pp. 165005–165005. DOI: 10.1103/PhysRevLett.107.165005.

[113] H. Ji, A. Bhattacharjee, S. Prager, W. Daughton, S. Bale, T. Carter, N. Crocker, J. Drake, J. Egedal, J. Sarff, J. Wallace, Y. Chen, R. Cutler, W. Fox, P. Heitzenroeder, M. Kalish, J. Jara-Almonte, C. Myers, Y. Ren, M. Yamada, and J. Yoo. "Status and Plans for the FLARE (Facility for Laboratory Reconnection Experiments) Project". In: *APS Meeting Abstracts*. 2015.

[114] W. Gekelman, E. Lawrence, A. Collette, S. Vincena, B. Van Compernolle, P. Pribyl, M. Berger, and J. Campbell. "Magnetic field line reconnection in the current systems of flux ropes and Alfvén waves". In: *Phys. Scr.* 2010.T142 (2010), p. 014032. ISSN: 1402-4896. DOI: 10.1088/0031-8949/2010/T142/014032.

[115] T. Dehaas, W. Gekelman, and B. van Compernolle. "Characteristic Dynamics of a Non-Linear Flux Rope". In: *APS Meeting Abstracts*. 2015.

[116] J. P. Sauppe and C. R. Sovinec. "Two-fluid and finite Larmor radius effects on helicity evolution in a plasma pinch". In: *Physics of Plasmas* 23.3 (2016), p. 032303. DOI: 10.1063/1.4942761.

[117] I. V. Khalzov, F. Ebrahimi, D. D. Schnack, and V. V. Mirnov. "Minimum energy states of the cylindrical plasma pinch in single-fluid and Hall magnetohydrodynamics". In: *Physics of Plasmas* 19.1 (2012), pp. 012111–012111. DOI: 10.1063/1.3676600.

Appendix A

FLUXTUBE STABILITY CODE

This appendix contains the source code used in the analysis and to generate figs. 3.3, 3.4, 3.5, and 3.6 and instructions for setting up and running the code. The source code can also be found at Zenodo
https://doi.org/10.5281/zenodo.230489 [42].

A.1 Dependencies

- python 2.7.12
- numpy 1.11.2
- numba 0.30.0
- scipy 0.18.1
- matplotlib 1.5.3
- seaborn 0.7.1
- sqlite 3.13
- gitpython 2.1.0

The dependencies can be installed with the anaconda python distribution. `gitpython` can be installed with pip.

A.2 Setup

Create three directories `figures`, `output`, `source`. Git clone the repository or copy the files into `source`.

A.3 Important files

recreate_paper_data.sh recreates the data underlying figs. 3.5 and 3.6.

skin_core_scanner.py is a script file that runs a stability scan over $\bar{k}$ - $\bar{\lambda}$ space for an equilibrium defined by a core skin current profile of a given core current to total current ratio ϵ. This file has a command line interface. Help can be accessed with `python skin_core_scanner.py -help`. `newcomb.py` determines the external stability of a single equilibrium (single pair of $\bar{k}$ and $\bar{\lambda}$).

paper_figures.py recreates the figs. 3.3, 3.4, 3.5, and 3.6. This file has a command line interface. Help can be accessed with `python paper_figures.py -help`.

A.4 Run

Run the bash script `recreate_paper_data.sh`. This will create a SQL database in output called `output.db` that keeps track of all runs and input parameters of `skin_core_scanner.py` and run `skin_core_scanner.py` three times to do a stability scan for three values of ϵ. The output data is stored in dated directories in `output`. `output.db` can be opened, e.g., with the Firefox extension SQLite Manager and the run parameters can be inspected.

Generate paper figures with `python paper_figures ../output/[timestamp] ../output/[timestamp] ../output/[timestamp]`, where [timestamp] is the dated directory name. The order should be from oldest to newest date.

A.5 newcomb.py description

Examines the equilibrium. If the equilibrium has a singularity, the Frobenius method is used to determine a small solution at an radius r greater than the radius of the singularity. If the singularity is Suydam unstable no attempt is made to calculate external stability. If there is no Suydam instability a Frobenius power series solution close to $r = 0$ is chosen or if the integration does not start at $r = 0$ a given ξ is used as boundary condition. Only the last interval is integrated. ξ and ξ' are plugged into the potential energy equation to determine stability.

A.6 Source code

A.6.1 init_database.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 21 13:53:21 2016

@author: jensv

Create sql run database.
"""

import sqlite3
import os

assert os.path.exists('../output'), ("../output does not exist."
                                     "Please place repo next to"
                                     "'output' directory")
assert not os.path.exists('../output/output.db'), ("database "
                                                   "../output/output.db "
                                                   "already exists.")
connection = sqlite3.connect('../output/output.db')
cursor = connection.cursor()
cursor.execute("CREATE TABLE Runs(datetime TEXT, "
               "points_core INTEGER, "
               "points_transition INTEGER, "
               "points_skin INTEGER, "
               "core_radius REAL, "
               "transition_width REAL, "
               "skin_width REAL, k_bar_start REAL, "
               "k_bar_end REAL, k_bar_num INTEGER, "
               "lambda_bar_start REAL, "
               "lambda_bar_end REAL, "
               "lambda_bar_num INTEGER, epsilon REAL, "
               "git_commit TEXT, python_call TEXT)")
```

```
cursor.close()
connection.commit()
connection.close()
```

A.6.2 skin_core_scanner.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 15 22:05:21 2015

@author: jensv

Script to scan k_bar-lambda_bar stability space of profile type.
"""

import equil_solver as es
import newcomb as new
import numpy as np
from copy import deepcopy
from datetime import datetime
import os
import json
import sys
import call_provenance as cp
import sqlite3
import argparse


def scan_lambda_k_space(lambda_bar_space, k_bar_space,
                        xi_factor=1., magnetic_potential_energy_ratio=1.,
                        offset=1E-3, r_0=0., init_value=(0.0, 1.0),
                        rtol=None, max_step=1E-2,
                        nsteps=1000, method='lsoda', suppress_output=True,
                        diagnose=False, stiff=False, use_jac=True,
                        adapt_step_size=True, sing_search_points=1000,
                        profile_type='default', no_git=False,
                        **kwargs):
    r"""
    Scans space given by lambda_a_space and k_a_space for m=0, 1 stability and
    saves several 2D numpy arrays (stability maps) to an npz file.
    """
    lambda_a_space = lambda_bar_space
    k_a_space = k_bar_space
    call_parameters = locals()
    func_name = 'scan_lambda_k_space'

    suydam_end_offset = offset

    k_a_points = np.linspace(k_a_space[0], k_a_space[1], num=k_a_space[2])
    lambda_a_points = np.linspace(lambda_a_space[0], lambda_a_space[1],
                                  num=lambda_a_space[2])

    lambda_a_mesh, k_a_mesh = np.meshgrid(lambda_a_points, k_a_points)

```

```
49      mesh_shape = lambda_a_mesh.shape
50
51      delta_map = {-1: np.zeros(mesh_shape),
52                   0: np.zeros(mesh_shape)}
53
54      sub_stability_maps = {-1: np.ones(mesh_shape),
55                            0: np.ones(mesh_shape)}
56
57      stability_maps = {'external': deepcopy(sub_stability_maps),
58                        'd_w': deepcopy(sub_stability_maps),
59                        'd_w_raw': deepcopy(sub_stability_maps),
60                        'suydam': deepcopy(sub_stability_maps)}
61
62      for i, lambda_a in enumerate(lambda_a_points):
63          print('lambda_bar = %.3f' % lambda_a)
64          for j, k_a in enumerate(k_a_points):
65              for m in [-1, 0]:
66                  results = None
67                  if profile_type == 'default':
68                      profile = es.UnitlessSmoothedCoreSkin(k_bar=k_a,
69                                                            lambda_bar=lambda_a,
70                                                            **kwargs)
71
72
73                      params = {'k': k_a, 'm': float(m), 'r_0': r_0, 'a': 1.,
74                                'b': 'infinity'}
75                      params_wo_splines = deepcopy(params)
76                      params.update(profile.get_tck_splines())
77                      params.update({'xi_factor': xi_factor,
78                                     'magnetic_potential_energy_ratio': magnetic_potential_energy_ratio,
79                                     'beta_0': profile.beta_0(),
80                                     'core_radius': profile.core_radius,
81                                     'transition_width': profile.transition_width,
82                                     'skin_width': profile.skin_width,
83                                     'points_core': profile.points_core,
84                                     'points_transition': profile.points_transition,
85                                     'points_skin': profile.points_skin,
86                                     'epsilon': profile.epsilon})
87
88                  elif profile_type == 'diffuse_core_skin':
89                      profile = es.UnitlessExponentialDecaySkin(k_bar=k_a,
90                                                                lambda_bar=lambda_a,
91                                                                **kwargs)
92                      params = {'k': k_a, 'm': float(m), 'r_0': r_0, 'a': 1.,
93                                'b': 'infinity'}
94                      params_wo_splines = deepcopy(params)
95                      params.update(profile.get_tck_splines())
96                      params.update({'xi_factor': xi_factor,
97                                     'magnetic_potential_energy_ratio': magnetic_potential_energy_ratio,
98                                     'beta_0': profile.beta_0(),
99                                     'core_radius': profile.core_radius,
100                                    'transition_width': None,
101                                    'skin_width': profile.skin_width,
102                                    'points_core': profile.points_core,
103                                    'points_transition': None,
104                                    'points_skin': profile.points_skin,
105                                    'epsilon': profile.epsilon})
```

```
106                 results = new.stability(params, offset, suydam_end_offset,
107                                         sing_search_points=sing_search_points,
108                                         suppress_output=suppress_output,
109                                         xi_given=init_value,
110                                         rtol=rtol, max_step=max_step,
111                                         nsteps=nsteps, method=method,
112                                         diagnose=diagnose, stiff=stiff,
113                                         use_jac=use_jac,
114                                         adapt_step_size=adapt_step_size)
115                 stable_external = results[0]
116                 stable_suydam = results[1]
117                 delta_w = results[2]
118                 xi = results[4]
119                 xi_der = results[5]
120
121                 delta = xi_der[-1] / xi[-1]
122
123                 delta_map[m][j][i] = delta
124
125                 if delta_w is not None:
126                     stability_maps['d_w'][m][j][i] = delta_w
127                     stability_maps['d_w_raw'][m][j][i] = delta_w
128                 else:
129                     stability_maps['d_w'][m][j][i] = np.nan
130                     stability_maps['d_w_raw'][m][j][i] = np.nan
131
132                 if not stable_external:
133                     stability_maps['external'][m][j][i] = 0.
134                 if stable_external and delta_w is None:
135                     stability_maps['external'][m][j][i] = -1.
136                 if not stable_suydam:
137                     stability_maps['suydam'][m][j][i] = 0.
138
139     #normalize
140     for m in [-1, 0]:
141         stability_maps['d_w'][m] = (stability_maps['d_w'][m] /
142                                     np.nanmax(np.abs(stability_maps['d_w'][m])))
143
144     params_wo_splines.update({'lambda_a_space': lambda_a_space,
145                               'k_a_space': k_a_space,
146                               'sing_search_points': sing_search_points,
147                               'offset': offset,
148                               'suydam_end_offset': suydam_end_offset})
149     params_wo_splines.update(kwargs)
150
151     date = datetime.now().strftime('%Y-%m-%d-%H-%M')
152     if os.getcwd().endswith('ipython_notebooks'):
153         path = '../../output/' + date
154         sql_db = '../../output/output.db'
155     else:
156         path = '../output/' + date
157         sql_db = '../output/output.db'
158     os.mkdir(path)
159
160     track_provenance(sql_db, func_name, call_parameters, date, params,
161                      lambda_a_space, k_a_space, no_git)
162
```

```
163     params_wo_splines.pop('dr', None)
164     with open(path+'/params.txt', 'w') as params_file:
165         json.dump(params_wo_splines, params_file)
166     np.savez(path+'/meshes.npz', lambda_a_mesh=lambda_a_mesh,
167              k_a_mesh=k_a_mesh,
168              external_m_neg_1=stability_maps['external'][-1],
169              external_m_0=stability_maps['external'][0],
170              d_w_m_neg_1=stability_maps['d_w'][-1],
171              d_w_m_0=stability_maps['d_w'][0],
172              d_w_raw_m_neg_1=stability_maps['d_w_raw'][-1],
173              d_w_raw_m_0=stability_maps['d_w_raw'][0],
174              suydam_m_0=stability_maps['suydam'][0],
175              suydam_m_neg_1=stability_maps['suydam'][-1],
176              delta_m_0=delta_map[0],
177              delta_m_neg_1=delta_map[-1]
178              )
179
180     print('Saved in Directory:' + str(date))
181
182     return lambda_a_mesh, k_a_mesh, stability_maps
183
184 def track_provenance(sql_db, func_name, call_parameters, date, params,
185                      lambda_a_space, k_a_space, no_git):
186     r"""
187     Save parameters, call, and git commit to make results reproducible and allow
188     easy scanning.
189     """
190     call = func_name + '('
191     for key in call_parameters.keys():
192         call += key + '=' + str(call_parameters[key]) + ', '
193     call = call[:-2] + ')'
194     if  no_git:
195         git_commit = None
196     else:
197         call, git_commit = cp.call_and_git_commit(call=call, call_path=os.getcwd())
198     assert os.path.exists(sql_db), ("Run database does not exist."
199                                     "Try running init_database.py")
200     connection = sqlite3.connect(sql_db)
201     cursor = connection.cursor()
202     cursor.execute("INSERT INTO Runs(datetime, points_core, points_transition"+
203                    ", points_skin, core_radius, transition_width, skin_width,"+
204                    "k_bar_start, k_bar_end, k_bar_num, lambda_bar_start," +
205                    "lambda_bar_end, lambda_bar_num, epsilon, git_commit," +
206                    "python_call) " +
207                    "VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?"+
208                    ")", (
209                          date,
210                          params['points_core'],
211                          params['points_transition'],
212                          params['points_skin'],
213                          params['core_radius'],
214                          params['transition_width'],
215                          params['skin_width'],
216                          k_a_space[0],
217                          k_a_space[1],
218                          int(k_a_space[2]),
219                          lambda_a_space[0],
```

```
220                             lambda_a_space[1],
221                             int(lambda_a_space[2]),
222                             params['epsilon'],
223                             git_commit,
224                             call))
225         connection.commit()
226         cursor.close()
227         connection.close()
228
229     def parse_args():
230         parser = argparse.ArgumentParser(description="scan specified"
231                                          "k_bar-lambda_bar space for specified"
232                                          "profiles")
233         parser.add_argument('--no_git', help='Do not record git commit',
234                             action='store_true', default=False)
235         parser.add_argument('--k_bar_space', help='specify k_bar space',
236                             nargs=3, type=float, metavar=('START', 'END', 'POINTS'),
237                             default=[0.01, 1.5, 50])
238         parser.add_argument('--lambda_bar_space',
239                             help="specify lambda_bar space:",
240                             metavar=('START', 'END', 'POINTS'),
241                             nargs=3, type=float, default=[0.01, 5.0, 50])
242         parser.add_argument('--epsilon', help='core current to total current ratio',
243                             type=float, default=0.5)
244         parser.add_argument('--core_radius_norm', help='normalized core radius',
245                             type=float, default=0.6)
246         parser.add_argument('--transition_width_norm',
247                             help='normalized transition width',
248                             type=float, default=0.175)
249         parser.add_argument('--skin_width_norm', help='normalized skin width',
250                             type=float, default=0.05)
251         parser.add_argument('--points_core', help="number of points from which"
252                                                   "to build core region splines",
253                             type=int, default=50)
254         parser.add_argument('--points_transition',
255                             help="number of points from which to build"
256                             "transition splines", type=int, default=50)
257         parser.add_argument('--points_skin',
258                             help="number of points from which to build"
259                             "skin splines", type=int, default=50)
260         parser.add_argument('--offset', help="offset after singularities at which"
261                                              "to continue integrating from"
262                                              "power series solutions",
263                             type=float, default=1e-3)
264         parser.add_argument('--rtol', help="relative tolerance setting for"
265                                            "integrator",
266                             type=float, default=None)
267         parser.add_argument('--max_step', help="max step for integrator",
268                             type=float, default=1e-2)
269         parser.add_argument('--nsteps', help='number of steps for integrator',
270                             type=int, default=1000)
271         parser.add_argument('--method', help="integration method"
272                             "from scipy.integrate.ode", type=str, default='lsoda')
273         parser.add_argument('--suppress_output', help="flag to suppress output",
274                             default=False, action='store_true')
275         parser.add_argument('--diagnose', help="flag to make code try to integrate"
276                                                "over whole fluxtube radius.",
```

```
                        default=False, action='store_true')
    parser.add_argument('--stiff', help="flag to pass stiff flag to integrator",
                        default=False, action='store_true')
    parser.add_argument('--use_jac', help="flag to use Jacobian"
                                          "with integrator",
                        action="store_true")
    parser.add_argument('--adapt_step_size',
                        help="flag to adapt the stepsize requirements"
                        "for the integrator to the fluxtube region",
                        action="store_true")
    parser.add_argument('--sing_search_points',
                        help="number of points to divide fluxtube radius by in"
                             "search for singularities",
                        type=int, default=1000)

    args = parser.parse_args()
    args.k_bar_space[2] = int(args.k_bar_space[2])
    args.lambda_bar_space[2] = int(args.lambda_bar_space[2])
    return args

def main(args):
    r"""
    Run skin_core_scanner from command line call.
    """
    scan_lambda_k_space(**vars(args))

if __name__ == "__main__":
    args = parse_args()
    main(args)
```

A.6.3 paper_figures.py

```
import numpy as np
from scipy.special import kv, kvp

import matplotlib.pyplot as plt
from matplotlib import colors
import seaborn as sns
sns.set_style('ticks')
sns.set_context('paper')

import argparse
import os
import lambda_k_plotting as plot
reload(plot)
import analytic_condition as ac
from analytic_condition import conditions
import equil_solver as es


"""parser = argparse.ArgumentParser(description='make paper plots')
parser.add_argument('high_epsilon_path',
                    help="path of high epsilon profile stability results")
parser.add_argument('mid_epsilon_path',
```

```
                    help="path of mid epsilon profile stability results")
parser.add_argument('low_epsilon_path',
                    help="path of low epsilon profile stability results")
args = parser.parse_args()
high_epsilon_path = os.path.join(args.high_epsilon_path,
                                 'meshes.npz')
mid_epsilon_path = os.path.join(args.mid_epsilon_path,
                                'meshes.npz')
low_epsilon_path = os.path.join(args.low_epsilon_path,
                                'meshes.npz')
"""

## Figure 1 ##
##############

fig, axes = plt.subplots(2, 2,
                         figsize=(6.69, 6.69),
                         sharex=False,
                         sharey=False)

example_ax = axes[0][0]
kink_ax = axes[0][1]
sausage_ax = axes[1][0]
delta_ax = axes[1][1]

## Example Plot ##
##################

lambda_bar = np.linspace(0, 4.25, 500)
k_bar = np.linspace(0, 1.5, 500)
mesh = np.meshgrid(lambda_bar, k_bar)
lambda_bar_mesh, k_bar_mesh = mesh[0], mesh[1]


d_w_sausage = conditions(k_bar_mesh,
                         lambda_bar_mesh,
                         epsilon=0.1,
                         m=0,
                         delta=0.1)
d_w_kink = conditions(k_bar_mesh,
                      lambda_bar_mesh,
                      epsilon=0.1,
                      m=1,
                      delta=0.1)

stability_sausage = d_w_sausage < 0
stability_kink = d_w_kink < 0
stability = stability_kink.astype(float)
stability[stability_sausage] = 2


cmap = colors.ListedColormap(['white',
                              'lightgrey',
                              'lightgrey'])

example_ax.contourf(lambda_bar_mesh, k_bar_mesh, stability, cmap=cmap,
                    levels=[0., 0.5, 1.5, 2.], hatches=[None, '/', 'x'])
```

```
contour_lines = example_ax.contour(lambda_bar_mesh, k_bar_mesh, stability,
                                   levels=[0., 1.5, 2.], colors='black', linewidths=2)

example_ax.clabel(contour_lines,
                  manual=([1.5, 1.1],
                          [2.5, 0.75]),
                  fmt={0.: ' Kink Boundary ', 1.5: ' Sausage Boundary '}, fontsize=10)

kruskal_shafranov = lambda_bar_mesh > 2 * k_bar_mesh

contour_lines = example_ax.contour(lambda_bar_mesh, k_bar_mesh, kruskal_shafranov,
                                   levels=[0.5], colors='black', linewidths=2)
contour_lines.collections[0].set_linestyle('—')
example_ax.clabel(contour_lines,
                  manual=([[1.8, 0.95]]),
                  fmt={0.5: 'Kruskal—Shafranov'}, fontsize=10)

#example_ax.plot([0, 3.], [0., 1.5], '--', c='black', lw=5)

example_ax.set_xlabel(r'$\bar{\lambda}$')
#example_ax.xaxis.labelpad = 20
plt.setp(example_ax.get_xticklabels())
example_ax.set_xticks(np.arange(0., 5, 1.))

example_ax.set_ylabel(r'$\bar{k}$', rotation='horizontal')
plt.setp(example_ax.get_yticklabels())
example_ax.set_yticks(np.arange(0., 2.0, 0.5))

example_ax.plot(2*np.sqrt(2), 0.0, 'o', lw=2, color='black')
example_ax.annotate('Tayler', color='black',
                    xy=(2*np.sqrt(2), 0),
                    xytext=(2*np.sqrt(2) + 0.025, 0.25),
                    rotation=45, fontsize=10)

sns.despine()

"""
## Kink Plot ##
###############

delta = 0.0
stability = np.zeros(lambda_bar_mesh.shape) - 0.2
for i, epsilon in enumerate(np.arange(0.1, 1.9, 0.3)):

    d_w_kink = conditions(k_bar_mesh,
                          lambda_bar_mesh,
                          epsilon=epsilon,
                          m=1,
                          delta=delta)

    stability_epsilon = d_w_kink < 0
    stability[stability_epsilon] = epsilon

levels = np.arange(0.1, 1.9, 0.3)
contour = kink_ax.contourf(lambda_bar_mesh,
                           k_bar_mesh,
                           stability,
```

```
                                levels=levels)
contour.cmap.set_over(sns.xkcd_rgb['red brown'])
contour.cmap.set_under('white')
contour.set_clim(-0.1, 0.)
colors_list = ['grey', 'white', 'white', 'white', 'white']
contour_lines = kink_ax.contour(lambda_bar_mesh,
                                k_bar_mesh,
                                stability,
                                levels=levels,
                                colors=colors_list,
                                linewidths=2)
kink_ax.clabel(contour_lines,
                               manual=([1.6, 1.3],
                                       [2.1, 1.2],
                                       [2.3, 1.1],
                                       [2.4, 0.8],
                                       [2.4, 0.5]),
                                fmt=r'$ \epsilon = %.1f $', colors=colors_list,
                                fontsize=12)
for line in contour_lines.collections:
    line.set_linestyle('solid')

line_x = np.linspace(0, 3, 50)
line_y = np.linspace(0, 1.5, 50)

line_y_masked = np.ma.masked_inside(line_y, 1.05, 1.18)

kink_ax.plot(line_x, line_y_masked, '--', c='black', lw=2)


kink_ax.plot([0, 3.], [0., 1.5], '--', c='black', lw=2)
kink_ax.set_xlabel(r'$\bar{\lambda}$')
#kink_ax.xaxis.labelpad = 20
plt.setp(kink_ax.get_xticklabels())
kink_ax.set_xticks(np.arange(0., 5, 1.))

kink_ax.set_ylabel(r'$\bar{k}$', rotation='horizontal')
plt.setp(kink_ax.get_yticklabels())
kink_ax.set_yticks(np.arange(0., 2.0, 0.5))

sns.despine()

## Sausage Plot ##
###################

delta = -0.7
stability = np.zeros(lambda_bar_mesh.shape) - 0.2
for i, epsilon in enumerate(np.arange(0, 1.2, 0.2)):

    d_w_sausage = conditions(k_bar_mesh,
                             lambda_bar_mesh,
                             epsilon=epsilon,
                             m=0,
                             delta=delta)

    stability_epsilon = d_w_sausage < 0
    stability[stability_epsilon] = epsilon
```

```
194
195 levels = np.arange(0., 1.2, 0.2)
196 contour = sausage_ax.contourf(lambda_bar_mesh,
197                                k_bar_mesh,
198                                stability,
199                                levels=levels)
200 contour.cmap.set_over('darkgreen')
201 contour.cmap.set_under('white')
202 contour.set_clim(-0.2, -0.1)
203
204 colors_list = ['grey', 'white', 'white', 'white']
205
206 contour_lines = sausage_ax.contour(lambda_bar_mesh,
207                                    k_bar_mesh,
208                                    stability,
209                                    levels=levels,
210                                    colors=colors_list,
211                                    linewidths=2)
212 for line in contour_lines.collections:
213     line.set_linestyle('solid')
214
215
216 sausage_ax.clabel(contour_lines,
217                                manual=([2.3, 1.3],
218                                        [1.75, 0.55],
219                                        [2.0, 0.4],
220                                        [2.1, 0.2]),
221                                fmt=r'$ \epsilon = %.1f$', fontsize=12)
222
223 sausage_ax.plot([0, 3.], [0., 1.5], '--', c='black', lw=2)
224 sausage_ax.set_xlabel(r'$\bar{\lambda}$')
225 #sausage_ax.xaxis.labelpad = 20
226 plt.setp(sausage_ax.get_xticklabels())
227 sausage_ax.set_xticks(np.arange(0., 5, 1.))
228
229 sausage_ax.set_ylabel(r'$\bar{k}$', rotation='horizontal')
230 plt.setp(sausage_ax.get_yticklabels())
231 sausage_ax.set_yticks(np.arange(0., 2.0, 0.5))
232 sns.despine()
233
234
235 ## Delta dependence ##
236 ######################
237
238 epsilon = 0.2
239 stability = np.zeros(lambda_bar_mesh.shape) - 1.2
240 for i, delta in enumerate(np.arange(-1.0, 1.6, 0.5)):
241
242     d_w_delta = conditions(k_bar_mesh,
243                            lambda_bar_mesh,
244                            epsilon=epsilon,
245                            m=0,
246                            delta=delta)
247
248     stability_delta = d_w_delta < 0
249     stability[stability_delta] = delta
250
```

```
levels = np.arange(-1.0, 1.6, 0.5)
contour = delta_ax.contourf(lambda_bar_mesh,
                            k_bar_mesh,
                            stability,
                            levels=levels,
                            extend='both')
levels = np.arange(-1.0, 1.6, 0.5)

colors_list = ['grey', 'white', 'white', 'white', 'white']

contour_lines = delta_ax.contour(lambda_bar_mesh,
                                 k_bar_mesh,
                                 stability,
                                 levels=levels,
                                 colors=colors_list,
                                 linewidths=2)
contour.cmap.set_over('darkgreen')
contour.cmap.set_under('white')
contour.set_clim(-1.3, -1.0)

for line in contour_lines.collections:
    line.set_linestyle('solid')

sausage_ax.clabel(contour_lines,
                  manual=([2., 0.85],
                          [2.4, 0.75],
                          [2.8, 0.65],
                          [3.0, 0.55],
                          [3.3, 0.45]),
                  fmt=r' $ \delta = \mathbf{%.1f}$ ',
                  colors=colors_list, fontsize=12)

delta_ax.plot([0, 3.], [0., 1.5], '--', c='black', lw=2)
delta_ax.set_xlabel(r'$\bar{\lambda}$')
#delta_ax.xaxis.labelpad = 20
plt.setp(delta_ax.get_xticklabels())
delta_ax.set_xticks(np.arange(0., 5, 1.))

delta_ax.set_ylabel(r'$\bar{k}$', rotation='horizontal')
plt.setp(delta_ax.get_yticklabels())
delta_ax.set_yticks(np.arange(0., 2.0, 0.5))

plt.tight_layout()

plt.figtext(0.001, 0.97, '(a)')
plt.figtext(0.001, 0.5, '(c)')
plt.figtext(0.49, 0.97, '(b)')
plt.figtext(0.49, 0.5, '(d)')

fig.savefig('../figures/figure1.eps', dpi=300)


## Fig 2 Profiles ##
####################
def normalized_single_plot(profile, axes, ylim, letter, styles=['-', '--', '-.'],
                           legend_loc=None):
    r"""
```

```
    """
    splines = profile.get_splines()
    beta_spl = profile.beta(profile.r)
    r1 = np.linspace(0, 1., 250)

    j_skin = profile.j_skin
    b_z0 = profile.b_z0


    axes.yaxis.tick_left()
    axes.xaxis.tick_bottom()

    j_z_line = axes.plot(r1, splines['j_z'](r1)/j_skin, lw=2.5, label=r'$\bar{j}_z$', ls=styles[0])
    b_theta_line = axes.plot(r1, splines['b_theta'](r1), lw=2.5, label=r'$\bar{B}_\theta$', ls=styles[1])
    p_line = axes.plot(r1, splines['pressure'](r1), lw=2.5, label=r'$\bar{p}$', ls=styles[2])
    #j_z_line = axes.plot(r1, splines['j_z'](r1)/j_skin, lw=2.5, ls=styles[0])
    #b_theta_line = axes.plot(r1, splines['b_theta'](r1), lw=2.5, ls=styles[1])
    #p_line = axes.plot(r1, splines['pressure'](r1), lw=2.5, ls=styles[2])
    axes.set_xlim(0, 1.)

    axes.set_ylim(0., ylim)

    if legend_loc:
        lgd = axes.legend(bbox_to_anchor=legend_loc,
                          loc='upper center',
                          borderaxespad=0.25,
                          ncol=3,
                          frameon=True,
                          handlelength=1.8,
                          columnspacing=1.1)
    else:
        lgd = None

    #axes.annotate('core', xy=(0.27, ylim*4.5/7.))
    #axes.annotate('transition', xy=(0.624, ylim*4.5/7.))
    #axes.annotate('skin', xy=(0.772, ylim*4.5/7.))
    #axes.annotate('transition', xy=(0.85, ylim*4.5/7.))

    axes.set_xlabel(r'$r$')
    axes.axvline(x=0.6, color='grey', ls='--', lw=2.5, alpha=0.5)
    axes.axvline(x=0.775, color='grey', ls='--', lw=2.5, alpha=0.5)
    axes.axvline(x=0.825, color='grey', ls='--', lw=2.5, alpha=0.5)
    axes.yaxis.set_ticks(np.arange(0.0, 1.8, 0.7))

    plt.setp(axes.get_xticklabels())
    plt.setp(axes.get_yticklabels())
    return lgd

fig = plt.figure(figsize=(3.75,5.5))
axes1 = plt.subplot2grid((3, 2), (0, 0), colspan=2, rowspan=1)
axes2 = plt.subplot2grid((3, 2), (1, 0), colspan=2, rowspan=1)
axes3 = plt.subplot2grid((3, 2), (2, 0), colspan=2, rowspan=1)
fig.subplots_adjust(wspace=0.05)
fig.subplots_adjust(hspace=0.5)

profile = es.UnitlessSmoothedCoreSkin(k_bar=1, lambda_bar=1, epsilon=0.7,
                                      core_radius_norm=0.6,
```

```
365                                     transition_width_norm=0.175,
366                                     skin_width_norm=0.05)
367
368 lgd = normalized_single_plot(profile, axes1, 1.8, 'a', legend_loc=[0.51, 1.8])
369
370 profile = es.UnitlessSmoothedCoreSkin(k_bar=1, lambda_bar=1, epsilon=0.5,
371                                     core_radius_norm=0.6,
372                                     transition_width_norm=0.175,
373                                     skin_width_norm=0.05)
374
375 normalized_single_plot(profile, axes2, 1.8, 'b')
376
377 profile = es.UnitlessSmoothedCoreSkin(k_bar=1, lambda_bar=1, epsilon=0.1,
378                                     core_radius_norm=0.6,
379                                     transition_width_norm=0.175,
380                                     skin_width_norm=0.05)
381
382 normalized_single_plot(profile, axes3, 1.8, 'c')
383 #axes1.legend(loc='best')
384 #axes2.legend(loc='best')
385 #axes3.legend(loc='best')
386 #plt.figlegend([l1, l2, l3],
387 #            [r'$\bar{j}_z$',
388 #             r'$\bar{B}_\theta$',
389 #             r'$\bar{p}$'], 'upper left',
390 #            bbox_to_anchor = (0.5, 0.5))
391
392 #plt.tight_layout()
393 plt.figtext(-0.0, 0.90, '(a)')
394 plt.figtext(-0.0, 0.60, '(b)')
395 plt.figtext(-0.0, 0.30, '(c)')
396 #plt.show()
397 fig.subplots_adjust(hspace=0.75)
398 sns.despine()
399 plt.savefig('../figures/figure2.eps', dpi=300,
400             bbox_extra_artists=(lgd,), bbox_inches='tight')
401
402 ## Fig 3 Numerical Stability Space ##
403 ####################################
404
405 from mpl_toolkits.axes_grid1 import make_axes_locatable
406
407 from matplotlib.colors import SymLogNorm, BoundaryNorm
408 from matplotlib.ticker import FormatStrFormatter, FixedFormatter
409 import matplotlib.patches as patches
410 import matplotlib.ticker as ticker
411
412 def plot_lambda_k_space_dw(axes, filename, epsilon, name, mode_to_plot='m_neg_1',
413                            show_points=False, lim=None, levels=None, log=True,
414                            linthresh=1E-7, bounds=(1.5, 3.0), norm=True,
415                            analytic_compare=False,
416                            label_pos=((0.5, 0.4), (2.1, 0.4), (2.8, 0.2)),
417                            delta_values=[-1,0,1],
418                            interpolate=False,
419                            cmap=None, hatch=False,
420                            figsize=None,
421                            save_as=None,
```

```
                            return_ax=False,
                            hatch_sausage_gap=False):

    epsilon_case = np.load(filename)
    lambda_a_mesh = epsilon_case['lambda_a_mesh']
    k_a_mesh = epsilon_case['k_a_mesh']
    external_m_neg_1 = epsilon_case['d_w_m_neg_1']
    external_sausage = epsilon_case['d_w_m_0']
    epsilon_case.close()

    if hatch_sausage_gap:
        external_sausage_gap = np.where((lambda_a_mesh > 3.) & (external_sausage > 0))
        external_sausage[external_sausage_gap] = np.nan

    instability_map = {'m_0': external_sausage,
                       'm_neg_1': external_m_neg_1}


    kink_pal = sns.blend_palette([sns.xkcd_rgb["dandelion"],
                                  sns.xkcd_rgb["white"]], 7, as_cmap=True)
    kink_pal = sns.diverging_palette(73, 182, s=72, l=85, sep=1, n=9, as_cmap=True)
    sausage_pal = sns.blend_palette(['orange', 'white'], 7, as_cmap=True)
    sausage_pal = sns.diverging_palette(49, 181, s=99, l=78, sep=1, n=9, as_cmap=True)

    if cmap:
        instability_palette = {'m_0': cmap,
                               'm_neg_1': cmap}
    else:
        instability_palette = {'m_0': sausage_pal,
                               'm_neg_1': kink_pal}




    if interpolate:
        instability_map['m_neg_1'] = interpolate_nans(lambda_a_mesh,
                                                      k_a_mesh,
                                                      instability_map['m_neg_1']
                                                      )

    values = instability_map[mode_to_plot]

    if norm:
        values = values / np.nanmax(np.abs(values))
    else:
        values = values

    if levels:
        if log:
            plot = axes.contourf(lambda_a_mesh, k_a_mesh, values,
                                 cmap=instability_palette[mode_to_plot],
                                 levels=levels, norm=SymLogNorm(linthresh))
            divider = make_axes_locatable(axes)
            cax = divider.append_axes("right", size="5%", pad=0.1)
            cbar = plt.colorbar(plot, cax=cax, label=r'$\delta W$')
            cbar.ax.yaxis.set_ticks_position('right')
            cbar.set_label(label=r'$\delta W$', size=10, rotation=0, labelpad=1)
```

```
479             contourlines = axes.contour(lambda_a_mesh, k_a_mesh,
480                                         values, levels=levels,
481                                         colors='grey',
482                                         norm=SymLogNorm(linthresh), linewidths=0.8)
483
484         else:
485             norm = BoundaryNorm(levels, 256)
486             plot = axes.contourf(lambda_a_mesh, k_a_mesh, values,
487                                  cmap=instability_palette[mode_to_plot],
488                                  levels=levels, norm=norm)
489             divider = make_axes_locatable(axes)
490             cax = divider.append_axes("right", size="5%", pad=0.1)
491             cbar = plt.colorbar(plot, cax=cax, label=r'$\delta W$')
492             cbar.set_label(label=r'$\delta W$', size=10, rotation=0, labelpad=1)
493             contourlines = axes.contour(lambda_a_mesh, k_a_mesh,
494                                         values, levels=levels,
495                                         colors='grey', linewidths=0.8)
496     else:
497         if log:
498             plot = axes.contourf(lambda_a_mesh, k_a_mesh, values,
499                                  cmap=instability_palette[mode_to_plot],
500                                  norm=SymLogNorm(linthresh))
501             divider = make_axes_locatable(axes)
502             cax = divider.append_axes("right", size="5%", pad=0.1)
503             cbar = plt.colorbar(plot, cax=cax, label=r'$\delta W$')
504             cbar.set_label(label=r'$\delta W$', size=10, rotation=0, labelpad=1)
505             contourlines = axes.contour(lambda_a_mesh, k_a_mesh,
506                                         values, colors='grey',
507                                         norm=SymLogNorm(linthresh), linewidths=0.8)
508         else:
509             plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
510                                 cmap=instability_palette[mode_to_plot])
511             divider = make_axes_locatable(axes)
512             cax = divider.append_axes("right", size="5%", pad=0.1)
513             cbar = plt.colorbar(plot, cax=cax, label=r'$\delta W$')
514             cbar.set_label(label=r'$\delta W$', size=10, rotation=0, labelpad=1)
515             contourlines = axes.contour(lambda_a_mesh, k_a_mesh,
516                                         values, colors='grey', linewidths=0.8)
517
518     if lim:
519         plot.set_clim(lim)
520
521     cbar.add_lines(contourlines)
522
523     cbar_lines = cbar.lines[0]
524     number_of_lines = len(cbar_lines.get_linewidths())
525
526     linewidths = []
527     linestyles = []
528     cbar_zero_line = 4
529     for line in xrange(number_of_lines):
530         if line < cbar_zero_line:
531             linewidths.append(0.8)
532             linestyles.append('--')
533         elif line == cbar_zero_line:
534             linewidths.append(0.8)
535             linestyles.append('-')
```

```
536             else:
537                 linewidths.append(0.8)
538                 linestyles.append('-')
539 
540         cbar_lines.set_linewidths(linewidths)
541         cbar_lines.set_linestyles(linestyles)
542 
543         axes.plot([0.01, 0.1, 1.0, 2.0, 3.0],
544                   [0.005, 0.05, 0.5, 1.0, 1.5], color='black', lw=1.5)
545 
546         axes.set_axis_bgcolor(sns.xkcd_rgb['white'])
547 
548         lambda_bar_analytic = np.linspace(0.01, 4., 750)
549         k_bar_analytic = np.linspace(0.01, 1.5, 750)
550         (lambda_bar_mesh_analytic,
551          k_bar_mesh_analytic) = np.meshgrid(lambda_bar_analytic, k_bar_analytic)
552 
553         if analytic_compare:
554             analytic_comparison(mode_to_plot, k_bar_mesh_analytic,
555                                 lambda_bar_mesh_analytic, epsilon, label_pos)
556 
557         if show_points:
558             axes.scatter(lambda_a_mesh, k_a_mesh, marker='o', c='b', s=2)
559 
560         axes.set_ylim(0.01, bounds[0])
561         axes.set_xlim(0.01, bounds[1])
562         axes.set_xticks(np.arange(0., 4.5, 1.0))
563         axes.set_yticks(np.arange(0., 2.0, 0.5))
564         plt.setp(axes.get_xticklabels(), fontsize=10)
565         plt.setp(axes.get_yticklabels(), fontsize=10)
566         axes.set_ylabel(r'$\bar{k}$', rotation='horizontal', fontsize=10)
567         axes.set_xlabel(r'$\bar{\lambda}$', fontsize=10)
568         cbar.ax.tick_params(labelsize=8)
569 
570         def my_formatter_fun(x):
571             if x == 0:
572                 return r'$0$'
573             if np.sign(x) > 0:
574                 return r'$10^{%i}$' % np.int(np.log10(x))
575             else:
576                 return r'$-10^{%i}$' % np.int(np.log10(np.abs(x)))
577         labels = [my_formatter_fun(level) for level in levels]
578         cbar.ax.set_yticklabels(labels)
579         sns.despine(ax=axes)
580 
581         if hatch:
582             xmin, xmax = axes.get_xlim()
583             ymin, ymax = axes.get_ylim()
584             xy = (xmin,ymin+0.1)
585             width = xmax - xmin
586             height = ymax - ymin
587             p = patches.Rectangle(xy, width, height, hatch='X'*2,
588                                   zorder=-10, edgecolor='#FEFFE0',
589                                   facecolor='#6C2605')
590             axes.add_patch(p)
591         if hatch_sausage_gap:
592             xmin, xmax = axes.get_xlim()
```

```
            ymin, ymax = axes.get_ylim()
            xmin = 3.
            xy = (xmin, ymin+0.01)
            width = xmax - xmin
            height = ymax - ymin
            p = patches.Rectangle(xy, width, height, hatch='X'*2,
                                  zorder=-10, edgecolor='#FEFFE0',
                                  facecolor='#005B31')
            axes.add_patch(p)
        cbar.ax.yaxis.set_ticks_position('right')


fig = plt.figure(figsize=(6.69, 6.69))

kink_ax1 = plt.subplot2grid((9, 6), (0, 0), colspan=3, rowspan=3)
sausage_ax1 = plt.subplot2grid((9, 6), (0, 3), colspan=3, rowspan=3)
kink_ax2 = plt.subplot2grid((9, 6), (3, 0), colspan=3, rowspan=3)
sausage_ax2 = plt.subplot2grid((9, 6), (3, 3), colspan=3, rowspan=3)
kink_ax3 = plt.subplot2grid((9, 6), (6, 0), colspan=3, rowspan=3)
sausage_ax3 = plt.subplot2grid((9, 6), (6, 3), colspan=3, rowspan=3)


## High epsilon
plot_lambda_k_space_dw(kink_ax1, high_epsilon_path,
                       1., 'ep12-m1', mode_to_plot='m_neg_1',
                       levels=[-1, -1e-1, -1e-2, -1e-3,
                               0, 1e-3, 1e-2, 1e-1, 1],
                       norm=True, analytic_compare=False,
                       log=True,
                       label_pos=None,
                       interpolate=False, cmap="YlOrBr_r",
                       hatch=True,
                       bounds=(1.5, 4.3))


plot_lambda_k_space_dw(sausage_ax1, high_epsilon_path,
                       1., 'ep12-m1', mode_to_plot='m_0',
                       levels=[-1, -1e-1, -1e-2, -1e-3,
                               0, 1e-3, 1e-2, 1e-1, 1],
                       norm=True, analytic_compare=False,
                       log=True,
                       label_pos=None,
                       interpolate=False, cmap="YlGn_r",
                       bounds=(1.5, 4.3),
                       hatch_sausage_gap=True)


## Mid epsilon
plot_lambda_k_space_dw(kink_ax2, mid_epsilon_path,
                       1., 'ep12-m1', mode_to_plot='m_neg_1',
                       levels=[-1, -1e-1, -1e-2, -1e-3,
                               0, 1e-3, 1e-2, 1e-1, 1],
                       norm=True, analytic_compare=False,
                       log=True,
                       label_pos=None,
                       interpolate=False, cmap="YlOrBr_r",
```

```
                        hatch=True,
                        bounds=(1.5, 4.3))


plot_lambda_k_space_dw(sausage_ax2, mid_epsilon_path,
                        1., 'ep12-m1', mode_to_plot='m_0',
                        levels=[-1, -1e-1, -1e-2, -1e-3,
                                0, 1e-3, 1e-2, 1e-1, 1],
                        norm=True, analytic_compare=False,
                        log=True,
                        label_pos=None,
                        interpolate=False, cmap="YlGn_r",
                        hatch_sausage_gap=True,
                        bounds=(1.5, 4.3))




## Low epsilon
plot_lambda_k_space_dw(kink_ax3, low_epsilon_path,
                        1., 'ep12-m1', mode_to_plot='m_neg_1',
                        levels=[-1, -1e-1, -1e-2, -1e-3,
                                0, 1e-3, 1e-2, 1e-1, 1],
                        norm=True, analytic_compare=False,
                        log=True,
                        label_pos=None,
                        interpolate=False, cmap="YlOrBr_r",
                        bounds=(1.5, 4.3),
                        hatch=True)

plot_lambda_k_space_dw(sausage_ax3, low_epsilon_path,
                        1., 'ep12-m1', mode_to_plot='m_0',
                        levels=[-1, -1e-1, -1e-2, -1e-3,
                                    0, 1e-3, 1e-2, 1e-1, 1],
                        norm=True, analytic_compare=False,
                        log=True,
                        label_pos=None,
                        bounds=(1.5, 4.3),
                        interpolate=False, cmap="YlGn_r",
                        hatch=True)



plt.tight_layout(pad=0.5, w_pad=0.1, h_pad=0.5)

plt.figtext(0.01, 0.98, '(a)', fontsize=10)
plt.figtext(0.01, 0.655, '(b)', fontsize=10)
plt.figtext(0.01, 0.33, '(c)', fontsize=10)
plt.figtext(0.49, 0.98, '(d)', fontsize=10)
plt.figtext(0.49, 0.655, '(e)', fontsize=10)
plt.figtext(0.49, 0.33, '(f)', fontsize=10)

plt.savefig('../figures/figure3.eps', dpi=300)

### Ratio Plots ###
####################

```

```
def sausage_kink_ratio(axes, filename, xy_limits=None, cmap=None, save_as=None,
                       levels=None, zero_line=2, label_lines=False, cbar_zero_line=2):
    r"""
    Plot ratio of sausage and kink potential energies.
    """
    meshes = np.load(filename)
    lambda_bar_mesh = meshes['lambda_a_mesh']
    k_bar_mesh = meshes['k_a_mesh']
    external_m_neg_1 = meshes['d_w_m_neg_1']
    external_sausage = meshes['d_w_m_0']
    meshes.close()

    sausage_stable_region = np.invert((external_sausage < 0))
    ratio = np.abs(external_sausage / external_m_neg_1)
    ratio[sausage_stable_region] = np.nan
    ratio_log = np.log10(ratio)

    if not cmap:
        cmap = sns.light_palette(sns.xkcd_rgb['red orange'],
                                 as_cmap=True)
    if levels:
        contours = axes.contourf(lambda_bar_mesh, k_bar_mesh,
                                 ratio_log, cmap=cmap, levels=levels)
    else:
        contours = axes.contourf(lambda_bar_mesh, k_bar_mesh,
                                 ratio_log, cmap=cmap)

    divider = make_axes_locatable(axes)
    cax = divider.append_axes("right", size="5%", pad=0.1)
    colorbar = plt.colorbar(contours, cax=cax, format=FormatStrFormatter(r'$10^{%i}$'))
    colorbar.set_label(r'$\frac{\delta W_{m=0}}{\delta W_{m=1}}$', rotation=0,
                       labelpad=10,
                       fontsize=10)
    if levels:
        lines = axes.contour(lambda_bar_mesh, k_bar_mesh,
                             ratio_log, colors='black', levels=levels, linewidths=1)
    else:
        lines = axes.contour(lambda_bar_mesh, k_bar_mesh,
                             ratio_log, colors='black', linewidths=1)
    plt.setp(lines.collections[zero_line], linewidth=2)
    colorbar.add_lines(lines)

    cbar_lines = colorbar.lines[0]
    number_of_lines = len(cbar_lines.get_linewidths())

    linewidths = []
    linestyles = []
    for line in xrange(number_of_lines):
        if line < cbar_zero_line:
            linewidths.append(1)
            linestyles.append('--')
        elif line == cbar_zero_line:
            linewidths.append(2)
            linestyles.append('-')
        else:
            linewidths.append(1)
            linestyles.append('-')
```

```
764
765         cbar_lines.set_linewidths(linewidths)
766         cbar_lines.set_linestyles(linestyles)
767
768         axes.plot([0, 3.], [0., 1.5], '--', c='black', lw=2)
769         axes.set_xlabel(r'$\bar{\lambda}$', fontsize=10)
770         plt.setp(axes.get_xticklabels(), fontsize=10)
771         axes.set_xticks(np.arange(0., 4.5, 1.0))
772
773         axes.set_ylabel(r'$\bar{k}$', rotation='horizontal', fontsize=10)
774         plt.setp(axes.get_yticklabels(), fontsize=10)
775         axes.set_yticks(np.arange(0., 2.0, 0.5))
776
777         if label_lines:
778             plt.clabel(lines)
779
780         if xy_limits:
781             axes.set_ylim((xy_limits[0], xy_limits[1]))
782             axes.set_xlim((xy_limits[2], xy_limits[3]))
783         plt.setp(axes.get_xticklabels(), fontsize=10)
784         plt.setp(axes.get_yticklabels(), fontsize=10)
785         axes.set_xlim((0, 4.3))
786         sns.despine(ax=axes)
787         colorbar.ax.yaxis.set_ticks_position('right')
788         colorbar.ax.tick_params(labelsize=8)
789
790     fig = plt.figure(figsize=(3.75,6.69))
791
792     ratio_ax1 = plt.subplot2grid((9, 3), (0, 0), colspan=3, rowspan=3)
793     ratio_ax2 = plt.subplot2grid((9, 3), (3, 0), colspan=3, rowspan=3)
794     ratio_ax3 = plt.subplot2grid((9, 3), (6, 0), colspan=3, rowspan=3)
795
796     sausage_kink_ratio(ratio_ax1, high_epsilon_path,
797                        cmap="Greys",
798                        zero_line=2, label_lines=False,
799                        cbar_zero_line=2)
800
801     sausage_kink_ratio(ratio_ax2, mid_epsilon_path,
802                        cmap="Greys",
803                        zero_line=1, label_lines=False,
804                        cbar_zero_line=1)
805
806     sausage_kink_ratio(ratio_ax3, low_epsilon_path,
807                        cmap="Greys",
808                        levels=[-4, -2, 0, 2, 4, 6, 15],
809                        zero_line=2, label_lines=False,
810                        cbar_zero_line=2)
811     plt.tight_layout()
812
813     plt.figtext(0.05, 0.96, '(a)', fontsize=10)
814     plt.figtext(0.05, 0.64, '(b)', fontsize=10)
815     plt.figtext(0.05, 0.32, '(c)', fontsize=10)
816     plt.savefig('../figures/figure4.eps', dpi=300)
817
818     ## Critical Lambda Bar Plot ##
819     ##############################
820
```

```
821 def lambda_crit(s_star):
822     return 2.*s_star/np.sqrt(4. - s_star**2)
823
824 fig, axes = plt.subplots(figsize=(3.37,3.37))
825 s_star = np.linspace(0, 5, 200)
826 critical_value = np.zeros(s_star.size)
827 critical_value[s_star < 2] = lambda_crit(s_star[s_star < 2])
828 critical_value[s_star >= 2] = np.nan
829 axes.plot(s_star, critical_value, color='black', lw=2)
830 axes.set_xlabel(r'$S^\ast$')
831 axes.set_ylabel(r'$\bar{\lambda}_{crit}$')
832 axes.set_ylim(0, 5)
833 axes.axhline(2*np.sqrt(2), linestyle='--', color='black', lw=2)
834 rectangle = patches.Rectangle((-1, 2.25), 10, 10, hatch='/',
835                               facecolor='none', edgecolor='grey')
836 axes.add_patch(rectangle)
837 plt.tight_layout()
838 plt.savefig('../figures/figure5.eps', dpi=300)
839 """
```

A.6.4 call_provenance.py

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Oct 24 13:01:44 2015
4
5  @author: jensv
6  """
7
8  from git import Repo
9  import os
10
11
12 def call_and_git_commit(call='', call_path=None):
13     """
14     Returns a program call and git HEAD commit hash of a git repo located in
15     the supplied call directory. If no directory is passed the current
16     directory is used.
17     """
18     if call_path is None:
19         call_path = os.getcwd()
20     repo = Repo(path=call_path, search_parent_directories=True)
21     commit = repo.commit('HEAD')
22     git_commit = commit.hexsha
23     return call, git_commit
```

A.6.5 equil_solver.py

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Fri May 23 10:19:28 2014
```

```
4
5   @author: Jens von der Linden
6
7   Create various profiles.
8   The analysis in the paper exclusivly focuses on UnitlessSmoothedCoreSkin
9   profile.
10
11  ParbolicNu2,  NuCurrentProfile recreate the profiles from Wesson's
12  Tokamak book. In the large aspect ratio limit k->0 these should
13  allow to recreate his stability plots.
14  The HardCoreZPinch is discussed in Freidberg's Ideal MHD book.
15  """
16
17
18  from __future__ import print_function, unicode_literals, division
19  from __future__ import absolute_import
20  from future import standard_library, utils
21  from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
22                               int, map, next, oct, open, pow, range, round,
23                               str, super, zip)
24  """Python 3.x compatability"""
25
26  import sys
27  import numpy as np
28  from numpy import atleast_1d
29  from scipy.interpolate import splev
30  import sympy as sp
31  from collections import OrderedDict
32  import scipy.interpolate as interp
33  import scipy.integrate as inte
34
35
36  class EquilSolver(object):
37      r"""
38      General Equilibrium Solver parent implements spline generation and safety
39      factor calculation for various profiles.
40      """
41
42      def set_splines(self, param_points):
43          r"""
44          Returns Splines requested with a dictionary of spline names and point
45          generating functions.
46          """
47          splines = {}
48          r = self.r
49          for key, value in param_points.items():
50              splines[key] = interp.InterpolatedUnivariateSpline(r,
51                                                                 value(self.r),
52                                                                 k=3)
53              self.splines = splines
54
55      def get_splines(self):
56          r"""
57          Returns splines of instance.
58          """
59          return self.splines
60
```

```
61      def get_tck_splines(self):
62          r"""
63          Returns tck splines lists, to be used with scipy procedural spline
64          interface.
65          """
66          return self.tck_splines
67
68      def q(self, r):
69          r"""
70          Returns safety factor evaluated at points.
71          """
72          if r[0] == 0.:
73              q_to_return = np.ones(r.size)*self.q0
74              q_to_return[1:] = r[1:]*self.k*self.b_z(r[1:])/self.b_theta(r[1:])
75          else:
76              q_to_return = r*self.k*self.b_z(r)/self.b_theta(r)
77          return q_to_return
78
79      def rho(self, r):
80          r"""
81          Return density. Set to one evrywhere. Density plays no role for
82          Newcomb. However, the general eigenvalue problem has stability issues.
83          """
84          return np.ones(r.size)
85
86      def convert_spline_objects_to_tck(self, spline_dict):
87          r"""
88          Returns tck tuples for use with procedural scipy spline interface,
89          which is slightly faster than the object-oriented interface.
90
91          Parameters
92          ----------
93          spline_dict: dict
94              dict of splines
95
96          Returns
97          -------
98          splines_tck: dict
99              dict of tck_splines
100         """
101         tck_splines = {}
102         for key in spline_dict.keys():
103             tck = spline_dict[key]._eval_args
104             tck_splines.update({key: tck})
105         return tck_splines
106
107
108 class ParabolicNu2(EquilSolver):
109     r"""
110     Creates splines for parabolic current profile pinch.
111     """
112
113     def __init__(self, a=1, points=500, q0=1.0, k=1, b_z0=1, temp=1.0,
114                  qa=None, mu_0=1.):
115         r"""
116         Initalize parameters defining parabolic pinch and create splines.
117         """
```

```
118         self.r = np.linspace(0, a, points)
119         if qa is not None:
120             q0 = qa/3.
121         self.q0 = q0
122         self.mu_0 = mu_0
123         self.k = k
124         self.b_z0 = b_z0
125         self.temp = temp
126         self.j0 = self.get_j0()
127         param_points = {'j_z': self.j_z, 'b_theta': self.b_theta,
128                         'b_z': self.b_z, 'p_prime': self.p_prime,
129                         'pressure': self.pressure, 'q': self.q,
130                         'rho': self.rho}
131         self.set_splines(param_points)
132
133     def get_j0(self):
134         r"""
135         Return j0 for a b_z and q0 paramters of pinch.
136         """
137         self.j0 = 1
138         return self.k*2.*self.b_z(0)/(self.q0*self.mu_0)
139
140     def j_z(self, r):
141         r"""
142         Return parabolic axial current profile.
143         """
144         return self.j0*(1 - r**2)**2
145
146     def b_theta(self, r):
147         r"""
148         Return azimuthhal magnetic field for a parabolic current profile pinch.
149         """
150         j0 = self.j0
151         return self.mu_0*(j0*r/2 - j0*r**3/2. + j0*r**5/6.)
152
153     def b_z(self, r):
154         r"""
155         Return constant axial magnetic field.
156         """
157         b_z0 = self.b_z0
158         r = np.asarray(r)
159         return np.ones(r.size) * b_z0
160
161     def p_prime(self, r):
162         r"""
163         Returns pressure_prime profile for a arabolic current profile pinch.
164         """
165         j0 = self.j0
166         return self.mu_0*(-j0**2*r/2. + 3.*j0**2*r**3/2.
167                           - 5.*j0**2*r**5/3. + 5.*j0**2*r**7/6.
168                           - j0**2*r**9/6.)
169
170     def pressure(self, r):
171         r"""
172         Returns pressure profile for a arabolic current profile pinch.
173         """
174         j0 = self.j0
```

```
175         return self.mu_0*(47.*j0**2/720. - j0**2*r**2/4. + 3.*j0**2*r**4/8.
176                           - 5.*j0**2*r**6/18. + 5.*j0**2*r**8/48.
177                           - j0**2*r**10/60.)
178
179
180 class NuCurrentConstructor(object):
181     r"""
182     Constructors for arbitrary nu current profiles.
183     """
184     def __init__(self, a=1., nu=2):
185         j0, r, mu_0, k, b_z, q0, qa = sp.symbols('j_0 r mu_0 k b_z \
186                                                  q_0 q_a')
187         current_sym = j0*(1 - r**2)**nu
188         b_theta_sym = mu_0*sp.integrate(current_sym*r, r, conds='none')/r
189         p_prime_sym = -current_sym*b_theta_sym
190         pressure_sym = sp.integrate(p_prime_sym, r, conds='none')
191         pressure_norm_sym = pressure_sym - pressure_sym.subs(r, a)
192         q_sym = sp.cancel(r*k*b_z/b_theta_sym)
193         q0_sym = q_sym.subs(r, 0)
194         qa_sym = q_sym.subs(r, a)
195         j0_defined_by_q0_sym = sp.solve(q0_sym - q0, j0)[0]
196         j0_defined_by_qa_sym = sp.solve(qa_sym - qa, j0)[0]
197
198         # create lambda functions of expressions
199         self.current_func = sp.lambdify((r, j0), current_sym,
200                                         modules=str('numpy'))
201         self.b_theta_func = sp.lambdify((r, j0, mu_0), sp.cancel(b_theta_sym),
202                                         modules=str('numpy'))
203         self.p_prime_func = sp.lambdify((r, j0, mu_0), sp.cancel(p_prime_sym),
204                                         modules=str('numpy'))
205         self.pressure_func = sp.lambdify((r, j0, mu_0), pressure_norm_sym,
206                                          modules=str('numpy'))
207         self.q_func = sp.lambdify((r, k, b_z, j0, mu_0), q_sym,
208                                   modules=str('numpy'))
209         self.j0_defined_by_q0_func = sp.lambdify((k, b_z, mu_0, q0),
210                                                  j0_defined_by_q0_sym,
211                                                  modules=str('numpy'))
212         self.j0_defined_by_qa_func = sp.lambdify((k, b_z, mu_0, qa),
213                                                  j0_defined_by_qa_sym,
214                                                  modules=str('numpy'))
215
216
217 class NuCurrentProfile(EquilSolver):
218     r"""
219     Nu current profiles as described in Wesson's Tokamak book under
220     large aspect ratio tokamak limit.
221     """
222
223     def __init__(self, nu_constructor=NuCurrentConstructor(), nu=2, a=1,
224                  points=500, q0=1.0, k=1., b_z0=1., temp=1.0, qa=None, mu_0=1.):
225         r"""
226
227         """
228
229         self.get_j_z = nu_constructor.current_func
230         self.get_b_theta = nu_constructor.b_theta_func
231         self.get_p_prime = nu_constructor.p_prime_func
```

```
232         self.get_pressure = nu_constructor.pressure_func
233         self.get_q = nu_constructor.q_func
234         self.get_j0_given_q0 = nu_constructor.j0_defined_by_q0_func
235         self.get_j0_given_qa = nu_constructor.j0_defined_by_qa_func
236 
237         self.r = np.linspace(0, a, points)
238         self.nu = nu
239         if qa is not None:
240             self.j0 = self.get_j0_given_qa(k, b_z0, mu_0, qa)
241         else:
242             self.j0 = self.get_j0_given_q0(k, b_z0, mu_0, q0)
243         self.q0 = q0
244         self.mu_0 = mu_0
245         self.k = k
246         self.b_z0 = b_z0
247         self.temp = temp
248         param_points = {'j_z': self.j_z, 'b_theta': self.b_theta,
249                         'b_z': self.b_z, 'p_prime': self.p_prime,
250                         'pressure': self.pressure, 'q': self.q,
251                         'rho': self.rho}
252         self.set_splines(param_points)
253 
254     def j_z(self, r):
255         r"""
256         Return nu axial current profile.
257         """
258         j_z_value = self.get_j_z(r, self.j0)
259         if isinstance(j_z_value, int) or isinstance(j_z_value, float):
260             j_z_value = np.ones(r.size)*j_z_value
261         return j_z_value
262 
263     def b_theta(self, r):
264         r"""
265         Return azimuthhal magnetic field for a nu current profile pinch.
266         """
267         return self.get_b_theta(r, self.j0, self.mu_0)
268 
269     def b_z(self, r):
270         r"""
271 
272         """
273         b_z0 = self.b_z0
274         r = np.asarray(r)
275         return np.ones(r.size) * self.b_z0
276 
277     def p_prime(self, r):
278         r"""
279 
280         """
281         return self.get_p_prime(r, self.j0, self.mu_0)
282 
283     def pressure(self, r):
284         r"""
285 
286         """
287         return self.get_pressure(r, self.j0, self.mu_0)
288 
```

```
289     def q(self, r):
290         r"""
291         Returns safety factor evaluated at points.
292         """
293         q_value = self.get_q(r, self.k, self.b_z0, self.j0, self.mu_0)
294         if isinstance(q_value, int) or isinstance(q_value, float):
295             q_value = np.ones(r.size)*q_value
296         return q_value
297 
298 
299 class NewcombConstantPressure(EquilSolver):
300     """
301     Creates splines describing the constant pressure profile at the end of
302     Newcomb's 1960 paper.
303     """
304 
305     def __init__(self, a=0.1, r_0i=0.5, k=1, b_z0=0.1, b_thetai=0.1,
306                  points=500):
307         r"""
308         Initialize parameters defining parabolic pinch and create splines.
309         """
310         self.r = np.linspace(a, r_0i, points)
311         self.a = a
312         self.r_0i = r_0i
313         self.k = k
314         self.b_z0 = b_z0
315         self.b_thetai = b_thetai
316         self.beta_0 = 0.
317         r = self.r
318         param_points = {'j_z': self.get_j_z, 'b_theta': self.b_theta,
319                         'b_z': self.b_z, 'p_prime': self.p_prime,
320                         'pressure': self.pressure, 'q': self.q,
321                         'rho': self.rho}
322 
323         self.set_splines(param_points)
324 
325         b_theta_prime = self.splines['b_theta'].derivative()
326         b_theta_prime_prime = b_theta_prime.derivative()
327         b_z_prime = self.splines['b_z'].derivative()
328         q_prime = self.splines['q'].derivative()
329 
330         self.splines.update({'b_theta_prime': b_theta_prime,
331                              'b_theta_prime_prime': b_theta_prime_prime,
332                              'b_z_prime': b_z_prime, 'q_prime': q_prime})
333 
334 
335         self.tck_splines = self.convert_spline_objects_to_tck(self.splines)
336 
337     def get_j_z(self, r):
338         r"""
339         Return zero current.
340         """
341         r = np.asarray(r)
342         return np.zeros(r.size)
343 
344     def b_theta(self, r):
345         r"""
```

```
346         Return b_theta current profile as given in Newcomb's paper.
347         """
348         return self.b_thetai*self.r_0i/r
349
350     def b_z(self, r):
351         r"""
352         Return constant b_z field.
353         """
354         return np.ones(r.size)*self.b_z0
355
356     def p_prime(self, r):
357         r"""
358         Return pressure_prime profile.
359         """
360         r = np.asarray(r)
361         return np.zeros(r.size)
362
363     def pressure(self, r):
364         r"""
365         Return pressure.
366         """
367         r = np.asarray(r)
368         return np.ones(r.size)
369
370
371 class SmoothedCoreSkin(EquilSolver):
372     r"""
373     Creates splines describing a smooth skin and core current profile.
374     """
375     def __init__(self, points_core=20, points_transition=50, points_skin=20,
376                  core_radius=0.7, transition_width=0.1, skin_width=0.1, k=1.,
377                  j_core=0.1, epsilon=0.3, lambda_bar=0.5, mu_0=1., q0=1.1,
378                  b_z0=0.1, determinator='j_core'):
379         r"""
380         Initialize parameters defining smooth skin and core profile
381         and create splines.
382         """
383         self.mu_0 = mu_0
384         self.q0 = q0
385
386         self.points_core = points_core
387         self.points_transition = points_transition
388         self.points_skin = points_skin
389         self.core_radius = core_radius
390         self.transition_width = transition_width
391         self.skin_width = skin_width
392
393         mask = np.ones(points_transition + 2, dtype=bool)
394         mask[[0, -1]] = False
395         self.r1 = np.linspace(0., core_radius, points_core)
396         r2 = np.linspace(core_radius, core_radius + transition_width,
397                          points_transition + 2)
398         self.r2 = r2[mask]
399         self.r3 = np.linspace(core_radius + transition_width,
400                               core_radius + transition_width + skin_width,
401                               points_skin)
402         r4 = np.linspace(core_radius + transition_width + skin_width,
```

```
403                         core_radius + 2*transition_width + skin_width,
404                         points_transition + 2)
405         self.r4 = r4[mask]
406         self.r = np.concatenate((self.r1, self.r2, self.r3, self.r4))
407 
408         self.k = k
409         self.epsilon = epsilon
410         self.lambda_bar = lambda_bar
411 
412         if determinator == 'q0':
413             self.b_z0 = b_z0
414             self.j_core = self.get_j_z_core()
415             self.ratio = self.get_ratio()
416             self.j_skin = j_core*self.ratio
417             self.j_skin = self.get_j_z_skin()
418 
419         if determinator == 'j_core':
420             self.j_core = j_core
421             self.ratio = self.get_ratio()
422             self.j_skin = j_core*self.ratio
423             self.current = self.get_current()
424             self.b_z0 = self.get_b_z()
425 
426         param_points = OrderedDict([('j_z', self.j_z),
427                                     ('b_theta', self.b_theta),
428                                     ('b_z', self.b_z),
429                                     ('p_prime', self.p_prime),
430                                     ('pressure', self.pressure),
431                                     ('q', self.q),
432                                     ('rho', self.rho)])
433 
434         self.set_splines(param_points)
435 
436     def smooth(self, x1, x2, g1, g2, x):
437         """
438         Smoothing method by Alan Glasser.
439         """
440         delta_x = (x2 - x1) / 2.
441         x_bar = (x2 + x1) / 2.
442         delta_g = (g2 - g1) / 2.
443         g_bar = (g1 + g2) / 2.
444         z = (x - x_bar) / delta_x
445         return g_bar + self.smooth_f(z)*delta_g
446 
447     def smooth_f(self, z):
448         r"""
449         Smoothing polynominal by Alan Glasser.
450         """
451         return z/8.*(3.*z**4 - 10.*z**2 + 15.)
452 
453     def get_j_z_core(self):
454         r"""
455         """
456         return (2.*self.b_z0*self.k/(self.mu_0*self.q0))
457 
458     def get_ratio(self):
459         r"""
```

```
460         Returs j_z_skin based on j_z_core, geometry and epsilon of pinch.
461         """
462         a = self.core_radius + 2*self.transition_width + self.skin_width
463         term1 = -7.*a**2
464         term2 = 7.*a*self.skin_width
465         term3 = 14.*a*self.transition_width
466         term4 = 7.*a**2*self.epsilon
467         term5 = -14.*a*self.skin_width*self.epsilon
468         term6 = 7.*self.skin_width**2*self.epsilon
469         term7 = -21.*a*self.transition_width*self.epsilon
470         term8 = 21.*self.skin_width*self.transition_width*self.epsilon
471         term9 = 16.*self.transition_width**2*self.epsilon
472         denominator = (7.*(2.*a - self.skin_width -
473                         2.*self.transition_width) *
474                         (self.skin_width + self.transition_width) *
475                         self.epsilon)
476         return -(term1 + term2 + term3 + term4 + term5 + term6 +
477                  term7 + term8 + term9) / denominator
478
479     def get_current(self):
480         r"""
481         """
482         a = self.core_radius + 2*self.transition_width + self.skin_width
483         term1 = 7.*a**2*self.j_core
484         term2 = -14.*a*self.skin_width*self.j_core
485         term3 = 14.*a*self.skin_width*self.j_skin
486         term4 = -21.*a*self.transition_width*self.j_core
487         term5 = 14.*a*self.transition_width*self.j_skin
488         term6 = 7.*self.skin_width**2*self.j_core
489         term7 = -7.*self.skin_width**2*self.j_skin
490         term8 = 21.*self.skin_width*self.transition_width*self.j_core
491         term9 = -21.*self.skin_width*self.transition_width*self.j_skin
492         term10 = 16.*self.transition_width**2*self.j_core
493         term11 = -14.*self.transition_width**2*self.j_skin
494         return np.pi/7.*(term1 + term2 + term3 + term4 + term5 + term6 +
495                          term7 + term8 + term9 + term10 + term11)
496
497     def get_b_z(self):
498         r"""
499         Returns b_z based on j_z, geometry and lambda_bar of pinch.
500         """
501         a = self.core_radius + 2.*self.transition_width + self.skin_width
502         return self.current*self.mu_0/(np.pi*a**2*self.lambda_bar)
503
504     def j_z(self, dummy_r):
505         r"""
506         For now always returns complete j_z.
507         """
508         total_points = (self.points_core + 2*self.points_transition +
509                         self.points_skin)
510
511         points1 = self.points_core
512         points2 = self.points_core + self.points_transition
513         points3 = self.points_core + self.points_transition + self.points_skin
514         points4 = (self.points_core + 2*self.points_transition +
515                    self.points_skin)
516
```

```
517         boundary1 = self.core_radius
518         boundary2 = self.core_radius + self.transition_width
519         boundary3 = self.core_radius + self.transition_width + self.skin_width
520         boundary4 = (self.core_radius + 2*self.transition_width +
521                      self.skin_width)
522 
523         j_z = np.zeros(total_points)
524         j_z[:points1] = self.j_core
525         j_z[points1:points2] = self.smooth(boundary1, boundary2, self.j_core,
526                                            self.j_skin,
527                                            self.r[points1:points2])
528         j_z[points2:points3] = self.j_skin
529         j_z[points3:points4] = self.smooth(boundary3, boundary4, self.j_skin,
530                                            0., self.r[points3:points4])
531         return j_z
532 
533     def b_theta(self, r):
534         r"""
535         Return b_theta at given r values.
536         """
537         b_theta_r_integrator = inte.ode(b_theta_r_prime_func)
538         b_theta_r_integrator.set_integrator('lsoda')
539         b_theta_r_integrator.set_f_params(self.splines['j_z'], self.mu_0)
540         b_theta_r_integrator.set_initial_value(0., t=0.)
541         b_theta_array = np.empty(r.size)
542         b_theta_array[0] = 0.
543         for i, position in enumerate(r[1:]):
544             if b_theta_r_integrator.successful():
545                 b_theta_r_integrator.integrate(position)
546                 b_theta_array[i+1] = (b_theta_r_integrator.y/position)
547             else:
548                 break
549         return np.array(b_theta_array)
550 
551     def b_z(self, r):
552         r"""
553         Returns constant axial magnetic field.
554         """
555         return np.ones(r.size)*self.b_z0
556 
557     def p_prime(self, r):
558         r"""
559         Return pressure_prime at given r values. To be used for integration.
560         """
561         return -self.splines['b_theta'](r)*self.splines['j_z'](r)
562 
563     def pressure(self, r):
564         r"""
565         Return pressure_prime at given r values. To be used for integration.
566         """
567         pressure_integrator = inte.ode(p_prime_func)
568         pressure_integrator.set_integrator('lsoda')
569         pressure_integrator.set_initial_value(0., 0.)
570         pressure_integrator.set_f_params(self.splines['j_z'],
571                                          self.splines['b_theta'])
572         pressure_unnorm = np.empty(r.size)
573         pressure_unnorm[0] = 0.
```

```
574             for i, position in enumerate(r[1:]):
575                 if pressure_integrator.successful():
576                     pressure_integrator.integrate(t=position)
577                     pressure_unnorm[i+1] = pressure_integrator.y
578                 else:
579                     break
580             pressure_norm = pressure_unnorm - pressure_unnorm[-1]
581             return pressure_norm
582 
583         def get_beta(self, r):
584             r"""
585             """
586             beta_numerator = 2.*self.mu_0*self.splines['pressure'](self.r)
587             beta_denominator = ((self.splines['b_z'](self.r))**2 +
588                                 (self.splines['b_theta'](self.r))**2)
589             return interp.InterpolatedUnivariateSpline(self.r, beta_numerator/beta_denominator)
590 
591         def get_beta_average(self, r):
592             pass
593 
594         def q(self, r):
595             r"""
596             Returns safety factor evaluated at points.
597             """
598             if r[0] == 0.:
599                 q0 = self.k*self.b_z0/(0.5*self.mu_0*self.j_core)
600                 q_to_return = np.ones(r.size)*q0
601                 q_to_return[1:] = (r[1:]*self.k*self.splines['b_z'](r[1:]) /
602                                    self.splines['b_theta'](r[1:]))
603             else:
604                 q_to_return = (r*self.k*self.splines['b_z'](r) /
605                                self.splines['b_theta'](r))
606             return q_to_return
607 
608 
609     class UnitlessSmoothedCoreSkin(EquilSolver):
610         r"""
611         Creates splines describing a smooth skin and core current profile.
612         """
613         def __init__(self, points_core=20, points_transition=50, points_skin=20,
614                      core_radius_norm=0.7, transition_width_norm=0.1,
615                      skin_width_norm=0.1, k_bar=1.,
616                      epsilon=0.3, lambda_bar=0.5):
617             r"""
618             Initialize parameters defining smooth skin and core profile
619             and create splines.
620             """
621             self.points_core = points_core
622             self.points_transition = points_transition
623             self.points_skin = points_skin
624             self.core_radius = core_radius_norm
625             self.transition_width = transition_width_norm
626             self.skin_width = skin_width_norm
627             self.r_bar = self.core_radius + 2*self.transition_width + self.skin_width
628             self.r = self.r_points()
629 
630             self.k_bar = k_bar
```

```
631         self.epsilon = epsilon
632         self.lambda_bar = lambda_bar
633 
634         self.splines = {}
635 
636         self.b_z0 = 1.
637         self.make_spline('b_z', self.r, self.b_z(self.r))
638 
639         self.j_core = 1.
640         self.j_skin = self.get_j_skin_norm()
641         self.make_spline('j_z', self.r, self.j_z(self.r))
642 
643         self.b_theta_integrand_array = self.b_theta_integrand(self.r)
644 
645         self.A = self.get_A()
646 
647         self.make_spline('b_theta', self.r, self.b_theta(self.r))
648 
649         self.make_spline('pressure', self.r, self.pressure(self.r))
650         self.make_spline('p_prime', self.r, self.p_prime(self.r))
651 
652         self.q_0 = self.get_q_0()
653         self.make_spline('q', self.r, self.q(self.r))
654         self.make_spline('beta', self.r, self.beta(self.r))
655 
656         self.make_spline('rho', self.r, self.rho(self.r))
657 
658         b_theta_prime = self.splines['b_theta'].derivative()
659         b_theta_prime_prime = b_theta_prime.derivative()
660         b_z_prime = self.splines['b_z'].derivative()
661         q_prime = self.splines['q'].derivative()
662 
663         self.splines.update({'b_theta_prime': b_theta_prime,
664                              'b_theta_prime_prime': b_theta_prime_prime,
665                              'b_z_prime': b_z_prime, 'q_prime': q_prime})
666 
667         self.tck_splines = self.convert_spline_objects_to_tck(self.splines)
668 
669 
670     def r_points(self):
671         r"""
672         """
673         (points_core, points_transition,
674          points_skin)                    = (self.points_core,
675                                             self.points_transition,
676                                             self.points_skin)
677         (core_radius, transition_width,
678          skin_width)                     = (self.core_radius,
679                                             self.transition_width,
680                                             self.skin_width)
681         mask1 = np.ones(points_transition + 2, dtype=bool)
682         mask2 = np.ones(points_transition + 2, dtype=bool)
683         mask1[[0, -1]] = False
684         mask2[0] = False
685         self.r1 = np.linspace(0., core_radius, points_core)
686         r2 = np.linspace(core_radius, core_radius +
687                          transition_width, points_transition + 2)
```

```
688             self.r2 = r2[mask1]
689             self.r3 = np.linspace(core_radius + transition_width,
690                                   core_radius + transition_width +
691                                   skin_width, points_skin)
692             r4 = np.linspace(core_radius + transition_width +
693                              skin_width,
694                              core_radius + 2*transition_width +
695                              skin_width, points_transition + 2)
696             self.r4 = r4[mask2]
697             r = np.concatenate((self.r1, self.r2, self.r3, self.r4))
698             return r
699
700         def make_spline(self, key, r, values):
701             r"""
702             """
703             self.splines[key] = interp.InterpolatedUnivariateSpline(r,
704                                                                     values,
705                                                                     k=3)
706         def smooth(self, x1, x2, g1, g2, x):
707             """
708             Smoothing method by Alan Glasser.
709             """
710             delta_x = (x2 - x1) / 2.
711             x_bar = (x2 + x1) / 2.
712             delta_g = (g2 - g1) / 2.
713             g_bar = (g1 + g2) / 2.
714             z = (x - x_bar) / delta_x
715             return g_bar + self.smooth_f(z)*delta_g
716
717         def smooth_f(self, z):
718             r"""
719             Smoothing polynominal by Alan Glasser.
720             """
721             return z/8.*(3.*z**4 - 10.*z**2 + 15.)
722
723         def get_j_skin_norm(self):
724             r"""
725             Returs j_z_skin based on j_z_core, geometry and epsilon of pinch.
726             """
727             (epsilon, skin_width,
728              transition_width, r_bar) = (self.epsilon, self.skin_width,
729                                          self.transition_width, self.r_bar)
730
731             term1 = 16.*skin_width**2*epsilon
732             term2 = 21.*skin_width*transition_width*epsilon
733             term3 = -21*skin_width*epsilon*r_bar
734             term4 = 14.*skin_width*r_bar
735             term5 = 7.*transition_width**2*epsilon
736             term6 = -14.*transition_width*r_bar*epsilon
737             term7 = 7.*transition_width*r_bar
738             term8 = 7.*epsilon*r_bar**2
739             term9 = -7.*r_bar**2
740             numerator = (term1 + term2 + term3 + term4 + term5 + term6 + term7 +
741                          term8 + term9)
742
743             factor1 = 7.*epsilon
744             factor2 = (skin_width + transition_width)
```

```
745         factor3 = (2.*skin_width + transition_width - 2.*r_bar)
746
747         denominator = factor1*factor2*factor3
748
749         return self.j_core*numerator/denominator
750
751     def j_z(self, dummy_r):
752         r"""
753         For now always returns complete j_z.
754         """
755         total_points = (self.points_core + 2*self.points_transition +
756                         self.points_skin)+1
757
758         points1 = self.points_core
759         points2 = self.points_core + self.points_transition
760         points3 = self.points_core + self.points_transition + self.points_skin
761         points4 = (self.points_core + 2*self.points_transition +
762                    self.points_skin)+1
763
764         boundary1 = self.core_radius
765         boundary2 = self.core_radius + self.transition_width
766         boundary3 = self.core_radius + self.transition_width + self.skin_width
767         boundary4 = (self.core_radius + 2*self.transition_width +
768                      self.skin_width)
769
770         j_z = np.zeros(total_points)
771         j_z[:points1] = self.j_core
772         j_z[points1:points2] = self.smooth(boundary1, boundary2, 1.,
773                                            self.j_skin,
774                                            self.r[points1:points2])
775         j_z[points2:points3] = self.j_skin
776         j_z[points3:points4] = self.smooth(boundary3, boundary4, self.j_skin,
777                                            0., self.r[points3:points4])
778         return j_z
779
780     def b_theta_integrand(self, r):
781         r"""
782         """
783         b_theta_r_integrator = inte.ode(b_theta_r_prime_func)
784         b_theta_r_integrator.set_integrator('lsoda')
785         b_theta_r_integrator.set_f_params(self.splines['j_z']._eval_args, 1.0)
786         b_theta_r_integrator.set_initial_value(0., t=0.)
787         b_theta_integrand_array = np.empty(r.size)
788         b_theta_integrand_array[0] = 0.
789         for i, position in enumerate(r[1:]):
790             if b_theta_r_integrator.successful():
791                 b_theta_r_integrator.integrate(position)
792                 b_theta_integrand_array[i+1] = (b_theta_r_integrator.y/position)
793             else:
794                 break
795         return b_theta_integrand_array
796
797     def get_A(self):
798         return self.lambda_bar*self.b_z0/(2*self.b_theta_integrand_array[-1])
799
800     def get_q_0(self):
801         r"""
```

```
802             """
803             return 2.*self.k_bar*self.b_z0/(self.A*self.j_core)
804
805         def b_theta(self, r):
806             r"""
807             Return b_theta at given r values.
808             """
809             b_theta_array = self.b_theta_integrand_array*self.A
810             return b_theta_array
811
812         def b_z(self, r):
813             r"""
814             Returns constant axial magnetic field.
815             """
816             return np.ones(r.size)*self.b_z0
817
818         def p_prime(self, r):
819             r"""
820             Return pressure_prime at given r values. To be used for integration.
821             """
822             return -self.B*self.splines['b_theta'](r)*self.splines['j_z'](r)
823
824         def pressure(self, r):
825             r"""
826             Return pressure_prime at given r values. To be used for integration.
827             """
828             pressure_integrator = inte.ode(p_prime_func_reverse)
829             pressure_integrator.set_integrator('lsoda')
830             pressure_integrator.set_initial_value(0., 0.)
831             pressure_integrator.set_f_params(self.splines['j_z']._eval_args,
832                                              self.splines['b_theta']._eval_args)
833             pressure_reverse = np.empty(r.size)
834             pressure_reverse[0] = 0.
835             r_reverse_diffs = np.cumsum(np.diff(self.r)[::-1])
836             for i, position in enumerate(r_reverse_diffs):
837                 if pressure_integrator.successful():
838                     pressure_integrator.integrate(t=position)
839                     pressure_reverse[i+1] = pressure_integrator.y
840                 else:
841                     break
842             pressure = pressure_reverse[::-1]
843             self.B = 1./pressure[0]
844             pressure_norm = pressure*self.B
845             return pressure_norm
846
847         def beta_0(self):
848             r"""
849             """
850             return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](0)
851
852         def beta(self, r):
853             r"""
854             """
855             return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](r)
856
857         def q(self, r):
858             r"""
```

```
859         Returns safety factor evaluated at points.
860         """
861         if r[0] == 0.:
862             q_to_return = np.ones(r.size)*self.q_0
863             q_to_return[1:] = (r[1:]*self.k_bar*self.splines['b_z'](r[1:]) /
864                                self.splines['b_theta'](r[1:]))
865         else:
866             q_to_return = (r*self.k_bar*self.splines['b_z'](r) /
867                            self.splines['b_theta'](r))
868         return q_to_return
869
870
871 class UnitlessSmoothedCoreSkinLinearBz(EquilSolver):
872     r"""
873     Creates splines describing a smooth skin and core current profile.
874     """
875     def __init__(self, points_core=20, points_transition=50, points_skin=20,
876                  core_radius_norm=0.7, transition_width_norm=0.1,
877                  skin_width_norm=0.1, k_bar=1.,
878                  epsilon=0.3, lambda_bar=0.5, b_z_factor=0.9):
879         self.points_core = points_core
880         self.points_transition = points_transition
881         self.points_skin = points_skin
882         self.core_radius = core_radius_norm
883         self.transition_width = transition_width_norm
884         self.skin_width = skin_width_norm
885         self.r_bar = self.core_radius + 2*self.transition_width + self.skin_width
886         self.r = self.r_points()
887
888         self.k_bar = k_bar
889         self.epsilon = epsilon
890         self.lambda_bar = lambda_bar
891
892         self.splines = {}
893
894         self.b_z_factor = b_z_factor
895         self.b_z0 = 1.*b_z_factor
896         self.make_spline('b_z', self.r, self.b_z(self.r))
897
898         self.j_core = 1.
899         self.j_skin = self.get_j_skin_norm()
900         self.make_spline('j_z', self.r, self.j_z(self.r))
901
902         self.b_theta_integrand_array = self.b_theta_integrand(self.r)
903
904         self.A = self.get_A()
905
906         self.make_spline('j_theta', self.r, self.j_theta(self.r))
907
908         self.make_spline('b_theta', self.r, self.b_theta(self.r))
909
910         self.make_spline('pressure', self.r, self.pressure(self.r))
911         self.make_spline('p_prime', self.r, self.p_prime(self.r))
912
913         self.q_0 = self.get_q_0()
914         self.make_spline('q', self.r, self.q(self.r))
915         self.make_spline('beta', self.r, self.beta(self.r))
```

```
916
917             self.make_spline('rho', self.r, self.rho(self.r))
918
919             b_theta_prime = self.splines['b_theta'].derivative()
920             b_theta_prime_prime = b_theta_prime.derivative()
921             b_z_prime = self.splines['b_z'].derivative()
922             q_prime = self.splines['q'].derivative()
923
924             self.splines.update({'b_theta_prime': b_theta_prime,
925                                  'b_theta_prime_prime': b_theta_prime_prime,
926                                  'b_z_prime': b_z_prime, 'q_prime': q_prime})
927
928             self.tck_splines = self.convert_spline_objects_to_tck(self.splines)
929
930
931         def r_points(self):
932             r"""
933             """
934             (points_core, points_transition,
935              points_skin)                    = (self.points_core,
936                                                 self.points_transition,
937                                                 self.points_skin)
938             (core_radius, transition_width,
939              skin_width)                     = (self.core_radius,
940                                                 self.transition_width,
941                                                 self.skin_width)
942             mask1 = np.ones(points_transition + 2, dtype=bool)
943             mask2 = np.ones(points_transition + 2, dtype=bool)
944             mask1[[0, -1]] = False
945             mask2[0] = False
946             self.r1 = np.linspace(0., core_radius, points_core)
947             r2 = np.linspace(core_radius, core_radius +
948                              transition_width, points_transition + 2)
949             self.r2 = r2[mask1]
950             self.r3 = np.linspace(core_radius + transition_width,
951                                   core_radius + transition_width +
952                                   skin_width, points_skin)
953             r4 = np.linspace(core_radius + transition_width +
954                              skin_width,
955                              core_radius + 2*transition_width +
956                              skin_width, points_transition + 2)
957             self.r4 = r4[mask2]
958             r = np.concatenate((self.r1, self.r2, self.r3, self.r4))
959             return r
960
961         def make_spline(self, key, r, values):
962             r"""
963             """
964             self.splines[key] = interp.InterpolatedUnivariateSpline(r,
965                                                                     values,
966                                                                     k=3)
967         def smooth(self, x1, x2, g1, g2, x):
968             """
969             Smoothing method by Alan Glasser.
970             """
971             delta_x = (x2 - x1) / 2.
972             x_bar = (x2 + x1) / 2.
```

```
973             delta_g = (g2 - g1) / 2.
974             g_bar = (g1 + g2) / 2.
975             z = (x - x_bar) / delta_x
976             return g_bar + self.smooth_f(z)*delta_g
977
978         def smooth_f(self, z):
979             r"""
980             Smoothing polynominal by Alan Glasser.
981             """
982             return z/8.*(3.*z**4 - 10.*z**2 + 15.)
983
984         def get_j_skin_norm(self):
985             r"""
986             Returs j_z_skin based on j_z_core, geometry and epsilon of pinch.
987             """
988             (epsilon, skin_width,
989              transition_width, r_bar) = (self.epsilon, self.skin_width,
990                                          self.transition_width, self.r_bar)
991
992             term1 = 16.*skin_width**2*epsilon
993             term2 = 21.*skin_width*transition_width*epsilon
994             term3 = -21*skin_width*epsilon*r_bar
995             term4 = 14.*skin_width*r_bar
996             term5 = 7.*transition_width**2*epsilon
997             term6 = -14.*transition_width*r_bar*epsilon
998             term7 = 7.*transition_width*r_bar
999             term8 = 7.*epsilon*r_bar**2
1000            term9 = -7.*r_bar**2
1001            numerator = (term1 + term2 + term3 + term4 + term5 + term6 + term7 +
1002                         term8 + term9)
1003
1004            factor1 = 7.*epsilon
1005            factor2 = (skin_width + transition_width)
1006            factor3 = (2.*skin_width + transition_width - 2.*r_bar)
1007
1008            denominator = factor1*factor2*factor3
1009
1010            return self.j_core*numerator/denominator
1011
1012        def j_z(self, dummy_r):
1013            r"""
1014            For now always returns complete j_z.
1015            """
1016            total_points = (self.points_core + 2*self.points_transition +
1017                            self.points_skin)+1
1018
1019            points1 = self.points_core
1020            points2 = self.points_core + self.points_transition
1021            points3 = self.points_core + self.points_transition + self.points_skin
1022            points4 = (self.points_core + 2*self.points_transition +
1023                       self.points_skin)+1
1024
1025            boundary1 = self.core_radius
1026            boundary2 = self.core_radius + self.transition_width
1027            boundary3 = self.core_radius + self.transition_width + self.skin_width
1028            boundary4 = (self.core_radius + 2*self.transition_width +
1029                         self.skin_width)
```

```
1030
1031            j_z = np.zeros(total_points)
1032            j_z[:points1] = self.j_core
1033            j_z[points1:points2] = self.smooth(boundary1, boundary2, 1.,
1034                                               self.j_skin,
1035                                               self.r[points1:points2])
1036            j_z[points2:points3] = self.j_skin
1037            j_z[points3:points4] = self.smooth(boundary3, boundary4, self.j_skin,
1038                                               0., self.r[points3:points4])
1039            return j_z
1040
1041        def b_theta_integrand(self, r):
1042            r"""
1043            """
1044            b_theta_r_integrator = inte.ode(b_theta_r_prime_func)
1045            b_theta_r_integrator.set_integrator('lsoda')
1046            b_theta_r_integrator.set_f_params(self.splines['j_z']._eval_args, 1.0)
1047            b_theta_r_integrator.set_initial_value(0., t=0.)
1048            b_theta_integrand_array = np.empty(r.size)
1049            b_theta_integrand_array[0] = 0.
1050            for i, position in enumerate(r[1:]):
1051                if b_theta_r_integrator.successful():
1052                    b_theta_r_integrator.integrate(position)
1053                    b_theta_integrand_array[i+1] = (b_theta_r_integrator.y/position)
1054                else:
1055                    break
1056            return b_theta_integrand_array
1057
1058        def get_A(self):
1059            return self.lambda_bar*self.b_z0/(2*self.b_theta_integrand_array[-1])
1060
1061        def get_q_0(self):
1062            r"""
1063            """
1064            return 2.*self.k_bar*self.b_z0/(self.A*self.j_core)
1065
1066        def j_theta(self, r):
1067            return 1./self.A*self.splines['b_z'].derivative()(r)
1068
1069        def b_theta(self, r):
1070            r"""
1071            Return b_theta at given r values.
1072            """
1073            b_theta_array = self.b_theta_integrand_array*self.A
1074            return b_theta_array
1075
1076        def b_z(self, r):
1077            r"""
1078            Returns constant axial magnetic field.
1079            """
1080            return (1.-self.b_z_factor)/(1.)*r+self.b_z_factor
1081
1082        def p_prime(self, r):
1083            r"""
1084            Return pressure_prime at given r values. To be used for integration.
1085            """
1086            return self.B*(self.splines['b_z'](r)*self.splines['j_theta'](r) -
```

```
1087                        self.splines['b_theta'](r)*self.splines['j_z'](r))
1088
1089      def pressure(self, r):
1090          r"""
1091          Return pressure_prime at given r values. To be used for integration.
1092          """
1093          pressure_integrator = inte.ode(p_prime_func_reverse_j_theta)
1094          pressure_integrator.set_integrator('lsoda')
1095          pressure_integrator.set_initial_value(0., 0.)
1096          pressure_integrator.set_f_params(self.splines['j_z']._eval_args,
1097                                           self.splines['b_theta']._eval_args,
1098                                           self.splines['j_theta']._eval_args,
1099                                           self.splines['b_z']._eval_args)
1100          pressure_reverse = np.empty(r.size)
1101          pressure_reverse[0] = 0.
1102          r_reverse_diffs = np.cumsum(np.diff(self.r)[::-1])
1103          for i, position in enumerate(r_reverse_diffs):
1104              if pressure_integrator.successful():
1105                  pressure_integrator.integrate(t=position)
1106                  pressure_reverse[i+1] = pressure_integrator.y
1107              else:
1108                  break
1109          pressure = pressure_reverse[::-1]
1110          self.B = 1./pressure[0]
1111          pressure_norm = pressure*self.B
1112          return pressure_norm
1113
1114      def beta_0(self):
1115          r"""
1116          """
1117          return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](0)
1118
1119      def beta(self, r):
1120          r"""
1121          """
1122          return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](r)
1123
1124      def q(self, r):
1125          r"""
1126          Returns safety factor evaluated at points.
1127          """
1128          if r[0] == 0.:
1129              q_to_return = np.ones(r.size)*self.q_0
1130              q_to_return[1:] = (r[1:]*self.k_bar*self.splines['b_z'](r[1:]) /
1131                                 self.splines['b_theta'](r[1:]))
1132          else:
1133              q_to_return = (r*self.k_bar*self.splines['b_z'](r) /
1134                             self.splines['b_theta'](r))
1135          return q_to_return
1136
1137
1138  class UnitlessSmoothedCoreSkinSmoothJTheta(EquilSolver):
1139      r"""
1140      Creates splines describing a smooth skin and core current profile.
1141      """
1142      def __init__(self, points_core=20, points_transition=50, points_skin=20,
1143                   core_radius_norm=0.7, transition_width_norm=0.1,
```

```
1144                    skin_width_norm=0.1, k_bar=1.,
1145                    epsilon=0.3, lambda_bar=0.5, j_theta_peak=1., j_theta_start=0., j_theta_peak_pos=0.5):
1146            self.points_core = points_core
1147            self.points_transition = points_transition
1148            self.points_skin = points_skin
1149            self.core_radius = core_radius_norm
1150            self.transition_width = transition_width_norm
1151            self.skin_width = skin_width_norm
1152            self.r_bar = self.core_radius + 2*self.transition_width + self.skin_width
1153            self.r = self.r_points()
1154
1155            self.k_bar = k_bar
1156            self.epsilon = epsilon
1157            self.lambda_bar = lambda_bar
1158
1159            self.splines = {}
1160
1161            self.b_z0 = 1.
1162
1163            self.j_core = 1.
1164            self.j_skin = self.get_j_skin_norm()
1165            self.make_spline('j_z', self.r, self.j_z(self.r))
1166
1167            self.b_theta_integrand_array = self.b_theta_integrand(self.r)
1168
1169            self.A = self.get_A()
1170
1171            self.j_theta_peak = j_theta_peak
1172            self.j_theta_start = j_theta_start
1173            self.j_theta_peak_pos = j_theta_peak_pos
1174            self.make_spline('j_theta', self.r, self.j_theta(self.r))
1175            self.make_spline('b_z', self.r, self.b_z(self.r))
1176
1177            self.make_spline('b_theta', self.r, self.b_theta(self.r))
1178
1179            self.make_spline('pressure', self.r, self.pressure(self.r))
1180            self.make_spline('p_prime', self.r, self.p_prime(self.r))
1181
1182            self.q_0 = self.get_q_0()
1183            self.make_spline('q', self.r, self.q(self.r))
1184            self.make_spline('beta', self.r, self.beta(self.r))
1185
1186            self.make_spline('rho', self.r, self.rho(self.r))
1187
1188            b_theta_prime = self.splines['b_theta'].derivative()
1189            b_theta_prime_prime = b_theta_prime.derivative()
1190            b_z_prime = self.splines['b_z'].derivative()
1191            q_prime = self.splines['q'].derivative()
1192
1193            self.splines.update({'b_theta_prime': b_theta_prime,
1194                                 'b_theta_prime_prime': b_theta_prime_prime,
1195                                 'b_z_prime': b_z_prime, 'q_prime': q_prime})
1196
1197            self.tck_splines = self.convert_spline_objects_to_tck(self.splines)
1198
1199
1200        def r_points(self):
```

```
        r"""
        """
        (points_core, points_transition,
         points_skin)                    = (self.points_core,
                                            self.points_transition,
                                            self.points_skin)
        (core_radius, transition_width,
         skin_width)                     = (self.core_radius,
                                            self.transition_width,
                                            self.skin_width)
        mask1 = np.ones(points_transition + 2, dtype=bool)
        mask2 = np.ones(points_transition + 2, dtype=bool)
        mask1[[0, -1]] = False
        mask2[0] = False
        self.r1 = np.linspace(0., core_radius, points_core)
        r2 = np.linspace(core_radius, core_radius +
                         transition_width, points_transition + 2)
        self.r2 = r2[mask1]
        self.r3 = np.linspace(core_radius + transition_width,
                              core_radius + transition_width +
                              skin_width, points_skin)
        r4 = np.linspace(core_radius + transition_width +
                         skin_width,
                         core_radius + 2*transition_width +
                         skin_width, points_transition + 2)
        self.r4 = r4[mask2]
        r = np.concatenate((self.r1, self.r2, self.r3, self.r4))
        return r

    def make_spline(self, key, r, values):
        r"""
        """
        self.splines[key] = interp.InterpolatedUnivariateSpline(r,
                                                                 values,
                                                                 k=3)

    def smooth(self, x1, x2, g1, g2, x):
        """
        Smoothing method by Alan Glasser.
        """
        delta_x = (x2 - x1) / 2.
        x_bar = (x2 + x1) / 2.
        delta_g = (g2 - g1) / 2.
        g_bar = (g1 + g2) / 2.
        z = (x - x_bar) / delta_x
        return g_bar + self.smooth_f(z)*delta_g

    def smooth_f(self, z):
        r"""
        Smoothing polynominal by Alan Glasser.
        """
        return z/8.*(3.*z**4 - 10.*z**2 + 15.)

    def get_j_skin_norm(self):
        r"""
        Returs j_z_skin based on j_z_core, geometry and epsilon of pinch.
        """
```

```
        (epsilon, skin_width,
         transition_width, r_bar) = (self.epsilon, self.skin_width,
                                     self.transition_width, self.r_bar)

        term1 = 16.*skin_width**2*epsilon
        term2 = 21.*skin_width*transition_width*epsilon
        term3 = -21*skin_width*epsilon*r_bar
        term4 = 14.*skin_width*r_bar
        term5 = 7.*transition_width**2*epsilon
        term6 = -14.*transition_width*r_bar*epsilon
        term7 = 7.*transition_width*r_bar
        term8 = 7.*epsilon*r_bar**2
        term9 = -7.*r_bar**2
        numerator = (term1 + term2 + term3 + term4 + term5 + term6 + term7 +
                     term8 + term9)

        factor1 = 7.*epsilon
        factor2 = (skin_width + transition_width)
        factor3 = (2.*skin_width + transition_width - 2.*r_bar)

        denominator = factor1*factor2*factor3

        return self.j_core*numerator/denominator

    def j_z(self, dummy_r):
        r"""
        For now always returns complete j_z.
        """
        total_points = (self.points_core + 2*self.points_transition +
                        self.points_skin)+1

        points1 = self.points_core
        points2 = self.points_core + self.points_transition
        points3 = self.points_core + self.points_transition + self.points_skin
        points4 = (self.points_core + 2*self.points_transition +
                   self.points_skin)+1

        boundary1 = self.core_radius
        boundary2 = self.core_radius + self.transition_width
        boundary3 = self.core_radius + self.transition_width + self.skin_width
        boundary4 = (self.core_radius + 2*self.transition_width +
                     self.skin_width)

        j_z = np.zeros(total_points)
        j_z[:points1] = self.j_core
        j_z[points1:points2] = self.smooth(boundary1, boundary2, 1.,
                                           self.j_skin,
                                           self.r[points1:points2])
        j_z[points2:points3] = self.j_skin
        j_z[points3:points4] = self.smooth(boundary3, boundary4, self.j_skin,
                                           0., self.r[points3:points4])
        return j_z

    def b_theta_integrand(self, r):
        r"""
        """
        b_theta_r_integrator = inte.ode(b_theta_r_prime_func)
```

```
1315            b_theta_r_integrator.set_integrator('lsoda')
1316            b_theta_r_integrator.set_f_params(self.splines['j_z']._eval_args, 1.0)
1317            b_theta_r_integrator.set_initial_value(0., t=0.)
1318            b_theta_integrand_array = np.empty(r.size)
1319            b_theta_integrand_array[0] = 0.
1320            for i, position in enumerate(r[1:]):
1321                if b_theta_r_integrator.successful():
1322                    b_theta_r_integrator.integrate(position)
1323                    b_theta_integrand_array[i+1] = (b_theta_r_integrator.y/position)
1324                else:
1325                    break
1326            return b_theta_integrand_array
1327
1328        def get_A(self):
1329            return self.lambda_bar*self.b_z0/(2*self.b_theta_integrand_array[-1])
1330
1331        def get_q_0(self):
1332            r"""
1333            """
1334            return 2.*self.k_bar*self.b_z0/(self.A*self.j_core)
1335
1336        def j_theta(self, r):
1337            peak = np.where(r>self.j_theta_peak_pos)[0][0]
1338            start = np.where(r>self.j_theta_start)[0][0]
1339            j_theta = np.zeros(r.shape)
1340            j_theta[start:peak] = self.smooth(self.j_theta_start,
1341                                              self.j_theta_peak_pos, 0.,
1342                                              self.j_theta_peak, r[start:peak])
1343            j_theta[peak:] = self.smooth(self.j_theta_peak_pos, 1.,
1344                                         self.j_theta_peak, 0, r[peak:])
1345            return j_theta
1346
1347        def b_theta(self, r):
1348            r"""
1349            Return b_theta at given r values.
1350            """
1351            b_theta_array = self.b_theta_integrand_array*self.A
1352            return b_theta_array
1353
1354        def b_z(self, r):
1355            r"""
1356            Returns constant axial magnetic field.
1357            """
1358            b_z_integrator = inte.ode(j_theta_func)
1359            b_z_integrator.set_integrator('lsoda')
1360            b_z_integrator.set_f_params(self.splines['j_theta'], 1.0)
1361            b_z_integrator.set_initial_value(0, t=0.)
1362            b_z = np.ones(r.size)*self.b_z0
1363            for i, position in enumerate(r):
1364                if b_z_integrator.successful():
1365                    b_z_integrator.integrate(position)
1366                    b_z[i] += self.A*b_z_integrator.y
1367                else:
1368                    break
1369            return b_z
1370
1371        def p_prime(self, r):
```

```
1372            r"""
1373            Return pressure_prime at given r values. To be used for integration.
1374            """
1375            return self.B*(self.splines['b_z'](r)*self.splines['j_theta'](r) -
1376                           self.splines['b_theta'](r)*self.splines['j_z'](r))
1377
1378        def pressure(self, r):
1379            r"""
1380            Return pressure_prime at given r values. To be used for integration.
1381            """
1382            pressure_integrator = inte.ode(p_prime_func_reverse_j_theta)
1383            pressure_integrator.set_integrator('lsoda')
1384            pressure_integrator.set_initial_value(0., 0.)
1385            pressure_integrator.set_f_params(self.splines['j_z']._eval_args,
1386                                             self.splines['b_theta']._eval_args,
1387                                             self.splines['j_theta']._eval_args,
1388                                             self.splines['b_z']._eval_args)
1389            pressure_reverse = np.empty(r.size)
1390            pressure_reverse[0] = 0.
1391            r_reverse_diffs = np.cumsum(np.diff(self.r)[::-1])
1392            for i, position in enumerate(r_reverse_diffs):
1393                if pressure_integrator.successful():
1394                    pressure_integrator.integrate(t=position)
1395                    pressure_reverse[i+1] = pressure_integrator.y
1396                else:
1397                    break
1398            pressure = pressure_reverse[::-1]
1399            self.B = 1./pressure[0]
1400            pressure_norm = pressure*self.B
1401            return pressure_norm
1402
1403        def beta_0(self):
1404            r"""
1405            """
1406            return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](0)
1407
1408        def beta(self, r):
1409            r"""
1410            """
1411            return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](r)
1412
1413        def q(self, r):
1414            r"""
1415            Returns safety factor evaluated at points.
1416            """
1417            if r[0] == 0.:
1418                q_to_return = np.ones(r.size)*self.q_0
1419                q_to_return[1:] = (r[1:]*self.k_bar*self.splines['b_z'](r[1:]) /
1420                                   self.splines['b_theta'](r[1:]))
1421            else:
1422                q_to_return = (r*self.k_bar*self.splines['b_z'](r) /
1423                               self.splines['b_theta'](r))
1424            return q_to_return
1425
1426
1427    class UnitlessExponentialDecaySkin(UnitlessSmoothedCoreSkin):
1428        r"""
```

```
1429        Creates splines describing a smooth skin and core current profile.
1430        """
1431        def __init__(self, points_core=20, points_skin=20, k_bar=1.,
1432                     epsilon=0.3, lambda_bar=0.5):
1433            r"""
1434            Initialize parameters defining smooth skin and core profile
1435            and create splines.
1436            """
1437            self.points_core = points_core
1438            self.points_skin = points_skin
1439            self.core_radius = 0.6
1440            self.skin_width = 0.4
1441            self.r_bar = self.core_radius + self.skin_width
1442            self.r = self.r_points()
1443
1444            self.k_bar = k_bar
1445            self.epsilon = epsilon
1446            self.lambda_bar = lambda_bar
1447
1448            self.splines = {}
1449
1450            self.b_z0 = 1.
1451            self.make_spline('b_z', self.r, self.b_z(self.r))
1452
1453            self.j_core = 1.
1454            self.j_skin = self.get_j_skin_norm(epsilon)
1455            self.make_spline('j_z', self.r, self.j_z(self.r))
1456
1457            self.b_theta_integrand_array = self.b_theta_integrand(self.r)
1458
1459            self.A = self.get_A()
1460
1461            self.make_spline('b_theta', self.r, self.b_theta(self.r))
1462
1463            self.make_spline('pressure', self.r, self.pressure(self.r))
1464            self.make_spline('p_prime', self.r, self.p_prime(self.r))
1465
1466            self.q_0 = self.get_q_0()
1467            self.make_spline('q', self.r, self.q(self.r))
1468            self.make_spline('beta', self.r, self.beta(self.r))
1469
1470            self.make_spline('rho', self.r, self.rho(self.r))
1471
1472            b_theta_prime = self.splines['b_theta'].derivative()
1473            b_theta_prime_prime = b_theta_prime.derivative()
1474            b_z_prime = self.splines['b_z'].derivative()
1475            q_prime = self.splines['q'].derivative()
1476
1477            self.splines.update({'b_theta_prime': b_theta_prime,
1478                                 'b_theta_prime_prime': b_theta_prime_prime,
1479                                 'b_z_prime': b_z_prime, 'q_prime': q_prime})
1480
1481            self.tck_splines = self.convert_spline_objects_to_tck(self.splines)
1482
1483        def r_points(self):
1484            r"""
1485            """
```

```
        (points_core,
         points_skin) = (self.points_core,
                         self.points_skin)
        (core_radius,
         skin_width) = (self.core_radius,
                        self.skin_width)
        self.r1 = np.linspace(0., core_radius, points_core, endpoint=False)
        self.r2 = np.linspace(core_radius, core_radius + skin_width,
                              points_skin)
        #print('r_points_build', core_radius, poin)
        r = np.concatenate((self.r1, self.r2))
        return r

    def get_j_skin_norm(self, epsilon,
                        i_core=0.806207671074,
                        slope=1.0053096491487337,
                        offset=0.18521395596186707):
        r"""
        Returns j_z_skin based on j_z_core, geometry and epsilon of pinch.
        """
        return (i_core/epsilon - i_core - offset) / slope

    def j_z(self, dummy_r):
        r"""

        """
        r = self.r
        (core_radius,
         skin_width) = (self.core_radius,
                        self.skin_width)
        skin_peak = skin_width/2
        skin_peak_radius = core_radius + skin_peak
        radius = core_radius + skin_width
        core = self.j_core * np.exp(-2*r**2)
        skin = interp.BPoly.from_derivatives([core_radius,
                                              skin_peak_radius,
                                              radius],
                                             [[np.exp(-2*core_radius**2),
                                             -4*core_radius*np.exp(-2*core_radius**2)],
                                             [self.j_skin, 0.], [0.,0.]])
        j_z = core
        indexes = np.where(r >= core_radius)
        j_z[indexes] = skin(r[indexes])
        return j_z


class UnitlessTanhCoreSkin(EquilSolver):
    r"""
    Creates splines describing a smooth skin and core current profile.
    """
    def __init__(self, points_core=100, points_transition=100, points_skin=100,
                 core_radius_norm=0.7, transition_width_norm=0.1,
                 skin_width_norm=0.1, k_bar=1.,
                 epsilon=0.3, lambda_bar=0.5):
        r"""
        Initialize parameters defining smooth skin and core profile
        and create splines.
```

```
1543            """
1544            self.points_core = points_core
1545            self.points_transition = points_transition
1546            self.points_skin = points_skin
1547            self.core_radius = core_radius_norm
1548            self.transition_width = transition_width_norm
1549            self.skin_width = skin_width_norm
1550            self.r_bar = self.core_radius + 2*self.transition_width + self.skin_width
1551            self.r = self.r_points()
1552
1553            self.k_bar = k_bar
1554            self.epsilon = epsilon
1555            self.lambda_bar = lambda_bar
1556
1557            self.splines = {}
1558
1559            self.b_z0 = 1.
1560            self.make_spline('b_z', self.r, self.b_z(self.r))
1561
1562            self.j_core = 1.
1563            self.j_skin = self.get_j_skin_norm()
1564            self.make_spline('j_z', self.r, self.j_z(self.r))
1565
1566            self.b_theta_integrand_array = self.b_theta_integrand(self.r)
1567
1568            self.A = self.get_A()
1569
1570            self.make_spline('b_theta', self.r, self.b_theta(self.r))
1571
1572            self.make_spline('pressure', self.r, self.pressure(self.r))
1573            self.make_spline('p_prime', self.r, self.p_prime(self.r))
1574
1575            self.q_0 = self.get_q_0()
1576            self.make_spline('q', self.r, self.q(self.r))
1577            self.make_spline('beta', self.r, self.beta(self.r))
1578
1579            self.make_spline('rho', self.r, self.rho(self.r))
1580
1581
1582        def r_points(self):
1583            r"""
1584            """
1585            (points_core, points_transition,
1586             points_skin)                    = (self.points_core,
1587                                                self.points_transition,
1588                                                self.points_skin)
1589            (core_radius, transition_width,
1590             skin_width)                     = (self.core_radius,
1591                                                self.transition_width,
1592                                                self.skin_width)
1593            mask1 = np.ones(points_transition + 2, dtype=bool)
1594            mask2 = np.ones(points_transition + 2, dtype=bool)
1595            mask1[[0, -1]] = False
1596            mask2[0] = False
1597            self.r1 = np.linspace(0., core_radius, points_core)
1598            r2 = np.linspace(core_radius, core_radius +
1599                             transition_width, points_transition + 2)
```

```
1600            self.r2 = r2[mask1]
1601            self.r3 = np.linspace(core_radius + transition_width,
1602                                  core_radius + transition_width +
1603                                  skin_width, points_skin)
1604            r4 = np.linspace(core_radius + transition_width +
1605                             skin_width,
1606                             core_radius + 2*transition_width +
1607                             skin_width, points_transition + 2)
1608            self.r4 = r4[mask2]
1609            r = np.concatenate((self.r1, self.r2, self.r3, self.r4))
1610            return r
1611
1612        def make_spline(self, key, r, values):
1613            r"""
1614            """
1615            self.splines[key] = interp.InterpolatedUnivariateSpline(r,
1616                                                                    values,
1617                                                                    k=3)
1618        def smooth(self, x1, x2, g1, g2, x):
1619            """
1620            Smoothing method by Alan Glasser.
1621            """
1622            delta_x = (x2 - x1) / 2.
1623            x_bar = (x2 + x1) / 2.
1624            delta_g = (g2 - g1) / 2.
1625            g_bar = (g1 + g2) / 2.
1626            z = (x - x_bar) / delta_x
1627            return g_bar + self.smooth_f(z)*delta_g
1628
1629        def smooth_f(self, z):
1630            r"""
1631            Smoothing polynominal by Alan Glasser.
1632            """
1633            return z/8.*(3.*z**4 - 10.*z**2 + 15.)
1634
1635        def get_j_skin_norm(self):
1636            r"""
1637            Returs j_z_skin based on j_z_core, geometry and epsilon of pinch.
1638            """
1639            (epsilon, skin_width,
1640             transition_width, r_bar) = (self.epsilon, self.skin_width,
1641                                         self.transition_width, self.r_bar)
1642
1643            term1 = 16.*skin_width**2*epsilon
1644            term2 = 21.*skin_width*transition_width*epsilon
1645            term3 = -21*skin_width*epsilon*r_bar
1646            term4 = 14.*skin_width*r_bar
1647            term5 = 7.*transition_width**2*epsilon
1648            term6 = -14.*transition_width*r_bar*epsilon
1649            term7 = 7.*transition_width*r_bar
1650            term8 = 7.*epsilon*r_bar**2
1651            term9 = -7.*r_bar**2
1652            numerator = (term1 + term2 + term3 + term4 + term5 + term6 + term7 +
1653                         term8 + term9)
1654
1655            factor1 = 7.*epsilon
1656            factor2 = (skin_width + transition_width)
```

```
1657            factor3 = (2.*skin_width + transition_width - 2.*r_bar)
1658
1659            denominator = factor1*factor2*factor3
1660
1661            return self.j_core*numerator/denominator
1662
1663        def j_z(self, dummy_r):
1664            r"""
1665            For now always returns complete j_z.
1666            """
1667            total_points = (self.points_core + 2*self.points_transition +
1668                            self.points_skin)+1
1669
1670            points1 = self.points_core
1671            points2 = self.points_core + self.points_transition
1672            points3 = self.points_core + self.points_transition + self.points_skin
1673            points4 = (self.points_core + 2*self.points_transition +
1674                       self.points_skin)+1
1675
1676            boundary1 = self.core_radius
1677            boundary2 = self.core_radius + self.transition_width
1678            boundary3 = self.core_radius + self.transition_width + self.skin_width
1679            boundary4 = (self.core_radius + 2*self.transition_width +
1680                         self.skin_width)
1681
1682            j_z = np.zeros(total_points)
1683            j_z[:points1] = self.j_core
1684            j_z[points1:points2] = np.tanh((self.r[points1:points2]-boundary1)/(self.transition_width)*4*np.pi-2*np.pi)*(self.j_skin - self.
                      j_core)/2. + 1. *(self.j_skin - self.j_core)/2. + self.j_core
1685            j_z[points2:points3] = self.j_skin
1686            j_z[points3:points4] = np.tanh((self.r[points3:points4]-boundary3)/(self.transition_width)*4*np.pi-2*np.pi)*(0. - self.j_skin)
                      /2. + 1. *(0. - self.j_skin)/2. + self.j_skin
1687            return j_z
1688
1689        def b_theta_integrand(self, r):
1690            r"""
1691            """
1692            b_theta_r_integrator = inte.ode(b_theta_r_prime_func)
1693            b_theta_r_integrator.set_integrator('lsoda')
1694            b_theta_r_integrator.set_f_params(self.splines['j_z'], 1.0)
1695            b_theta_r_integrator.set_initial_value(0., t=0.)
1696            b_theta_integrand_array = np.empty(r.size)
1697            b_theta_integrand_array[0] = 0.
1698            for i, position in enumerate(r[1:]):
1699                if b_theta_r_integrator.successful():
1700                    b_theta_r_integrator.integrate(position)
1701                    b_theta_integrand_array[i+1] = (b_theta_r_integrator.y/position)
1702                else:
1703                    break
1704            return b_theta_integrand_array
1705
1706        def get_A(self):
1707            return self.lambda_bar*self.b_z0/(2*self.b_theta_integrand_array[-1])
1708
1709        def get_q_0(self):
1710            r"""
1711            """
```

```
1712            return 2.*self.k_bar*self.b_z0/(self.A*self.j_core)
1713
1714        def b_theta(self, r):
1715            r"""
1716            Return b_theta at given r values.
1717            """
1718            b_theta_array = self.b_theta_integrand_array*self.A
1719            return b_theta_array
1720
1721        def b_z(self, r):
1722            r"""
1723            Returns constant axial magnetic field.
1724            """
1725            return np.ones(r.size)*self.b_z0
1726
1727        def p_prime(self, r):
1728            r"""
1729            Return pressure_prime at given r values. To be used for integration.
1730            """
1731            return -self.B*self.splines['b_theta'](r)*self.splines['j_z'](r)
1732
1733        def pressure(self, r):
1734            r"""
1735            Return pressure_prime at given r values. To be used for integration.
1736            """
1737            pressure_integrator = inte.ode(p_prime_func_reverse)
1738            pressure_integrator.set_integrator('lsoda')
1739            pressure_integrator.set_initial_value(0., 0.)
1740            pressure_integrator.set_f_params(self.splines['j_z'],
1741                                             self.splines['b_theta'])
1742            pressure_reverse = np.empty(r.size)
1743            pressure_reverse[0] = 0.
1744            r_reverse_diffs = np.cumsum(np.diff(self.r)[::-1])
1745            for i, position in enumerate(r_reverse_diffs):
1746                if pressure_integrator.successful():
1747                    pressure_integrator.integrate(t=position)
1748                    pressure_reverse[i+1] = pressure_integrator.y
1749                else:
1750                    print('integration failed')
1751                    break
1752            pressure = pressure_reverse[::-1]
1753            self.B = 1./pressure[0]
1754            pressure_norm = pressure*self.B
1755            return pressure_norm
1756
1757        def beta_0(self):
1758            r"""
1759            """
1760            return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](0)
1761
1762        def beta(self, r):
1763            r"""
1764            """
1765            return 2.*self.A/(self.b_z0**2*self.B)*self.splines['pressure'](r)
1766
1767        def q(self, r):
1768            r"""
```

```
1769            Returns safety factor evaluated at points.
1770            """
1771            if r[0] == 0.:
1772                q_to_return = np.ones(r.size)*self.q_0
1773                q_to_return[1:] = (r[1:]*self.k_bar*self.splines['b_z'](r[1:]) /
1774                                   self.splines['b_theta'](r[1:]))
1775            else:
1776                q_to_return = (r*self.k_bar*self.splines['b_z'](r) /
1777                               self.splines['b_theta'](r))
1778            return q_to_return
1779
1780
1781    class HardCoreZPinch(EquilSolver):
1782        r"""
1783        Create profiles for the Hard-Core Z-pinch as described in Freidberg Ideal
1784        MHD.
1785        """
1786        def __init__(self, i_c=0.1, i_p=0.2, r_c=0.1, r_a=1.0, k=1.0, points=500,
1787                     mu_0=1.):
1788            self.k = k
1789            self.i_c = i_c
1790            self.i_p = i_p
1791            self.r_c = r_c
1792            self.r_a = r_a
1793            self.current = i_c + i_p
1794            self.k_p = 3./2.*(5./3.)**(5./2.)
1795            self.k_i = mu_0*((i_c + i_p)/(2.*np.pi*r_c))**2
1796            self.mu_0 = mu_0
1797
1798            self.r = np.linspace(r_c, r_a, points)
1799
1800            param_points = {'j_z': self.j_z,
1801                            'b_theta': self.b_theta,
1802                            'b_z': self.b_z,
1803                            'p_prime': self.p_prime,
1804                            'pressure': self.pressure,
1805                            'q': self.q,
1806                            'rho': self.rho,
1807                            'stability': self.stability_criterion}
1808
1809            self.set_splines(param_points)
1810
1811        def b_z(self, r):
1812            r"""
1813            Returns axial field.
1814            """
1815            r = np.asarray(r)
1816            return np.zeros(r.size)
1817
1818        def b_theta(self, r):
1819            r"""
1820            Returns azimuthal field.
1821            """
1822            r = np.asarray(r)
1823            x = r**2 / self.r_c**2
1824            b_theta = np.sqrt(self.k_i/x - 2.*self.mu_0*self.k_p*(9.*x-5.) /
1825                              (3*x*x**(3./2.)))
```

```
1826            return b_theta
1827
1828        def pressure(self, r):
1829            r"""
1830            Returns pressure.
1831            """
1832            r = np.asarray(r)
1833            x = r**2 / self.r_c**2
1834            return self.k_p*(x - 1.)/x**(5./2.)
1835
1836        def p_prime(self, r):
1837            r"""
1838            Returns derivative of pressure.
1839            """
1840            r = np.asarray(r)
1841            r_sym, r_c, k_p = sp.symbols('r r_c K_p')
1842            x = r_sym**2 / self.r_c**2
1843            pressure_sym = k_p*(x - 1)/x**(2.5)
1844            p_prime_func = sp.lambdify((r_sym, r_c, k_p),
1845                                       sp.diff(pressure_sym, r_sym), modules=np)
1846            return p_prime_func(r, self.r_c, self.k_p)
1847
1848        def j_z(self, r):
1849            r"""
1850            Returns axial current.
1851            """
1852            r = np.asarray(r)
1853
1854            r_sym, r_c, k_p, k_i = sp.symbols('r r_c K_p K_i')
1855            x = r_sym**2 / r_c**2
1856            b_theta = sp.sqrt(k_i/x - 2.*k_p*(9.*x - 5.)/(3*x*x**(1.5)))
1857            j_z_sym = sp.diff(r_sym*b_theta, r_sym)/(r_sym*self.mu_0)
1858            j_z_func = sp.lambdify((r_sym, r_c, k_p, k_i), j_z_sym,
1859                                   modules=np)
1860            return j_z_func(r, self.r_c, self.k_p, self.k_i)
1861
1862        def beta(self, r):
1863            r"""
1864            Returns Beta
1865            """
1866            r = np.asarray(r)
1867            return 32./2.*np.pi**2*self.r_c**2/(self.mu_0*self.current)*self.k_p
1868
1869        def stability_criterion(self, r):
1870            r"""
1871            Returns stability criterion. If array is  greater than 0 any where the
1872            profile is unstable.
1873            """
1874            r = np.asarray(r)
1875            x_sym, r_c, current, k_p = sp.symbols('x r_c I K_p')
1876            beta = 32./3.*sp.pi**2*r_c**2/(self.mu_0*current**2)*k_p
1877            b_theta_norm_sq = 1. - beta/4. * (9.*x_sym - 5.)/x_sym**(1.5)
1878            stab_sym = sp.diff(b_theta_norm_sq/x_sym**(0.5), x_sym)
1879            stab_func = sp.lambdify((x_sym, r_c, k_p, current), stab_sym,
1880                                    modules=np)
1881            return stab_func(r**2/self.r_c**2, self.r_c, self.k_p, self.current)
1882
```

```
1883
1884 class SharpCoreSkin(EquilSolver):
1885     pass
1886
1887
1888 def b_theta_r_prime_func(r, y, j_z, mu_0):
1889     r"""
1890     Return b_theta_r_prime at given r values. To be used for integration.
1891     """
1892     r_arr = np.asarray(r)
1893     r_arr = atleast_1d(r_arr).ravel()
1894     return splev(r_arr, j_z)*r_arr*mu_0
1895
1896
1897 def p_prime_func(r, y, j_z, b_theta):
1898     r"""
1899     Return pressure_prime at given r values. To be used for integration.
1900     """
1901     return -splev(r, b_theta)*splev(r, j_z)
1902
1903 def p_prime_func_reverse_j_theta(r, y, j_z, b_theta, j_theta, b_z):
1904     r"""
1905     Return negative pressure_prime at given r values. To be used for reverse integration.
1906     """
1907     r_arr = np.asarray(r)
1908     r_arr = atleast_1d(r_arr).ravel()
1909     return splev(1.-r_arr, b_theta)*splev(1.-r_arr, j_z) - splev(1.-r_arr, b_z)*splev(1.-r_arr, j_theta)
1910
1911 def p_prime_func_reverse(r, y, j_z, b_theta):
1912     r"""
1913     Return negative pressure_prime at given r values. To be used for reverse integration.
1914     """
1915     r_arr = np.asarray(r)
1916     r_arr = atleast_1d(r_arr).ravel()
1917     return splev(1.-r_arr, b_theta)*splev(1.-r_arr, j_z)
1918
1919 def j_theta_func(r, y, j_theta_spline):
1920     return j_theta_spline(r)
```

A.6.6 newcomb.py

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sun Feb 08 15:36:58 2015
4
5  @author: Jens von der Linden
6  """
7
8  from __future__ import print_function
9  from __future__ import division
10 from __future__ import absolute_import
11 from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
12                               int, map, next, oct, open, pow, range, round,
13                               str, super, zip)
```

```
14  """Python 3.x compatibility"""
15
16  import sys
17  from scipy.interpolate import splev
18
19  import numpy as np
20  from numpy import atleast_1d
21  import scipy.integrate
22
23  import newcomb_init as init
24  import singularity_frobenius as frob
25  import find_singularties as find_sing
26  import external_stability as ext
27  import newcomb_f as new_f
28  import newcomb_g as new_g
29
30
31
32  def stability(params, offset, suydam_offset, suppress_output=False,
33                method='lsoda', rtol=None, max_step=None, nsteps=None,
34                xi_given=[0., 1.], diagnose=False, sing_search_points=10000,
35                f_func=new_f.newcomb_f_16, g_func=new_g.newcomb_g_18_dimless_wo_q,
36                skip_external_stability=False, stiff=False, use_jac=True,
37                adapt_step_size=False, adapt_min_steps=500):
38      r"""
39      Determine external stability.
40
41      Parameters
42      ----------
43      params: dict
44          equilibrium parameters including spline coefficients
45      offset : float
46          offset after which to start integrating after singularties
47      suydam_offset : float
48          offset after which to start integrating after suydam unstable
49          singularties
50      suppress_output: boolean
51          flag to suppress diagnostic print statements
52      method: string
53          integration method to use. Either an integrator in scipy.integate.ode
54          or 'odeint' for scipy.integrate.odeint
55      rtol : float
56          passed to ode solver relative tolerance setting for ODE integrator
57      max_step: float
58          option passed to ode solver, limit for max step size
59      nsteps: int
60          option passed to ode solder, maximum number of steps allowed during call
61          to solver.
62      xi_given: tuple of floats
63          Initial condition used for xi if equilibrim does not start ar r=0
64      diagnose: bool
65          flag to print out more diagnostc statements during integration
66      sing_search_points: int
67          number of points to divide domain by in search for singularities
68      f_func: function
69          python function to use to calculate f
70      g_func: function
```

```
71          python function to use to calculate given
72      skip_external_stability: bool
73          flag to skip external stability
74      stiff: bool
75          flag to indicate stiff equation to integrator
76      use_jac: bool
77          flag to have integrator use Jacobian
78      adapt_step_size: bool
79          flag to use adaptive step size integrator
80      adapt_min_steps: int
81          specify minimum number of steps
82      Returns
83      -------
84      stable_external : bool
85          True if delta_w > 0
86      suydam_stable : bool
87          True if all singularties (except r=0) are Suydam stable
88      delta_w : float
89          total perturbed potential energy
90      missing_end_params:
91          diagnostic ouput no longer used.
92      xi : ndarray
93          xi at interval boundary (only last element is relevant)
94      xi_der : ndarray
95          derivative of xi at interval boundary (only last element is relevant)
96      r_array : ndarray
97          array of r values at which xi is integrated
98      Notes
99      -----
100     Examines the equilibrium. If the equilibrium has a singularity, the
101     Frobenius method is used to determine a small solution at an r > than
102     instability. If the singularity is suydam unstable no attempt is made to
103     calulate external stability.
104     If there is no Frobenius instability power series solution
105     close to r=0 is chosen or if the integration does not start at r=0 a given
106     xi is used as boundary condition.
107     Only the last interval is integrated.
108     To save time xi and xi_der are only evaluated at r=a (under the hood the
109     integrator is evaluating xi and xi_der across the interval).
110     Xi and xi_der are plugged into the potential energy equation to determine
111     stability.
112     """
113     params.update({'f_func': f_func, 'g_func': g_func})
114     missing_end_params = None
115
116     if params['m'] == -1:
117         sing_params = {'a': params['r_0'], 'b': params['a'],
118                        'points': sing_search_points, 'k': params['k'],
119                        'm': params['m'], 'b_z_spl': params['b_z'],
120                        'b_z_prime_spl': params['b_z_prime'],
121                        'b_theta_spl': params['b_theta'],
122                        'b_theta_prime_spl': params['b_theta_prime'],
123                        'p_prime_spl': params['p_prime'], 'offset': offset,
124                        'tol': 1E-2, 'beta_0': params['beta_0']}
125         (interval,
126          starts_with_sing,
127          suydam_stable,
```

```
128             suydam_unstable_interval) = intervals_with_singularties(suppress_output,
129                                                                     **sing_params)
130         else:
131             suydam_stable = True
132             starts_with_sing = False
133             suydam_unstable_interval = False
134             interval = [params['r_0'], params['a']]
135
136         interval, init_value = setup_initial_conditions(interval, starts_with_sing,
137                                                         offset, suydam_offset,
138                                                         xi_given=xi_given,
139                                                         **params)
140         if not suydam_unstable_interval:
141             if skip_external_stability:
142                 (xi, xi_der, r_array) = newcomb_int(params, interval,
143                                                     init_value, method,
144                                                     diagnose, max_step,
145                                                     nsteps, rtol,
146                                                     skip_external_stability=True,
147                                                     stiff=stiff,
148                                                     use_jac=use_jac,
149                                                     adapt_step_size=adapt_step_size,
150                                                     adapt_min_steps=adapt_min_steps)
151                 return xi, xi_der
152
153             (stable_external, delta_w,
154              missing_end_params, xi, xi_der,
155              r_array) = newcomb_int(params, interval, init_value, method,
156                                     diagnose, max_step, nsteps, rtol,
157                                     stiff=stiff, use_jac=use_jac,
158                                     adapt_step_size=adapt_step_size,
159                                     adapt_min_steps=adapt_min_steps)
160         else:
161             msg = ("Last singularity is suydam unstable. " +
162                    "Unable to deterime external instability at k = %.3f."
163                    % params['k'])
164             print(msg)
165             delta_w = None
166             stable_external = None
167             xi = np.asarray([np.nan])
168             xi_der = np.asarray([np.nan])
169             r_array = np.asarray([np.nan])
170         return (stable_external, suydam_stable, delta_w, missing_end_params, xi,
171                 xi_der, r_array)
172
173
174     def newcomb_der(r, y, k, m, b_z_spl, b_z_prime_spl, b_theta_spl,
175                     b_theta_prime_spl, p_prime_spl, q_spl, q_prime_spl,
176                     f_func, g_func, beta_0):
177         r"""
178         Returns derivatives of Newcomb's Euler-Lagrange equation expressed as a set
179         of 2 first order ODEs.
180
181         Parameters
182         ----------
183         r : float
184             radius for which to find derivative
```

```
    y : ndarray (2)
        values of :math:`\xi` and :math:`f \xi'`
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z_spl : scipy spline tck tuple
        axial magnetic field
    b_theta_spl : scipy spline tck tuple
        azimuthal magnetic field
    b_theta_prime_spl: scipy spline tck tuple
        radial derivative of azimuthal magnetic field
    p_prime_spl : scipy spline tck tuple
        derivative of pressure
    q_spl : scipy spline tck tuple
        safety factor
    f_func : function
        function which returns f of Newcomb's Euler-Lagrange equation
    g_func : function
        function which returns f of Newcomb's Euler-Lagrange equation
    beta_0 : float
        pressure ratio on axis

    Returns
    -------
    y_prime : ndarray of floats (2)
        derivatives of y

    Notes
    -----
    The system of ODEs representing the Euler-Lagrange equations is

    .. math::

       \frac{d \xi}{dr} &= \xi' \\
       \frac{d (f \xi')}{dr} &= g \xi
    Reference
    ---------
    Newcomb (1960) Hydromagnetic Stability of a diffuse linear pinch.
    """
    y_prime = np.zeros(2)

    r_arr = np.asarray(r)
    r_arr = atleast_1d(r_arr).ravel()

    g_params = {'r': r, 'k': k, 'm': m, 'b_z': splev(r_arr, b_z_spl),
                'b_theta': splev(r_arr, b_theta_spl),
                'p_prime': splev(r_arr, p_prime_spl),
                'beta_0': beta_0}

    f_params = {'r': r, 'k': k, 'm': m, 'b_z': splev(r_arr, b_z_spl),
                'b_theta': splev(r_arr, b_theta_spl), 'q': splev(r_arr, q_spl)}

    y_prime[0] = y[1] / f_func(**f_params)
    y_prime[1] = y[0]*g_func(**g_params)
    return y_prime

```

```
242
243 def newcomb_jac(r, y, k, m, b_z_spl, b_z_prime_spl, b_theta_spl,
244                 b_theta_prime_spl, p_prime_spl, q_spl, q_prime_spl,
245                 f_func, g_func, beta_0):
246     r"""
247     Jacobian of Newcomb's Euler-Lagrange equation.
248     """
249     r_arr = np.asarray(r)
250     r_arr = atleast_1d(r_arr).ravel()
251     g_params = {'r': r, 'k': k, 'm': m, 'b_z': splev(r_arr, b_z_spl),
252                 'b_theta': splev(r_arr, b_theta_spl),
253                 'p_prime': splev(r_arr, p_prime_spl),
254                 'beta_0': beta_0}
255
256     f_params = {'r': r, 'k': k, 'm': m, 'b_z': splev(r_arr, b_z_spl),
257                 'b_theta': splev(r_arr, b_theta_spl), 'q': splev(r_arr, q_spl)}
258
259     jac = np.zeros((2,2))
260     jac[0,1] = 1. / f_func(**f_params)
261     jac[1,0] = g_func(**g_params)
262     return jac
263
264 def newcomb_der_for_odeint(y, r, *args):
265     r"""
266     odeint uses a derivative function with y and r passed as arguments in
267     reverse order.
268     """
269     return newcomb_der(r, y, *args)
270
271
272 def divide_by_f(r, xi_der_f, k, m, b_z_spl, b_theta_spl, q_spl, f_func):
273     r"""
274     Divides :math:`y[1]=f \xi'` by f to recover :math:`\xi`.
275     """
276     r_arr = np.asarray(r)
277     r_arr = atleast_1d(r_arr).ravel()
278     f_params = {'r': r, 'k': k, 'm': m, 'b_z': splev(r_arr, b_z_spl),
279                 'b_theta': splev(r_arr, b_theta_spl), 'q': splev(r_arr, q_spl)}
280     return xi_der_f / f_func(**f_params)
281
282
283 def intervals_with_singularties(suppress_output, **sing_params):
284     r"""
285     Determines if an interval starts with a singularity, is Suydam unstable.
286     """
287     starts_with_sing = False
288     suydam_unstable_interval = False
289     suydam_stable = False
290     interval = [sing_params['a'], sing_params['b']]
291     (sings,
292      sings_wo_0, intervals) = find_sing.identify_singularties(**sing_params)
293
294     if not sings_wo_0.size == 0:
295         if not suppress_output:
296             print("Non-geometric singularties identified at r =", sings_wo_0)
297         interval = [sings_wo_0[-1], sing_params['b']]
298         starts_with_sing = True
```

```
299     # Check singularties for Suydam stability
300     suydam_result = check_suydam(sings_wo_0, **sing_params)
301     if suydam_result.size != 0:
302         suydam_stable = False
303         if sings_wo_0.size > 0 and abs(suydam_result[-1] - sings_wo_0[-1]) < 1e-8:
304             if not suppress_output:
305                 print("Profile is Suydam unstable at r =", suydam_result)
306             suydam_unstable_interval = True
307     else:
308         suydam_stable = True
309     return interval, starts_with_sing, suydam_stable, suydam_unstable_interval
310
311
312 def setup_initial_conditions(interval, starts_with_sing, offset,
313                              suydam_offset, xi_given=[0., 1.], **params):
314     r"""
315     Returns the initial condition to use for integrating an interval.
316     """
317
318     if interval[0] == 0.:
319         interval[0] += offset
320         if interval[0] > interval[1]:
321             interval[0] = interval[1]
322         init_params = dict(params)
323         r_arr = np.asarray(interval[0])
324         r_arr = atleast_1d(r_arr).ravel()
325         init_params.update({'b_z': splev(r_arr, params['b_z']),
326                             'b_theta': splev(r_arr, params['b_theta']),
327                             'q': splev(r_arr, params['q'])})
328         init_value = init.init_geometric_sing(interval[0], **init_params)
329     else:
330         if starts_with_sing:
331             if interval[0]+suydam_offset > interval[1]:
332                 suydam_offset = interval[1] - interval[0]
333             frob_params = {'offset': suydam_offset, 'b_z_spl': params['b_z'],
334                            'b_z_prime_spl': params['b_z_prime'],
335                            'b_theta_spl': params['b_theta'],
336                            'b_theta_prime_spl': params['b_theta_prime'],
337                            'q_spl': params['q'], 'f_func': new_f.newcomb_f_16,
338                            'p_prime_spl': params['p_prime'],
339                            'beta_0': params['beta_0'], 'r_sing': interval[0]}
340             xi_given = frob.sing_small_solution(**frob_params)
341             interval[0] += suydam_offset
342             init_params = dict(params)
343             r_arr = np.asarray(interval[0])
344             r_arr = atleast_1d(r_arr).ravel()
345             init_params.update({'b_z': splev(r_arr, params['b_z']),
346                                 'b_theta': splev(r_arr, params['b_theta']),
347                                 'q': splev(r_arr, params['q'])})
348             init_value = init.init_xi_given(xi_given, r_arr, **init_params)
349
350         else:
351             init_params = dict(params)
352             init_params.pop('r')
353             init_params.update({'b_z': splev(r_arr, params['b_z']),
354                                 'b_theta': splev(r_arr, params['b_theta']),
355                                 'q': splev(r_arr, params['q'])})
```

```
356             init_value = init.init_xi_given(xi_given, interval[0], **init_params)
357     return interval, init_value
358
359
360 def check_suydam(r, b_z_spl, b_z_prime_spl, b_theta_spl, b_theta_prime_spl,
361                  p_prime_spl, beta_0, **kwargs):
362     r"""
363     Return radial positions at which the Euler–Lagrange equation is singular
364     and Suydam's criterion is violated.
365
366     Parameters
367     ----------
368     r : ndarray of floats (M)
369         positions at which f=0.
370     b_z_spl : scipy spline object
371         axial magnetic field
372     b_theta_spl : scipy spline object
373         azimuthal magnetic field
374     p_prime_spl : scipy spline object
375         derivative of pressure
376     beta_0 : float
377         beta on axis
378     Returns
379     -------
380     unstable_r : ndarray of floats (N)
381         positions at which plasma column is suydam unstable
382     """
383     params = {'r': r, 'b_z_spl': b_z_spl, 'b_z_prime_spl': b_z_prime_spl,
384               'b_theta_spl': b_theta_spl,
385               'b_theta_prime_spl': b_theta_prime_spl,
386               'p_prime_spl': p_prime_spl, 'beta_0': beta_0}
387     unstable_mask = np.invert(frob.sings_suydam_stable(**params))
388     return r[unstable_mask]
389
390
391 def newcomb_int(params, interval, init_value, method, diagnose, max_step,
392                 nsteps, rtol, skip_external_stability=False, stiff=False,
393                 use_jac=True, adapt_step_size=False, adapt_min_steps=500):
394     r"""
395     Integrates newcomb's euler Lagrange equation in a given interval with lsoda
396     either with the scipy.ode object oriented interface or with scipy.odeint.
397     """
398     missing_end_params = None
399     #print('k_bar', params['k'], 'interval:', interval[0], interval[1], init_value)
400     args = (params['k'], params['m'], params['b_z'], params['b_z_prime'],
401             params['b_theta'], params['b_theta_prime'], params['p_prime'],
402             params['q'], params['q_prime'], params['f_func'], params['g_func'],
403             params['beta_0'])
404
405     if adapt_step_size:
406         interval_list = [interval]
407         max_step_list = [10.**(np.floor(np.log10(1 - params['core_radius']))-1)]
408         nsteps_for_list = 10.**(np.abs(np.log10(max_step_list[0]))+.5) * (1 - params['core_radius'])
409         nsteps_for_list = adapt_min_steps if nsteps_for_list < adapt_min_steps else nsteps_for_list
410         nsteps_list = [nsteps_for_list]
411         if interval[0] < params['core_radius']:
412             interval_list.insert(0, [interval[0], params['core_radius']])
```

```
413             interval_list[1] = [params['core_radius'], interval[1]]
414             max_step_list.insert(0, max_step)
415             nsteps_list.insert(0, nsteps)
416     else:
417         interval_list = [interval]
418         max_step_list = [max_step]
419         nsteps_list = [nsteps]
420 
421 
422     for i, interval in enumerate(interval_list):
423         max_step = max_step_list[i]
424         nsteps = nsteps_list[i]
425         if diagnose:
426             r_array = np.linspace(interval[0], interval[1], 250)
427         else:
428             r_array = np.asarray(interval)
429 
430 
431         if method == 'lsoda_odeint':
432             if 'core_radius' in params.keys():
433                 transition_points = np.asarray([params['core_radius'],
434                                                 params['core_radius'] +
435                                                 params['transition_width'],
436                                                 params['core_radius'] +
437                                                 params['transition_width'] +
438                                                 params['skin_width']])
439                 tcrit = np.asarray(transition_points[np.less(interval[0], transition_points)])
440             else:
441                 tcrit = None
442 
443             integrator_args = {}
444             if rtol is not None:
445                 integrator_args['rtol'] = rtol
446             if nsteps is not None:
447                 integrator_args['mxstep'] = nsteps
448             if max_step is not None:
449                 integrator_args['hmax'] = max_step
450             if use_jac:
451                 results = scipy.integrate.odeint(newcomb_der_for_odeint,
452                                                  np.asarray(init_value),
453                                                  np.asarray(r_array),
454                                                  Dfun=newcomb_jac,
455                                                  tcrit=tcrit,
456                                                  args=args,
457                                                  **integrator_args)
458             else:
459                 results = scipy.integrate.odeint(newcomb_der_for_odeint,
460                                                  np.asarray(init_value),
461                                                  np.asarray(r_array),
462                                                  tcrit=tcrit,
463                                                  args=args,
464                                                  **integrator_args)
465             xi = np.asarray([results[:, 0]]).ravel()
466             xi_der_f = np.asarray([results[:, 1]]).ravel()
467         else:
468             if use_jac:
469                 integrator = scipy.integrate.ode(newcomb_der, jac=newcomb_jac)
```

```
470             else:
471                 integrator = scipy.integrate.ode(newcomb_der)
472 
473             integrator_args = {}
474             if rtol is not None:
475                 integrator_args['rtol'] = rtol
476             if nsteps is not None:
477                 integrator_args['nsteps'] = nsteps
478             if max_step is not None:
479                 integrator_args['max_step'] = max_step
480             if stiff:
481                 integrator_args['method'] = 'bdf'
482 
483             integrator.set_integrator(method, **integrator_args)
484             integrator.set_f_params(*args)
485             integrator.set_jac_params(*args)
486             integrator.set_initial_value(init_value, interval[0])
487             results = np.empty((r_array.size, 2))
488             results[0] = init_value
489             for i, r in enumerate(r_array[1:]):
490                 integrator.integrate(r)
491                 results[i+1, :] = integrator.y
492                 if not integrator.successful():
493                     results[i+1:, :] = [np.nan, np.nan]
494                     break
495             xi = results[:, 0]
496             xi_der_f = results[:, 1]
497         init_value = [xi[-1], xi_der_f[-1]]
498 
499     xi_der = divide_by_f(r_array,
500                          xi_der_f,
501                          params['k'],
502                          params['m'],
503                          params['b_z'],
504                          params['b_theta'],
505                          params['q'],
506                          params['f_func'])
507 
508     if np.all(np.isfinite(results[-1])):
509         if skip_external_stability:
510             return xi, xi_der, r_array
511         (stable_external,
512          delta_w) = ext.external_stability_from_analytic_condition(params,
513                                                                    xi[-1],
514                                                                    xi_der[-1],
515                                                                    without_sing=True,
516                                                                    dim_less=True)
517         #print(delta_w)
518     else:
519         msg = ("Integration to plasma edge did not succeed. " +
520                "Can not determine external stability at k = %.3f."
521                % params['k'])
522         print(msg)
523         missing_end_params = params
524         stable_external = None
525         delta_w = None
526     return (stable_external, delta_w, missing_end_params, xi, xi_der, r_array)
```

A.6.7 newcomb_init.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Aug 18 11:31:46 2014

@author: Jens von der Linden

Collection of init functions to set initial xi and xi prime for newcomb
integrations.
"""

from __future__ import print_function
from __future__ import division
from __future__ import absolute_import
from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
                             int, map, next, oct, open, pow, range, round,
                             str, super, zip)
"""Python 3.x compatibility"""

import numpy as np

def init_geometric_sing(r, k, m, b_z, b_theta, q, f_func, xi_factor, *args, **kwargs):
    r"""
    Return xi found from Frobenius method at a geometric singularity (i.e. r=0)

    Parameters
    ----------
    r : float
        radius
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z : float
        axial magnetic field
    b_z_prime : float
        derivative of axial magnetic field
    b_theta : float
        azimuthal magnetic field
    b_theta_prime : float
        derivative of azimuthal magnetic field
    p_prime : float
        derivative of pressure
    q : float
        safety factor
    q_prime : float
        derivative of safety factor
    f_func : function
        function used to return Newcomb's g

    Returns
```

```
51      -------
52      y : ndarray of flaots (2)
53          intial values for linear set of ODEs representing Euler-Lagrange
54          equation. :math:`y(0)=\xi'` and :math:`y(1)=f\xi'`
55
56      Notes
57      -----
58      The expressions for xi and xi_prime are from power series expansion as
59      described in the Newcomb's paper, Goedbloed and Freidberg's MHD books.
60      """
61      y = np.zeros(2)
62
63      f_params = {'r': r, 'k': k, 'm': m, 'b_z': b_z, 'b_theta': b_theta, 'q': q}
64
65      if m == 0:
66          y[0] = r
67          y[1] = 1.
68      else:
69          y[0] = r**(abs(m) - 1.)
70          y[1] = (abs(m) - 1.)*r**(abs(m) - 2.)
71          #y[0] = r**(m - 1)
72          #y[1] = (m - 1)*r**(m - 2)
73
74      y[1] = f_func(**f_params)*y[1]
75      return y*xi_factor
76
77
78  def init_xi_given(xi, r, k, m, b_z, b_theta, q, f_func, xi_factor, *args, **kwargs):
79      r"""
80      Return y intizlized with given xi and xi_prime.
81
82      Parameters
83      ----------
84      r : float
85          radius
86      k : float
87          axial periodicity number
88      m : float
89          azimuthal periodicity number
90      b_z : float
91          axial magnetic field
92      b_z_prime : float
93          derivative of axial magnetic field
94      b_theta : float
95          azimuthal magnetic field
96      b_theta_prime : float
97          derivative of azimuthal magnetic field
98      p_prime : float
99          derivative of pressure
100     q : float
101         safety factor
102     q_prime : float
103         derivative of safety factor
104     f_func : function
105         function used to return Newcomb's f
106
107     Returns
```

```
108         ———————
109         y : ndarray of flaots (2)
110             intial values for linear set of ODEs representing Euler-Lagrange
111             equation. :math:`y(0)=\xi'` and :math:`y(1)=f\xi'`
112
113         Notes
114         —————
115         The expressions for xi and xi_prime are from power series expansion as
116         described in the Newcomb's paper, Goedbloed and Freidberg's MHD books.
117         """
118         y = np.zeros(2)
119
120         f_params = {'r': r, 'k': k, 'm': m, 'b_z': b_z, 'b_theta': b_theta, 'q': q}
121
122         y[0] = xi[0]
123         y[1] = xi[1] * f_func(**f_params)
124
125         return y*xi_factor
126
127
128     def init_r_sing_glasser(r, k, m, b_z, b_theta, xi_factor, *args, **kwargs):
129         r"""
130         Returns initial condition for r close to zero.
131
132         Parameters
133         ——————————
134         r : float
135             radial position
136
137         k : float
138             axial periodicity number
139
140         m : float
141             azimuthal periodicity number
142
143         b_z : scipy spline
144             axial magnetic field
145
146         b_theta : scipy spline
147             azimuthal magnetic field
148
149         Returns
150         ———————
151         xi : ndarray
152              newcomb's xi
153
154         Notes
155         —————
156         Implements initial condition Alan used in his code.
157
158         .. math ::
159             u(0) &= r^{m - 1} \\
160             u(1) &= u(0) \frac{(k B_{z} r + m B_{\theta})^{2}}{m^{2}} (m-1)
161
162         Reference
163         —————————
164         Alan Glasser's newcomb.f code.
```

```
    """
    xi = np.zeros(2)
    xi[0] = r**(m - 1)
    xi[1] = xi[0]*((k*b_z(r)*r + m*b_theta(r))/m)**2*(m - 1)
    return xi*xi_factor
```

A.6.8 newcomb_f.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Aug 06 13:43:55 2014

@author: Jens von der Linden
"""

from __future__ import print_function
from __future__ import division
from __future__ import absolute_import
from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
                             int, map, next, oct, open, pow, range, round,
                             str, super, zip)
"""Python 3.x compatibility"""

from numba import jit, float64


def newcomb_f_16(r, k, m, b_z, b_theta, **kwargs):
    r"""
    Return f from Newcomb's paper.

    Parameters
    ----------
    r : ndarray of floats
        radius
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z : ndarray of floats
        axial magnetic field
    b_theta : ndarray of floats
        azimuthal mangetic field

    Returns
    -------
    f : ndarray of floats
        f from Newcomb's paper

    Notes
    -----
    Implements equation:
    :math:`f = \frac{r (k r B_{z}+ m B_{\theta})^2}{k^{2} r^{2} + m^{2}}`


```

```
47      Reference
48      ----------
49      Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
50      Equation (16)
51      """
52      params_num = {'r': r, 'k': k, 'm': m, 'b_z': b_z, 'b_theta': b_theta}
53      params_denom = {'r': r, 'k': k, 'm': m}
54      #if type(r) == float:
55      #     all_f_g.all_f.append([r, r*f_num_wo_r(**params_num)/f_denom(**params_denom)])
56      return r*f_num_wo_r(**params_num)/f_denom(**params_denom)
57
58
59  def jardin_f_8_78(r, k, m, b_z, b_theta, q):
60      r"""
61      Return f from Newcomb's paper.
62
63      Parameters
64      ----------
65      r : ndarray of floats
66          radius
67      k : float
68          axial periodicity number
69      m : float
70          azimuthal periodicity number
71      b_z : ndarray of floats
72          axial magnetic field
73      b_theta : ndarray of floats
74          azimuthal mangetic field
75      q: ndarray of floats
76          safety factor
77
78      Returns
79      -------
80      f : ndarray of floats
81          f from Newcomb's paper
82
83      Reference
84      ----------
85      Jardin (2010) Computational Methods in Plasma Physics
86      Equation (8.78)
87      """
88      return r*b_theta**2*(m - k*q)**2/(k**2*r**2 + m**2)
89
90
91  #@jit(float64(float64, float64, float64))
92  @jit
93  def f_denom(r, k, m):
94      r"""
95      Return denominator of f from Newcomb's paper.
96
97      Parameters
98      ----------
99      r: ndarray of floats
100         radius
101     k: float
102         axial periodicity number
103     m: float
```

```
104             azimuthal periodicity number
105
106         Returns
107         -------
108         f_denom: ndarray of floats
109             denominator of f from Newcomb's paper
110
111         Notes
112         -----
113         f denominator :math:`k^{2}r^{2}+m^{2}`
114
115         Reference
116         ---------
117         Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
118         Equation (16)
119         """
120         return k**2*r**2 + m**2
121
122
123     #@jit(float64(float64, float64, float64, float64, float64))
124     @jit
125     def f_num_wo_r(r, k, m, b_z, b_theta):
126         r"""
127         Return numerator of f without r from Newcomb's paper.
128
129         Parameters
130         ----------
131         r : ndarray of floats
132             radius
133         k : float
134             axial periodicity number
135         m : float
136             azimuthal periodicity number
137         b_z : ndarray of floats
138             axial magnetic field
139         b_theta : ndarray of floats
140             azimuthal mangetic field
141
142         Returns
143         -------
144         f_num_wo_r : ndarray of floats
145             numerator of f without r from Newcomb's paper
146
147         Notes
148         -----
149         f numerator without r: :math:`(k r B_{z}+m B_{\theta})^{2}`
150
151         Reference
152         ---------
153         Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
154         Equation (16)
155         """
156         return (k*r*b_z + m*b_theta)**2
157
158
159     def f_prime(r, k, m, b_z, b_theta, b_theta_prime, **kwargs):
160         r"""
```

```
161     return f_prime
162     """
163     term1 = -2.*k**2*r**2/(k**2*r**2 + m**2)**2 * (b_z*k*r + m*b_theta)**2
164     term2 = r/(k**2*r**2 + m**2)*(2.*b_z*k + 2.*m*b_theta_prime)*(b_z*k*r + m*b_theta)
165     term3 = 1./(k**2*r**2 + m**2)*(b_z*k*r + m*b_theta)**2
166     return term1 + term2 + term3
```

A.6.9 newcomb_g.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Wed Aug 06 14:39:57 2014
4
5   @author: Jens von der Linden
6   """
7
8   from __future__ import print_function
9   from __future__ import division
10  from __future__ import absolute_import
11  from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
12                               int, map, next, oct, open, pow, range, round,
13                               str, super, zip)
14  """Python 3.x compatibility"""
15
16  from newcomb_f import f_denom, f_num_wo_r
17  from numba import jit, float64
18
19
20  def jardin_g_8_80(r, k, m, b_z, b_z_prime, b_theta, b_theta_prime, p_prime, q,
21                    q_prime):
22      r"""
23      Return g from Newcomb's paper.
24
25      Parameters
26      ----------
27      r : ndarray of floats
28          radius
29      k : float
30          axial periodicity number
31      m : float
32          azimuthal periodicity number
33      b_z : ndarray of floats
34          axial magnetic field
35      b_theta : ndarray of floats
36          azimuthal mangetic field
37      b_theta_prime: ndarray of floats
38          azimuthal magnetic field
39      p_prime : ndarray of floats
40          pressure prime
41      q : ndarray of floats
42          safety factor
43      q_prime : ndarray of floats
44          derivative of safety factor
45      mu_0 : float
```

```
46          magnetic permeability of free space
47
48      Returns
49      -------
50      g : ndarray of floats
51          g from Newcomb's paper
52
53      Reference
54      ---------
55      Jardin (2010) Computational Methods in Plasma Physics
56      Equation (8.80)
57      """
58      term1 = (2.*k**2*r**2)/(k**2*r**2+m**2)*p_prime
59      term2 = b_theta**2/r*(m-k*q)**2*(k**2*r**2+m**2-1.)/(k**2*r**2+m**2)
60      term3 = 2*k**2*r*b_theta**2/(k**2*r**2+m**2)**2*(k**2*q**2-m**2)
61      return term1 + term2 + term3
62
63
64  def jardin_g_8_79(r, k, m, b_z, b_z_prime, b_theta, b_theta_prime, p_prime, q,
65                    q_prime):
66      r"""
67      Return g from Newcomb's paper.
68
69      Parameters
70      ----------
71      r : ndarray of floats
72          radius
73      k : float
74          axial periodicity number
75      m : float
76          azimuthal periodicity number
77      b_z : ndarray of floats
78          axial magnetic field
79      b_theta : ndarray of floats
80          azimuthal mangetic field
81      b_theta_prime: ndarray of floats
82          azimuthal magnetic field
83      p_prime : ndarray of floats
84          pressure prime
85      q : ndarray of floats
86          safety factor
87      q_prime : ndarray of floats
88          derivative of safety factor
89      mu_0 : float
90          magnetic permeability of free space
91
92      Returns
93      -------
94      g : ndarray of floats
95          g from Newcomb's paper
96
97      Reference
98      ---------
99      Jardin (2010) Computational Methods in Plasma Physics
100     Equation (8.79)
101     """
102     term1 = 1./r*b_theta**2/(k**2*r**2+m**2)*(k*q + m)**2
```

```
103         term2 = b_theta**2/r*(m - k*q)
104         term3 = 2.*b_theta/r*(r*b_theta_prime + b_theta)
105         der_term1 = -2.*k**2*r*b_theta**2/(k**2*r**2 + m**2)**2*(k**2*q**2 -
106                                                                  m**2)
107         der_term2 = 2.*k**2*b_theta**2/(k**2*r**2 + m**2)*q*q_prime
108         der_term3 = 2.*b_theta_prime/(k**2*r**2 + m**2)*((k**2*q**2 - m**2) *
109                                                          b_theta)
110         return term1 + term2 - term3 - der_term1 - der_term2 - der_term3
111
112
113     def newcomb_g_17(r, k, m, b_z, b_z_prime, b_theta, b_theta_prime, p_prime, q,
114                      q_prime):
115         r"""
116         Return g from Newcomb's paper.
117
118         Parameters
119         ----------
120         r : ndarray of floats
121             radius
122         k : float
123             axial periodicity number
124         m : float
125             azimuthal periodicity number
126         b_z : ndarray of floats
127             axial magnetic field
128         b_theta : ndarray of floats
129             azimuthal mangetic field
130         b_theta_prime: ndarray of floats
131             azimuthal magnetic field
132         p_prime : ndarray of floats
133             pressure prime
134         q : ndarray of floats
135             safety factor
136         q_prime : ndarray of floats
137             derivative of safety factor
138         mu_0 : float
139             magnetic permeability of free space
140
141         Returns
142         -------
143         g : ndarray of floats
144             g from Newcomb's paper
145
146         Reference
147         ---------
148         Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
149         Equation (17)
150         """
151         term1 = 1./r*(k*r*b_z - m*b_theta)**2/(k**2*r**2 + m**2)
152         term2 = 1./r*(k*r*b_z - m*b_theta)**2
153         term3 = 2.*b_theta/r*(r*b_theta_prime + b_theta)
154         der_term1 = -2.*k**2*r/(k**2*r**2 + m**2)**2*(k**2*r**2*b_z**2 -
155                                                       m**2*b_theta**2)
156         der_term2 = 1./(k**2*r**2 + m**2)*(2.*k**2*r**2*b_z*b_z_prime +
157                                            2.*k**2*r*b_z**2 -
158                                            2.*m**2*b_theta*b_theta_prime)
159         return term1 + term2 - term3 - der_term1 - der_term2
```

```
160
161
162 def newcomb_g_18(r, k, m, b_z, b_theta, p_prime, mu_0, **kwargs):
163     r"""
164     Return g from Newcomb's paper.
165
166     Parameters
167     ----------
168     r : ndarray of floats
169         radius
170     k : float
171         axial periodicity number
172     m : float
173         azimuthal periodicity number
174     b_z : ndarray of floats
175         axial magnetic field
176     b_theta : ndarray of floats
177         azimuthal mangetic field
178     p_prime : ndarray of floats
179         pressure prime
180     q : ndarray of floats
181         safety factor
182     mu_0 : float
183         magnetic permeability of free space
184
185     Returns
186     -------
187     g : ndarray of floats
188         g from Newcomb's paper
189
190     Notes
191     -----
192     Implements equation
193
194     .. math::
195         \frac{2 k^{2} r^{2}}{k^{2} r^{2} + m^{2}} \frac{dP}{dr} +
196         \frac{1}{r}(k r B_{z}+m B_{\theta})^{2}
197         \frac{k^{2}r^{2}+m^{2}-1}{k^{2}r^{2}+m^{2}}+
198         \frac{2k^{2}r}{(k^{2}r^{2}+m^{2})^{2}}
199         (k^{2}r^{2}B_{z}^{2}-m^{2}B_{\theta}^{2})
200
201     Reference
202     ---------
203     Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
204     Equation (18)
205     """
206     term1 = 2.*k**2*r**2/(f_denom(r, k, m))*p_prime*mu_0
207     term2 = (1./r*f_num_wo_r(r, k, m, b_z, b_theta)*(k**2*r**2+m**2-1.) /
208              f_denom(r, k, m))
209     term3 = (2.*k**2*r/f_denom(r, k, m)**2 *
210              (k**2*r**2*b_z**2 - m**2*b_theta**2))
211     #if type(r) == float:
212     #    all_f_g.all_g.append([r, term1 + term2 + term3])
213     return term1 + term2 + term3
214
215
216 def newcomb_g_18_dimless(r, k, m, b_z, p_prime, q, beta_0, **kwargs):
```

```
    r"""
    Return g from Newcomb's paper in dimensionless form. g only depends on b_z,
    b_theta is caclulated from q and b_z.

    Parameters
    ----------
    r : float
        radial position
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z : float
        axial magnetic field
    p_prime : float
        pressure gradient
    q : safety factor
    beta_0 : float
        beta on axis

    Returns
    -------
    g : dimensionless g

    Notes
    -----
    Implements equation

    .. math::
        \beta_{0} \frac{k^{2} r^{2}}{k^{2} r^{2} + m^{2}} \frac{dp}{dr} +
        r(k B_{z})^{2} \frac{k^{2}r^{2}+m^{2}-1}{k^{2}r^{2}+m^{2}}+
        \frac{1}{r}(m B_{\theta})^{2}\frac{k^{2}r^{2}+m^{2}-1}{k^{2}r^{2}+m^{2}}+
        2kmB_{z}B_{\theta}\frac{k^{2}r^{2}+m^{2}-1}{k^{2}r^{2}+m^{2}}+
        \frac{2k^{4}r^{3}B_{z}^{2}}{k^{2}r^{2}+m^{2}}-
        \frac{2k^{2}m^{2}rB_{\theta}^{2}}{k^{2}r^{2}+m^{2}}^{2}

    """
    term1 = beta_0*k**2*r**2/f_denom(r, k, m)*p_prime
    term2 = r*(k*b_z)**2*(k**2*r**2 + m**2-1.)/f_denom(r, k, m)
    term3 = m**2*1./r*(b_z*r*k/q)**2*(k**2*r**2+m**2-1.)/ f_denom(r, k, m)
    term4 = 2.*m*k**2*r*b_z**2/q*(k**2*r**2 + m**2-1.)/(k**2*r**2 + m**2)
    term5 = 2*k**4*r**3*b_z**2/f_denom(r, k, m)**2
    term6 = -m**2*2.*k**4*r**3*b_z**2/(q**2*f_denom(r, k, m)**2)
    #if type(r) == float:
    #    all_f_g.all_g.append([r, term1 + term2 + term3 + term4 + term5 + term6])
    #    all_f_g.all_g_term1.append([r, term1])
    #    all_f_g.all_g_term2.append([r, term2])
    #    all_f_g.all_g_term3.append([r, term3])
    #    all_f_g.all_g_term4.append([r, term4])
    #    all_f_g.all_g_term5.append([r, term5])
    #    all_f_g.all_g_term6.append([r, term6])
    return term1 + term2 + term3 + term4 + term5 + term6


#@jit(float64(float64, float64, float64, float64, float64, float64, float64))
#@jit
def newcomb_g_18_dimless_wo_q(r, k, m, b_z, b_theta, p_prime, beta_0):
```

```
    r"""
    Return g from Newcomb's paper in dimensionless form.

    Parameters
    ----------
    r : float
        radial position
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z : float
        axial magnetic field
    p_prime: float
        pressure gradient
    beta_0 : float
        beta on axis

    Returns
    -------
    g : dimensionless g

    Notes
    -----
    Implements equation

    .. math::
        \frac{2 k^{2} r^{2}}{k^{2} r^{2} + m^{2}} \frac{dP}{dr} +
        \frac{1}{r}(k r B_{z}+m B_{\theta})^{2}
        \frac{k^{2}r^{2}+m^{2}-1}{k^{2}r^{2}+m^{2}}+
        \frac{2k^{2}r}{(k^{2}r^{2}+m^{2})^{2}}
        (k^{2}r^{2}B_{z}^{2}-m^{2}B_{\theta}^{2})eturn g from Newcomb's paper in dimensionless form.
    """
    term1 = beta_0*k**2*r**2/f_denom(r, k, m)*p_prime
    term2 = r*(k*b_z)**2*(k**2*r**2 + m**2-1.)/f_denom(r, k, m)
    term3 = 1./r*(m*b_theta)**2*(k**2*r**2 + m**2-1.)/ f_denom(r, k, m)
    term4 = 2.*m*k*b_z*b_theta*(k**2*r**2 + m**2-1.)/f_denom(r, k, m)
    term5 = 2.*k**4*r**3*b_z**2/f_denom(r, k, m)**2
    term6 = -2.*k**2*m**2*r*b_theta**2/f_denom(r, k, m)**2
    #if type(r) == type(r):
    #    all_f_g.all_g.append([r, term1 + term2 + term3 + term4 + term5 + term6])
    #    all_f_g.all_g_term1.append([r, term1])
    #    all_f_g.all_g_term2.append([r, term2])
    #    all_f_g.all_g_term3.append([r, term3])
    #    all_f_g.all_g_term4.append([r, term4])
    #    all_f_g.all_g_term5.append([r, term5])
    #    all_f_g.all_g_term6.append([r, term6])
    #    all_f_g.all_pressure_prime.append([r, p_prime])
    #    all_f_g.all_b_theta.append([r, b_theta])
    #    all_f_g.all_b_z.append([r, b_z])
    #    all_f_g.all_beta_0.append([r, beta_0])
    #    all_f_g.all_m.append([r, m])
    #    all_f_g.all_k.append([r, k])
    return term1 + term2 + term3 + term4 + term5 + term6
```

A.6.10 find_singularities

```
# -*- coding: utf-8 -*-
"""
Created on Mon Sep 22 20:27:04 2014

@author: Jens von der Linden

Identify singularities in Euler-Lagrange
by searching for zeros of f.
"""

from __future__ import print_function
from __future__ import division
from __future__ import absolute_import
from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
                             int, map, next, oct, open, pow, range, round,
                             str, super, zip)
"""Python 3.x compatibility"""

import numpy as np
import scipy.optimize as opt
from scipy.interpolate import splev


def identify_singularties(a, b, points, k, m, b_z_spl, b_theta_spl, offset,
                          tol, **kwargs):
    """
    Return list of singular points.

    Parameters
    ----------
    a : float
        radial start of pinch
    b : float
        radial end of pinch
    points : int
        number of points through which to divide f
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z_spl : scipy spline object
        axial magnetic field
    b_theta_spl : scipy spline object
        azimuthal magnetic field

    Returns
    -------
    zero_positions: ndarray of floats (M)
        radial positions at which r equals zero.

    Notes
    -----
    Singular points are found by dividing f into intervals checking for sign
    changes and then running a zero funding method from the scipy optimize
```

```
    module.
    """
    params = (k, m, b_z_spl, b_theta_spl)
    r = np.linspace(a, b, points)
    zero_positions = []

    sign = np.sign(f_relevant_part(r, k, m, b_z_spl, b_theta_spl))
    for i in range(points-1):
        if abs(sign[i] + sign[i+1]) < 1e-8:
            zero_pos = opt.brentq(f_relevant_part, r[i], r[i+1], args=params)
            zero = f_relevant_part(r[i], k, m, b_z_spl, b_theta_spl)
            if np.isnan(zero) or abs(zero) > tol:
                continue
            else:
                zero_positions.append(zero_pos)

    sings = np.array(zero_positions)

    if not sings.size == 0 and abs(sings[0]) < 1e-8:
        sings_wo_0 = np.delete(sings, 0)
    else:
        sings_wo_0 = sings

    integration_points = np.insert((float(a), float(b)), 1, sings_wo_0)
    intervals = [[integration_points[i],
                  integration_points[i+1]]
                 for i in range(integration_points.size-1)]

    return sings, sings_wo_0, intervals


def f_relevant_part(r, k, m, b_z_spl, b_theta_spl):
    """
    Return relevant part of f for singularity detection.

    Parameters
    ----------
    r : ndarray of floats
        radial points
    k : float
        axial periodicity number
    m : float
        azimuthal periodicity number
    b_z_spl : scipy spline object
        axial magnetic field
    b_theta_spl : scipy spline object
        azimuthal magnetic field

    Returns
    -------
    f_relvant_part : ndarray of floats
        The relevant part o    eigenfunctions = []
    eigen_ders = []
    rs_list = []f Newcomb's f for determining f=0. The term that can
        make f=0 when :math:`r \neq 0`

    Notes
```

```
112         -----
113         The relevant part of f is:
114         .. math::
115            k r B_{z} + m B_{\theta}
116         """
117         b_theta = splev(r, b_theta_spl)
118         b_z = splev(r, b_z_spl)
119         return f_relevant_part_func(r, k, m, b_z, b_theta)
120
121
122     def f_relevant_part_func(r, k, m, b_z, b_theta):
123         """
124         Return relevant part of f for singularity detection. Could be complied be
125         with numba.
126
127         Parameters
128         ----------
129         r : ndarray of floats
130             radial points
131         k : float
132             axial periodicity number
133         m : float
134             azimuthal periodicity number
135         b_z : ndarray of floats
136             axial magnetic field
137         b_theta : ndarray of floats
138             azimuthal magnetic field
139
140         Returns
141         -------
142         f_relvant_part : ndarray of floats
143             The relevant part of Newcomb's f for determining f=0. The term that can
144             make f=0 when :math:`r \neq 0`
145
146         Notes
147         -----
148         The relevant part of f is:
149         .. math::
150            k r B_{z} + m B_{\theta}
151         """
152         return k*r*b_z + m*b_theta
153
154
155     def f_relevant_part_der(r, k, m, b_z_spl, b_theta_prime_spl):
156         r"""
157         Prefunction to derivative of f_relvent_part.
158         """
159         b_z = b_z_spl(r)
160         b_theta_prime = splev(r, b_theta_prime_spl)
161         return f_relevant_part_der_func(r, k, m, b_z, b_theta_prime)
162
163
164     def f_relevant_part_der_func(r, k, m, b_z, b_theta_prime):
165         r"""
166         Derivative of f_relevant_part.
167         """
168         return k*b_z + m*b_theta_prime
```

```
169
170
171 def f_relevant_part_der_2(r, m, b_theta_prime_prime_spl):
172     r"""
173     Prefunction to Second derivative of f_relvent_part.
174     """
175     b_theta_prime_prime = splev(r, b_theta_prime_prime_spl)
176     return f_relevant_part_der_2_func(r, m, b_theta_prime_prime)
177
178
179 def f_relevant_part_der_2_func(r, m, b_theta_prime_prime):
180     r"""
181     Second derivative of f_relevant_part.
182     """
183     return m*b_theta_prime_prime
```

A.6.11 singularity_frobenius.py

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Aug 21 13:23:31 2014
4
5  @author: Jens von der Linden
6
7  Implements Frobenius expansion around a singularity to determine the "small"
8  solution and check the Suydam condition.
9  """
10
11 from __future__ import print_function
12 from __future__ import division
13 from __future__ import absolute_import
14 from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
15                              int, map, next, oct, open, pow, range, round,
16                              str, super, zip)
17 """Python 3.x compatibility"""
18
19 import numpy as np
20 from scipy.interpolate import splev
21
22
23 def alpha_func(r, b_z, b_z_prime, b_theta, b_theta_prime, **kwargs):
24     r"""
25     Return alpha for Frobenius solution.
26
27     Parameters
28     ----------
29     r : ndarray of floats
30         radial position of singularity
31     b_z : ndarray of floats
32         axial magnetic field
33     b_z_prime : ndarray of floats
34         derivative of axial magnetic field
35     b_theta : ndarry of floats
36         azimuthal magnetic field
```

```
37      b_theta_prime : ndarry of floats
38          derivative of azimuthal magnetic field
39
40      Returns
41      _______
42      alpha : ndarray of floats
43          beta of quadratic equation for the exponents of the Forbenius solutions
44      """
45      mu = b_theta/(r*b_z)
46      mu_prime = (r*b_z*b_theta_prime - b_theta*(b_z + r*b_z_prime)) / (r*b_z)**2
47      return r*b_theta**2*b_z**2/(b_theta**2 + b_z**2)*(mu_prime / mu)**2
48
49
50  def beta_func(b_z, b_theta, p_prime, beta_0, **kwargs):
51      r"""
52      Return beta for Frobenius solution.
53
54      Parameters
55      __________
56      b_z : ndarray of floats
57          axial magnetic field
58      b_theta: ndarray of floats
59          azimuhtal magnetic field
60      p_prime : ndarry of floats
61          derivative of pressure
62      Returns
63      _______
64      beta : ndarray of floats
65          beta of quadratic equation for the exponents of the Forbenius solutions
66      """
67      return 2.*beta_0*b_theta**2/(b_theta**2 + b_z**2) * p_prime
68
69
70  def nu_1_2(alpha, beta, imaginary=False, **kwargs):
71      r"""
72      Return exponents of Frobenius solution.
73
74      Paramters
75      _________
76      alpha : ndarray of floats
77          alpha of quadratic equation for the exponents of the Frobenius
78          solutions
79      beta : ndarray of floats
80          beta of quadratic equation for the exponents of the Forbenius solutions
81      Returns
82      _______
83      nu_1 : ndarray of floats
84           exponent 1 of Frobenius solution
85      nu_2 : ndarray of floats
86           exponent 1 of Frobenius solution
87      """
88      nu_1 = -0.5 + np.emath.sqrt(0.25 + beta/alpha)
89      nu_2 = -0.5 - np.emath.sqrt(0.25 + beta/alpha)
90      return nu_1, nu_2
91
92
93  def sings_alpha_beta(r, b_z_spl, b_z_prime_spl, b_theta_spl, b_theta_prime_spl,
```

```
94                          p_prime_spl, beta_0):
95      r"""
96      Returns alpha and beta at the singularties.
97
98      Parameters
99      ----------
100     r : ndarray of floats
101         positions of singularities
102     b_z_spl : scipy spline object
103         axial magnetic field
104     b_theta_spl : scipy spline object
105         azimuthal magnetic field
106     p_prime_spl : scipy spline object
107         derivative of pressure
108
109     Returns
110     -------
111     alpha : ndarry of floats
112         alpha of quadratic equation for the exponents of the Frobenius
113         solutions
114     beta : ndarray of floats
115         beta of quadratic equation for the exponents of the Forbenius solutions
116     """
117     if r.size == 0:
118         return np.empty(0), np.empty(0)
119     else:
120         b_z = splev(r, b_z_spl)
121         b_z_prime = splev(r, b_z_prime_spl)
122         b_theta = splev(r, b_theta_spl)
123         b_theta_prime = splev(r, b_theta_prime_spl)
124         p_prime = splev(r, p_prime_spl)
125         params = {'r': r, 'b_z': b_z, 'b_z_prime': b_z_prime,
126                   'b_theta': b_theta, 'b_theta_prime': b_theta_prime,
127                   'p_prime': p_prime, 'beta_0': beta_0}
128         alpha = alpha_func(**params)
129         beta = beta_func(**params)
130         return alpha, beta
131
132
133 def sings_suydam_stable(r, b_z_spl, b_z_prime_spl, b_theta_spl,
134                         b_theta_prime_spl, p_prime_spl, beta_0):
135     r"""
136     Returns bool array. True for singularities that are Suydam stable and False
137     for unstable singularities.
138
139     Parameters
140     ----------
141     r : ndarray of floats
142         singularity positions
143     b_z_spl : scipy spline object
144         axial magnetic field
145     b_theta_spl : scipy spline object
146         azimuthal magnetic field
147     p_prime_spl :scipy spline object
148         derivative of pressure
149
150     Returns
```

```
151         -------
152         stable_mask : ndarray of bool
153             bool array of Suydam stable singularities can be used as a mask for
154             indexing.
155         """
156         alpha, beta = sings_alpha_beta(r, b_z_spl, b_z_prime_spl, b_theta_spl,
157                                        b_theta_prime_spl, p_prime_spl, beta_0)
158         return suydam_stable(alpha, beta)
159
160
161     def sing_small_solution(r_sing, offset, b_z_spl, b_z_prime_spl, b_theta_spl,
162                             b_theta_prime_spl, p_prime_spl, q_spl, f_func, beta_0,
163                             r_request=None):
164         r"""
165         Returns small solution of Frobenius expansion near singularity in y form of
166         ODE set.
167
168         Parameters
169         ----------
170         r_sing : ndarray of floats
171             position of singularities
172         offset : float
173             offset from singularity for integration
174         b_z_spl : scipy spline object
175             axial magnetic field
176         b_theta_spl : scipy spline object
177             azimuthal magnetic field
178         p_prime_spl : scipy spline object
179             derivative of pressure
180         q_spl : scipy spline object
181             safety factor
182         f_func : function
183             function
184
185         Returns
186         -------
187         small_sol : ndarry of floats (2)
188             small solution xi and xi_prime to be used in an init function.
189
190         References
191         ----------
192         Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
193         """
194         alpha, beta = sings_alpha_beta(r_sing, b_z_spl, b_z_prime_spl, b_theta_spl,
195                                        b_theta_prime_spl, p_prime_spl, beta_0)
196
197         nu_1, nu_2 = nu_1_2(alpha, beta)
198         if not r_request:
199             r = r_sing + offset
200         else:
201             r = r_request
202         small_sol = small_solution(r_sing + offset, r_sing, nu_1, nu_2)
203         return small_sol
204
205
206     def suydam_stable(alpha, beta):
207         r"""
```

```
208         Returns suydam condition.
209
210         Parameters
211         ----------
212         alpha : ndarray of floats
213             alpha of quadratic equation for the exponents of the Frobenius
214             solutions
215         beta : scipy spline
216             beta of quadratic equation for the exponents of the Forbenius solutions
217
218         Returns
219         -------
220         suydam : ndarray of bool
221             suydam stable
222
223         Notes
224         -----
225         This ithe criterion for oscillatory solutions near singularity.
226
227         Reference
228         ---------
229         Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch
230         eq (5).
231         Jardin (2010) Computational Mehtods in Plasma Physics. eq (8.84)
232         """
233         return alpha + 4.*beta > 0.
234
235
236     def small_solution(r, r_sing, nu_1, nu_2):
237         r"""
238         Returns xi and f*xi_der of the small solution close to a singularity.
239
240         Parameters
241         ----------
242         r : float
243             position at which small solution is desired
244         r_sing : float
245             position of singularity
246         nu_1 : float
247             exponent 1 of Frobenius solution
248         nu_2 : float
249             exponent 2 of Frobenius solution
250
251         Returns
252         -------
253         small_sol : ndarry of floats (2)
254             small solution near singularity
255
256         Reference
257         ---------
258         Newcomb (1960) Hydromagnetic Stability of a Diffuse Linear Pinch (p. 243)
259         """
260         x = np.abs(r - r_sing)
261         if nu_1 < nu_2:
262             return np.array([np.real(x**nu_2), np.real(nu_2*x**(nu_2 - 1.))])
263         else:
264             return np.array([np.real(x**nu_1), np.real(nu_1*x**(nu_1 - 1.))])
```

A.6.12 external_stability.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Sep 08 20:42:30 2014

@author: Jens von der Linden

Calculate dW and determine external stability.
"""

from __future__ import print_function
from __future__ import division
from __future__ import absolute_import
from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
                             int, map, next, oct, open, pow, range, round,
                             str, super, zip)
"""Python 3.x compatibility"""

import scipy.special as spec
from scipy.interpolate import splev
import analytic_condition as ac

def external_stability(params, xi, xi_der, dim_less=False):
    r"""
    Returns external external stability and dW.
    """
    a = params['a']
    b_z = splev(a, params['b_z'])
    b_theta = splev(a, params['b_theta'])
    m = params['m']
    k = params['k']
    magnetic_potential_energy_ratio = params['magnetic_potential_energy_ratio']

    if params['b'] == 'infinity':
        lambda_term = lambda_infinity(**{'a': a, 'k': k, 'm': m})
    else:
        assert isinstance(params['b'], float), "b must be 'infinity' or of \
                                               type float."
        lambda_term = lambda_boundary(**{'a': a, 'b': b, 'k': k, 'm': m})
    f_term = capital_f(**{'a': a, 'k': k, 'm': m, 'b_theta': b_theta,
                          'b_z': b_z})
    f_adjoint_term = f_adjoint(**{'a': a, 'k': k, 'm': m, 'b_theta': b_theta,
                                  'b_z': b_z})
    k_0_sq_term = k_0(**{'k': k, 'm': m, 'a': a})

    term1 = f_term**2*a*xi_der/(k_0_sq_term*xi)
    term2 = f_term*f_adjoint_term/(k_0_sq_term)
    term3 = a**2*f_term**2*lambda_term
    delta_w = (term1+term2*term3)*xi**2
    if dim_less:
        delta_w = magnetic_potential_energy_ratio * delta_w
```

```
51      stable = delta_w > 0
52      return stable, delta_w
53
54  def lambda_infinity(a, k, m):
55      r"""
56      Return lambda term for wall at infinity.
57      """
58      k_a = spec.kv(m, abs(k)*a)
59      k_a_prime = spec.kvp(m, abs(k)*a)
60
61      return -k_a/(abs(k*a)*k_a_prime)
62
63
64  def lambda_boundary(a, b, k, m):
65      r"""
66      Return lambda term for wall at radius b.
67      """
68      k_a = spec.kv(m, abs(k)*a)
69      k_a_prime = spec.kvp(m, abs(k)*a)
70      k_b_prime = spec.kvp(m, abs(k)*b)
71      i_a = spec.iv(m, abs(k)*a)
72      i_a_prime = spec.ivp(m, abs(k)*a)
73      i_b_prime = spec.ivp(m, abs(k)*b)
74
75      factor1 = -k_a/(abs(k*a)*k_a_prime)
76      factor2_num = 1. - k_b_prime*i_a/(i_b_prime*k_a)
77      factor2_denom = 1. - k_b_prime*i_a_prime/(i_b_prime*k_a_prime)
78      return factor1*factor2_num/factor2_denom
79
80
81  def k_0(k, m, a):
82      r"""
83      Return k_0 term.
84      """
85      return k**2 + m**2 / a**2
86
87
88  def f_adjoint(a, k, m, b_theta, b_z):
89      r"""
90      Return adjoint F.
91      """
92      return k*b_z - m*b_theta/a
93
94
95  def capital_f(a, k, m, b_theta, b_z):
96      r"""
97      Return F.
98      """
99      return k*b_z + m*b_theta/a
100
101
102 def external_stability_from_notes(params, xi, xi_der, dim_less=False):
103     a = params['a']
104     b_z = splev(a, params['b_z'])
105     b_theta = splev(a, params['b_theta'])
106     m = params['m']
107     k_bar = params['k']
```

```
108         magnetic_potential_energy_ratio = params['magnetic_potential_energy_ratio']
109
110         term_params = {'a': a, 'k_bar': k_bar, 'm': m, 'b_z': b_z,
111                        'b_theta': b_theta, 'xi': xi, 'xi_der': xi_der}
112         delta_w = (plasma_term_from_notes(**term_params) -
113                    vacuum_term_from_notes(**term_params))
114         if dim_less:
115             delta_w = magnetic_potential_energy_ratio * delta_w
116         stable = delta_w > 0
117
118         return stable, delta_w
119
120
121     def plasma_term_from_notes(a, k_bar, m, b_z, b_theta, xi, xi_der):
122         r"""
123         Returns plasma energy term as in my derivation.
124         """
125         f_term = a*(k_bar*b_z + m*b_theta)**2/(k_bar**2 + m**2)
126         h_term = (k_bar**2*b_z**2 - m**2*b_theta**2)/(k_bar**2 + m**2)
127         return xi**2 * (f_term*xi_der / xi + h_term)
128
129
130     def vacuum_term_from_notes(a, k_bar, m, b_z, b_theta, xi, xi_der):
131         r"""
132         Returns vacuum energy term as in my derivation.
133         """
134         k_a = spec.kv(m, abs(k_bar))
135         k_a_prime = spec.kvp(m, abs(k_bar))
136         term1 = (k_bar*b_z + m*b_theta)**2
137         term2 = xi**2/k_bar*k_a/k_a_prime
138         return term1*term2
139
140     def external_stability_from_analytic_condition(params, xi, xi_der,
141                                                    without_sing=True,
142                                                    dim_less=False):
143         r"""
144         Returns delta_w as given in the analytic condition.
145
146         Optionally remove singularity by mutiplying by xi^2.
147
148         Note
149         ----
150         When xi goes to zero, this condition is singular due to the delta=xi'/xi
151         term.
152         """
153         a = params['a']
154         b_z = splev(a, params['b_z'])
155         b_theta_vacuum = splev(a, params['b_theta'])
156         m = -params['m']
157         k_bar = params['k']
158         lambda_bar = 2*b_theta_vacuum / (b_z*a)
159         delta = xi_der*a / xi
160
161         if without_sing:
162             dW = ac.conditions_without_interface_wo_sing(k_bar, lambda_bar, m, xi,
163                                                          xi_der, a)
164         else:
```

```
        dW = ac.conditions_without_interface(k_bar, lambda_bar, m, delta)
    stable = dW > 0
    return stable, dW
```

A.6.13 lambda_k_plotting.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 16 00:30:53 2015

@author: jensv

Module for examining stability spaces.
"""

from __future__ import print_function, unicode_literals, division
from __future__ import absolute_import
from future import standard_library, utils
from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
                             int, map, next, oct, open, pow, range, round,
                             str, super, zip)

import numpy as np
from scipy.special import kv, kvp
import analytic_condition as ac
from scipy.interpolate import griddata

import matplotlib.pyplot as plt
from matplotlib.colors import SymLogNorm, BoundaryNorm
from matplotlib.ticker import FormatStrFormatter, FixedFormatter
import matplotlib.patches as patches
import matplotlib.ticker as ticker
import seaborn as sns
sns.set_style('white')
sns.set_context('poster')


def plot_lambda_k_space_dw(filename, epsilon, name, mode_to_plot='m_neg_1',
                           show_points=False, lim=None, levels=None, log=True,
                           linthresh=1E-7, bounds=(1.5, 3.0), norm=True,
                           analytic_compare=False,
                           label_pos=((0.5, 0.4), (2.1, 0.4), (2.8, 0.2)),
                           delta_values=[-1,0,1],
                           interpolate=False,
                           cmap=None, hatch=False,
                           figsize=None,
                           save_as=None,
                           return_ax=False,
                           axes=None):
    r"""
    Plot the delta_w of external instabilities in the lambda-k space.
    """
    if figsize:
        fig = plt.figure(figsize=figsize)
```

```
    epsilon_case = np.load(filename)
    lambda_a_mesh = epsilon_case['lambda_a_mesh']
    k_a_mesh = epsilon_case['k_a_mesh']
    external_m_neg_1 = epsilon_case['d_w_m_neg_1']
    external_sausage = epsilon_case['d_w_m_0']
    epsilon_case.close()

    instability_map = {'m_0': external_sausage,
                       'm_neg_1': external_m_neg_1}


    kink_pal = sns.blend_palette([sns.xkcd_rgb["dandelion"],
                                  sns.xkcd_rgb["white"]], 7, as_cmap=True)
    kink_pal = sns.diverging_palette(73, 182, s=72, l=85, sep=1, n=9, as_cmap=True)
    sausage_pal = sns.blend_palette(['orange', 'white'], 7, as_cmap=True)
    sausage_pal = sns.diverging_palette(49, 181, s=99, l=78, sep=1, n=9, as_cmap=True)

    if cmap:
        instability_palette = {'m_0': cmap,
                               'm_neg_1': cmap}
    else:
        instability_palette = {'m_0': sausage_pal,
                               'm_neg_1': kink_pal}




    if interpolate:
        instability_map['m_neg_1'] = interpolate_nans(lambda_a_mesh,
                                                      k_a_mesh,
                                                      instability_map['m_neg_1']
                                                      )

    values = instability_map[mode_to_plot]

    if norm:
        values = values / np.nanmax(np.abs(values))
    else:
        values = values

    if levels:
        if log:
            plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
                                cmap=instability_palette[mode_to_plot],
                                levels=levels, norm=SymLogNorm(linthresh))
            cbar = plt.colorbar(label=r'$\delta W$')
            cbar.set_label(label=r'$\delta W$', size=45, rotation=0, labelpad=30)
            contourlines = plt.contour(lambda_a_mesh, k_a_mesh,
                                       values, levels=levels,
                                       colors='grey',
                                       norm=SymLogNorm(linthresh))

        else:
            norm = BoundaryNorm(levels, 256)
            plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
                                cmap=instability_palette[mode_to_plot],
                                levels=levels, norm=norm)
```

```
106             cbar = plt.colorbar(label=r'$\delta W$')
107             cbar.set_label(label=r'$\delta W$', size=45, rotation=0, labelpad=30)
108             contourlines = plt.contour(lambda_a_mesh, k_a_mesh,
109                                        values, levels=levels,
110                                        colors='grey')
111     else:
112         if log:
113             plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
114                                 cmap=instability_palette[mode_to_plot],
115                                 norm=SymLogNorm(linthresh))
116             cbar = plt.colorbar(label=r'$\delta W$')
117             cbar.set_label(label=r'$\delta W$', size=45, rotation=0, labelpad=30)
118             contourlines = plt.contour(lambda_a_mesh, k_a_mesh,
119                                        values, colors='grey',
120                                        norm=SymLogNorm(linthresh))
121         else:
122             plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
123                                 cmap=instability_palette[mode_to_plot])
124             cbar = plt.colorbar(label=r'$\delta W$')
125             cbar.set_label(label=r'$\delta W$', size=45, rotation=0, labelpad=30)
126             contourlines = plt.contour(lambda_a_mesh, k_a_mesh,
127                                        values, colors='grey')
128
129     if lim:
130         plot.set_clim(lim)
131
132     cbar.add_lines(contourlines)
133
134     plt.plot([0.01, 0.1, 1.0, 2.0, 3.0],
135              [0.005, 0.05, 0.5, 1.0, 1.5], color='black')
136
137     axes = plt.gca()
138     axes.set_axis_bgcolor(sns.xkcd_rgb['white'])
139
140     lambda_bar_analytic = np.linspace(0.01, 3., 750)
141     k_bar_analytic = np.linspace(0.01, 1.5, 750)
142     (lambda_bar_mesh_analytic,
143      k_bar_mesh_analytic) = np.meshgrid(lambda_bar_analytic, k_bar_analytic)
144
145     if analytic_compare:
146         analytic_comparison(mode_to_plot, k_bar_mesh_analytic,
147                             lambda_bar_mesh_analytic, epsilon, label_pos)
148
149     if show_points:
150         plt.scatter(lambda_a_mesh, k_a_mesh, marker='o', c='b', s=5)
151
152     plt.ylim(0.01, bounds[0])
153     plt.xlim(0.01, bounds[1])
154     axes = plt.gca()
155     axes.set_xticks(np.arange(0., 4.5, 1.))
156     axes.set_yticks(np.arange(0., 2.0, 0.5))
157     plt.setp(axes.get_xticklabels(), fontsize=30)
158     plt.setp(axes.get_yticklabels(), fontsize=30)
159     plt.ylabel(r'$\bar{k}$', fontsize=40, rotation='horizontal', labelpad=30)
160     plt.xlabel(r'$\bar{\lambda}$', fontsize=40)
161     cbar.ax.tick_params(labelsize=30)
162     def my_formatter_fun(x):
```

```
163             if x == 0:
164                 return r'$0$'
165             if np.sign(x) > 0:
166                 return r'$10^{%i}$' % np.int(np.log10(x))
167             else:
168                 return r'$-10^{%i}$' % np.int(np.log10(np.abs(x)))
169         labels = [my_formatter_fun(level) for level in levels]
170         cbar.ax.set_yticklabels(labels)
171         sns.despine(ax=axes)
172 
173         if hatch:
174             xmin, xmax = axes.get_xlim()
175             ymin, ymax = axes.get_ylim()
176             xy = (xmin,ymin)
177             width = xmax - xmin
178             height = ymax - ymin
179             p = patches.Rectangle(xy, width, height, hatch='+', fill=None, zorder=-10)
180             axes.add_patch(p)
181 
182         plt.tight_layout()
183         if return_ax:
184             return axes, cbar
185         else:
186             plt.savefig('../../output/plots/' + name + '.png')
187             if save_as:
188                 plt.savefig(save_as)
189             plt.show()
190 
191 
192 def interpolate_nans(lambda_a, k_a, quantity):
193     r"""
194     Return mesh with nans interpolated from neighboring values.
195     """
196     index_to_keep = np.isnan(quantity.ravel())
197     interp_values = quantity.ravel()[~index_to_keep]
198     interp_k = k_a.ravel()[~index_to_keep]
199     interp_lambda = lambda_a.ravel()[~index_to_keep]
200     return griddata((interp_lambda, interp_k),
201                     interp_values,
202                     (lambda_a, k_a),
203                     method='linear')
204 
205 
206 def plot_dW_given_delta(filename, epsilon, name, mode_to_plot='m_neg_1',
207                         show_points=False, lim=None, levels=None, log=False,
208                         linthresh=1E-7, bounds=(1.5, 3.0), floor_norm=False,
209                         analytic_compare=False,
210                         label_pos=((0.5, 0.4), (2.1, 0.4), (2.8, 0.2)),
211                         delta_values=[-1,0,1],
212                         interpolate=False, with_interface=False):
213     r"""
214     Plot the delta_w of external instabilities in the lambda-k space.
215     """
216     epsilon_case = np.load(filename)
217     lambda_a_mesh = epsilon_case['lambda_a_mesh']
218     k_a_mesh = epsilon_case['k_a_mesh']
219 
```

```
    delta_mesh_sausage = epsilon_case['delta_m_0']
    delta_mesh_kink = epsilon_case['delta_m_neg_1']

    epsilon_case.close()

    if with_interface:
        external_sausage_norm = ac.conditions(k_a_mesh, lambda_a_mesh, epsilon,
                                              0, delta_mesh_sausage)
        external_m_neg_1_norm = ac.conditions(k_a_mesh, lambda_a_mesh, epsilon,
                                              1, delta_mesh_kink)
    else:
        external_sausage_norm = ac.conditions_without_interface(k_a_mesh,
                                                                 lambda_a_mesh,
                                                                 epsilon,
                                                                 0,
                                                                 delta_mesh_sausage)
        external_m_neg_1_norm = ac.conditions_without_interface(k_a_mesh,
                                                                 lambda_a_mesh,
                                                                 epsilon,
                                                                 1,
                                                                 delta_mesh_kink)

    instability_map = {'m_0': external_sausage_norm,
                       'm_neg_1': external_m_neg_1_norm}

    kink_pal = sns.blend_palette([sns.xkcd_rgb["dandelion"],
                                  sns.xkcd_rgb["white"]], 7, as_cmap=True)
    kink_pal = sns.diverging_palette(73, 182, s=72, l=85, sep=1, n=9, as_cmap=True)
    sausage_pal = sns.blend_palette(['orange', 'white'], 7, as_cmap=True)
    sausage_pal = sns.diverging_palette(49, 181, s=99, l=78, sep=1, n=9, as_cmap=True)
    instability_palette = {'m_0': sausage_pal,
                           'm_neg_1': kink_pal}

    if interpolate:
        instability_map['m_neg_1'] = interpolate_nans(lambda_a_mesh,
                                                      k_a_mesh,
                                                      instability_map['m_neg_1']
                                                      )

    values = instability_map[mode_to_plot]

    if floor_norm:
        values = np.clip(values, -100., 100.)
        values = values / -np.nanmin(values)
        values = np.clip(values, -1., 1.)
    else:
        values = values / -np.nanmin(values)

    if levels:
        if log:
            plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
                                cmap=instability_palette[mode_to_plot],
                                levels=levels, norm=SymLogNorm(linthresh))
        else:
            norm = BoundaryNorm(levels, 256)
            plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
                                cmap=instability_palette[mode_to_plot],
```

```
277                                 levels=levels, norm=norm)
278             cbar = plt.colorbar(label=r'$\delta W$',
279                                 format=FormatStrFormatter('%.0e'))
280             cbar.set_label(label=r'$\delta W$', size=45, rotation=0, labelpad=30)
281             contourlines = plt.contour(lambda_a_mesh, k_a_mesh, values,
282                                        levels=levels[:-1], colors='grey')
283             cbar.add_lines(contourlines)
284     else:
285         if log:
286             plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
287                                 cmap=instability_palette[mode_to_plot],
288                                 norm=SymLogNorm(linthresh))
289         else:
290             plot = plt.contourf(lambda_a_mesh, k_a_mesh, values,
291                                 cmap=instability_palette[mode_to_plot])
292
293     if lim:
294         plot.set_clim(lim)
295     plt.plot([0.01, 0.1, 1.0, 2.0, 3.0],
296              [0.005, 0.05, 0.5, 1.0, 1.5], color='black')
297
298     axes = plt.gca()
299     axes.set_axis_bgcolor(sns.xkcd_rgb['grey'])
300
301     lambda_bar = np.linspace(0.01, 3., 750)
302     k_bar = np.linspace(0.01, 1.5, 750)
303     lambda_bar_mesh, k_bar_mesh = np.meshgrid(lambda_bar, k_bar)
304
305     if analytic_compare:
306         analytic_comparison(mode_to_plot, k_bar_mesh, lambda_bar_mesh, epsilon,
307                             label_pos)
308
309     if show_points:
310         plt.scatter(lambda_a_mesh, k_a_mesh, marker='o', c='b', s=5)
311     plt.ylim(0.01, bounds[0])
312     plt.xlim(0.01, bounds[1])
313     axes = plt.gca()
314     plt.setp(axes.get_xticklabels(), fontsize=40)
315     plt.setp(axes.get_yticklabels(), fontsize=40)
316     plt.ylabel(r'$\bar{k}$', fontsize=45, rotation='horizontal', labelpad=30)
317     plt.xlabel(r'$\bar{\lambda}$', fontsize=45)
318     cbar.ax.tick_params(labelsize=40)
319     sns.despine(ax=axes)
320     plt.tight_layout()
321     plt.savefig('../../output/plots/' + name + '.png')
322     plt.show()
323
324
325 def analytic_comparison_flex(mode_to_plot, k_bar_mesh, lambda_bar_mesh, epsilon,
326                              delta_values, label_pos):
327     r"""
328     Add red lines indicating stability boundaries from analytical model.
329     """
330     line_labels = FixedFormatter(delta_values)
331
332     assert (mode_to_plot == 'm_neg_1' or
333             mode_to_plot == 'm_0'), ("Please specify mode_to_plot as either" +
```

```
                                    "m_neg_1 or m_0")

    if mode_to_plot == 'm_neg_1':
        m = 1
        color = 'red'
    if mode_to_plot == 'm_0':
        m = 0
        color = 'red'


    stability_kink_given_delta = []

    for delta in delta_values:
        stability_kink_given_delta.append(ac.conditions(k_bar_mesh,
                                                        lambda_bar_mesh,
                                                        epsilon,
                                                        m,
                                                        delta))


    stability_kink = stability_kink_given_delta[0] < 0
    stability_kink = stability_kink.astype(float)
    stability_kink[stability_kink_given_delta[0] >= 0] = -1.5
    stability_kink[stability_kink_given_delta[0] < 0] = -0.5
    value = 0.5

    for i in range(len(delta_values[1:])):
        stability_kink[stability_kink_given_delta[i] < 0] = value
        value += 1.

    levels = np.array(range(len(delta_values))) - 1

    cs = plt.contour(lambda_bar_mesh, k_bar_mesh, stability_kink,
                     levels=levels, colors=color, linewidths=5,
                     linestyles='dotted')

    line_labels = {}

    for i, level in enumerate(levels):
        line_labels.update({level: r'$\delta = $ %2.1f' % (delta_values[i])})
    print(levels, value, line_labels)
    plt.clabel(cs, fmt=line_labels, fontsize=40, manual=label_pos)
    return cs


def single_analytic_comparison(mode_to_plot,
                               k_bar_mesh,
                               lambda_bar_mesh,
                               epsilon,
                               delta_value,
                               label_pos):
    """
    Add contour of analytic stability condition.
    """

    line_labels = FixedFormatter(delta_value)

```

```
391         assert (mode_to_plot == 'm_neg_1' or
392                 mode_to_plot == 'm_0'), ("Please specify mode_to_plot as either" +
393                                          "m_neg_1 or m_0")
394 
395         if mode_to_plot == 'm_neg_1':
396             m = 1
397             color = 'red'
398         if mode_to_plot == 'm_0':
399             m = 0
400             color = 'red'
401 
402 
403         stability_kink_given_delta = []
404 
405         stability_kink_given_delta.append(ac.conditions(k_bar_mesh,
406                                                         lambda_bar_mesh,
407                                                         epsilon,
408                                                         m,
409                                                         delta_value))
410 
411 
412         stability_kink = stability_kink_given_delta[0] < 0
413         stability_kink = stability_kink.astype(float)
414         stability_kink[stability_kink_given_delta[0] >= 0] = -1.5
415         stability_kink[stability_kink_given_delta[0] < 0] = -0.5
416 
417         levels = [-1]
418 
419         cs = plt.contour(lambda_bar_mesh, k_bar_mesh, stability_kink,
420                          levels=levels, colors=color, linewidths=5,
421                          linestyles='dotted')
422 
423         line_labels = {}
424 
425         #plt.clabel(cs, fmt={-1: '%2.1f' % delta_value}, fontsize=40, manual=label_pos)
426         return cs
427 
428     def analytic_comparison(mode_to_plot, k_bar_mesh, lambda_bar_mesh, epsilon,
429                             label_pos, lines=None, colors=None):
430         r"""
431         Add red lines indicating stability boundaries from analytical model.
432         """
433         if not lines:
434             line_labels = FixedFormatter(['-1', '0', '1'])
435         else:
436             line_labels = FixedFormatter([str(line) for line in lines])
437 
438         assert (mode_to_plot == 'm_neg_1' or
439                 mode_to_plot == 'm_0'), ("Please specify mode_to_plot as either" +
440                                          "m_neg_1 or m_0")
441 
442 
443         if mode_to_plot == 'm_neg_1':
444             m = 1
445             if not colors:
446                 color = 'red'
447             else:
```

```
448             color = colors
449     if mode_to_plot == 'm_0':
450         m = 0
451         if not colors:
452             color = 'red'
453         else:
454             color = colors
455 
456     if not lines:
457         stability_kink_m_neg_1 = ac.conditions(k_bar_mesh, lambda_bar_mesh,
458                                                epsilon, m, -1.)
459         stability_kink_m_0 = ac.conditions(k_bar_mesh, lambda_bar_mesh,
460                                            epsilon, m, 0.)
461         stability_kink_m_1 = ac.conditions(k_bar_mesh, lambda_bar_mesh,
462                                            epsilon, m, 1)
463     else:
464         stability_kink_m_neg_1 = ac.conditions(k_bar_mesh, lambda_bar_mesh,
465                                                epsilon, m, lines[0])
466         stability_kink_m_0 = ac.conditions(k_bar_mesh, lambda_bar_mesh,
467                                            epsilon, m, lines[1])
468         stability_kink_m_1 = ac.conditions(k_bar_mesh, lambda_bar_mesh,
469                                            epsilon, m, lines[2])
470 
471 
472     stability_kink = stability_kink_m_neg_1 < 0
473     stability_kink = stability_kink.astype(float)
474     stability_kink[stability_kink_m_neg_1 >= 0] = -1.5
475     stability_kink[stability_kink_m_neg_1 < 0] = -0.5
476     stability_kink[stability_kink_m_0 < 0] = 0.5
477     stability_kink[stability_kink_m_1 < 0] = 1.5
478     cs = plt.contour(lambda_bar_mesh, k_bar_mesh, stability_kink,
479                      levels=[-1, 0, 1], colors=color, linewidths=10,
480                      linestyles='dotted')
481 
482     if not lines:
483         plt.clabel(cs, fmt={-1: r'$\delta = -1$', 0: r'$\delta = 0$',
484                             1: r'$\delta = 1$'}, fontsize=40, manual=label_pos)
485     else:
486        plt.clabel(cs, fmt={-1: r'$\delta =$ %' % lines[0], 0: r'$\delta =$ %' % lines[1],
487                            1: r'$\delta =$ %' % lines[2]}, fontsize=40, manual=label_pos)
488     return cs
489 
490 def plot_lambda_k_space_delta(filename, mode_to_plot,
491                               clip=False, delta_min=-1.5,
492                               delta_max=1., levels=None,
493                               interpolate=True, compare_analytic=False,
494                               epsilon=None, analytic_label_pos=None, lines=None,
495                               plot_numeric_boundary=False, cmap=None, analytic_color=None):
496     r"""
497     Plot values of delta in lambda k space.
498     """
499     data_meshes = np.load(filename)
500     lambda_mesh = data_meshes['lambda_a_mesh']
501     k_mesh = data_meshes['k_a_mesh']
502 
503 
504     if mode_to_plot == 0:
```

```
505             color = 'green'
506             delta_mesh = data_meshes['delta_m_0']
507             external_sausage = data_meshes['d_w_m_0']
508         else:
509             #color = sns.xkcd_rgb["dandelion"]
510             color = 'green'
511             delta_mesh = data_meshes['delta_m_neg_1']
512             external_kink = data_meshes['d_w_m_neg_1']
513
514         if interpolate:
515             delta_mesh = interpolate_nans(lambda_mesh,
516                                           k_mesh,
517                                           delta_mesh)
518
519
520         if clip:
521             delta_mesh = np.clip(delta_mesh, delta_min, delta_max)
522
523         if cmap:
524             colors = cmap
525         else:
526             colors = sns.light_palette(color, n_colors=6, reverse=True,
527                                        as_cmap=True)
528
529         if levels:
530             plt.contourf(lambda_mesh, k_mesh, delta_mesh, cmap=colors,
531                          levels=levels)
532         else:
533             plt.contourf(lambda_mesh, k_mesh, delta_mesh, cmap=colors)
534
535         cbar = plt.colorbar(label=r'$\delta$')
536         cbar.set_label(label=r'$\delta(\bar{\lambda},\bar{k})$', size=45, rotation=0, labelpad=30)
537
538         if levels:
539             contourlines = plt.contour(lambda_mesh, k_mesh, delta_mesh,
540                                        colors='grey', levels=levels)
541         else:
542             contourlines = plt.contour(lambda_mesh, k_mesh, delta_mesh,
543                                        colors='grey')
544
545         cbar.add_lines(contourlines)
546
547
548         if mode_to_plot == 0:
549             mode_to_plot = 'm_0'
550         else:
551             mode_to_plot = 'm_neg_1'
552
553         if compare_analytic:
554             if analytic_color:
555                 analytic_comparison(mode_to_plot, k_mesh, lambda_mesh, epsilon,
556                                     analytic_label_pos, lines=lines, colors=analytic_color)
557             else:
558                 analytic_comparison(mode_to_plot, k_mesh, lambda_mesh, epsilon,
559                                     analytic_label_pos, lines=lines)
560
561
```

```
562     if plot_numeric_boundary:
563         contour = plt.contour(lambda_mesh,
564                               k_mesh,
565                               external_sausage,
566                               levels=[0],
567                               colors='grey',
568                               linestyles='-.')
569
570
571
572     axes = plt.gca()
573     axes.set_axis_bgcolor(sns.xkcd_rgb['grey'])
574     plt.setp(axes.get_xticklabels(), fontsize=40)
575     plt.setp(axes.get_yticklabels(), fontsize=40)
576     plt.ylabel(r'$\bar{k}$', fontsize=45, rotation='horizontal', labelpad=30)
577     plt.xlabel(r'$\bar{\lambda}$', fontsize=45)
578     cbar.ax.tick_params(labelsize=40)
579     axes.set_xticks(np.arange(0., 5, 1.))
580     axes.set_yticks(np.arange(0., 2.0, 0.5))
581
582     plt.ylim(0.01, 1.5)
583     plt.xlim(0.01, 3.0)
584     sns.despine(ax=axes)
585     plt.tight_layout()
586
587 def sausage_kink_ratio(filename, xy_limits=None, cmap=None, save_as=None,
588                        levels=None, return_ax=False):
589     r"""
590     Plot ratio of sausage and kink potential energies.
591     """
592     meshes = np.load(filename)
593     lambda_bar_mesh = meshes['lambda_a_mesh']
594     k_bar_mesh = meshes['k_a_mesh']
595     external_m_neg_1 = meshes['d_w_m_neg_1']
596     external_sausage = meshes['d_w_m_0']
597     meshes.close()
598
599     sausage_stable_region = np.invert((external_sausage < 0))
600     ratio = np.abs(external_sausage / external_m_neg_1)
601     ratio[sausage_stable_region] = np.nan
602     ratio_log = np.log10(ratio)
603
604     if not cmap:
605         cmap = sns.light_palette(sns.xkcd_rgb['red orange'],
606                                  as_cmap=True)
607     if levels:
608         contours = plt.contourf(lambda_bar_mesh, k_bar_mesh,
609                                 ratio_log, cmap=cmap, levels=levels)
610     else:
611         contours = plt.contourf(lambda_bar_mesh, k_bar_mesh,
612                                 ratio_log, cmap=cmap)
613
614     colorbar = plt.colorbar(format=FormatStrFormatter(r'$10^{%i}$'))
615     colorbar.set_label(r'$\frac{\delta W_{m=0}}{\delta W_{m=-1}}$',
616                        size=35, rotation=0, labelpad=50)
617     if levels:
618         lines = plt.contour(lambda_bar_mesh, k_bar_mesh,
```

```
                            ratio_log, colors='grey', levels=levels)
    else:
        lines = plt.contour(lambda_bar_mesh, k_bar_mesh,
                            ratio_log, colors='grey')
    colorbar.add_lines(lines)

    axes = plt.gca()
    axes.plot([0, 3.], [0., 1.5], '—', c='black', lw=5)
    axes.set_xlabel(r'$\bar{\lambda}$', fontsize=40)
    plt.setp(axes.get_xticklabels(), fontsize=30)
    axes.set_xticks(np.arange(0., 4.5, 0.5))

    axes.set_ylabel(r'$\bar{k}$', fontsize=40)
    plt.setp(axes.get_yticklabels(), fontsize=30)
    axes.set_yticks(np.arange(0., 2.0, 0.5))

    if xy_limits:
        axes.set_ylim((xy_limits[0], xy_limits[1]))
        axes.set_xlim((xy_limits[2], xy_limits[3]))

    sns.despine()
    colorbar.ax.yaxis.set_ticks_position('right')
    colorbar.ax.tick_params(labelsize=30)
    plt.tight_layout()
    if return_ax:
        return axes, colobar
    else:
        if save_as:
            plt.savefig(save_as)
        plt.show()
```

A.6.14 analytic_condition.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov  4 13:25:43 2015

@author: jensv

Analytic stability condition derived for lengthening
current-carrying magnetic flux tube with core and skin
currents.
"""

import numpy as np
from scipy.special import kv, kvp

import matplotlib.pyplot as plt
from matplotlib import colors
import seaborn as sns
sns.set_style('ticks')
sns.set_context('poster')


```

```
22  def conditions(k_bar, lambda_bar, epsilon, m, delta):
23      r"""
24      Return analytic stability condition.
25
26      Parameters
27      ----------
28      k_bar : float
29          normalized inverse aspect ratio
30      lambda_bar : float
31          normalized current-to-magnetic flux ratio
32      epsilon : float
33          core to total current ratio
34      m : float
35          azimuthal periodicity number
36      delta : float
37          abruptness parameter
38
39      Returns
40      -------
41      delta_w : float
42          perturbed potential energy of marginal stability case
43      """
44      term1 = conditions_plasma_term(k_bar, lambda_bar, epsilon, m, delta)
45      term2 = conditions_interface_term(k_bar, lambda_bar, epsilon, m, delta)
46      term3 = conditions_vacuum_term(k_bar, lambda_bar, m, delta)
47      return term1 + term2 - term3
48
49
50  def conditions_without_interface(k_bar, lambda_bar, m, delta):
51      r"""
52      Return analytic stability condition minus interface term (term2).
53
54      Parameters
55      ----------
56      k_bar : float
57          normalized inverse aspect ratio
58      lambda_bar : float
59          normalized current-to-magnetic flux ratio
60      m : float
61          azimuthal periodicity number
62      delta : float
63          abruptness parameter
64
65      Returns
66      -------
67      delta_w_without_interface : float
68          perturbed potential energy without interface term.
69
70      Notes
71      -----
72      For profiles with current smoothly going to zero at the delta_w term is zero.
73      """
74      term1 = conditions_smooth_plasma_term(k_bar, lambda_bar, m, delta)
75      term3 = conditions_vacuum_term(k_bar, lambda_bar, m, delta)
76      return term1 - term3
77
78
```

```
def conditions_without_interface_wo_sing(k_bar, lambda_bar, m, xi,
                                         xi_der, a):
    r"""
    Multiply analytic expression with xi squared to avoid singularity.

    Parameters
    ----------
    k_bar : float
        normalized inverse aspect ratio
    lambda_bar : float
        normalized current-to-magnetic flux ratio
    m : float
        azimuthal periodicity number
    xi : float
        solution to Euler-Lagrange equation at boundary
    xi_der : float
        derivative of solution to Euler-Lagrange equation at boundary
    a : float
        radius of current-carrying magnetic flux tube
    Returns
    -------
    delta_w_without_interface_wo_sing : float
        perturbed potential energy without interface or singularity

    Notes
    -----
    delta can be singular when xi goes through zero. This form is multiplied
    by xi**2 to avoid singularity.
    """
    term1 = conditions_smooth_plasma_term_wo_sing(k_bar, lambda_bar,
                                                  m, xi, xi_der, a)
    term3 = conditions_vacuum_term_wo_sing(k_bar, lambda_bar, m, xi)
    return term1 - term3


def conditions_plasma_term(k_bar, lambda_bar, epsilon, m, delta):
    r"""
    Returns plasma term of analytic stability condition.

    Parameters
    ----------
    k_bar : float
        normalized inverse aspect ratio
    lambda_bar : float
        normalized current-to-magnetic flux ratio
    epsilon : float
        core to total current ratio
    m : float
        azimuthal periodicity number
    delta : float
        abruptness parameter

    Returns
    -------
    delta_w_plasma_term : float
        perturbed potential energy plasma term due to internal
        currents.
```

```
136     """
137     term1 = (2.*k_bar - m*epsilon*lambda_bar)*((delta + 1)*2.*k_bar -
138                                                (delta - 1)*m*epsilon *
139                                                lambda_bar)/(k_bar**2 + m**2)
140     return term1
141
142
143 def conditions_smooth_plasma_term_wo_sing(k_bar, lambda_bar, m, xi,
144                                           xi_der, a):
145     r"""
146     Multiply analytic expression with xi squared to avoid singularity.
147
148     Parameters
149     ----------
150     k_bar : float
151         normalized inverse aspect ratio
152     lambda_bar : float
153         normalized current-to-magnetic flux ratio
154     m : float
155         azimuthal periodicity number
156     xi : float
157         solution to Euler-Lagrange equation at boundary
158     xi_der : float
159         derivative of solution to Euler-Lagrange equation at boundary
160     a : float
161         radius of current-carrying magnetic flux tube
162
163     """
164     epsilon = 1.
165     term1 = (2.*k_bar - m*epsilon*lambda_bar)*((xi_der*a*xi + xi**2)*2.*k_bar -
166                                                (xi_der*a*xi - xi**2)*m*epsilon*
167                                                lambda_bar)/(k_bar**2 + m**2)
168     return term1
169
170
171 def conditions_smooth_plasma_term(k_bar, lambda_bar, m, delta):
172     r"""
173     Returns plasma term of analytic condition with epsilon set to 1. This
174     should be relvant for a profile that smoothly goes to zero current,
175     since b_v(a) = b_p(a) in that case.
176
177     Parameters
178     ----------
179     k_bar : float
180         normalized inverse aspect ratio
181     lambda_bar : float
182         normalized current-to-magnetic flux ratio
183     m : float
184         azimuthal periodicity number
185     delta : float
186         abruptness parameter
187
188     """
189     epsilon = 1.
190     term1 = conditions_plasma_term(k_bar, lambda_bar, epsilon, m, delta)
191     return term1
192
```

```
193
194 def conditions_interface_term(k_bar, lambda_bar, epsilon, m, delta):
195     r"""
196     Returns interface term of analytic stability condition.
197
198     Parameters
199     ----------
200     k_bar : float
201         normalized inverse aspect ratio
202     lambda_bar : float
203         normalized current-to-magnetic flux ratio
204     epsilon : float
205         core to total current ratio
206     m : float
207         azimuthal periodicity number
208     delta : float
209         abruptness parameter
210
211     """
212     term2 = (epsilon**2 - 1) * lambda_bar**2
213     return term2
214
215
216 def conditions_vacuum_term(k_bar, lambda_bar, m, delta):
217     r"""
218     Returns vacuum term of analytic stability condition.
219
220     Parameters
221     ----------
222     k_bar : float
223         normalized inverse aspect ratio
224     lambda_bar : float
225         normalized current-to-magnetic flux ratio
226     m : float
227         azimuthal periodicity number
228     delta : float
229         abruptness parameter
230     """
231     term3 = (m*lambda_bar - 2.*k_bar)**2/k_bar*(kv(m, np.abs(k_bar)) /
232                                                 kvp(m, np.abs(k_bar)))
233     return term3
234
235 def conditions_vacuum_term_wo_sing(k_bar, lambda_bar, m, xi):
236     r"""
237     Multiply analytic expression with xi squared to avoid singularity.
238
239     Parameters
240     ----------
241     k_bar : float
242         normalized inverse aspect ratio
243     lambda_bar : float
244         normalized current-to-magnetic flux ratio
245     m : float
246         azimuthal periodicity number
247     xi : float
248         Euler-Lagrange solution
249     """
```

```
250     term3 = xi**2 * (m*lambda_bar - 2.*k_bar)**2/k_bar*(kv(m, np.abs(k_bar)) /
251                                                         kvp(m, np.abs(k_bar)))
252     return term3
253
254
255 def condition_map(epsilon=0.5, delta=0.):
256     r"""
257     Draw filled contours of sausage (orange), kink(yellow), and stable (white)
258     regions for given epsilon and delta values.
259
260     Parameters
261     ----------
262     epsilon : float
263         core to total current ratio
264     delta : float
265         abruptness parameter
266     """
267     fig = plt.figure(figsize=(10,10))
268     lambda_bar = np.linspace(0., 3., 750)
269     k_bar = np.linspace(0, 1.5, 750)
270     lambda_bar_mesh, k_bar_mesh = np.meshgrid(lambda_bar, k_bar)
271
272     stability_kink = conditions(k_bar_mesh, lambda_bar_mesh, epsilon, 1., delta)
273     stability_kink = stability_kink < 0
274     stability_sausage = conditions(k_bar_mesh, lambda_bar_mesh, epsilon, 0., delta)
275     stability_sausage = stability_sausage < 0
276     stability_kink = stability_kink.astype(float)
277     stability_kink[stability_sausage] = 2
278
279     cmap = colors.ListedColormap([sns.xkcd_rgb["white"],
280                                   sns.xkcd_rgb["yellow"], sns.xkcd_rgb["orange"]])
281     plt.contourf(lambda_bar_mesh, k_bar_mesh, stability_kink,
282                  cmap=cmap, levels=[0., 0.5, 1.5, 2.])
283     plt.contour(lambda_bar_mesh, k_bar_mesh, stability_kink,
284                 levels=[0., 0.5, 1.5, 2.], colors='grey')
285     plt.plot([0, 3.], [0., 1.5], '--', c='black', lw=5)
286     axes = plt.gca()
287
288     plt.setp(axes.get_xticklabels(), fontsize=40)
289     plt.setp(axes.get_yticklabels(), fontsize=40)
290     plt.ylabel(r'$\bar{k}$', fontsize=45, rotation='horizontal', labelpad=25)
291     plt.xlabel(r'$\bar{\lambda}$', fontsize=45)
292     sns.despine()
293
294
295 def condition_map_variable_delta(filename, mode=1, epsilon=0.5,
296                                  conditions_func=conditions_without_interface):
297     r"""
298     Draw filled contours of sausage (orange), kink(yellow), and stable (white)
299     regions for given epsilon and delta values.
300     Delta values are loaded from a .npz mesh file.
301
302     Parameters
303     ----------
304     filename : string
305         filename from which to load lambda_bar, k_bar and delta values.
306     mode : int
```

```
307         azimuthal mode number 0 or 1
308     epsilon : float
309         core current to total current ratio
310     conditions_func : function
311         conditions function to use
312     """
313     data_meshes = np.load(filename)
314     lambda_bar_mesh = data_meshes['lambda_a_mesh']
315     k_bar_mesh = data_meshes['k_a_mesh']
316     delta_mesh = data_meshes['delta_m_0']
317
318     fig = plt.figure(figsize=(10,10))
319
320     cmap = colors.ListedColormap([sns.xkcd_rgb["white"],
321                                   sns.xkcd_rgb["yellow"],
322                                   sns.xkcd_rgb["orange"]])
323
324     stability_kink = conditions_func(k_bar_mesh, lambda_bar_mesh, epsilon, 1.,
325                                      delta_mesh)
326     stability_kink = stability_kink < 0
327     stability_sausage = conditions_func(k_bar_mesh, lambda_bar_mesh, epsilon, 0.,
328                                         delta_mesh)
329     stability_sausage = stability_sausage < 0
330
331     if mode == 0:
332         stability = stability_sausage
333         cmap = colors.ListedColormap([sns.xkcd_rgb["white"],
334                                       sns.xkcd_rgb["orange"]])
335     else:
336         stability = stability_kink
337         cmap = colors.ListedColormap([sns.xkcd_rgb["white"],
338                                       sns.xkcd_rgb["yellow"]])
339
340     plt.contourf(lambda_bar_mesh, k_bar_mesh, stability,
341                  cmap=cmap, levels=[0.5, 1.5])
342     plt.contour(lambda_bar_mesh, k_bar_mesh, stability,
343                 levels=[0.5, 1.5], colors='grey')
344     plt.plot([0, 3.], [0., 1.5], '—', c='black', lw=5)
345     axes = plt.gca()
346
347     plt.setp(axes.get_xticklabels(), fontsize=40)
348     plt.setp(axes.get_yticklabels(), fontsize=40)
349     plt.ylabel(r'$\bar{k}$', fontsize=45, rotation='horizontal', labelpad=25)
350     plt.xlabel(r'$\bar{\lambda}$', fontsize=45)
351     sns.despine()
352     plt.show()
```

A.6.15 diagnostic_plots.py

```
1   """
2   Created on Tue Nov 22 11:12:58 2016
3
4   @author: Jens von der Linden
5   """
```

```
6
7  from __future__ import print_function
8  from __future__ import division
9  from __future__ import absolute_import
10 from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
11                                int, map, next, oct, open, pow, range, round,
12                                str, super, zip)
13 """Python 3.x compatibility"""
14
15 import sys
16
17 import scipy.integrate
18
19 import singularity_frobenius as frob
20
21 import matplotlib.pyplot as plt
22 import seaborn as sns
23 sns.set_style('whitegrid')
24 sns.set_context('poster')
25
26 def plot_all_profiles_suydam(profile, normalize=False,
27                                mu_0=None, axes=None, title=None):
28     r"""
29     Plot radial profiles of current, magnetic field, pressure, safety_factor,
30     and Suydam criterion.
31
32     Parameters
33     ----------
34     profiles : profile_object
35         profile created with equil_solver
36     normalize : bool
37         boolean flag to normalize profiles to max values
38     mu_0 : float
39          magnetic permeability value (used for non-dimensionless profiles)
40     axes : matplotlib axes
41         plotting axes to use
42     title :
43         plot title
44     """
45     if not axes:
46         axes = plt.gca()
47     r = np.linspace(0, 1, 250)
48     splines = profile.get_splines()
49     j_z = splines['j_z'](r)
50     b_z = splines['b_z'](r)
51     b_z_prime = splines['b_z'].derivative()(r)
52     b_theta = splines['b_theta'](r)
53     b_theta_prime = splines['b_theta'].derivative()(r)
54     p_prime = splines['p_prime'](r)
55     p = splines['pressure'](r)
56     safety_factor = splines['q'](r)
57     safety_factor_prime = splines['q'].derivative()(r)
58     if mu_0:
59        beta_0 = mu_0
60     else:
61         beta_0 = profile.beta_0()
62     alpha = frob.alpha_func(r, b_z, b_z_prime, b_theta, b_theta_prime)
```

```
63      beta = frob.beta_func(b_z, b_theta, p_prime, beta_0)
64      suydam_mu = alpha + 4.*beta
65      suydam_q = r*b_z**2./(8. * beta_0)*(safety_factor_prime/safety_factor)**2. + p_prime
66      if 'j_theta' in splines.keys():
67          print('true')
68          j_theta = splines['j_theta'](r)
69      else:
70          print('false')
71          j_theta = None
72      if normalize:
73          j_z = j_z/np.max(np.abs(j_z))
74          b_theta = b_theta/np.max(np.abs(b_theta))
75          alpha = alpha/np.max(np.abs(alpha))
76          beta = beta/np.max(np.abs(beta))
77          suydam_q = suydam_q/np.max(np.abs(suydam_q))
78          suydam_mu = suydam_mu/np.nanmax(np.abs(suydam_mu))
79          p = p/np.max(np.abs(p))
80          p_prime =  p_prime/np.max(np.abs(p_prime))
81          safety_factor_prime =  safety_factor_prime/np.max(np.abs(safety_factor_prime))
82      if 'j_theta' in splines.keys():
83              j_theta = j_theta/np.nanmax(np.abs(j_theta))
84      axes.plot(r, j_z, c='#087804', label=r'$j_z$')
85      if 'j_theta' in splines.keys():
86          axes.plot(r, j_theta, c='#6fc276', label=r'$j_\theta$')
87      axes.plot(r, b_theta, c='#e50000', label=r'$B_\theta$')
88      axes.plot(r, p_prime, c='#7e1e9c', label=r"$p'$")
89      axes.plot(r, p, c='#bf77f6', label=r"$p$")
90      axes.plot(r, safety_factor, c='#000000', label=r"$q$")
91      axes.plot(r, safety_factor_prime, c='#7d7f7c', label=r"$q'$")
92      axes.plot(r[1:], suydam_q[1:], c='#acfffc', label=r"suydam_q")
93      #axes.plot(r[1:], suydam_mu[1:], c='#82cafc', label=r"suydam_mu")
94      if title:
95          axes.set_title(title)
96      axes.legend(loc='best')
97      return axes
98
99  def plot_suydam(profile, normalize=False):
100     r"""
101     Plot suydam stability conditions.
102
103     Parameters
104     ----------
105     profiles : profile_object
106         profile created with equil_solver
107     normalize : bool
108         boolean flag to normalize profiles to max value
109     """
110     axes = plt.gca()
111     r = np.linspace(0, 1, 250)
112     splines = profile.get_splines()
113     j_z = splines['j_z'](r)
114     b_z = splines['b_z'](r)
115     b_z_prime = splines['b_z'].derivative()(r)
116     b_theta = splines['b_theta'](r)
117     b_theta_prime = splines['b_theta'](derivative)(r)
118     p_prime = splines['p_prime'](r)
119     p = splines['pressure'](r)
```

```
120     safety_factor = splines['q'](r)
121     safety_factor_prime = splines['q'].derivative()(r)
122     beta_0 = profile.beta_0()
123     alpha = frob.alpha_func(r, b_z, b_z_prime, b_theta, b_theta_prime)
124     beta = frob.beta_func(b_z, b_theta, p_prime, beta_0)
125     suydam_mu = alpha + 4.*beta
126     suydam_q = r*b_z**2./(8. * beta_0)*(safety_factor_prime/safety_factor)**2. + p_prime
127     if normalize:
128         j_z = j_z/np.max(np.abs(j_z))
129         b_theta = b_theta/np.max(np.abs(b_theta))
130         alpha = alpha/np.max(np.abs(alpha))
131         beta = beta/np.max(np.abs(beta))
132         suydam_q = suydam_q/np.max(np.abs(suydam_q))
133         suydam_mu = suydam_mu/np.nanmax(np.abs(suydam_mu))
134         p_prime =  p_prime/np.max(np.abs(p_prime))
135         safety_factor_prime = safety_factor_prime/np.max(np.abs(safety_factor_prime))
136     #axes.plot(r, j_z, c='#087804', label=r'$j_z$')
137     #axes.plot(r, b_theta, c='#e50000', label=r'$B_\theta$')
138     #axes.plot(r, p_prime, c='#7e1e9c', label=r"$p'$")
139     #axes.plot(r, p, c='#bf77f6', label=r"$p$")
140     #axes.plot(r, safety_factor, c='#000000', label=r"$q$")
141     #axes.plot(r, safety_factor_prime, c='#7d7f7c', label=r"$q'$")
142     axes.plot(r, suydam_q, c='#acfffc', label=r"suydam_q")
143     axes.plot(r, suydam_mu, c='#82cafc', label=r"suydam_mu")
144     axes.legend(loc='best')
145     return suydam_q, suydam_mu, axes
```

A.6.16 test_equil_solver.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Thu Nov  5 09:44:38 2015
4
5   @author: jensv
6   """
7
8   import numpy as np
9   import numpy.testing as test
10  import scipy.integrate as integrate
11  import equil_solver as es
12  from scipy.interpolate import splev
13  from nose.tools import with_setup
14
15  test_equil = None
16
17  def setup_func():
18      r"""
19      Generate test equilibrium.
20      """
21      global test_equil
22      test_equil = es.UnitlessSmoothedCoreSkin(core_radius_norm=0.9,
23                                               transition_width_norm=0.033,
24                                               skin_width_norm=0.034,
25                                               epsilon=0.9, lambda_bar=.5)
```

```
26
27
28  def teardown_func():
29      pass
30
31
32  @with_setup(setup_func, teardown_func)
33  def test_epsilon():
34      r"""
35      Test that ratio of b_theta gives epsilon.
36      """
37      r_core = test_equil.core_radius
38      a = (test_equil.core_radius + 2.*test_equil.transition_width +
39           test_equil.skin_width)
40      b_theta_tck = test_equil.get_tck_splines()['b_theta']
41      epsilon_from_b_theta_ratio = (splev(r_core, b_theta_tck) /
42                                    splev(a, b_theta_tck))
43      test.assert_approx_equal(epsilon_from_b_theta_ratio, test_equil.epsilon,
44                               significant=3)
45
46
47  @with_setup(setup_func, teardown_func)
48  def test_lambda_bar():
49      r"""
50      Test that lambda bar is given by ratio of total current to magnetic flux.
51      """
52      a = (test_equil.core_radius + 2.*test_equil.transition_width +
53           test_equil.skin_width)
54      b_theta_tck = test_equil.get_tck_splines()['b_theta']
55      b_z_tck = test_equil.get_tck_splines()['b_z']
56      calculated_lambda = (2.*splev(a, b_theta_tck) /
57                           (splev(a, b_z_tck)))
58      test.assert_approx_equal(calculated_lambda, test_equil.lambda_bar,
59                               significant=3)
```

A.6.17 test_newcomb.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Tue May 20 14:05:57 2014
4
5   @author: Jens von der Linden
6
7   Tests based on relations in Goedbloed (2010) Principles of MHD, Jardin &
8   Newcomb paper.
9   All equation references are from these books unless ofterwise noted.
10  """
11
12  from __future__ import print_function
13  from __future__ import division
14  from __future__ import absolute_import
15  from future.builtins import (ascii, bytes, chr, dict, filter, hex, input,
16                               int, map, next, oct, open, pow, range, round,
17                               str, super, zip)
```

```
18  """Python 3.x compatibility"""
19
20  import math
21  import numpy as np
22  from numpy import atleast_1d as d1
23  from numpy.testing import assert_allclose, assert_array_less
24  import newcomb_g as g
25  import newcomb_f as f
26
27
28  class TestParabolic(object):
29      r"""
30      Compare f and g outputs for a parabolic current profile.
31      """
32
33      def setUp(self):
34          r"""
35          Create a parabolic axial current equilibrium profile.
36          """
37          import equil_solver as es
38          parabolic = es.ParabolicNu2()
39          params = {'mu0': 1.0, 'k': (2*math.pi)/1.0, 'n': 1.0, 'm': 1.0,
40                    'a': 1.0, 'b': 10.0, 'omega_sq': -5.0, 'gamma': 1.0,
41                    'constant': 1.0}
42          splines = parabolic.get_splines()
43          params.update(splines)
44          self.params = params
45          self.delta = 10E-5
46
47      def test_jardin_f_negative(self):
48          r"""
49          Test that f is never negative for all r.
50          """
51          k, m, b_z, b_theta, q = map(self.params.get, ['k', 'm', 'b_z',
52                                                        'b_theta', 'q'])
53          r = np.linspace(0, 1, 100)
54          f_params = {'r': r, 'k': d1(k), 'm': d1(m),
55                      'b_z': b_z(r), 'b_theta': b_theta(r),
56                      'q': q(r)}
57
58          assert_array_less(-1E-14, f.jardin_f_8_78(**f_params))
59
60      def test_jardin_f_newcomb_f(self):
61          r"""
62          Test that the jardin_f and newcomb_f expressions are equivalent.
63          """
64          k, m, b_z, b_theta, q = map(self.params.get, ['k', 'm', 'b_z',
65                                                        'b_theta', 'q'])
66          r = np.linspace(0, 1, 100)
67          f_params = {'r': r, 'k': d1(k), 'm': d1(m),
68                      'b_z': b_z(r), 'b_theta': b_theta(r), 'q': q(r)}
69
70          assert_allclose(f.jardin_f_8_78(**f_params),
71                          f.newcomb_f_16(**f_params), rtol=0.5)
72
73      def test_jardin_g_8_79_and_8_80(self):
74          r"""
```

```
75          Test that the jardin equation (8.79) and jardin (8.80) equation
76          are equivalent.
77          """
78          (k, m, b_z, b_theta,
79           q, p_prime) = map(self.params.get, ['k', 'm', 'b_z', 'b_theta', 'q',
80                                               'p_prime'])
81          r = np.linspace(0, 1, 100)
82          g_params = {'r': r, 'k': d1(k), 'm': d1(m),
83                      'b_z': b_z(r), 'b_z_prime': b_z.derivative()(r),
84                      'b_theta': b_theta(r),
85                      'b_theta_prime': b_theta.derivative()(r),
86                      'p_prime': p_prime(r), 'q': q(r),
87                      'q_prime': q.derivative()(r)}
88
89          assert_allclose(g.jardin_g_8_79(**g_params),
90                          g.jardin_g_8_80(**g_params), rtol=0.5)
91
92      def test_jardin_g_8_80_newcomb_g_18(self):
93          r"""
94          Test that the jardin g (8.80) and newcomb g (18) expressions are
95          equivalent.
96          """
97          (k, m, b_z, b_theta,
98           q, p_prime) = map(self.params.get, ['k', 'm', 'b_z', 'b_theta', 'q',
99                                               'p_prime'])
100         r = np.linspace(0, 1, 100)
101
102         g_params = {'r': r, 'k': d1(k), 'm': d1(m),
103                     'b_z': b_z(r), 'b_z_prime': b_z.derivative()(r),
104                     'b_theta': b_theta(r),
105                     'b_theta_prime': b_theta.derivative()(r),
106                     'p_prime': p_prime(r), 'q': q(r),
107                     'q_prime': q.derivative()(r)}
108
109         assert_allclose(g.jardin_g_8_80(**g_params),
110                         g.newcomb_g_18(**g_params), rtol=0.5)
111
112     def test_jardin_g_8_79_newcomb_g_18(self):
113         r"""
114         Test that the jardin g (8.79) and newcomb g (18) expressions are
115         equivalent.
116         """
117         (k, m, b_z, b_theta,
118          q, p_prime) = map(self.params.get, ['k', 'm', 'b_z', 'b_theta', 'q',
119                                              'p_prime'])
120         r = np.linspace(0, 1, 100)
121         g_params = {'r': r, 'k': d1(k), 'm': d1(m),
122                     'b_z': b_z(r), 'b_z_prime': b_z.derivative()(r),
123                     'b_theta': b_theta(r),
124                     'b_theta_prime': b_theta.derivative()(r),
125                     'p_prime': p_prime(r), 'q': q(r),
126                     'q_prime': q.derivative()(r)}
127
128         assert_allclose(g.jardin_g_8_79(**g_params),
129                         g.newcomb_g_18(**g_params), rtol=0.5)
130
131     def test_newcomb_g_17_newcomb_g_18(self):
```

```
132         r"""
133         Test that the newcomb g (17) and newcomb g (18) expressions are
134         equivalent.
135         """
136         (k, m, b_z, b_theta,
137          q, p_prime) = map(self.params.get, ['k', 'm', 'b_z', 'b_theta', 'q',
138                                              'p_prime'])
139         r = np.linspace(0, 1, 100)
140         g_params = {'r': r, 'k': d1(k), 'm': d1(m),
141                     'b_z': b_z(r), 'b_z_prime': b_z.derivative()(r),
142                     'b_theta': b_theta(r),
143                     'b_theta_prime': b_theta.derivative()(r),
144                     'p_prime': p_prime(r), 'q': q(r),
145                     'q_prime': q.derivative()(r)}
146         assert_allclose(g.newcomb_g_17(**g_params),
147                         g.newcomb_g_18(**g_params), rtol=0.5)
148
149
150     def test_suydam_q_suydam_alpha_beta(self):
151         r"""
152         Not implemented
153         """
154         pass
155         # r = np.linspace(0, 1, 100)
156         # assert_almost_equal(suydam_q(), suydam_alpha_beta())
157
158     def test_suydam_q_suydam_mu(self):
159         r"""
160         Not implemented
161         """
162         pass
163         # r = np.linspace(0, 1, 100)
164         # assert_almost_equal(suydam_q(), suydam_mu())
165
166     def test_jardin_small_solution_goedbloed_small_solution(self):
167         r"""
168         Not implemented
169         """
170         pass
171         # r = np.linspace(0, 1, 100)
172         # assert_almost_equal(jardin_small_solution(), newcomb_small_solution())
```

Appendix B

MOCHICONTROL & MOCHIFPGACONTROL CODE

This appendix contains instructions for setting up and running the MochiControl and MochiFPGAControl LabVIEW applications. The source code can also be found at Zenodo https://doi.org/10.5281/zenodo.495643 and https://doi.org/10.5281/zenodo.495787 [72, 74]

B.1 MochiControl

Mochi Lab uses this flexible National Instruments LabVIEW experiment control application, based on the producer-consumer design pattern, to allow operators real-time control over all experimental settings, including 16 timing channels with a dynamic timing map, 120 digitizer channels, and a settings save and load interface powered by MDSplus. The hierarchical MDSplus tree stores all experiment settings such as capacitor bank voltages and discharge times together with all data acquired through National Instruments PXI-6133 and NI-5752 digitizer cards, Tektronix TDS-2024B oscilloscopes, and a Princeton Instruments PI-MAX 3 fast framing camera. The experiment control application can also access stored settings through the MDSplus LabVIEW interface which leverages the object oriented and multi-threaded features of MDSplus. To control the experiment the application switches digital outputs of a PXI-6602 timing card connected through fiber optics to relays in the bias coil and plasma gun capacitor banks to achieve the desired charging voltage. The application transmits the desired experiment timings to the PXI-6602 cards which use on board clocks to generate optically isolated trigger pulses for the gas puff valves, oscilloscopes, bias coil and plasma gun capacitor banks. The optical isolation is achieved with TTL-optical transmitter circuits, and DC and fast optical-TTL receiver circuits at the experiment (Carroll 2016).

B.1.1 Requirements

Software Dependencies

The following software can be downloaded from the National Instruments website.

- LabVIEW 2015 (or later)
- DAQmx
- NI-Sync
- MathScript
- LabVIEW FPGA module
- FlexRIO drivers
- FlexRIO adapter drivers

The following software can be downloaded from mdsplus.org.

- MDSplus (the directory with the Vis has to be copied to the LabVIEW directory)

Hardware Setup

This code is designed for this specific hardware.

- PXI-1042Q chassis with
  - PXI-MXI-4 (communication with PC)
  - PXI-6653 (synchronization card)
  - 2x PXI-6602 (Timing cards with 8 channels each)
  - 3x PXI-6133 (Digitizers each 8 channel 2.5 MS/s 14 bit)

- PXIe-1082 chassis with
  - PXIe-8375 (communication with PC)
  - 3x PXIe-7962R with NI-5752 (Digitizers each 32 channel 50MS/s 12 bit)
  - PXIe-6672 (synchronization card)

B.1.2 Important Files

Experiment_Control.lvproj project file this file should be opened with LabVIEW

`Modules/mochifpgacontrol/FPGA bitfiles/MochiFPGAControl_2016-11-29_generic_register_setting_5f6bfda_K10_digital_filters.lvbitx` this bitfile must be loaded on the FPGAs with NI-MAX

B.1.3 Setup

- The code is setup to synchronize two chassis (one leader, one follower). The synchronization cards, one in each chassis should be connected with two coax cables. One coax cable should connect PFI 0 of the leader to PFI 1 of the leader and the other cable PFI 2 of the leader to PFI 0 of the follower. The bitfile has to be loaded onto the FPGA, one way to do this is with NI MAX and the update firmware option.
- The bitfile has to be loaded onto the FPGA, one way to do this is with NI MAX and the update firmware option.
- The NI components have to be named in NI-MAX as follows:
  - PXIe-1082 -> Chassis 1
  - PXIe-7962R -> RIO[0-2]
  - PXIe-6672 -> Sync0
  - PXI-6653 -> Sync

  - PXI-6602 -> Timing[0-1]
  - PXI-6133 -> DAQ[0-2]

- an MDSplus tree named 'op_tree' has to be created with the MochiModel software referenced below.

B.1.4 Adapting the code to new hardware

Adapting this code to new NI components requires significant work. Woodruff Scientific, Inc. designs and installs Controls, Data Access and Communication (CODAC) systems.
http://www.woodruffscientific.com/codac

B.1.5 Reference

- Jens von der Linden. (In preparation). Investigating the Dynamics of Canonical Flux Tubes (Doctoral dissertation). Chapter 4. University of Washington.

- Alexander Card. (In preparation). A new measuremnet of electron densities in the MOCHI LabJet experiment using an unequal path length, heterodyne interferometer (Master's thesis). University of Washington

- Alexander Card. (2017). MochiModel. Zenodo. https://doi.org/10.5281/zenodo.495631

- Jens von der Linden. (2017). MochiFPGAcontrol: High-throughput FPGA code for NI-5752 digitizers. Zenodo. http://doi.org/10.5281/zenodo.439654

- Evan Carroll (2016). Driving Flows in Laboratory Astrophysical Plasma Jets: The Mochi.LabJet Experiment (Master's thesis). University of Washington.

B.1.6 Acknowledgments

James Stuber and Woodruff Scientific, Inc. provided the lab with our initial NI hardware, code, and ongoing support.

B.1.7 Contact

Jens von der Linden jensv@uw.edu Alexander Card card@uw.edu

B.2 MochiFPGAControl

This LabVIEW project programs a PXIe-7962R field programmable arrays (FPGA) to read out 50Mhz 12 bit samples for 800 us from a NI-5752 digitizer.

The PXIe-7962R reads out the samples as 16-bit words, resulting in a data throughput of 3GB/s. This code achieves a throughput to the on board DRAM of 1.5 GB/s, limiting the acquisition time to 800us when the overflow data fills the on board block memory.

B.2.1 Requirements

Software Dependencies

All software can be downloaded from the National Instruments website.

- LabVIEW 2015 (or later)
- LabVIEW FPGA module
- FlexRIO drivers
- FlexRIO adapter drivers
- Xilinx Compile Tools
- NI-Sync

Hardware Requirements

This code is custom designed to this specific hardware. It can, however, serve as an example to be adapted to other NI FPGA based digitizers.

- NI-5752 digitizer
- PXIe-7962R FPGA
- Two NI chassis
- Two NI sync cards (e.g. PXIe 6672, PXI 6652)

The code is designed to work in a double chassis configuration with two synchronization cards. This double chassis requirement could be removed with minor edits in the host Vis.

B.2.2 Important Files

LabVIEW FPGA projects consist of FPGA VIs meant to be compiled to bitfiles and run on the FPGAs, and Host VIs to be run on a computer.

MochiFPGAControl.lvproj is the LabVIEW project file which should be opened with LabVIEW.

host_with_sub_vis_timed_pulse.vi is the recommended host test file. There are options to set the number of samples that should be acquired and to display the acquisition from one channel. An analog digital high pass filter can be set to on or off. It should be set to on. A digital high pass filter can be set with a binary number. The possible settings can be looked up in the AFE 5801.

FPGA Bitfiles/MOCHIFPGAControl_2016-11-29_generic_register_setting_5f6bfda_K10_digital_filters.lvbitx is the bitfile that has to be loaded onto the FPGA.

pxie_7962_fpga.vi is the FPGA vi. The bitfile is compiled from this file.

B.2.3 Setup

The HostVI is setup to synchronize two chassis (one leader, one follower) and trigger the FPGA. The FPGA should be located in the follower. The synchronization cards, one in each chassis should be connected with two coax cables. One coax cable should connect PFI 0 of the leader to PFI 1 of the leader and the other cable PFI 2 of the leader to PFI 0 of the follower. The bitfile has to be loaded onto the FPGA, one way to do this is with NI MAX and the update firmware option.

B.2.4 Reference

- Jens von der Linden. 2017. Investigating the Dynamics of Canonical Flux Tubes (Doctoral dissertation). Chapter 4. University of Washington.
- NI-5752 spec sheet.
- AFE-5801 (Texas Instruments) spec sheet. Each NI-5752 has 4x AFE-5801 ADC chips. The analog filter settings have to be set on ADC chip registers.
- NI LabVIEW High-Performance FPGA Developer's Guide. 2014.

B.2.5 Improvements

- Bitpacking the 16 bit words to 12 bits. This would reduce the data throughput to 2.25GB/s and increase the acquisition time. First steps are in the bitpacking branches in the GitHub repository (MochiLab/mochiFPGAcontrol).
- Use of K7 library to manage DRAM memory. There is a LabVIEW FPGA DRAM memory library that could simplify the read and write process to the DRAM. First steps are in the guys_version branch in the GitHub repository (MochiLab/mochiFPGAcontrol).

- Using digital outputs of the NI-5752 as counters. The clock in the digitizer adapter could be used as a driver for single-cycle timed loop. An upcount vi could count 20ns ticks and then turn on the digital output channels at given counts.

B.2.6 Acknowledgments

Thanks to Glenn Manlongat (National Instruments) for help specifying and selecting the hardware, and Guy McDonnell (National Instruments) for help and advice developing the FPGA code.

B.2.7 Contact

Jens von der Linden jensv@uw.edu

Appendix C

MOCHI.PSU1

This appendix contains figures and engineering drawings of the Mochi.PSU1 (power supply unit). Mochi.PSU1 consists of two identical capacitor banks of 30x 4μF NWL# 12094 capacitors. The capacitors are connected in parallel by two brass plates, one for the hot leads, and one for the ground leads. The brass plates plates are isolated from each other by two insulating high-density polyethylene (HDPE) plates which sandwich insulating hats for the capacitor bank leads, maximizing the tracking length between the leads. Each capacitor bank is switched by an individually triggered A size ignitron. Each ignitron is housed in a coaxial enclosure that is insulated from the bank and control electronics with an acrylic tube.

C.1 Overview figures

Figure C.1: Mochi.PSU1: Two capacitor banks in an 80-20 frame enclosed with ABS plastic panels.

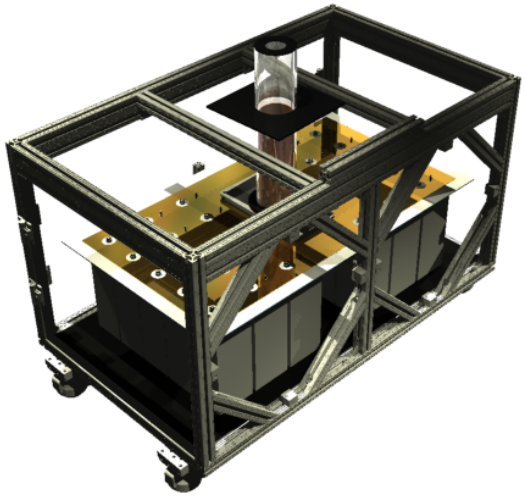


Figure C.2: One bank of PSU1: ABS panels have been removed. The acrylic tube around the ignitron coaxial housing is visible.

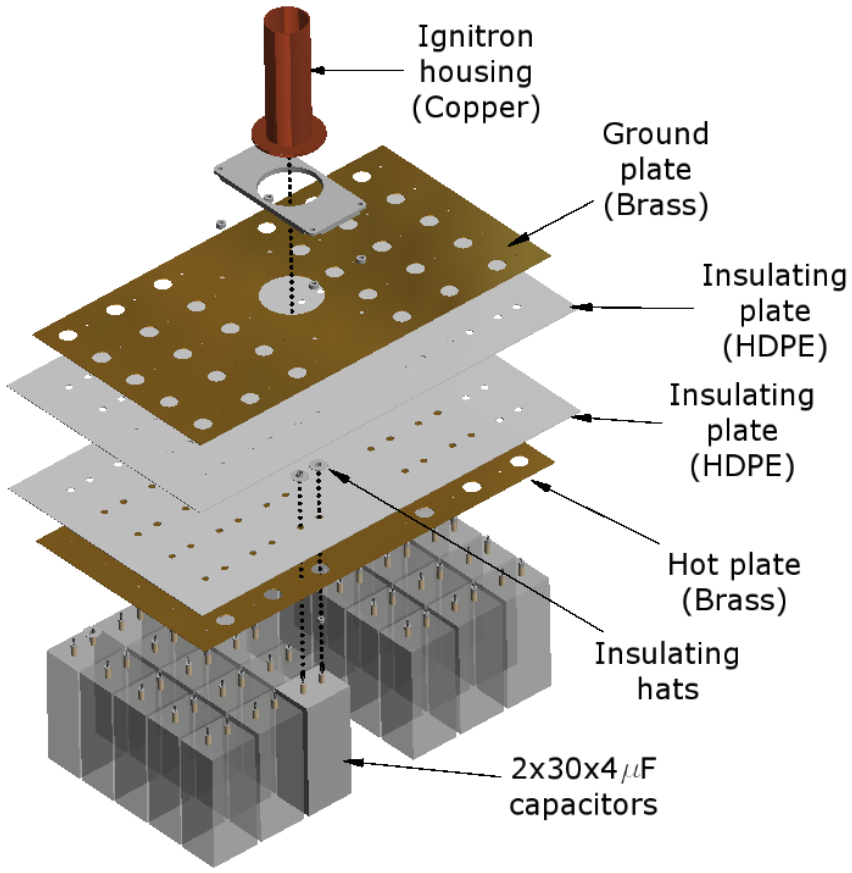
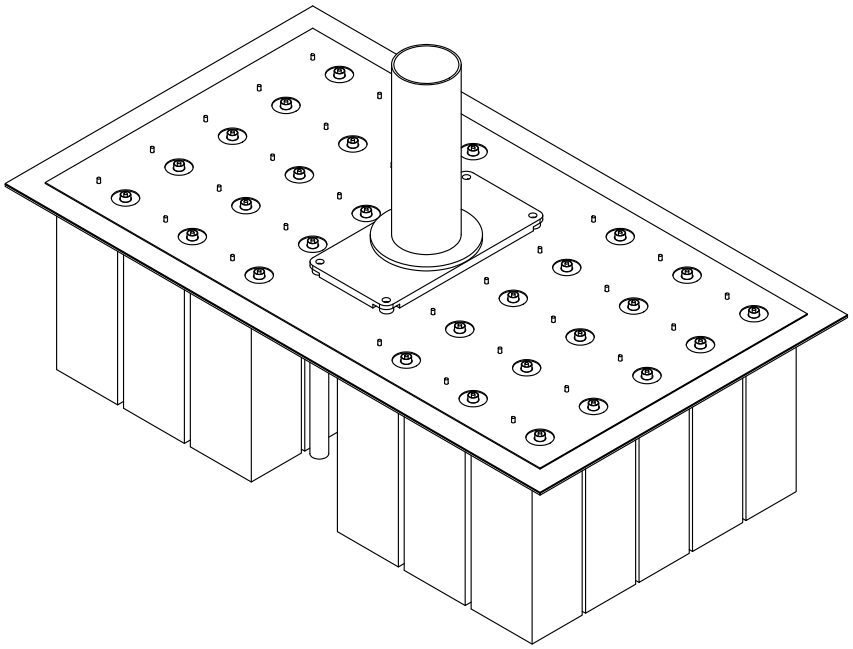


Figure C.3: Exploded view of capacitor bank assembly.

C.2 Drawings

C.2.1 Capacitor Bank Assembly



DRAWN	jens	11/19/2012	
CHECKED			
QA			
MFG			
APPROVED			

TITLE

SIZE C | DWG NO Capacitor Bank | REV

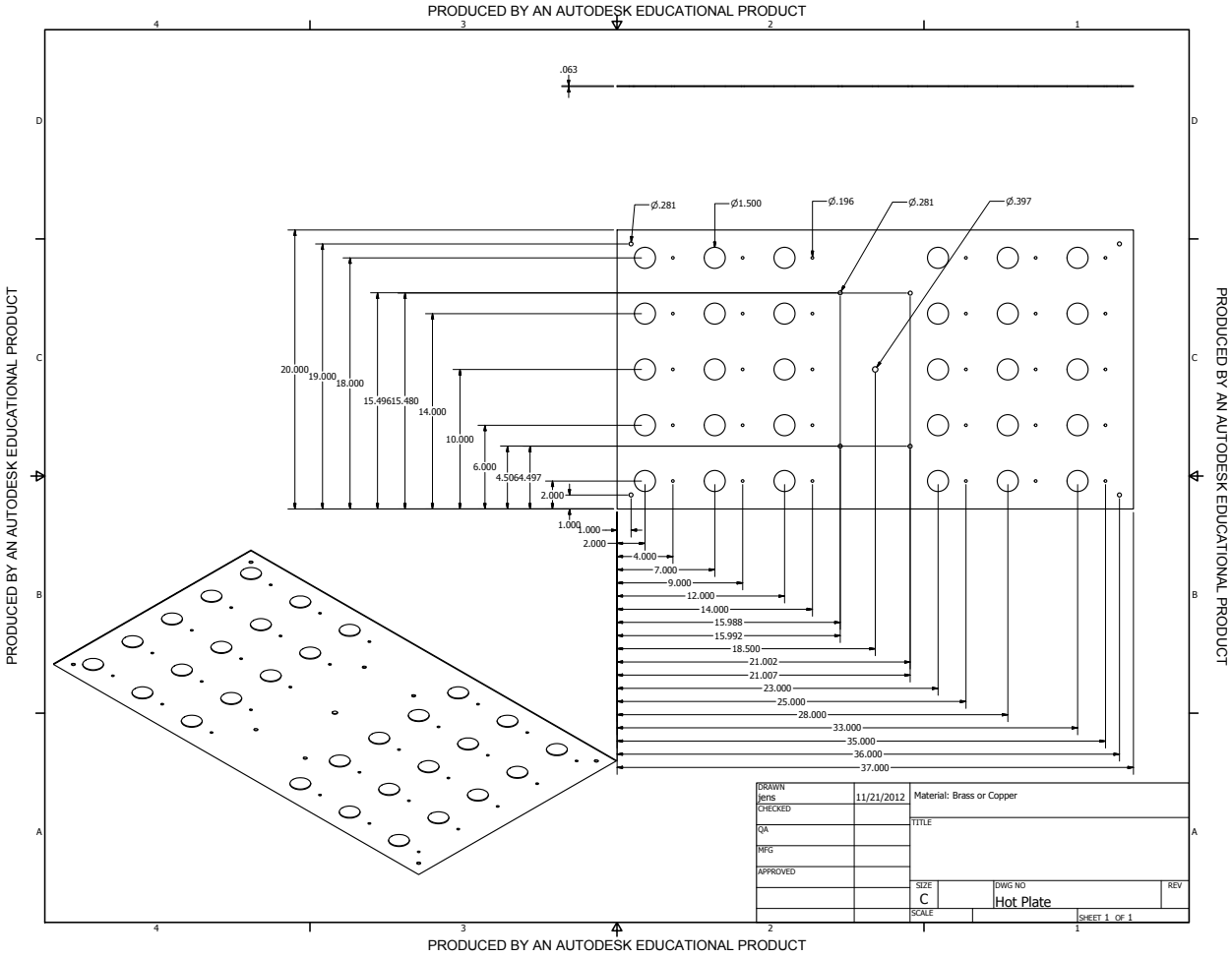
SCALE | SHEET 1 OF 1

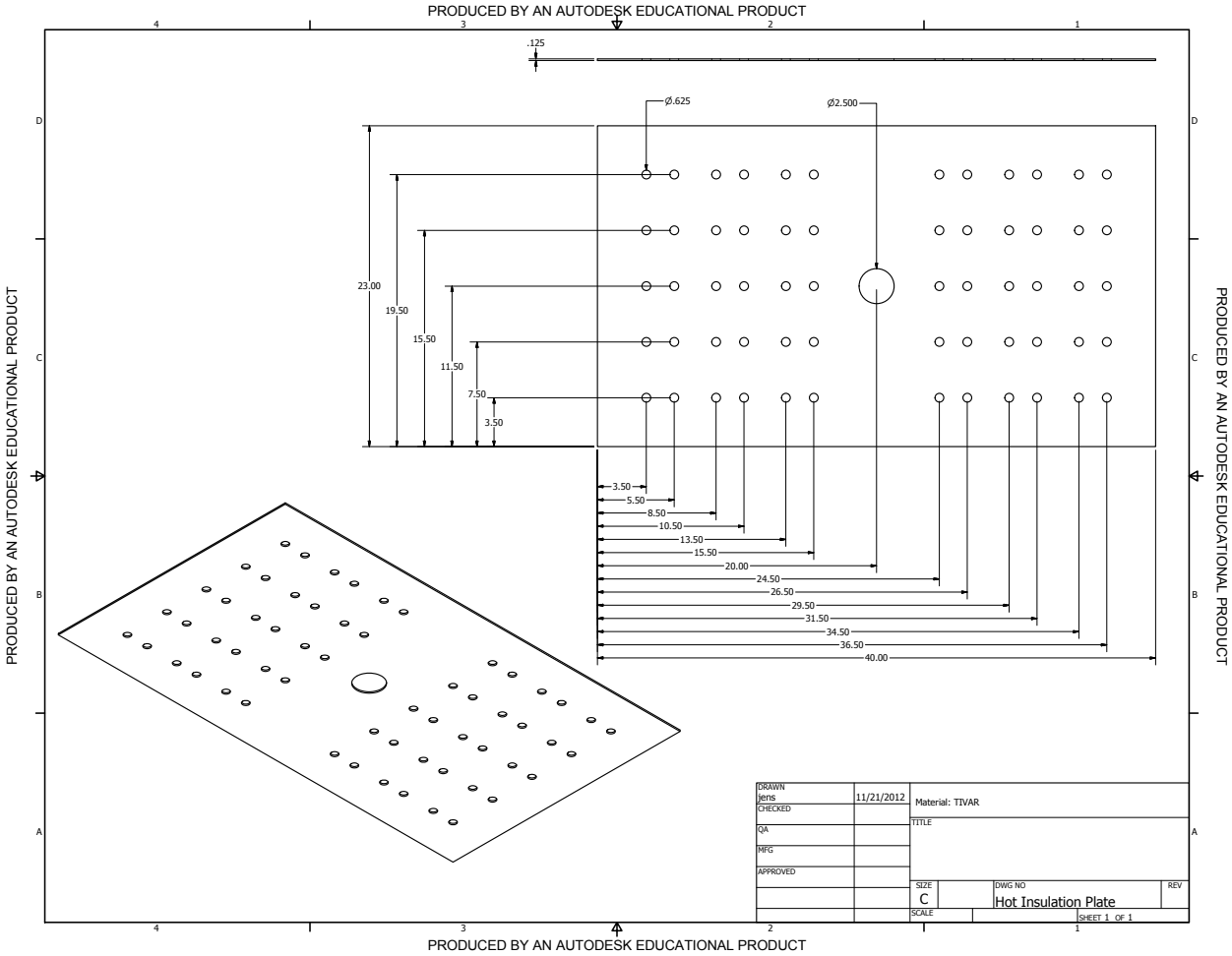
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

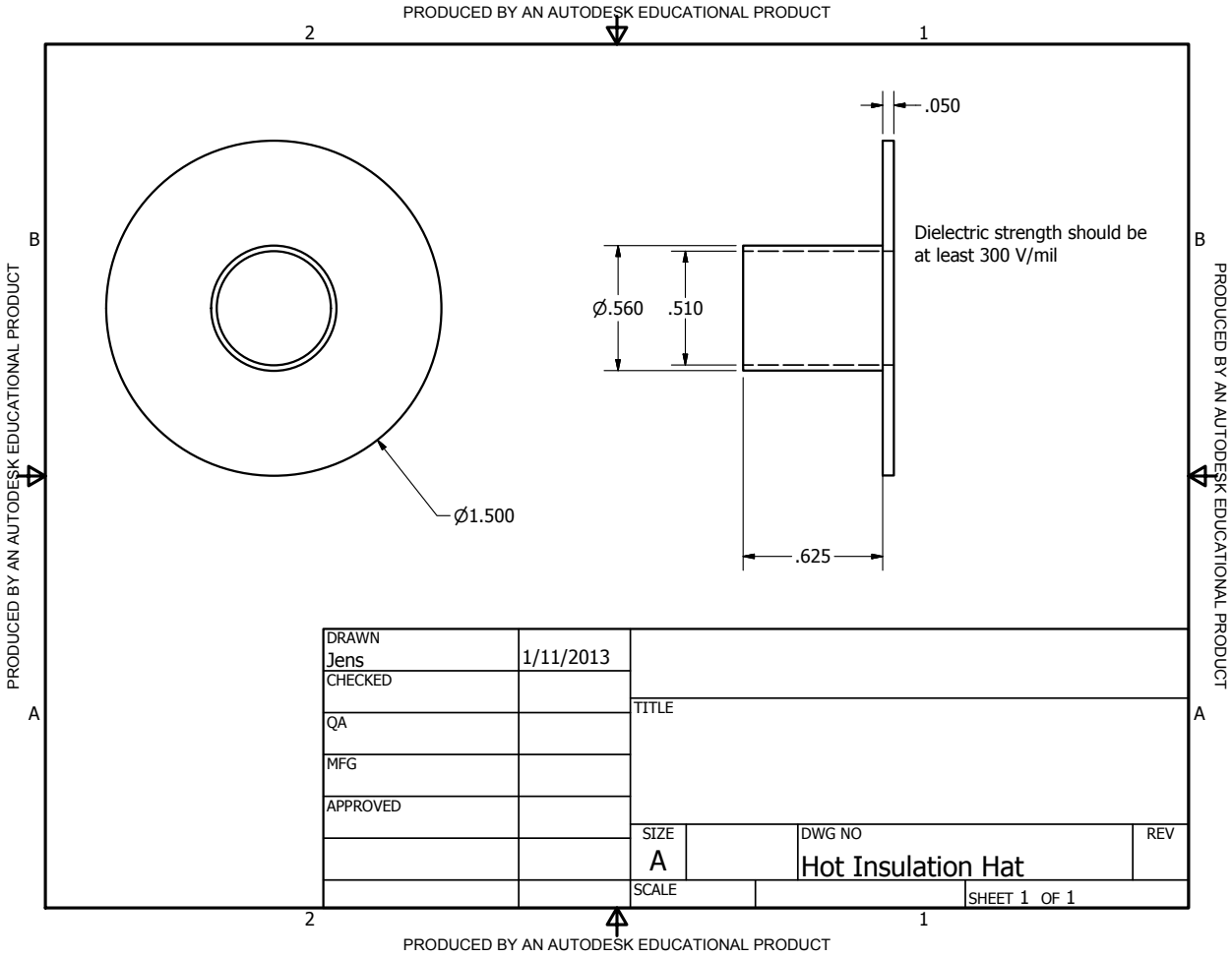
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

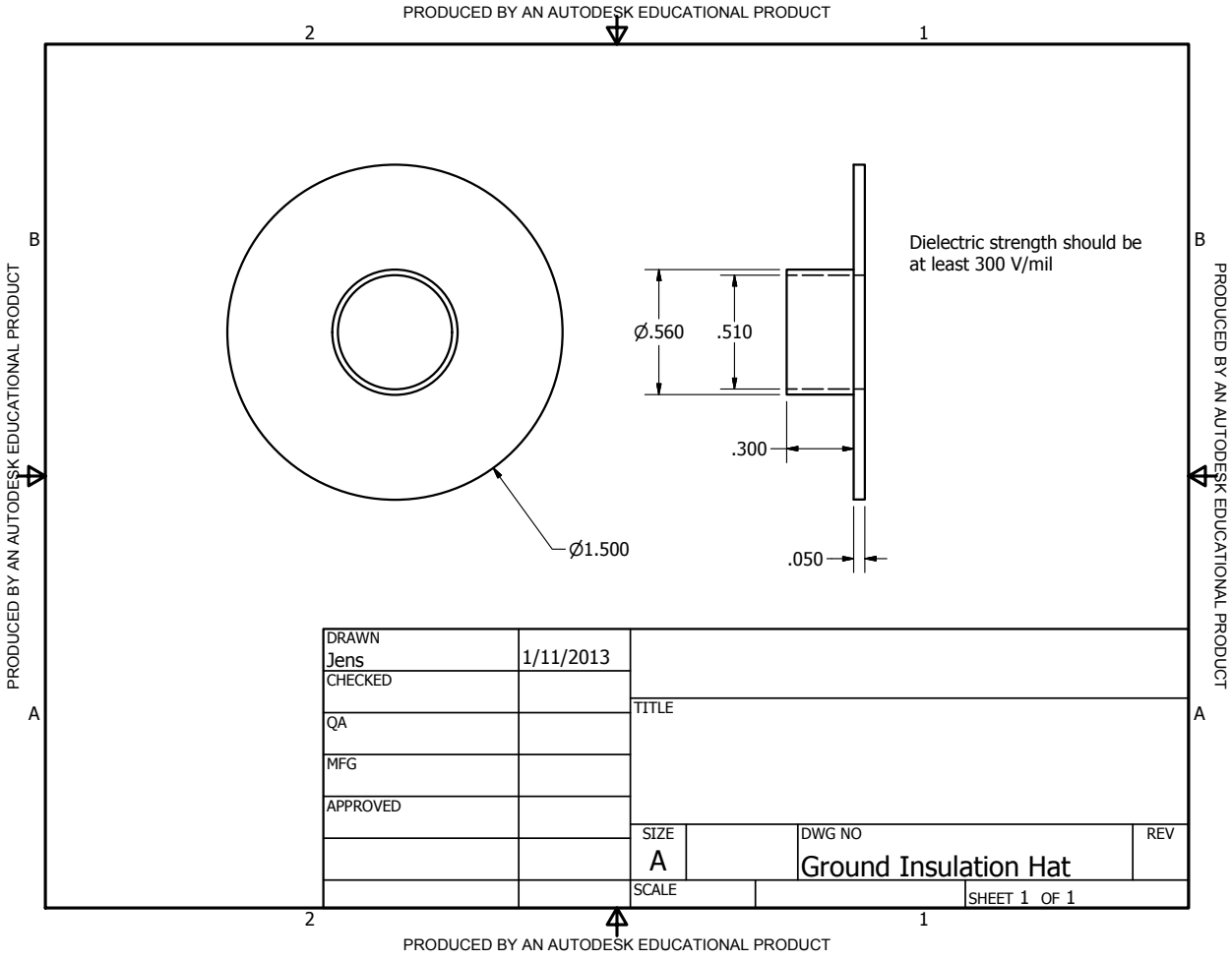
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

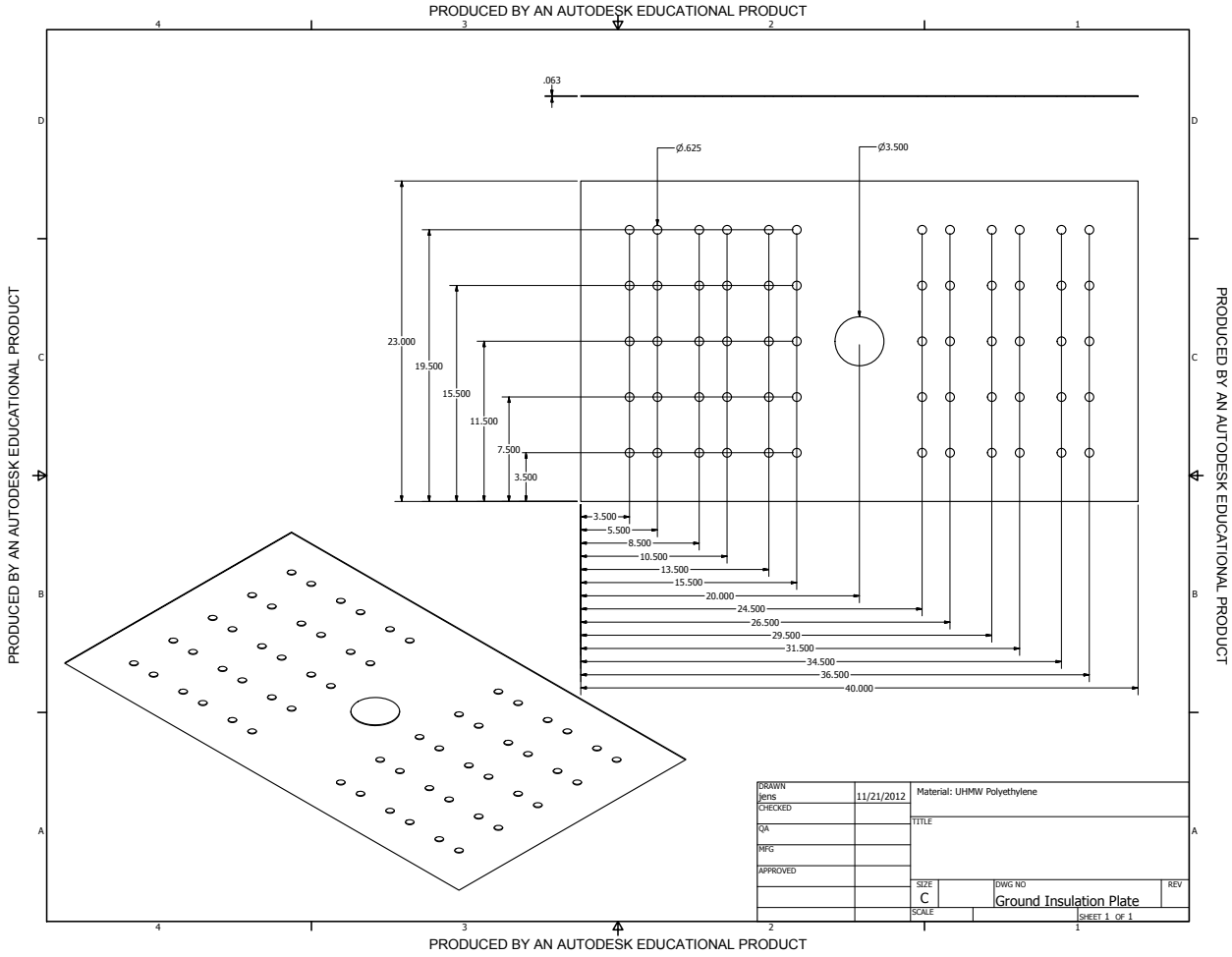
C.2.2 Plates

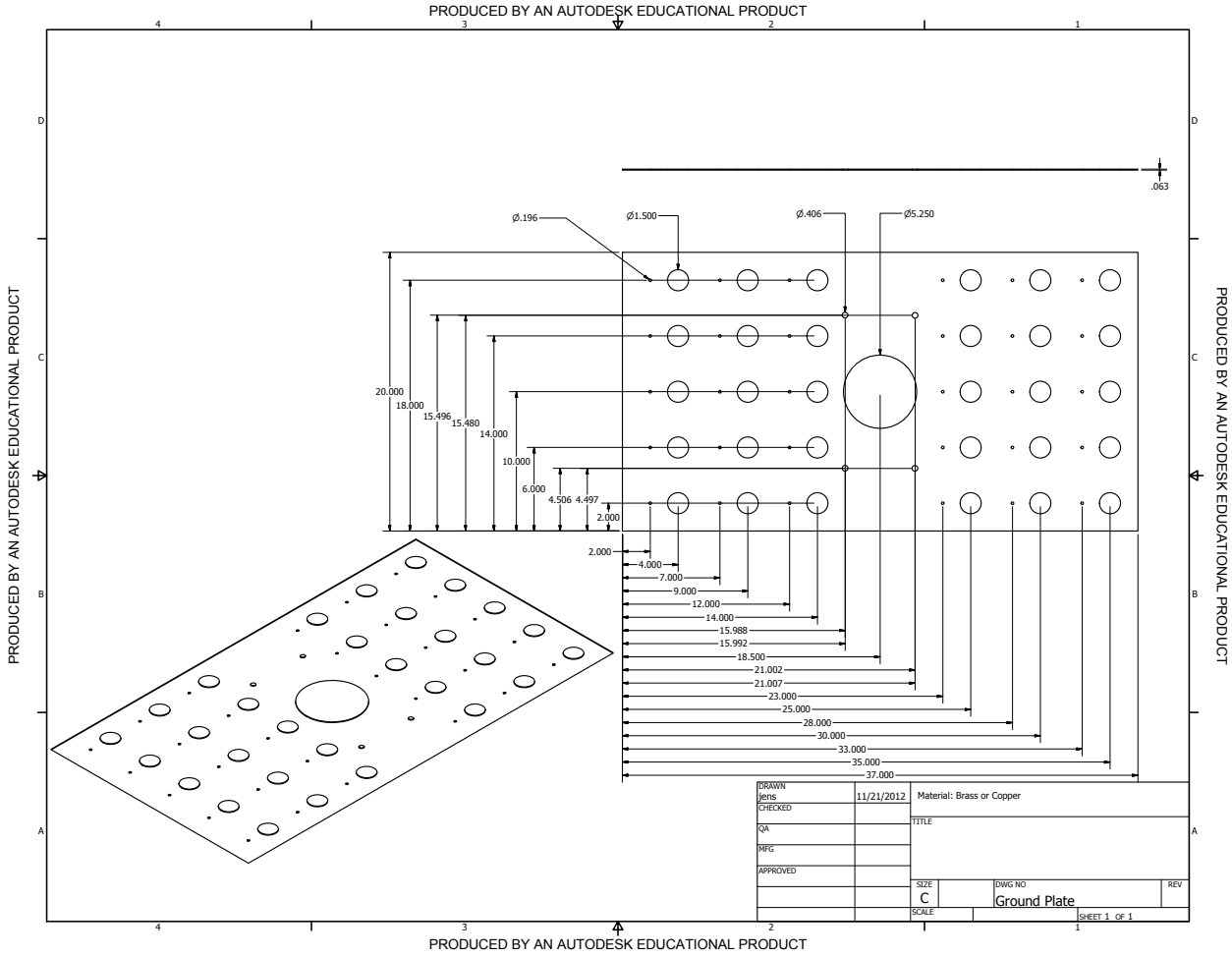

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
.063
Ø.281
Ø1.500
Ø.196
Ø.281
Ø.397
20.000
19.000
18.000
15.496
15.480
14.000
10.000
6.000
4.506
4.497
2.000
1.000
1.000
2.000
4.000
7.000
9.000
12.000
14.000
15.988
15.992
18.500
21.002
21.007
23.000
25.000
28.000
33.000
35.000
36.000
37.000
DRAWN
jens
11/21/2012
Material: Brass or Copper
CHECKED
QA
MFG
APPROVED
TITLE
SIZE
C
DWG NO
Hot Plate
REV
SCALE
SHEET 1 OF 1


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
.125
Ø.625
Ø2.500
23.00
19.50
15.50
11.50
7.50
3.50
3.50
5.50
8.50
10.50
13.50
15.50
20.00
24.50
26.50
29.50
31.50
34.50
36.50
40.00
DRAWN
jens
11/21/2012
Material: TIVAR
CHECKED
QA
MFG
APPROVED
TITLE
SIZE
C
DWG NO
Hot Insulation Plate
REV
SCALE
SHEET 1 OF 1
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

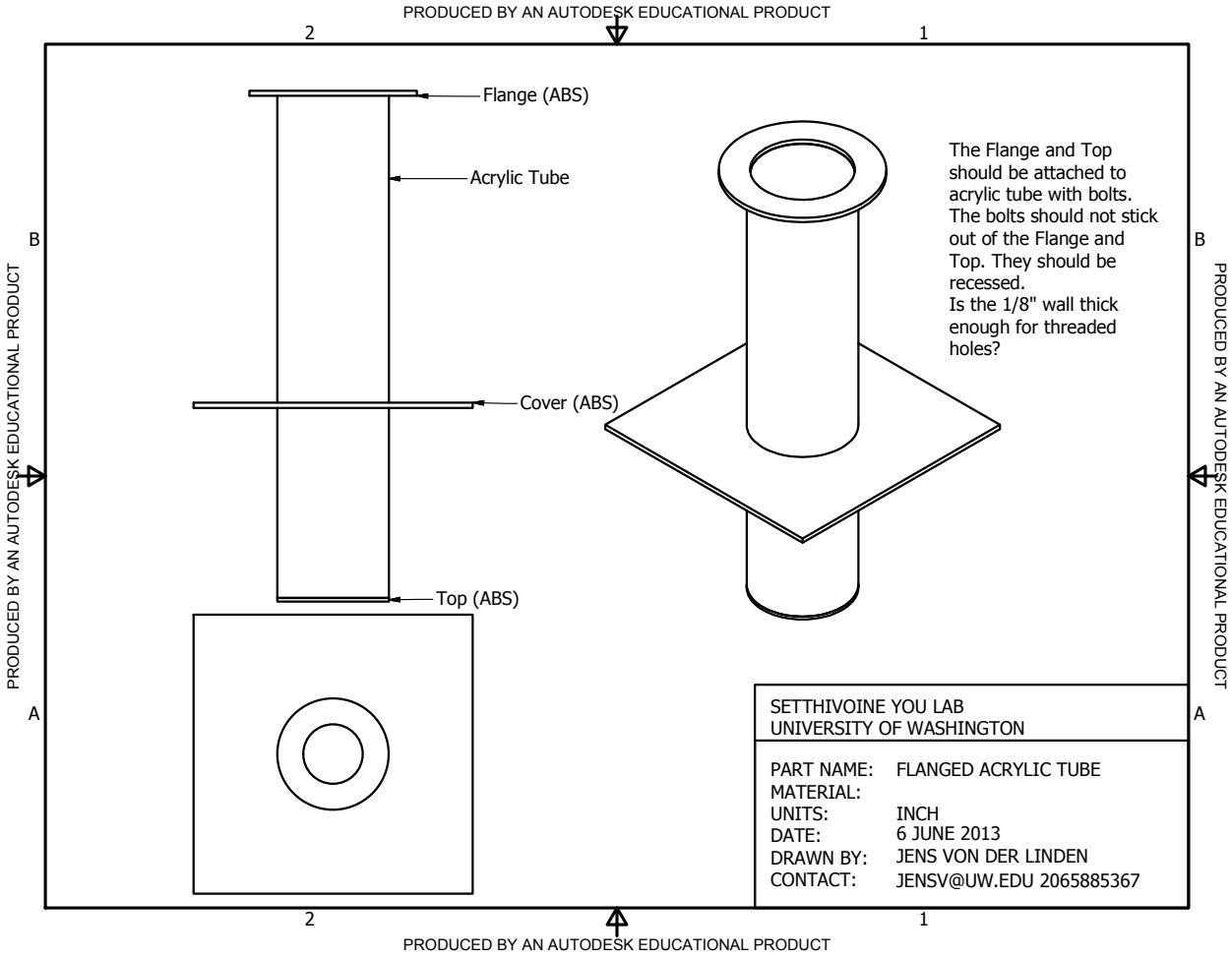

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
.050
Dielectric strength should be at least 300 V/mil
Ø.560
.510
Ø1.500
.625
DRAWN
Jens
1/11/2013
CHECKED
QA
MFG
APPROVED
TITLE
SIZE
A
DWG NO
Hot Insulation Hat
REV
SCALE
SHEET 1 OF 1
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

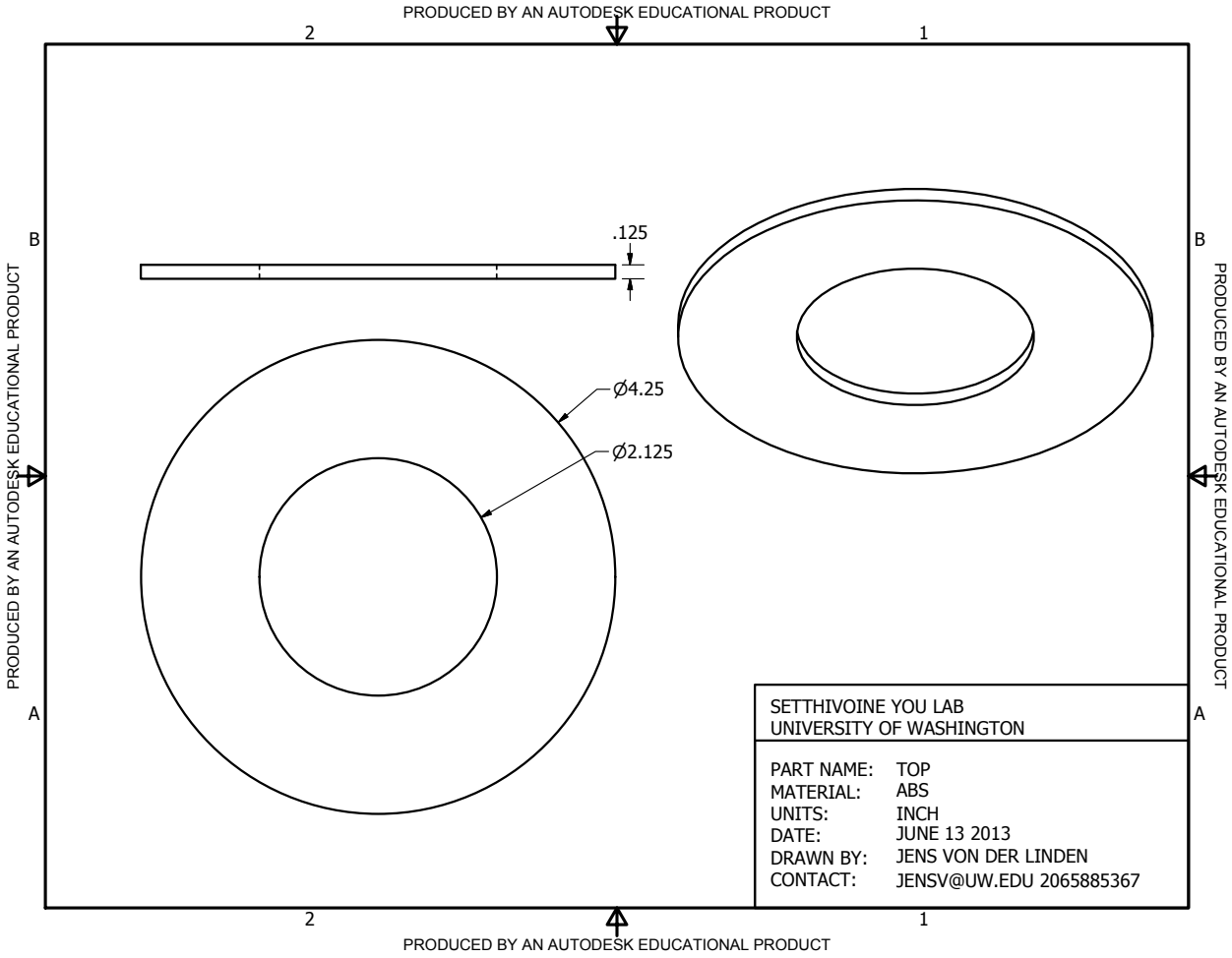

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
Dielectric strength should be at least 300 V/mil
Ø.560
.510
.300
.050
Ø1.500
DRAWN
Jens
1/11/2013
CHECKED
QA
MFG
APPROVED
TITLE
SIZE
A
DWG NO
Ground Insulation Hat
REV
SCALE
SHEET 1 OF 1

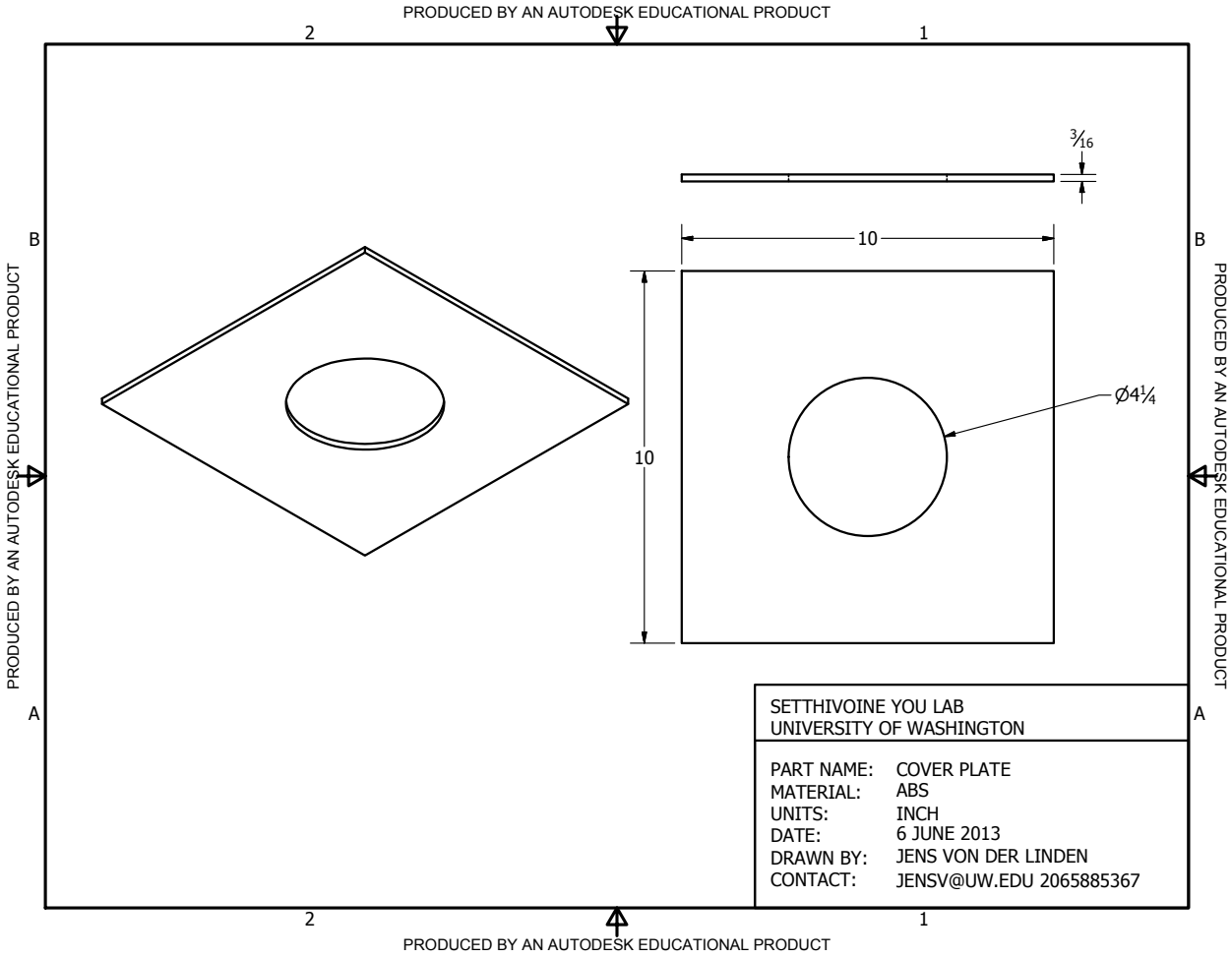

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
.063
Ø.625
Ø3.500
23.000
19.500
15.500
11.500
7.500
3.500
3.500
5.500
8.500
10.500
13.500
15.500
20.000
24.500
26.500
29.500
31.500
34.500
36.500
40.000
DRAWN
jens
11/21/2012
Material: UHMW Polyethylene
CHECKED
QA
MFG
APPROVED
TITLE
SIZE
C
DWG NO
Ground Insulation Plate
REV
SCALE
SHEET 1 OF 1

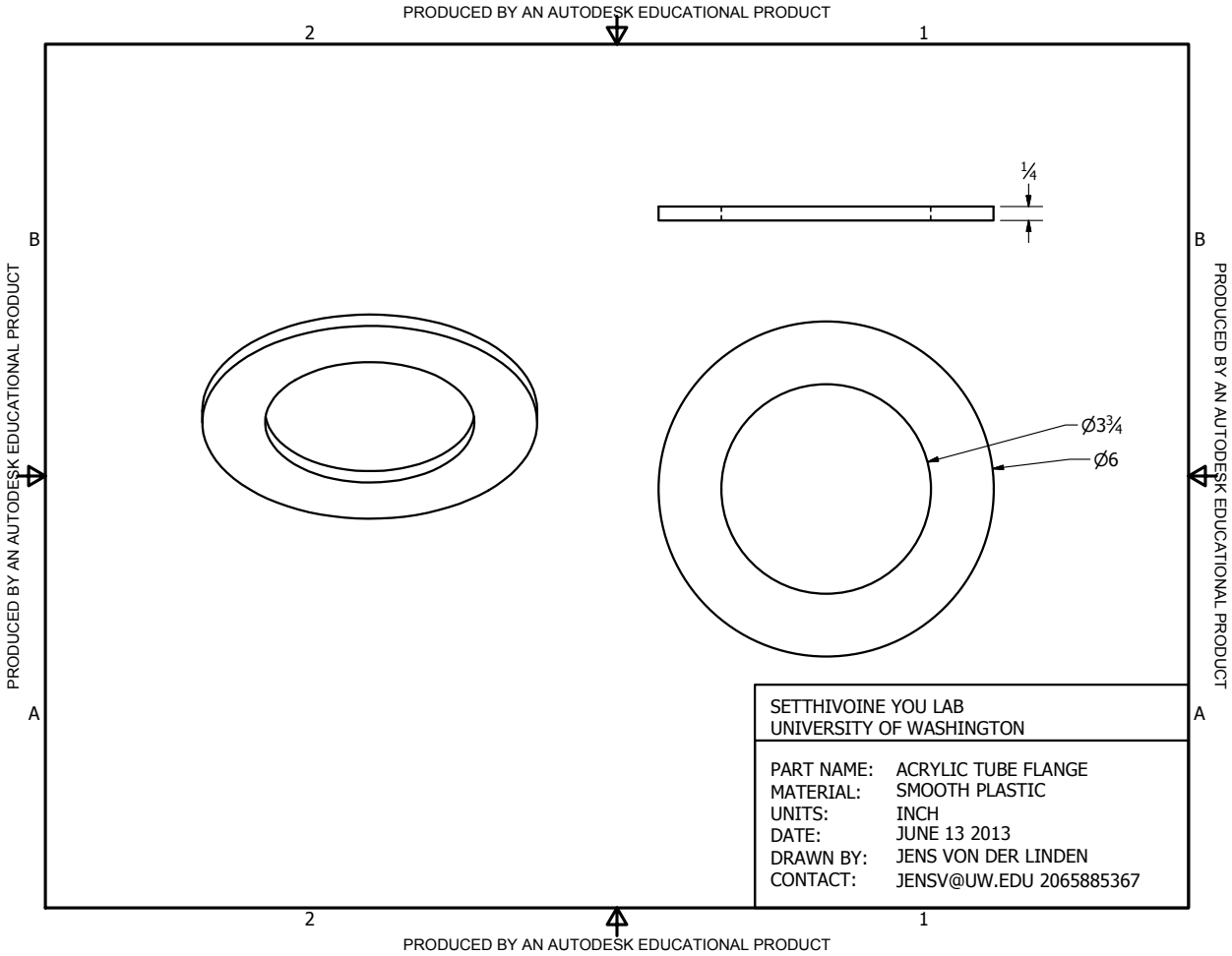

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
.063
Ø.196
Ø1.500
Ø.406
Ø5.250
20.000
18.000
15.496
15.480
14.000
10.000
6.000
4.506
4.497
2.000
2.000
4.000
7.000
9.000
12.000
14.000
15.988
15.992
18.500
21.002
21.007
23.000
25.000
28.000
30.000
33.000
35.000
37.000
DRAWN
jens
11/21/2012
Material: Brass or Copper
CHECKED
QA
MFG
APPROVED
TITLE
SIZE
C
DWG NO
Ground Plate
REV
SCALE
SHEET 1 OF 1

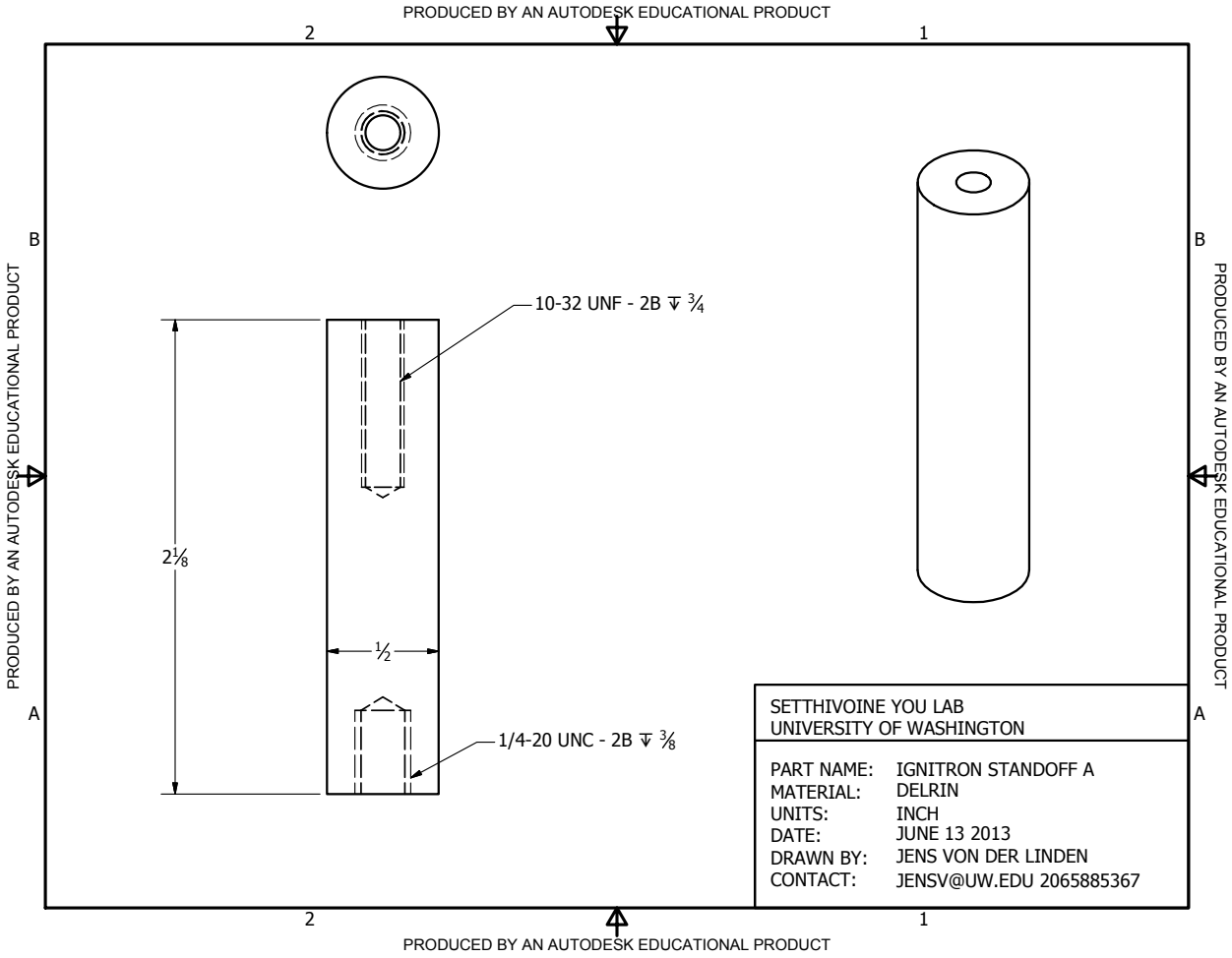
C.2.3 Ignitron cover parts

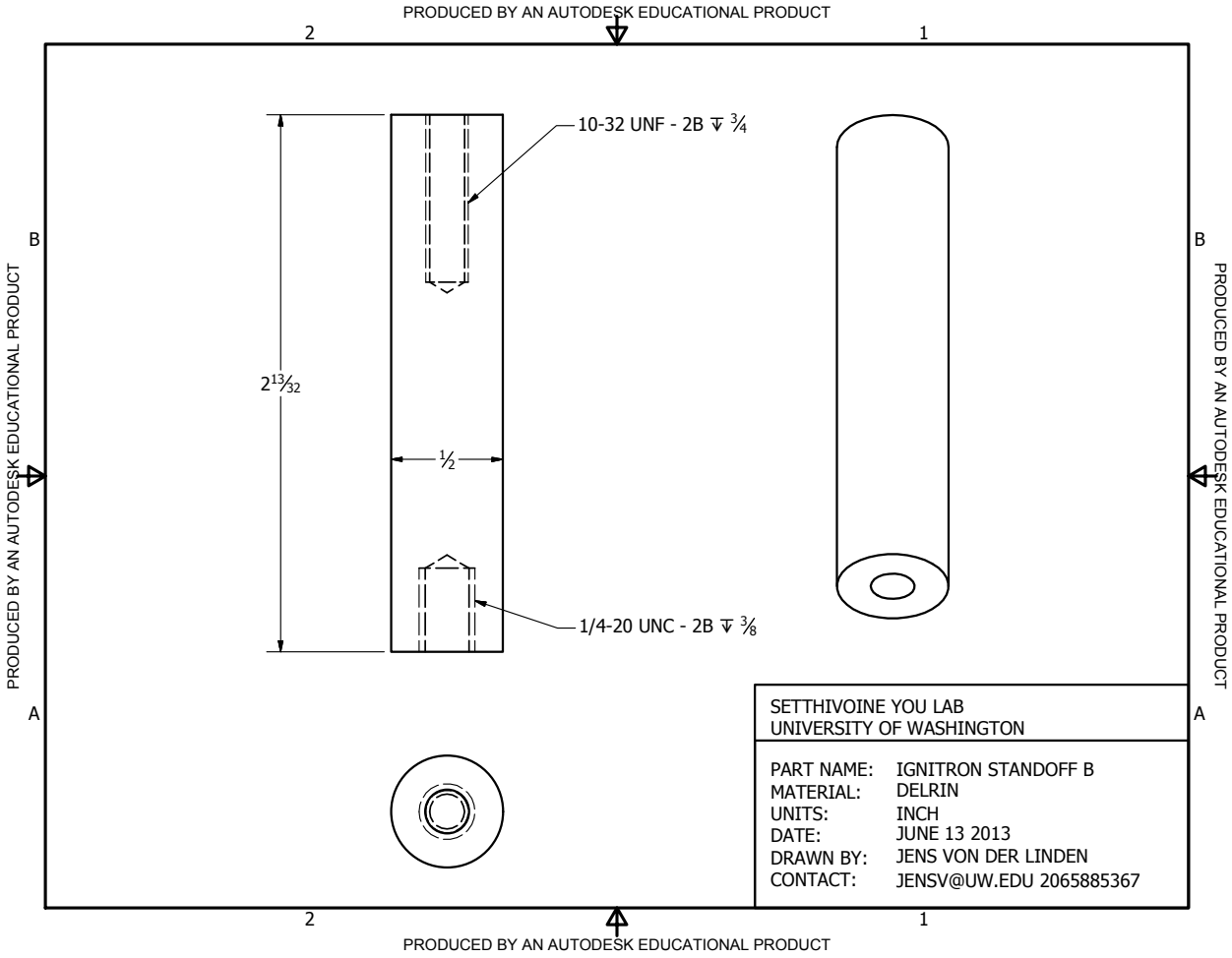

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
Flange (ABS)
Acrylic Tube
Cover (ABS)
Top (ABS)
The Flange and Top should be attached to acrylic tube with bolts. The bolts should not stick out of the Flange and Top. They should be recessed.
Is the 1/8" wall thick enough for threaded holes?
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: FLANGED ACRYLIC TUBE
MATERIAL:
UNITS: INCH
DATE: 6 JUNE 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
.125
Ø4.25
Ø2.125
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: TOP
MATERIAL: ABS
UNITS: INCH
DATE: JUNE 13 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367

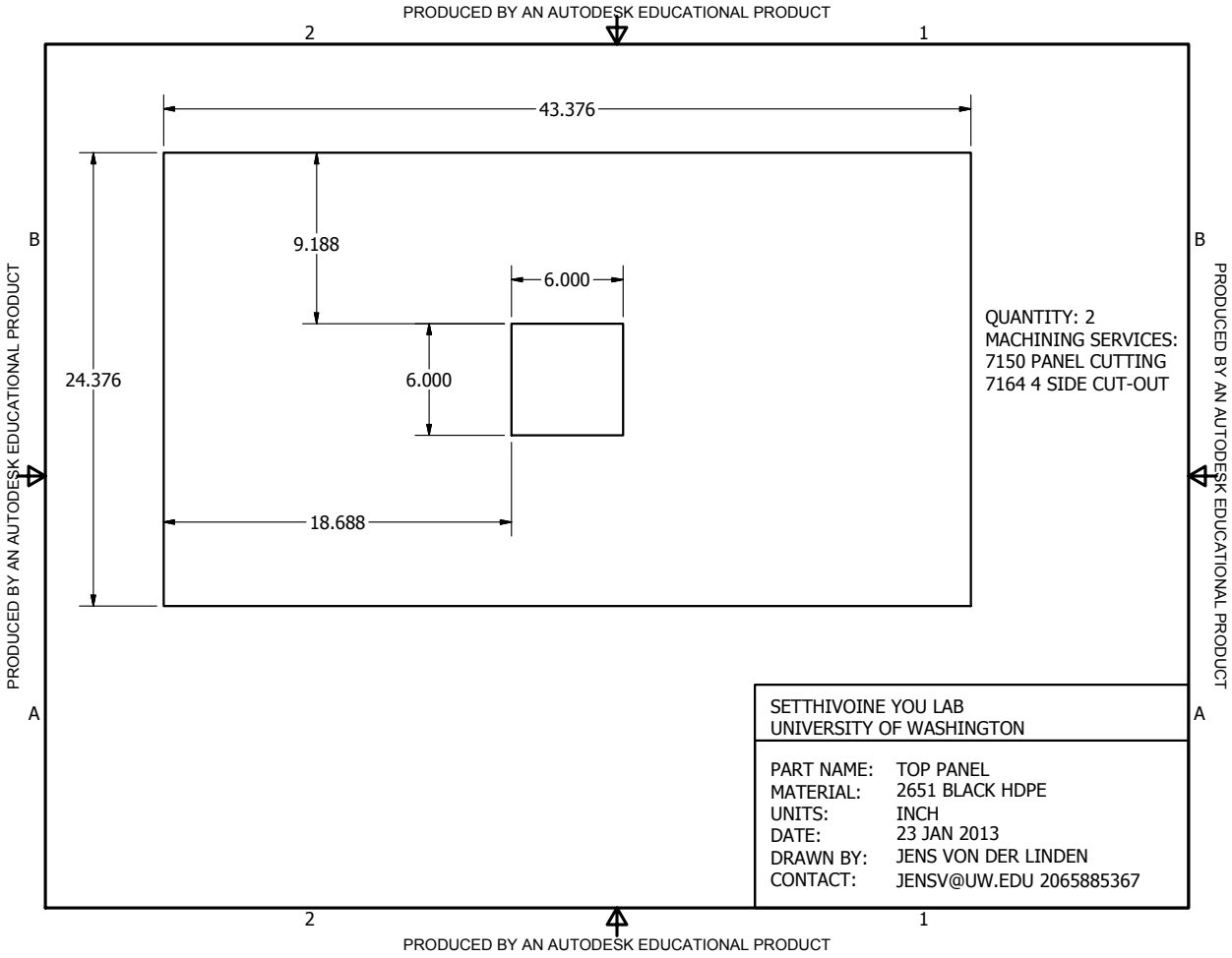

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
3/16
10
10
Ø4¼
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: COVER PLATE
MATERIAL: ABS
UNITS: INCH
DATE: 6 JUNE 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

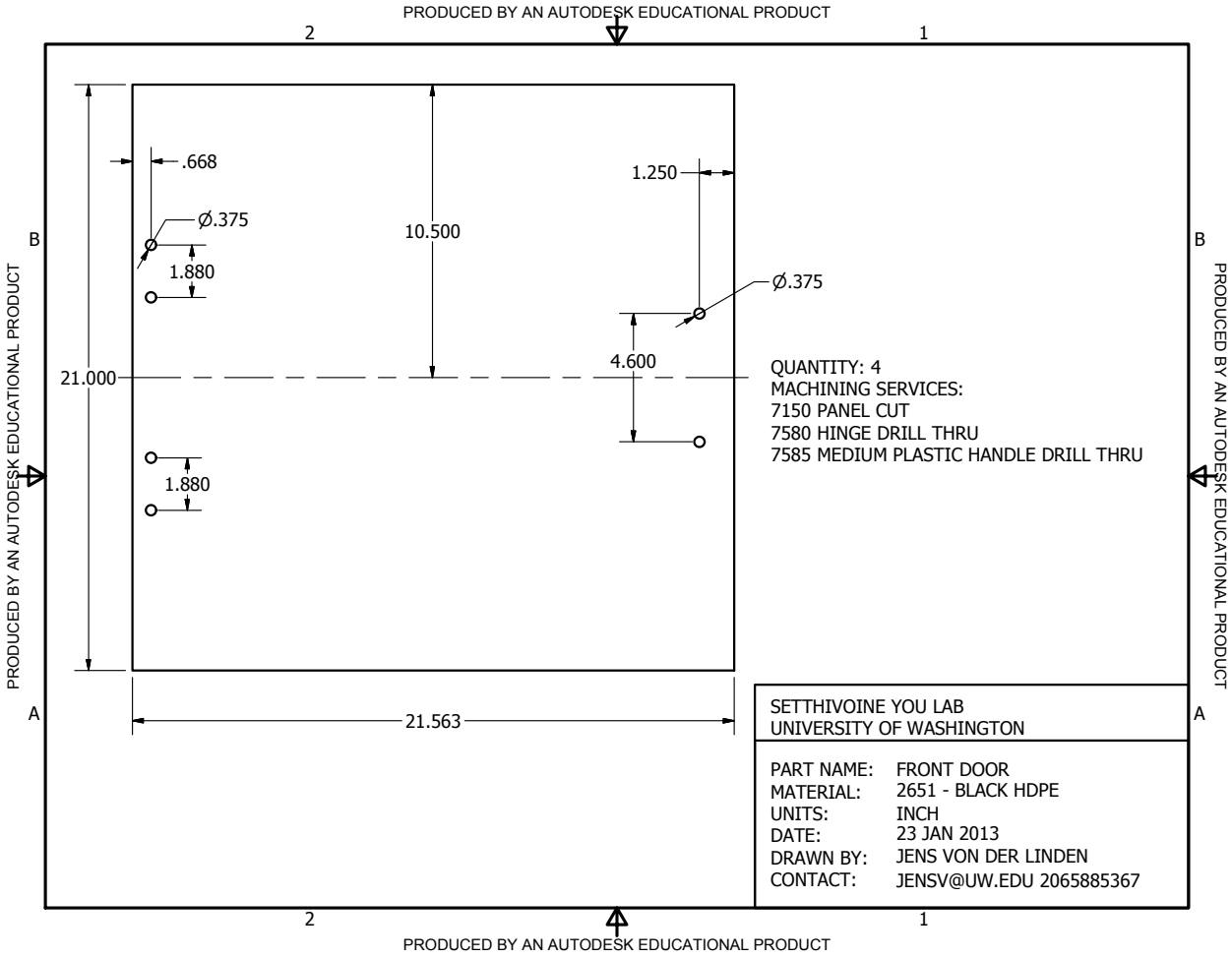

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
¼
Ø3¾
Ø6
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: ACRYLIC TUBE FLANGE
MATERIAL: SMOOTH PLASTIC
UNITS: INCH
DATE: JUNE 13 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367

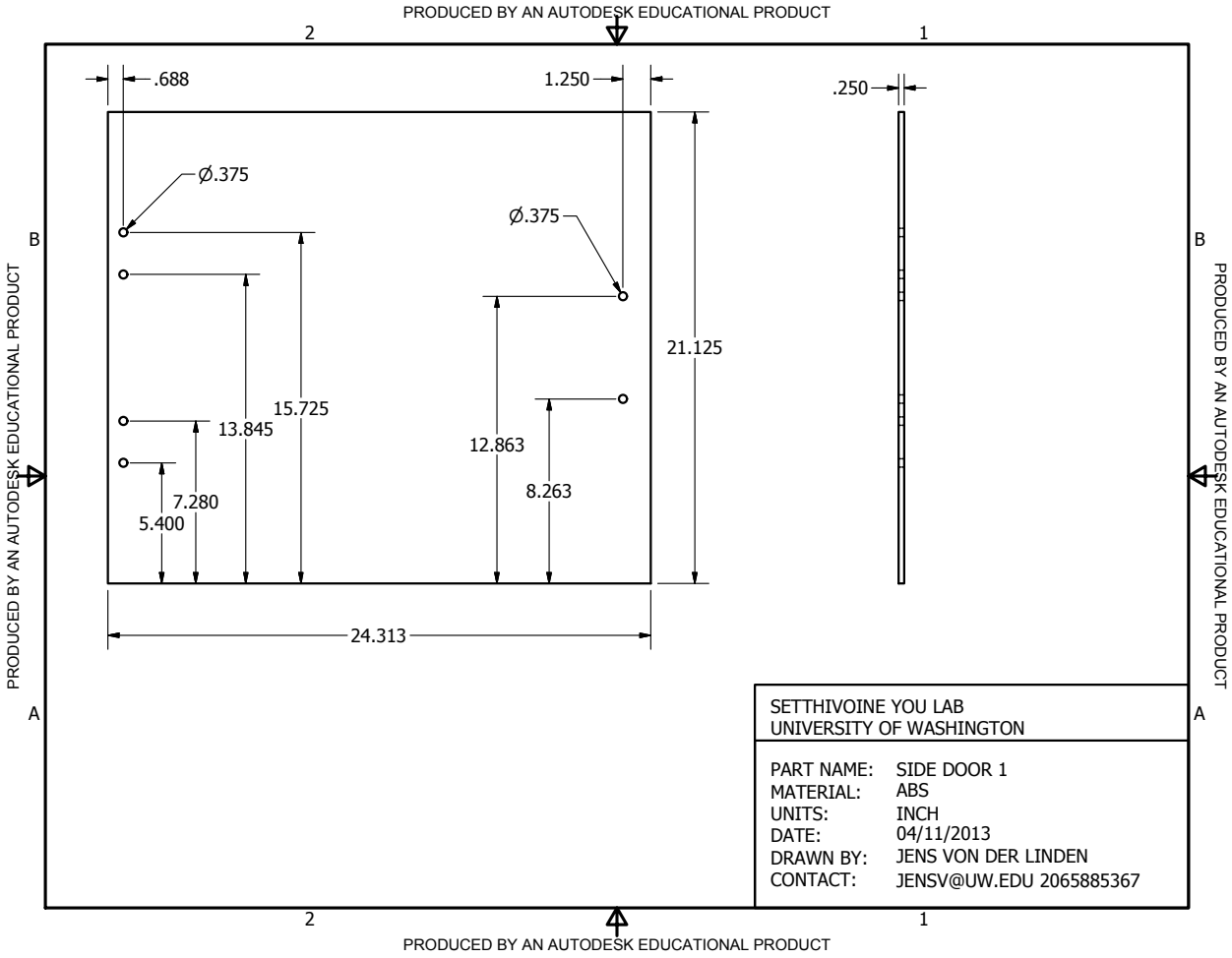

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
10-32 UNF - 2B ↧ ¾
2⅛
½
1/4-20 UNC - 2B ↧ ⅜
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: IGNITRON STANDOFF A
MATERIAL: DELRIN
UNITS: INCH
DATE: JUNE 13 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367

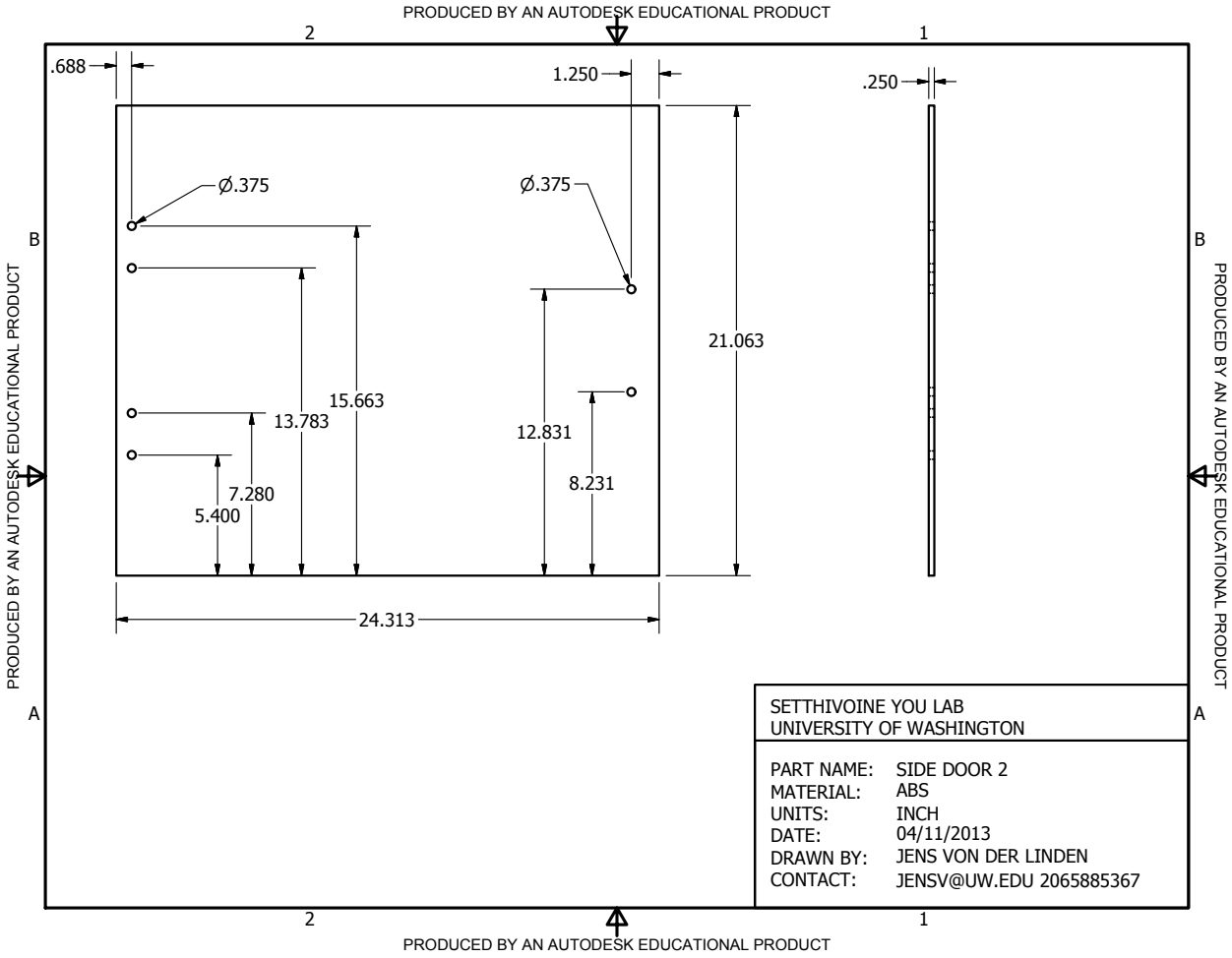

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
10-32 UNF - 2B ↧ ¾
2 13/32
½
1/4-20 UNC - 2B ↧ ⅜
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: IGNITRON STANDOFF B
MATERIAL: DELRIN
UNITS: INCH
DATE: JUNE 13 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367

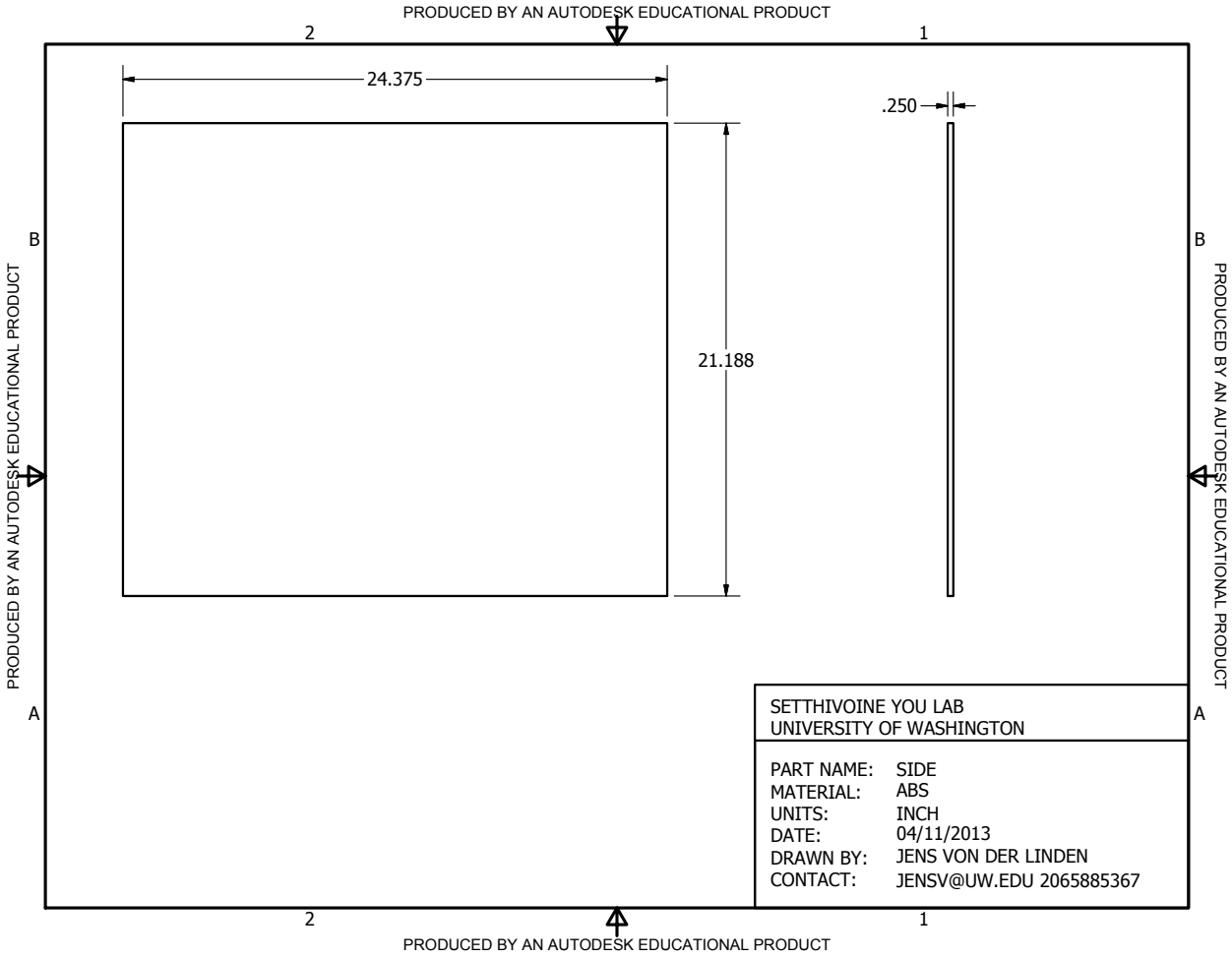
C.2.4 Panels

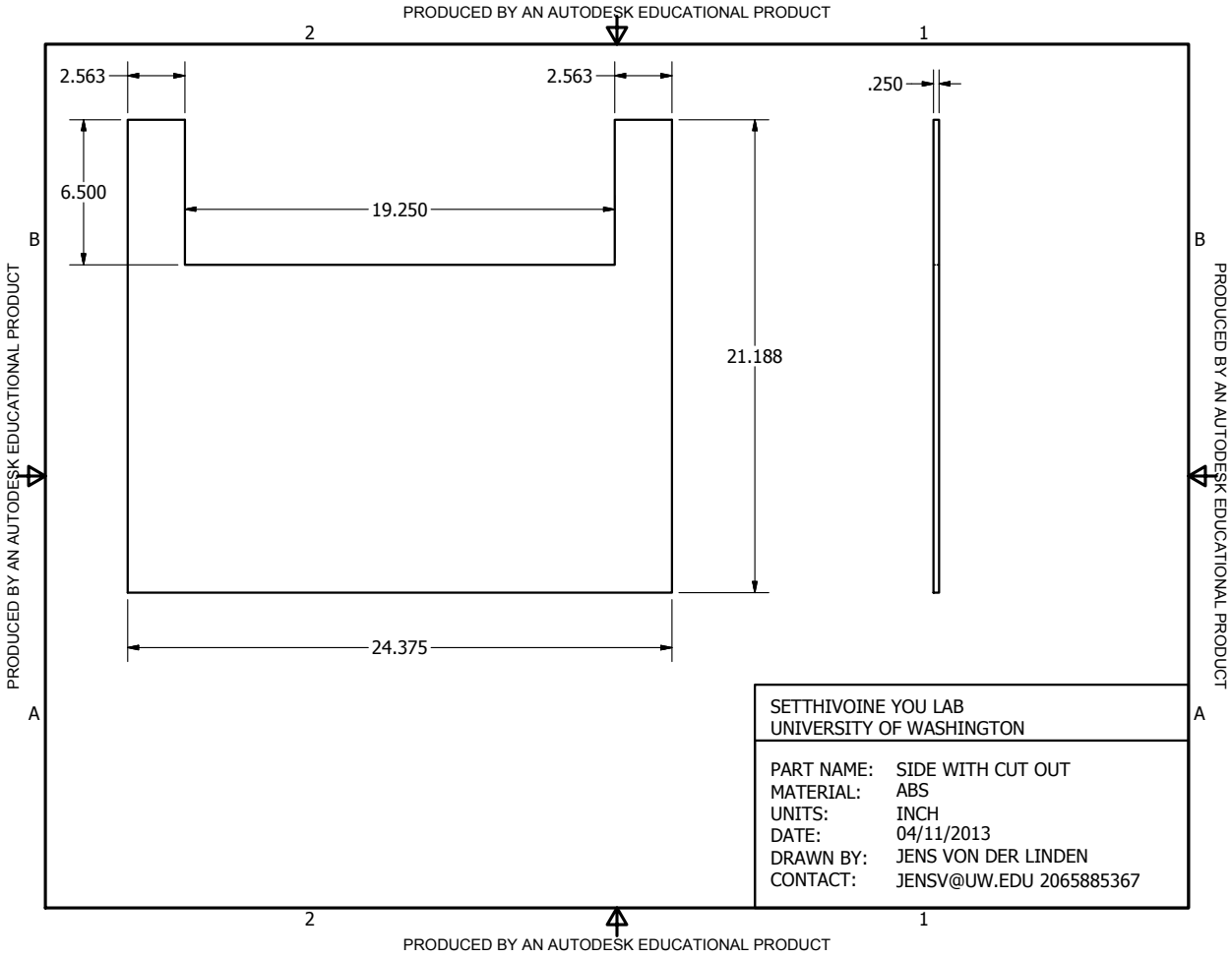

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
43.376
9.188
6.000
24.376
6.000
18.688
QUANTITY: 2
MACHINING SERVICES:
7150 PANEL CUTTING
7164 4 SIDE CUT-OUT
B
A
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: TOP PANEL
MATERIAL: 2651 BLACK HDPE
UNITS: INCH
DATE: 23 JAN 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

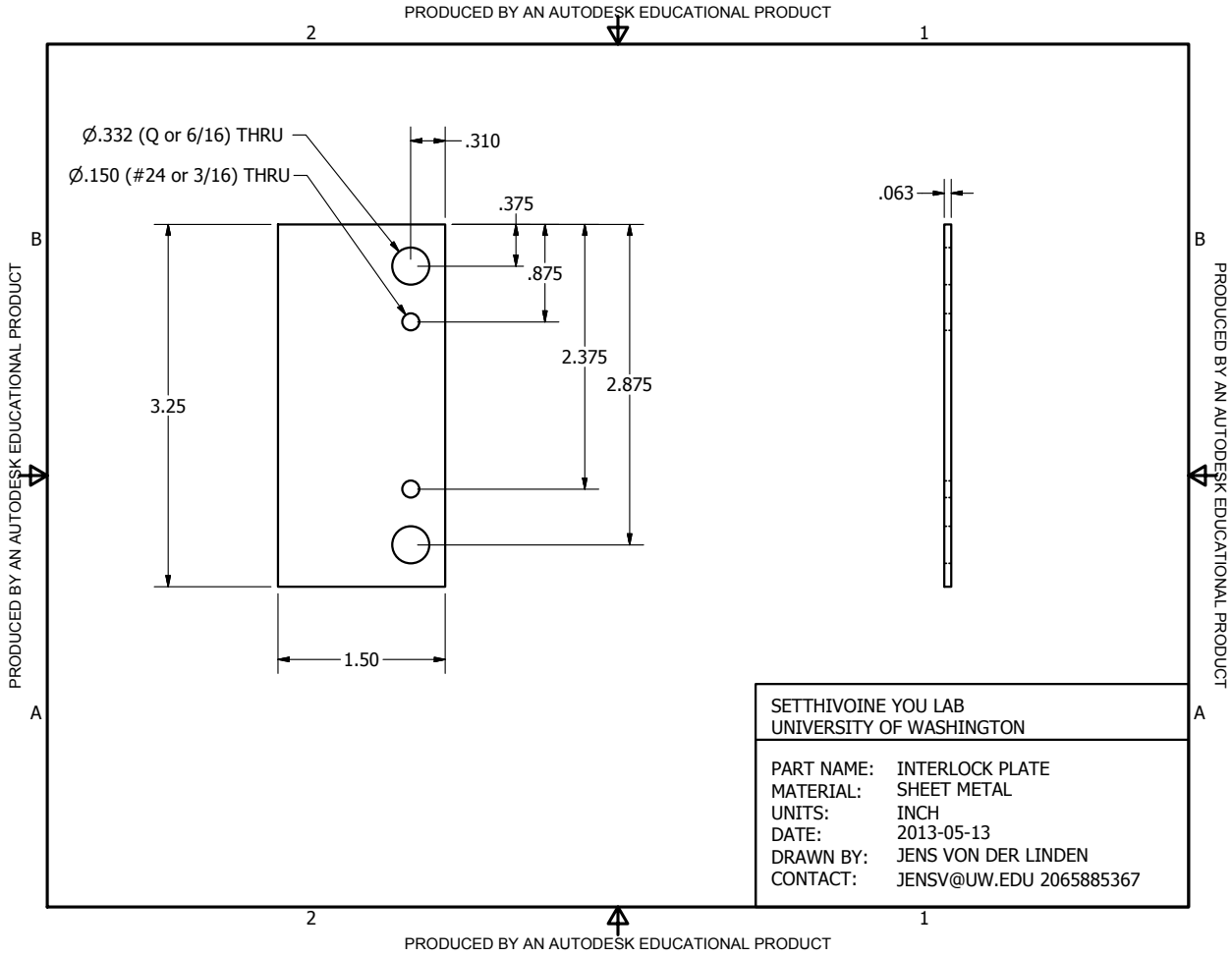

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
.668
Ø.375
1.880
10.500
1.250
Ø.375
4.600
21.000
1.880
21.563
QUANTITY: 4
MACHINING SERVICES:
7150 PANEL CUT
7580 HINGE DRILL THRU
7585 MEDIUM PLASTIC HANDLE DRILL THRU
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: FRONT DOOR
MATERIAL: 2651 - BLACK HDPE
UNITS: INCH
DATE: 23 JAN 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

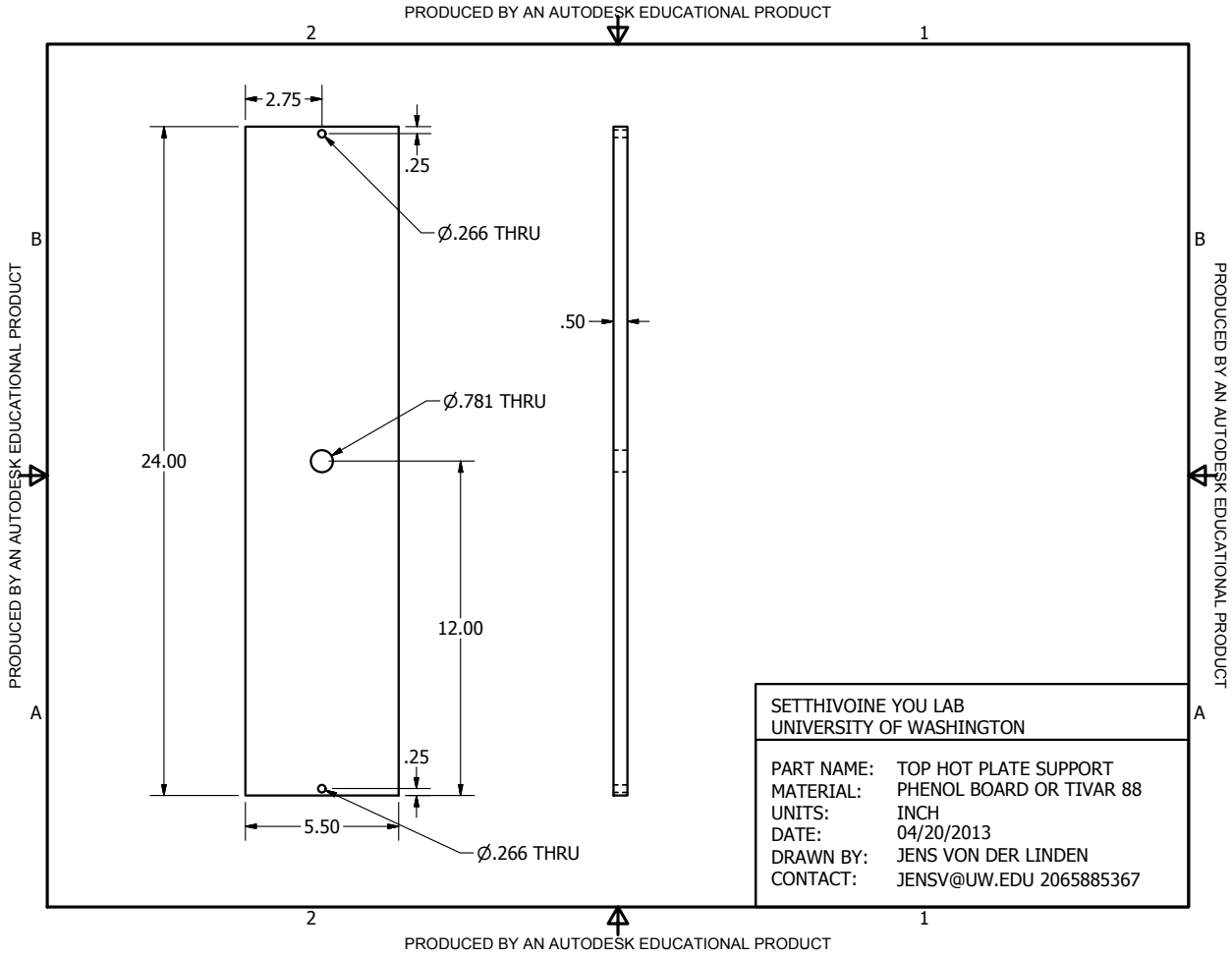

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
.688
1.250
.250
Ø.375
Ø.375
21.125
15.725
13.845
12.863
7.280
5.400
8.263
24.313
B
A
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: SIDE DOOR 1
MATERIAL: ABS
UNITS: INCH
DATE: 04/11/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

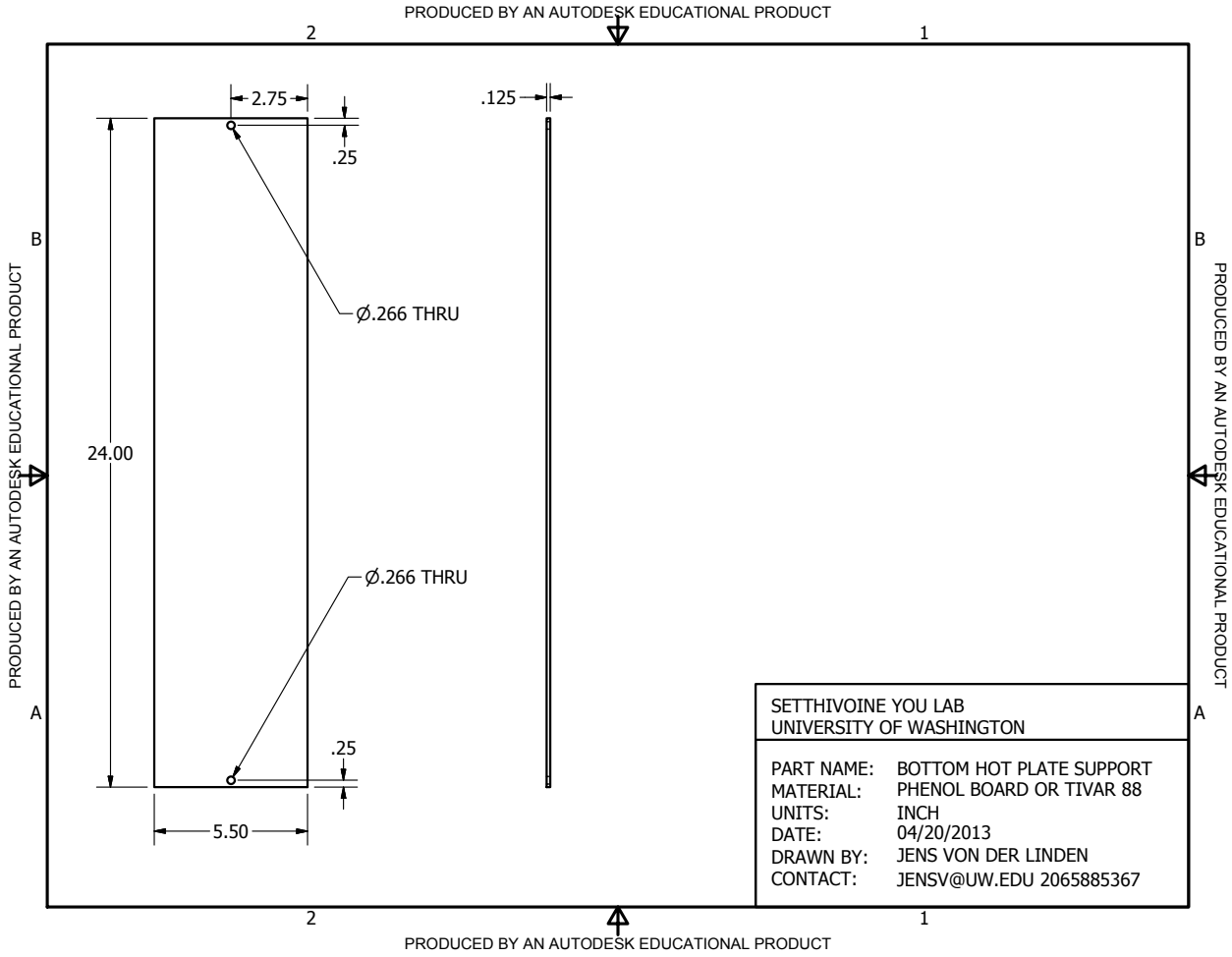

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
.688
1.250
.250
Ø.375
Ø.375
B
B
21.063
15.663
13.783
12.831
7.280
5.400
8.231
24.313
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
A
A
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: SIDE DOOR 2
MATERIAL: ABS
UNITS: INCH
DATE: 04/11/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
2
1
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

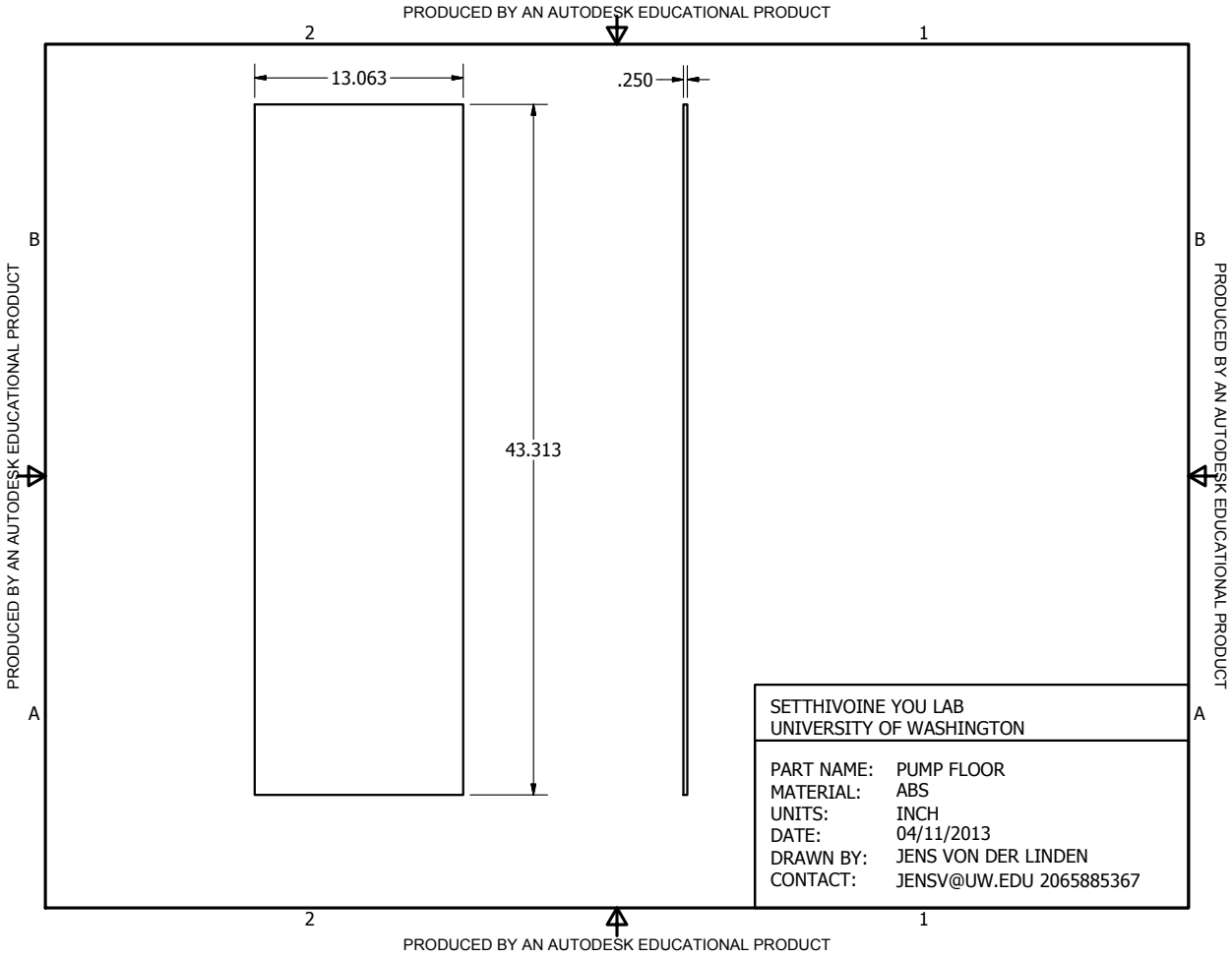

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
24.375
.250
21.188
B
A
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: SIDE
MATERIAL: ABS
UNITS: INCH
DATE: 04/11/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

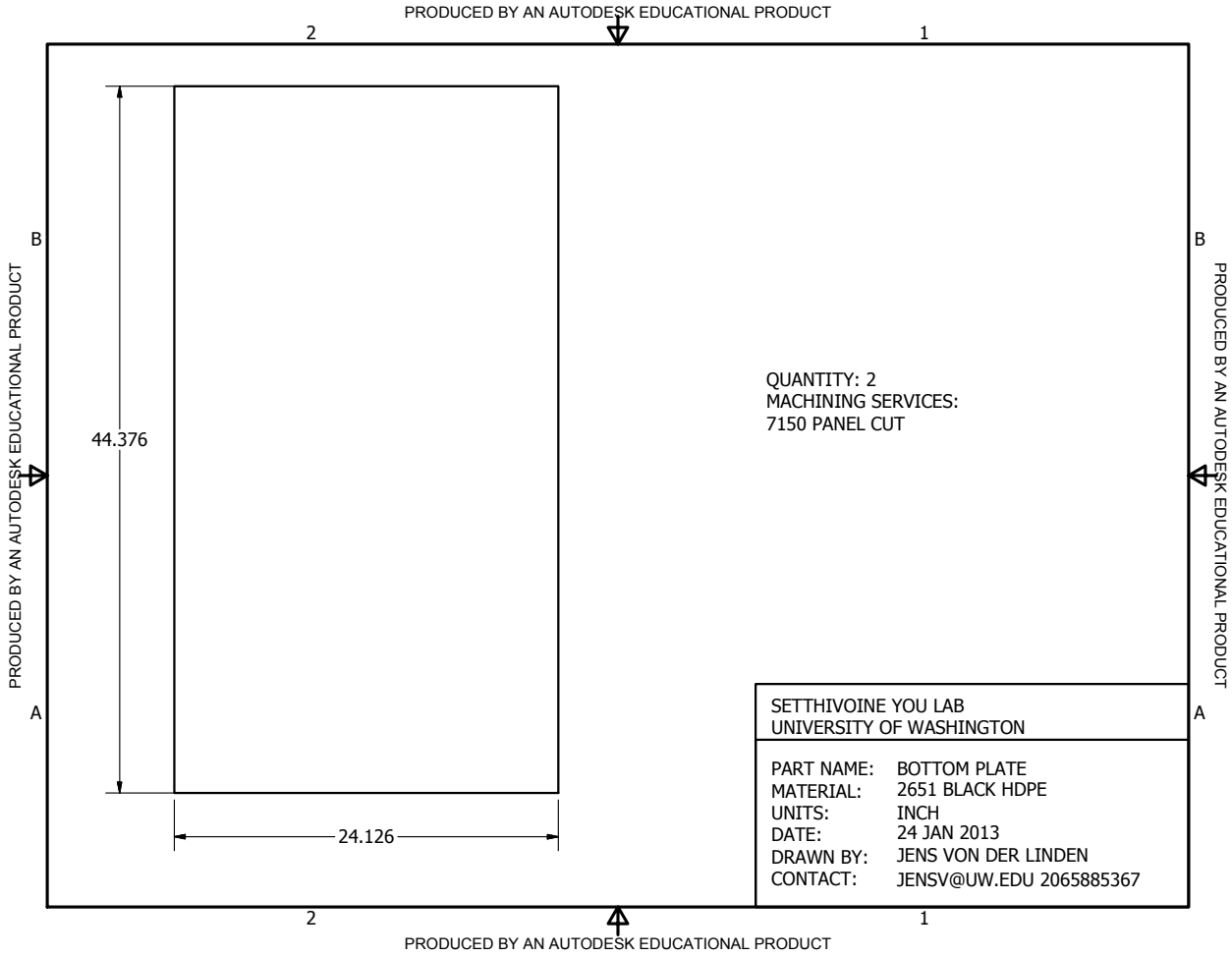

PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
2.563
2.563
.250
6.500
19.250
21.188
24.375
B
A
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: SIDE WITH CUT OUT
MATERIAL: ABS
UNITS: INCH
DATE: 04/11/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
Ø.332 (Q or 6/16) THRU
Ø.150 (#24 or 3/16) THRU
.310
.375
.875
2.375
2.875
3.25
1.50
.063
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: INTERLOCK PLATE
MATERIAL: SHEET METAL
UNITS: INCH
DATE: 2013-05-13
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2.75
.25
Ø.266 THRU
.50
Ø.781 THRU
24.00
12.00
.25
5.50
Ø.266 THRU
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: TOP HOT PLATE SUPPORT
MATERIAL: PHENOL BOARD OR TIVAR 88
UNITS: INCH
DATE: 04/20/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
2.75
.25
.125
Ø.266 THRU
24.00
Ø.266 THRU
.25
5.50
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: BOTTOM HOT PLATE SUPPORT
MATERIAL: PHENOL BOARD OR TIVAR 88
UNITS: INCH
DATE: 04/20/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
13.063
.250
43.313
B
A
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: PUMP FLOOR
MATERIAL: ABS
UNITS: INCH
DATE: 04/11/2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT


PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT
2
1
B
A
44.376
24.126
QUANTITY: 2
MACHINING SERVICES:
7150 PANEL CUT
SETTHIVOINE YOU LAB
UNIVERSITY OF WASHINGTON
PART NAME: BOTTOM PLATE
MATERIAL: 2651 BLACK HDPE
UNITS: INCH
DATE: 24 JAN 2013
DRAWN BY: JENS VON DER LINDEN
CONTACT: JENSV@UW.EDU 2065885367
PRODUCED BY AN AUTODESK EDUCATIONAL PRODUCT

Appendix D

CANONICAL FLUX TUBE AND HELICITY RECONSTRUCTION CODE

This appendix contains scripts used to reconstruct and plot canonical flux tubes in VisIt and calculate their helicity from experimental measurements of magnetic field, density, electron temperature and Mach numbers of a plasma. Only the main scripts are printed the underlying modules can be found at Zenodo
https://doi.org/10.5281/zenodo.581197 [84].

D.1 Dependencies

- python 2.7.12
- numpy 1.11.2
- scipy 0.18.1
- matplotlib 1.5.3
- seaborn 0.7.1
- sqlite 3.13
- vtk 6.3.0
- MDSplus 7.7.1 and mdsplus python module (alpha7.0.144)
- VisIt 2.10.3 and corresponding visit python module

The dependencies can be installed with the anaconda python distribution. The MDSplus can be found at http://www.mdsplus.org/index.php/Introduction. The mdsplus python module can be found at https://pypi.python.org/pypi/MDSplus. VisIt can be found at https://visit.llnl.gov. Instructions for installing the VisIt https://visit.llnl.gov/manuals.

D.2 RSX data and analysis scripts

This code processes both raw data and processed data from the RSX experiment. The RSX data and RSX data analysis codes have not been released yet. Without these inputs the code can not currently be run, however, it could be adapted to run on other datasets. When they are a note will be added to the Github repository and Zenodo on how to obtain them.

D.3 Directory structure

1. Create two directories `output`, `source`. 2. Git clone or copy the repo into `source`.

D.4 Important files

Each of these files has documentation that can be accessed with `python filename -help`

write_measurements_to_unstructured_grid.py reads processed Bdot and triple probe, and raw Mach probe measurements and writes them to unstructured vtk files files.

fit_field_null.py reads unstructured Bx and By fields and finds the field nulls in a measurement plane.

interpolate_measurements.py interpolates fields from unstructured measurements to rectilinear grid.

calculate_dependent_quantities.py calculates the dependent quantities needed to plot flux tubes and calculate helicity from interpolated measurements.

calculate_helicity.py calculates gauge-dependent and relative helicities.

vis_canonical_fluxtubes.py plots frames of canonical flux tube animations. Plot options include ion, electron canonical flux tubes, temperature and density isosurfaces, current contours in an x-y plane.

thesis_figures/thesis_figures.ipynb a jupyter notebook generating the figures for the 5th chapter of my thesis.

D.5 Reference

von der Linden. 2017. Investigating the Dynamics of Canonical Flux Tubes. (PhD thesis) (In preparation.)

D.6 Contact

Jens von der Linden jensv@uw.edu

D.7 Main Scripts

D.7.1 write_measurements_to_unstructured_grid.py

```
#! /Users/vonderlinden2/anaconda/bin/python
# -*- coding: utf-8 -*-
"""
Created on Feb 14 2017

@author: Jens von der Linden

Read processed measurements
and write to unstructured vtk files files.
The processed measurements are magnetic field, density, temperature
and Mach number.
Mach number is calculated directly from the raw MDSplus Mach probe
voltages, all other data is read from IDL output files from the RSX data anlysis
scripts.
Only measurement points slightly larger than the
intended interpolation spaces are kept and Delaunay triangulated
to define the unstructured grid.
"""
import argparse
import numpy as np
```

```
21  from datetime import date
22  from datetime import datetime
23  import os
24
25  from mach_probe_analysis import ion_current_to_mach_number as ic_to_mach
26  from read_from_sql import read_from_sql
27  from write_to_vtk import prepare_measurements as pm
28  from write_to_vtk import unstructured_grid as ug
29
30
31  def main(args):
32      r"""
33      Read processed measurements and write to unstructured vtk files.
34      """
35      now = datetime.now().strftime("%Y-%m-%d-%H-%M")
36      out_dir = '../output/boxed_unstructured_measurements/' + now
37      try:
38          os.makedirs(out_dir)
39      except:
40          pass
41
42      planes = [0.249, 0.302, 0.357, 0.416]
43      bx_measurements = pm.read_idl('bx')
44      by_measurements = pm.read_idl('by')
45      bz_measurements = pm.read_idl('bz')
46      te_measurements = pm.read_idl('te')
47      n_measurements = pm.read_idl('n')
48      mach_y_measurements, mach_z_measurements = pm.read_mach_probe_data(args)
49
50      bx_all_planes = pm.cut_and_average_quantity(bx_measurements,
51                                                  args.bx_extent, planes)
52      by_all_planes = pm.cut_and_average_quantity(by_measurements,
53                                                  args.by_extent, planes)
54      bz_all_planes = pm.cut_and_average_quantity(bz_measurements,
55                                                  args.bz_extent, planes)
56      n_all_planes = pm.cut_and_average_quantity(n_measurements,
57                                                 args.n_extent,
58                                                 planes,
59                                                 bounds=args.n_bounds)
60      te_all_planes = pm.cut_and_average_quantity(te_measurements, args.te_extent,
61                                                  planes, bounds=args.te_bounds)
62      mach_y_plane = pm.cut_and_average_quantity(mach_y_measurements, args.mach_y_extent,
63                                                 [0.416], bounds=args.mach_bounds)
64      mach_z_plane = pm.cut_and_average_quantity(mach_z_measurements, args.mach_z_extent,
65                                                 [0.416], bounds=args.mach_bounds)
66
67
68      n_three_planes = pm.remove_plane(0.302, n_all_planes)
69      te_three_planes = pm.remove_plane(0.302, te_all_planes)
70
71      ug.save_to_unstructured_grid(bx_all_planes, 'bx', out_dir,
72                                   prefix=args.output_prefix)
73      ug.save_to_unstructured_grid(by_all_planes, 'by', out_dir,
74                                   prefix=args.output_prefix)
75      ug.save_to_unstructured_grid(bz_all_planes, 'bz', out_dir,
76                                   prefix=args.output_prefix)
77      ug.save_to_unstructured_grid(te_three_planes, 'te', out_dir,
```

```
78                                  prefix=args.output_prefix)
79      ug.save_to_unstructured_grid(n_three_planes, 'n', out_dir,
80                                  prefix=args.output_prefix)
81      ug.save_to_unstructured_grid(mach_y_plane, 'mach_y', out_dir,
82                                  prefix=args.output_prefix)
83      ug.save_to_unstructured_grid(mach_z_plane, 'mach_z', out_dir,
84                                  prefix=args.output_prefix)
85
86  def parse_args():
87      r"""
88      Read Arguments.
89      """
90      parser = argparse.ArgumentParser(description='Create unstructured VTK from measurements')
91      parser.add_argument('--bx_extent',
92                          help='spatial extent of Bx measurements',
93                          nargs=6, type=float,
94                          default=[-0.032, 0.028, -0.022, 0.032, 0.249, 0.416])
95      parser.add_argument('--by_extent',
96                          help='spatial extent of By measurements',
97                          nargs=6, type=float,
98                          default=[-0.032, 0.028, -0.022, 0.032, 0.249, 0.416])
99      parser.add_argument('--bz_extent',
100                         help='spatial extent of Bz measurements',
101                         nargs=6, type=float,
102                         default=[-0.032, 0.028, -0.022, 0.032, 0.249, 0.416])
103     parser.add_argument('--te_extent',
104                         help='spatial extent of temperature measurements',
105                         nargs=6, type=float,
106                         default=[-0.026, 0.028, -0.03, 0.029, 0.249, 0.416])
107     parser.add_argument('--te_bounds',
108                         help='sensible bounds for temperature measurements',
109                         nargs=2, type=float,
110                         default=[1e-3, 1e3])
111     parser.add_argument('--n_extent',
112                         help='spatial extent of density measurements',
113                         nargs=6, type=float,
114                         default=[-0.026, 0.028, -0.03, 0.029, 0.249, 0.416])
115     parser.add_argument('--n_bounds',
116                         help='sensible bounds for density measurements',
117                         nargs=2, type=float,
118                         default=[1e3, 1e22])
119     parser.add_argument('--mach_time_steps',
120                         help='# of time steps to extract from one gyration', type=int,
121                         default=250)
122     parser.add_argument('--shot_database', help='path to shot database',
123                         default='/home/jensv/rsx/jens_analysis/helicity_tools/shots_database/shots.db')
124     parser.add_argument('--table_name', help='name of sql table',
125                         default='Shots')
126     parser.add_argument('--min_spectral',
127                         help=("minimum spectral energy around gyration"
128                               "frequency to include shot"),
129                         type=float,
130                         default=1.6e-8)
131     parser.add_argument('--mach_y_extent',
132                         help='spatial extent of mach measurements to include',
133                         nargs=6, type=float,
134                         default=[-0.052, 0.052, -0.022, 0.032, 0.249, 0.416])
```

```
135     parser.add_argument('--mach_z_extent',
136                         help='spatial extent of mach measurements to include',
137                         nargs=6, type=float,
138                         default=[-0.032, 0.032, -0.022, 0.032, 0.249, 0.416])
139     parser.add_argument('--mach_bounds',
140                         help='bounds on mach measurements', nargs=2, type=float,
141                         default=[-10, 10])
142     parser.add_argument('--output_prefix',
143                         help='prefix of output files',
144                         default='_boxed_unstructured_')
145     args = parser.parse_args()
146     return args
147
148 if __name__ == '__main__':
149     args = parse_args()
150     main(args)
```

D.7.2 fit_field_null.py

```
1   r"""
2   Fits field line null.
3
4   Created March 28 2017 by Jens von der Linden.
5
6   Fit field nulls of the Bx and By magnetic field with
7   iterative circle fitting to the field lines.
8
9   Reads unstructured vtk files and generates Bx and By interpolators.
10  """
11
12  import argparse
13  from scipy.interpolate import LinearNDInterpolator
14  import numpy as np
15  from scipy.optimize import leastsq
16  from scipy import odr
17  from scipy import ndimage
18  from scipy.integrate import odeint, dblquad
19  from datetime import datetime
20  import os
21  from scipy.interpolate import LinearNDInterpolator
22
23  from write_to_vtk.read_unstructured_vtk import read_unstructured_vtk
24  from write_to_vtk import structured_3d_vtk as struc_3d
25
26
27  def main(args):
28      r"""
29      Fit field nulls of the Bx and By magnetic field with
30      iterative circle fitting to the field lines.
31      """
32      now = datetime.now().strftime("%Y-%m-%d-%H-%M")
33      out_dir = '../output/' + args.output_prefix + '/' + now + '/'
34      try:
35          os.makedirs(out_dir)
```

```
36      except:
37          pass
38
39      in_dir = args.input_path + args.input_date + '/'
40      in_file = args.input_file_text
41
42      centroids = []
43
44
45      bxby_extents = {0: args.bxby_extent_0,
46                      1: args.bxby_extent_1,
47                      2: args.bxby_extent_2,
48                      3: args.bxby_extent_3}
49      bz_extents = {0: args.bz_extent_0,
50                    1: args.bz_extent_1,
51                    2: args.bz_extent_2,
52                    3: args.bz_extent_3}
53
54      bxby_extent = bxby_extents[args.plane_number]
55      bz_extent = bz_extents[args.plane_number]
56
57      for time_point in xrange(args.time_steps):
58          print time_point
59          time_str = str(time_point).zfill(4)
60          bx_points, bx_values = read_unstructured_vtk(in_dir + 'bx' +
61                                                       in_file + time_str + '.vtk')
62          by_points, by_values = read_unstructured_vtk(in_dir + 'by' +
63                                                       in_file + time_str + '.vtk')
64          #bz_points, bz_values = read_unstructured_vtk(in_dir + 'bz' +
65          #                                             in_file + time_str + '.vtk')
66          z_value = np.unique(bx_points[:, 2])[args.plane_number]
67
68          z_index = np.where(bx_points[:, 2] == z_value)[0]
69          bx_points = bx_points[z_index, :-1]
70          bx_values = bx_values[z_index]
71          z_index = np.where(by_points[:, 2] == z_value)[0]
72          by_points = by_points[z_index, :-1]
73          by_values = by_values[z_index]
74          #z_index = np.where(bz_points[:, 2] == z_value)[0]
75          #bz_points = bz_points[z_index, :-1]
76          #bz_values = bz_values[z_index]
77
78          bx_interpolator = struc_3d.get_interpolator(bx_points, bx_values)
79          by_interpolator = struc_3d.get_interpolator(by_points, by_values)
80          #bz_interpolator = struc_3d.get_interpolator(bz_points, bz_values)
81          grid_extent = [bxby_extent[0], bxby_extent[1],
82                         -0.02, bxby_extent[3]]
83          grid = np.meshgrid(np.linspace(grid_extent[0], grid_extent[1],
84                                         (grid_extent[1] - grid_extent[0])/
85                                         args.spatial_increment),
86                             np.linspace(grid_extent[2], grid_extent[3],
87                                         (grid_extent[3] - grid_extent[2])/
88                                         args.spatial_increment))
89
90          (centroid, center_points,
91           radii, streamlines,
92           max_index) = find_field_null(grid,
```

```
                                    bx_interpolator,
                                    by_interpolator,
                                    launch_point_step_factor=0.05,
                                    integration_length=20)
        centroids.append(centroid)

    centroids = np.asarray(centroids)
    np.savetxt(out_dir + '/field_nulls.txt', centroids,
               header=("magnetic field null positions in z plane # %d plane,"
                       "determined by"
                       "fitting circles to integrated field lines starting at max"
                       "magnitude and moving successive towards the center of circles."
                       % args.plane_number))


def d_l(l, t, interpolator_x, interpolator_y):
    r"""
    Returns d_l for the field line integrator.
    """
    return np.asarray([interpolator_x([l[0], l[1]])[0],
                       interpolator_y([l[0], l[1]])[0]])


def to_min(params, points):
    r"""
    Returns circle expression to minimize with least squares.
    """
    a = 2.*params[0]
    b = 2.*params[1]
    c = params[2]**2 - params[1]**2 - params[0]**2
    return a*points[0] + b*points[1] + c - points[0]**2 - points[1]**2


def find_field_null(grid, bx_interpolator, by_interpolator,
                    distance_thres=0.001, filter_size=2,
                    integration_length=10, integration_steps=100,
                    launch_point_step_factor=0.1, max_count=50,
                    params_guess=[0, 0, 0.01]):
    r"""
    Find Bx-By field null in a x-y plane
    by integrating field lines and fitting a circle to them.
    Move towards the center and iterate process.
    If leaving the measurement plane
    extrapolate from last fit circle.
    Start close to the Bx-By field max.
    """
    b_fields_x = bx_interpolator(grid[0][:, :], grid[1][:, :])
    b_fields_y = by_interpolator(grid[0][:, :], grid[1][:, :])
    b_fields = [b_fields_x, b_fields_y]
    x_min, x_max = grid[0].min(), grid[0].max()
    y_min, y_max = grid[1].min(), grid[1].max()
    magnitude = np.sqrt(b_fields[0][:, :]**2 + b_fields[1][:, :]**2)
    filtered_magnitude = ndimage.gaussian_filter(magnitude, filter_size)
    max_index = np.unravel_index( np.nanargmax(filtered_magnitude),
                                  filtered_magnitude.shape)
    center_points = []
    radii = []
```

```
150         center_points = []
151         streamlines = []
152         direction = [0, 0]
153         distance = 100
154         launch_point = (grid[0][:][max_index], grid[1][:][max_index])
155         count = 0
156         while distance >= distance_thres:
157             #print 'launch', launch_point
158             #print distance
159             t2 = np.linspace(0, integration_length, integration_steps)
160             t1 = np.linspace(0, -integration_length, integration_steps)
161             stream2 = odeint(d_l, launch_point, t2, args=(bx_interpolator, by_interpolator))
162             stream1 = odeint(d_l, launch_point, t1, args=(bx_interpolator, by_interpolator))
163             streamline = np.concatenate((stream1, stream2))
164             size0 = np.sum(np.invert(np.isnan(streamline[:, 0])))
165             size1 = np.sum(np.invert(np.isnan(streamline[:, 1])))
166             min_index = np.argmin([size0, size1])
167             min_size = [size0, size1][min_index]
168             streamline = streamline[np.invert(np.isnan(streamline[:, min_index]))].reshape(min_size, 2)
169             try:
170                 circle_params, success = leastsq(to_min, params_guess,
171                                                  args=np.asarray([streamline[:, 0],
172                                                                   streamline[:, 1]]))
173             except:
174                 break
175             direction = [circle_params[0] - launch_point[0], circle_params[1] - launch_point[1]]
176             distance = np.sqrt(direction[0]**2. + direction[1]**2.)
177             center_point = (circle_params[0], circle_params[1])
178             launch_point = [launch_point[0] + direction[0] * launch_point_step_factor,
179                             launch_point[1] + direction[1] * launch_point_step_factor]
180             center_points.append(center_point)
181             #print 'center', center_point
182             radii.append(circle_params[0])
183             streamlines.append(streamline)
184             if (launch_point[0] <= x_min or
185                 launch_point[0] >= x_max or
186                 launch_point[1] <= y_min or
187                 launch_point[1] >= y_max or
188                 count > max_count):
189                 break
190             count += 1
191         field_null = center_point
192         return field_null, center_points, radii, streamlines, max_index
193
194
195     def integrate_flux(centroid, radius, bz_interpolator, limits, bias_field=0.02):
196         r"""
197         Return axial magnetic flux and error estimate integrated
198         in a circle of given radius around a given centroid.
199         """
200         if (centroid[0] - radius < limits[0] or centroid[0] + radius > limits[1] or
201             centroid[1] - radius < limits[2] or centroid[1] + radius > limits[3]):
202             return -1
203         gfun = lambda x: -np.sqrt(radius**2 - (x-centroid[0])**2)
204         hfun = lambda x: np.sqrt(radius**2 - (x-centroid[0])**2)
205         bz_interpolator_bias  = lambda x, y: bz_interpolator(x, y) + bias_field
206         return dblquad(bz_interpolator_bias, centroid[0] - radius,
```

```
207                     centroid[0] + radius, gfun, hfun)
208
209
210 def parse_args():
211     r"""
212     Read arguments.
213     """
214     parser = argparse.ArgumentParser(description=("Create VTK files of"
215                                                   "interpolated measurements"))
216     parser.add_argument('--input_path',
217                         help='path to input files',
218                         default='../output/boxed_unstructured_measurements/')
219     parser.add_argument('--input_date',
220                         help='time stamp of input files',
221                         default='2017-04-11-21-07')
222     parser.add_argument('--input_file_text',
223                         help='input file name',
224                         default='_boxed_unstructured_')
225     parser.add_argument('--spatial_increment',
226                         help='Spatial increment of grids',
227                         type=float, default=0.001)
228     parser.add_argument('--derivative_increment',
229                         help=("spatial increment used to determine"
230                               "tetrahedron derivative of Delaunay"),
231                         type=float, default=0.0001)
232     parser.add_argument('--plane_number',
233                         help="z-Plane number on which to find field nulls.",
234                         type=int, default=0)
235     parser.add_argument('--bxby_extent_0',
236                         help='overlapping spatial extent of bx by',
237                         nargs=6, type=float,
238                         default=[-0.027, 0.025, -0.057, 0.040, 0.249, 0.416])
239     parser.add_argument('--bxby_extent_1',
240                         help='overlapping spatial extent of bx by',
241                         nargs=6, type=float,
242                         default=[-0.027, 0.027, -0.073, 0.041, 0.249, 0.416])
243     parser.add_argument('--bxby_extent_2',
244                         help='overlapping spatial extent of bx by',
245                         nargs=6, type=float,
246                         default=[-0.047, 0.031, -0.021, 0.028, 0.249, 0.416])
247     parser.add_argument('--bxby_extent_3',
248                         help='overlapping spatial extent of bx by',
249                         nargs=6, type=float,
250                         default=[-0.061, 0.031, -0.026, 0.03, 0.249, 0.416])
251     parser.add_argument('--bz_extent_0',
252                         help='spatial extent of bz',
253                         nargs=6, type=float,
254                         default=[-0.027, 0.025, -0.06, 0.041, 0.249, 0.416])
255     parser.add_argument('--bz_extent_1',
256                         help='spatial extent of bz',
257                         nargs=6, type=float,
258                         default=[-0.27, 0.027, -0.076, 0.041, 0.249, 0.416])
259     parser.add_argument('--bz_extent_2',
260                         help='spatial extent of bz',
261                         nargs=6, type=float,
262                         default=[-0.044, 0.031, -0.021, 0.03, 0.249, 0.416])
263     parser.add_argument('--bz_extent_3',
```

```
                        help='spatial extent of bz',
                        nargs=6, type=float,
                        default=[-0.072, 0.031, -0.026, 0.03, 0.249, 0.416])
    parser.add_argument('--output_prefix',
                        help='prefix of output files',
                        default='field_nulls')
    parser.add_argument('--bias_field_magnitude',
                        help='magnitude of axial bias magnetic field',
                        type=float,
                        default=0.02)
    parser.add_argument('--time_steps',
                        help='number of time steps', type=int,
                        default=250)
    args = parser.parse_args()
    return args


if __name__ == '__main__':
    args = parse_args()
    main(args)
```

D.7.3 interpolate_measurements.py

```
#! /Users/vonderlinden2/anaconda/bin/python
# -*- coding: utf-8 -*-
"""
Created on Mon Aug  1 18:07:14 2016

@author: Jens von der Linden

Interpolate unstructured fields to rectilinear grid.
Read unstructured fields from unstructured vtk file.
Write interpolated fields to rectilinear vtk file.
"""
import argparse
import numpy as np
from scipy.interpolate import LinearNDInterpolator
from datetime import date
from datetime import datetime
import os

from write_to_vtk.read_unstructured_vtk import read_unstructured_vtk
from write_to_vtk import structured_3d_vtk as struc_3d

def main(args):
    r"""
    Interpolate unstructured fields to rectilinear grid.
    """
    just_magnetic = args.just_magnetic
    now = datetime.now().strftime("%Y-%m-%d-%H-%M")
    out_dir = '../output/' + args.output_prefix + '/' + now + '/'
    try:
        os.makedirs(out_dir)
    except:
```

```
32          pass
33
34      in_dir = args.input_path + args.input_date + '/'
35      in_file = args.input_file_text
36
37      for time_point in xrange(args.time_steps):
38          print time_point
39          time_str = str(time_point).zfill(4)
40          bx_points, bx_values = read_unstructured_vtk(in_dir + 'bx' +
41                                                       in_file + time_str + '.vtk')
42          by_points, by_values = read_unstructured_vtk(in_dir + 'by' +
43                                                       in_file + time_str + '.vtk')
44          bz_points, bz_values = read_unstructured_vtk(in_dir + 'bz' +
45                                                       in_file + time_str + '.vtk')
46          if not just_magnetic:
47              n_points, n_values = read_unstructured_vtk(in_dir + 'n' +
48                                                         in_file + time_str + '.vtk')
49              te_points, te_values = read_unstructured_vtk(in_dir + 'te' +
50                                                           in_file + time_str + '.vtk')
51              mach_y_points, mach_y_values = read_unstructured_vtk(in_dir + 'mach_y' +
52                                                                   in_file + time_str + '.vtk')
53              mach_z_points, mach_z_values = read_unstructured_vtk(in_dir + 'mach_z' +
54                                                                   in_file + time_str + '.vtk')
55
56          bx_interpolator = struc_3d.get_interpolator(bx_points, bx_values)
57          by_interpolator = struc_3d.get_interpolator(by_points, by_values)
58          bz_interpolator = struc_3d.get_interpolator(bz_points, bz_values)
59          if not just_magnetic:
60              te_interpolator = struc_3d.get_interpolator(te_points, te_values)
61              n_interpolator = struc_3d.get_interpolator(n_points, n_values)
62              mach_y_interpolator = struc_3d.get_interpolator(mach_y_points[:, :2], mach_y_values)
63              mach_z_interpolator = struc_3d.get_interpolator(mach_z_points[:, :2], mach_z_values)
64
65          (x_min, x_max,
66           y_min, y_max,
67           z_min, z_max) = args.joint_extent
68
69          mesh = np.meshgrid(np.linspace(x_min, x_max,
70                                         np.ceil((x_max-x_min)/
71                                                 args.spatial_increment)),
72                             np.linspace(y_min, y_max,
73                                         np.ceil((y_max-y_min)/
74                                                 args.spatial_increment)),
75                             np.linspace(z_min, z_max,
76                                         np.ceil((z_max-z_min)/
77                                                 args.spatial_increment)))
78
79
80          bx_grad = struc_3d.triangulate_grad(mesh, bx_interpolator,
81                                              increment=args.derivative_increment)
82          by_grad = struc_3d.triangulate_grad(mesh, by_interpolator,
83                                              increment=args.derivative_increment)
84          bz_grad = struc_3d.triangulate_grad(mesh, bz_interpolator,
85                                              increment=args.derivative_increment)
86          if not just_magnetic:
87              te_grad = struc_3d.triangulate_grad(mesh, te_interpolator,
88                                                  increment=args.derivative_increment)
```

```
89              n_grad = struc_3d.triangulate_grad(mesh, n_interpolator,
90                                                 increment=args.derivative_increment)
91              plane_mesh = [mesh[0][:, :, 0], mesh[1][:, :, 0]]
92              mach_y_grad_plane = struc_3d.triangulate_grad(plane_mesh, mach_y_interpolator,
93                                                            increment=args.derivative_increment)
94              mach_z_grad_plane = struc_3d.triangulate_grad(plane_mesh, mach_z_interpolator,
95                                                            increment=args.derivative_increment)
96
97
98          bx, by, bz = struc_3d.vector_on_mesh((bx_interpolator,
99                                                by_interpolator,
100                                               bz_interpolator), mesh)
101         bx, by, bz = struc_3d.add_vacuum_field([bx, by, bz],
102                                                vacuum_field=args.bias_field_magnitude)
103         if not just_magnetic:
104             te = struc_3d.scalar_on_mesh(te_interpolator, mesh)
105             n = struc_3d.scalar_on_mesh(n_interpolator, mesh)
106             mach_y_plane = struc_3d.scalar_on_mesh(mach_y_interpolator,
107                                                    plane_mesh)
108             mach_z_plane= struc_3d.scalar_on_mesh(mach_z_interpolator,
109                                                   plane_mesh)
110
111             mach_y = np.repeat(mach_y_plane[:, :, np.newaxis],
112                                mesh[0].shape[2], axis=2)
113             mach_z = np.repeat(mach_z_plane[:, :, np.newaxis],
114                                mesh[0].shape[2], axis=2)
115
116             mach_y_dx = np.repeat(mach_y_grad_plane[0][:, :, np.newaxis],
117                                   mesh[0].shape[2], axis=2)
118             mach_y_dy = np.repeat(mach_y_grad_plane[1][:, :, np.newaxis],
119                                   mesh[0].shape[2], axis=2)
120             mach_y_dz = np.zeros(mesh[0].shape)
121
122             mach_z_dx = np.repeat(mach_z_grad_plane[0][:, :, np.newaxis],
123                                   mesh[0].shape[2], axis=2)
124             mach_z_dy = np.repeat(mach_z_grad_plane[0][:, :, np.newaxis],
125                                   mesh[0].shape[2], axis=2)
126             mach_z_dz = np.zeros(mesh[0].shape)
127
128             fields = ([bx] + [by] + [bz] + [n] + [te] +
129                       [mach_y] + [mach_z] +
130                       list(bx_grad) +
131                       list(by_grad) +
132                       list(bz_grad) +
133                       list(n_grad) +
134                       list(te_grad) +
135                       [mach_y_dx] + [mach_y_dy] + [mach_y_dz] +
136                       [mach_z_dx] + [mach_z_dy] + [mach_z_dz])
137
138             quantity_names = ['B_x', 'B_y', 'B_z',
139                               'n', 'Te',
140                               'mach_y', 'mach_z',
141                               'B_x_dx', 'B_x_dy', 'B_x_dz',
142                               'B_y_dx', 'B_y_dy', 'B_y_dz',
143                               'B_z_dx', 'B_z_dy', 'B_z_dz',
144                               'n_dx', 'n_dy', 'n_dz',
145                               'Te_dx', 'Te_dy', 'Te_dz',
```

```
146                             'mach_y_dx', 'mach_y_dy', 'mach_y_dz',
147                             'mach_z_dx', 'mach_z_dy', 'mach_z_dz']
148         else:
149             fields = ([bx] + [by] + [bz] +
150                       list(bx_grad) +
151                       list(by_grad) +
152                       list(bz_grad))
153
154             quantity_names = ['B_x', 'B_y', 'B_z',
155                             'B_x_dx', 'B_x_dy', 'B_x_dz',
156                             'B_y_dx', 'B_y_dy', 'B_y_dz',
157                             'B_z_dx', 'B_z_dy', 'B_z_dz']
158
159
160         x, y, z, variables = struc_3d.prepare_for_rectilinear_grid(mesh, fields,
161                                                                    quantity_names)
162
163         vtk_file_path = out_dir + args.output_prefix
164         struc_3d.write_fields_to_rectilinear_grid(vtk_file_path,
165                                                   x, y, z, variables,
166                                                   time_point)
167
168 def parse_args():
169     r"""
170     Read arguments.
171     """
172     parser = argparse.ArgumentParser(description=("Create VTK files of"
173                                                   "interpolated measurements"))
174     parser.add_argument('--input_path',
175                         help='path to input files',
176                         default='../output/boxed_unstructured_measurements/')
177     parser.add_argument('--input_date',
178                         help='time stamp of input files',
179                         default='2017-04-04-13-44')
180     parser.add_argument('--input_file_text',
181                         help='input file name',
182                         default='_boxed_unstructured_')
183     parser.add_argument('--spatial_increment',
184                         help='Spatial increment of output file grids',
185                         type=float, default=0.001)
186     parser.add_argument('--derivative_increment',
187                         help=("spatial increment used to determine"
188                               "tetrahedron derivative of Delaunay"),
189                         type=float, default=0.0001)
190     parser.add_argument('--joint_extent',
191                         help='overlapping spatial extent of all parameters',
192                         nargs=6, type=float,
193                         default=[-0.022, 0.024, -0.02, 0.029, 0.249, 0.416])
194     parser.add_argument('--output_prefix',
195                         help='prefix of output files',
196                         default='data_interp_to_rect_grid')
197     parser.add_argument('--bias_field_magnitude',
198                         help='magnitude of axial bias magnetic field',
199                         type=float,
200                         default=0.02)
201     parser.add_argument('--time_steps',
202                         help='number of time steps', type=int,
```

```
                        default=250)
    parser.add_argument('--just_magnetic',
                        help='only interpolate bdot measurements',
                        action='store_true', default=False)
    #parser.add_argument('--just_one_time_step',
    #                    help='only interpolate first time step',
    #                    action='store_true', default=False)
    args = parser.parse_args()
    return args

if __name__ == '__main__':
    args = parse_args()
    main(args)
```

D.7.4 calculate_dependent_quantities.py

```
#! /Users/vonderlinden2/anaconda/bin/python
# -*- coding: utf-8 -*-
"""
Created on Mon Aug  1 18:07:14 2016

@author: Jens von der Linden

Calculate the dependent fields from interpolated
measurements.
Optionally filter the fields with Gaussian kernels of given
standard deviation for sensitivity / uncertainty analysis.

The dependent fields are:
ion_vorticity,
ion_vorticity_smooth,
current,
current_smooth,
density_plane_normalized,
temperature_plane_normalized,
vector_potential,
b_field_ref,
vector_potential_ref,
ion_vorticity_ref,
ion_vorticity_smooth_ref,
ion_velocity_ref,
ion_velocity_smooth_ref
"""
import argparse
import numpy as np
from scipy.constants import mu_0
from scipy.constants import elementary_charge as q_e
from scipy.constants import proton_mass as m_i
from astropy.convolution import convolve, convolve_fft
from scipy.ndimage import gaussian_filter
from scipy import ndimage
from scipy.signal import fftconvolve
import scipy
from datetime import date
```

```
39  from datetime import datetime
40  import visit_writer
41  import os
42
43  from vector_comparison import read_rectilinear_vtk as rrv
44  from write_to_vtk import structured_3d_vtk as struc_3d
45  from invert_curl.invert_curl import devore_invert_curl
46  from write_to_vtk.reference_field import determine_reference_fields as ref_fields
47  from vector_calculus import vector_calculus as vc
48
49  def main(args):
50      r"""
51      Calculate the dependent fields from interpolated
52      measurements.
53      """
54      just_magnetic = args.just_magnetic
55      interpolate_nan = args.interpolate_nan
56      now = datetime.now().strftime("%Y-%m-%d-%H-%M")
57      out_dir = '../output/' + args.output_prefix + '/' + now + '/'
58      try:
59          os.makedirs(out_dir)
60      except:
61          pass
62      in_dir = args.input_path + args.input_date + '/'
63      in_file = args.input_file_text
64      input_path = in_dir + in_file
65
66      if not just_magnetic:
67          fields_to_read = ('B_x', 'B_y', 'B_z',
68                            'n', 'Te',
69                            'mach_y', 'mach_z',
70                            'B_x_dx', 'B_x_dy', 'B_x_dz',
71                            'B_y_dx', 'B_y_dy', 'B_y_dz',
72                            'B_z_dx', 'B_z_dy', 'B_z_dz',
73                            'n_dx', 'n_dy', 'n_dz',
74                            'Te_dx', 'Te_dy', 'Te_dz',
75                            'mach_y_dx', 'mach_y_dy', 'mach_y_dz',
76                            'mach_z_dx', 'mach_z_dy', 'mach_z_dz')
77          mach_fields_to_read = ('mach_y', 'mach_y_dx', 'mach_y_dy', 'mach_y_dz')
78      else:
79          fields_to_read = ('B_x', 'B_y', 'B_z',
80                            'B_x_dx', 'B_x_dy', 'B_x_dz',
81                            'B_y_dx', 'B_y_dy', 'B_y_dz',
82                            'B_z_dx', 'B_z_dy', 'B_z_dz')
83
84      u_x_time_points_plus = np.roll(np.arange(250), int(np.round(250*0.25)))
85      u_x_time_points_minus = np.roll(np.arange(250), -int(np.round(250*0.25)))
86      xlow, xhigh, ylow, yhigh, zlow, zhigh = args.to_cut
87
88
89      for time_point in xrange(args.time_steps):
90          print time_point
91          input_file = input_path + str(time_point).zfill(4) + '.vtk'
92          fields = {}
93          for field in fields_to_read:
94              read = rrv.read_scalar(input_file, field)
95              fields[field] = read[1][ylow:yhigh, xlow:xhigh, zlow:zhigh]
```

```
96              if args.filter_gaussian:
97                  fields[field] = filter_data(fields[field],
98                                             filter_sigma=args.filter_sigma,
99                                             filter_truncate=args.filter_truncate)
100
101         mesh = read[0]
102         mesh[0] = mesh[0][ylow:yhigh, xlow:xhigh, zlow:zhigh]
103         mesh[1] = mesh[1][ylow:yhigh, xlow:xhigh, zlow:zhigh]
104         mesh[2] = mesh[2][ylow:yhigh, xlow:xhigh, zlow:zhigh]
105
106         bx_grad = [fields['B_x_dx'], fields['B_x_dy'], fields['B_x_dz']]
107         by_grad = [fields['B_y_dx'], fields['B_y_dy'], fields['B_y_dz']]
108         bz_grad = [fields['B_z_dx'], fields['B_z_dy'], fields['B_z_dz']]
109
110         ## current
111         ##
112         current = 1/mu_0*np.asarray(vc.curl(None,
113                                             vector_grad=[bx_grad,
114                                                          by_grad,
115                                                          bz_grad]))
116         current_smooth = []
117         for direction in xrange(len(current)):
118             current_dir = np.array(current[direction])
119             for z_step in xrange(len(current[0, 0, :])):
120                 current_dir[:, :, z_step] = (gaussian_filter(current_dir[:, :, z_step],
121                                                              args.filter_width,
122                                                              mode='reflect'))
123             current_smooth.append(current_dir)
124
125         ## density and temperature
126         ##
127         if not just_magnetic:
128             density = fields['n']
129             grad_density = (fields['n_dx'], fields['n_dy'], fields['n_dz'])
130             temperature = fields['Te']
131             grad_temperature = (fields['Te_dx'], fields['Te_dy'], fields['Te_dz'])
132
133             density_plane_normalized = normalize_scalar_by_plane(density)
134             temperature_plane_normalized = normalize_scalar_by_plane(temperature)
135             density_smooth = boxcar_filter(density,
136                                            args.filter_width)
137             temperature_smooth = boxcar_filter(temperature,
138                                                args.filter_width)
139             density_smooth_plane_normalized = normalize_scalar_by_plane(density_smooth)
140             temperature_smooth_plane_normalized = normalize_scalar_by_plane(temperature_smooth)
141             density_constant = args.density_constant_factor*np.ones(density.shape)
142             temperature_smooth_norm = temperature_smooth/np.nanmax(temperature_smooth)
143             temperature_norm = (temperature/
144                                 np.nanmax(temperature))
145
146             ## velocity and vorticity
147             ##
148             mach_file = (input_path +
149                          str(u_x_time_points_plus[time_point]).zfill(4) +
150                          '.vtk')
151             mach_x_fields = {}
152             for field in mach_fields_to_read:
```

```
153                 read = rrv.read_scalar(mach_file, field)
154                 mach_x_fields[field] = read[1][ylow:yhigh, xlow:xhigh, zlow:zhigh]
155             mach_x_p = mach_x_fields['mach_y']
156             grad_mach_x_p = (mach_x_fields['mach_y_dx'],
157                              mach_x_fields['mach_y_dy'],
158                              mach_x_fields['mach_y_dz'])
159             mach_file = (input_path +
160                          str(u_x_time_points_minus[time_point]).zfill(4) +
161                          '.vtk')
162             mach_x_fields = {}
163             for field in mach_fields_to_read:
164                 read = rrv.read_scalar(mach_file, field)
165                 mach_x_fields[field] = read[1][ylow:yhigh, xlow:xhigh, zlow:zhigh]
166             mach_x_m = mach_x_fields['mach_y']
167             grad_mach_x_m = (mach_x_fields['mach_y_dx'],
168                              mach_x_fields['mach_y_dy'],
169                              mach_x_fields['mach_y_dz'])
170             mach_y = fields['mach_y']
171             grad_mach_y = (fields['mach_y_dx'],
172                            fields['mach_y_dy'],
173                            fields['mach_y_dz'])
174             mach_z = fields['mach_z']
175             grad_mach_z = (fields['mach_z_dx'],
176                            fields['mach_z_dy'],
177                            fields['mach_z_dz'])
178             te = temperature
179             u_i_x_plus = np.sqrt(te*q_e/m_i)*mach_x_p
180             u_i_x_minus = np.sqrt(te*q_e/m_i)*mach_x_m
181             u_i_y = np.sqrt(te*q_e/m_i)*mach_y
182             u_i_z = np.sqrt(te*q_e/m_i)*mach_z
183             ion_velocity_p = [u_i_x_plus, u_i_y, u_i_z]
184             ion_velocity_m = [u_i_x_minus, u_i_y, u_i_z]
185             grad_ion_velocity_x_p = grad_ion_velocity(mach_x_p, grad_mach_x_p,
186                                                       temperature, grad_temperature)
187             grad_ion_velocity_x_m = grad_ion_velocity(mach_x_m, grad_mach_x_m,
188                                                       temperature, grad_temperature)
189             grad_ion_velocity_y = grad_ion_velocity(mach_y, grad_mach_y,
190                                                     temperature, grad_temperature)
191             grad_ion_velocity_z = grad_ion_velocity(mach_z, grad_mach_z,
192                                                     temperature, grad_temperature)
193             ion_vorticity_p =  vc.curl(None,
194                                        vector_grad=[grad_ion_velocity_x_p,
195                                                     grad_ion_velocity_y,
196                                                     grad_ion_velocity_z])
197             ion_vorticity_m =  vc.curl(None,
198                                        vector_grad=[grad_ion_velocity_x_m,
199                                                     grad_ion_velocity_y,
200                                                     grad_ion_velocity_z])
201
202             ion_vorticity_p_smooth = []
203             ion_vorticity_m_smooth = []
204             for direction in xrange(len(ion_vorticity_p)):
205                 ion_vorticity_p_smooth.append(gaussian_filter(ion_vorticity_p[direction],
206                                                               args.filter_width, mode='reflect'))
207                 ion_vorticity_m_smooth.append(gaussian_filter(ion_vorticity_m[direction],
208                                                               args.filter_width,
209                                                               mode='reflect'))
```

```
210             ion_velocity = ion_velocity_p
211             ion_vorticity = ion_vorticity_p
212             ion_vorticity_smooth = ion_vorticity_p_smooth
213
214         ## Vector Potential
215         ##
216         b_field = [fields['B_x'], fields['B_y'], fields['B_z']]
217         vector_potential = devore_invert_curl(mesh,
218                                               b_field)
219
220         b_field_dynamic = np.array(b_field)
221         b_field_dynamic[2] = b_field[2] - args.bias_field_magnitude
222         vector_potential_dynamic = devore_invert_curl(mesh,
223                                                       b_field_dynamic)
224         ## Reference fields
225         ##
226         (vector_potential_ref,
227          b_field_ref,
228          b_scalar_potential_ref) = ref_fields(mesh, b_field,
229                                               return_scalar_ref=True)
230         (vector_potential_dynamic_ref,
231          b_field_dynamic_ref,
232          b_scalar_potential_dynamic_ref) = ref_fields(mesh, b_field_dynamic,
233                                                       return_scalar_ref=True)
234         if not just_magnetic:
235             (ion_velocity_smooth_ref,
236              ion_vorticity_smooth_ref,
237              i_scalar_potential_smooth_ref) = ref_fields(mesh, ion_vorticity_smooth,
238                                                          return_scalar_ref=True)
239             (ion_velocity_ref,
240              ion_vorticity_ref,
241              i_scalar_potential_ref) = ref_fields(mesh, ion_vorticity,
242                                                   return_scalar_ref=True)
243
244             fields = ([b_field[0]] + [b_field[1]] + [b_field[2]] +
245                       list(current_smooth) + list(current) +
246                       [density] + [temperature] +
247                       [density_smooth_plane_normalized] +
248                       [temperature_smooth_plane_normalized] +
249                       [density] + [temperature] +
250                       [density_plane_normalized] +
251                       [temperature_plane_normalized] +
252                       list(ion_velocity) +
253                       list(ion_vorticity_smooth) + list(ion_vorticity) +
254                       list(vector_potential) + list(b_field_ref) +
255                       list(vector_potential_ref) +
256                       list(ion_velocity_ref) +
257                       list(ion_vorticity_smooth_ref) +
258                       list(ion_vorticity_ref))
259
260             quantity_names = ['B_x', 'B_y', 'B_z',
261                               'j_x', 'j_y', 'j_z',
262                               'j_raw_x', 'j_raw_y', 'j_raw_z',
263                               'n', 'Te',
264                               'n_plane_normalized', 'Te_plane_normalized',
265                               'n_raw', 'Te_raw',
266                               'n_raw_plane_normalized',
```

```
                            'Te_raw_plane_normalized',
                            'u_i_x_plus', 'u_i_y', 'u_i_z',
                            'w_i_x_plus', 'w_i_y_plus', 'w_i_z_plus',
                            'w_i_raw_x_plus', 'w_i_raw_y_plus', 'w_i_raw_z_plus',
                            'A_x', 'A_y', 'A_z',
                            'B_ref_x', 'B_ref_y', 'B_ref_z',
                            'A_ref_x', 'A_ref_y', 'A_ref_z',
                            'u_i_ref_x', 'u_i_ref_y', 'u_i_ref_z',
                            'w_i_ref_x', 'w_i_ref_y', 'w_i_ref_z',
                            'w_i_raw_ref_x', 'w_i_raw_ref_y', 'w_i_raw_ref_z']

        else:
            fields = ([b_field[0]] + [b_field[1]] + [b_field[2]] +
                      list(current_smooth) + list(current) +
                      list(b_field_dynamic) +
                      list(b_field_dynamic_ref) +
                      list(vector_potential_dynamic) +
                      list(vector_potential_dynamic_ref) +
                      [b_scalar_potential_ref] +
                      [b_scalar_potential_dynamic_ref])

            quantity_names = ['B_x', 'B_y', 'B_z',
                              'j_x', 'j_y', 'j_z',
                              'j_raw_x', 'j_raw_y', 'j_raw_z',
                              'A_x', 'A_y', 'A_z',
                              'B_ref_x', 'B_ref_y', 'B_ref_z',
                              'A_ref_x', 'A_ref_y', 'A_ref_z',
                              'B_dynamic_x', 'B_dynamic_y', 'B_dynamic_z',
                              'B_dynamic_ref_x', 'B_dynamic_ref_y', 'B_dynamic_ref_z',
                              'A_dynamic_x', 'A_dynamic_y', 'A_dynamic_z',
                              'A_dynamic_ref_x', 'A_dynamic_ref_y', 'A_dynamic_ref_z',
                              'phi_b_ref',
                              'phi_b_dynamic_ref']


        for i, field in enumerate(fields):
            fields[i] = field.astype('float64')

        x, y, z, variables = struc_3d.prepare_for_rectilinear_grid(mesh, fields,
                                                                   quantity_names)
        vtk_file_path = out_dir + args.output_prefix
        struc_3d.write_fields_to_rectilinear_grid(vtk_file_path,
                                                  x, y, z, variables,
                                                  time_point)


def filter_data(data, filter_sigma=None,
                filter_truncate=None):
    r"""
    Filter (with Gaussian).
    """
    if filter_sigma:
        if filter_truncate:
            filtered = ndimage.gaussian_filter(data, filter_sigma,
                                               truncate=filter_truncate)
        else:
            filtered = ndimage.gaussian_filter(data, filter_sigma)
```

```
324         else:
325             filtered = data
326         return filtered
327 
328 
329     def parse_args():
330         r"""
331         Read arguments.
332         """
333         parser = argparse.ArgumentParser(description=("Create VTK files"
334                                                       "of canonical quantities"))
335         parser.add_argument('--input_path',
336                             help='path to input files',
337                             default='../output/data_interp_to_rect_grid/')
338         parser.add_argument('--input_date',
339                             help='time stamp of input files',
340                             default='2017-05-10-11-53')
341         parser.add_argument('--input_file_text',
342                             help='input file name',
343                             default='data_interp_to_rect_grid')
344         parser.add_argument('--output_prefix',
345                             help='prefix of output files',
346                             default='canonical_quantities')
347         parser.add_argument('--to_cut',
348                              help=("number of indicies to cut on"
349                                    "each side to remove NaNs e.g."
350                                    "xlow xhigh ylow yhigh zlow zhigh"),
351                             nargs=6, type=int,
352                             default=[2, -1, 3, -3, 0, -1])
353         parser.add_argument('--filter_width',
354                             help='width of boxcar filter for derivatives',
355                             type=float, default=5)
356         parser.add_argument('--interpolate_nan',
357                             help='use astropy option to interpolate nans before filtering' +
358                             ' may not work well because nans are usually located at edge.',
359                             type=bool, default=False)
360         parser.add_argument('--density_constant_factor',
361                             help='value used for constant density parameters ',
362                             type=float,
363                             default=1e18)
364         parser.add_argument('--bias_field_magnitude',
365                             help='magnitude of axial bias magnetic field,' +
366                                  'used to calculate dynamic field.',
367                             type=float,
368                             default=0.02)
369         parser.add_argument('--time_steps',
370                             help='# of time steps', type=int,
371                             default=250)
372         parser.add_argument('--just_magnetic',
373                             help='only calculate magnetic quantities',
374                             action='store_true',
375                             default=False)
376         parser.add_argument('--filter_gaussian',
377                             help="run with averaging on all measured quantities,"
378                             "used to average measurements for uncertainty analysis",
379                             default=False, action='store_true')
380         parser.add_argument('--filter_sigma',
```

```
381                         help='standard deviation of gaussian filter',
382                         type=float,
383                         default=3)
384     parser.add_argument('--filter_truncate',
385                         help='truncate Gaussian filter at this multiple of sigma',
386                         type=float,
387                         default=3)
388     args = parser.parse_args()
389     return args
390
391
392 def normalize_scalar_by_plane(scalar):
393     r"""
394     Return scalar normalized by the maximum in each z plane.
395
396     Notes
397     -----
398     To find the max in each plane take nanmax in x and y direction.
399     Each nanmax reduces the number of dimensions by 1.
400     """
401     scalar = np.array(scalar)
402     maxes = np.nanmax(np.nanmax(scalar, axis=0), axis=0)
403     return scalar / maxes[None, None, :]
404
405
406 def remove_edges_vector(quantity,
407                         x_start=2, x_end=None,
408                         y_start=0, y_end=-2,
409                         z_start=0, z_end=-1):
410     r"""
411     Returns vector field with edges removed.
412
413     Notes
414     -----
415     Usually used to remove nans from interpolation, if parts of the grid were outside
416     of the convex hull of the measurement points.
417     """
418     for index in xrange(len(quantity)):
419         quantity[index] = quantity[index][y_start:y_end, x_start:x_end, z_start:z_end]
420     return quantity
421
422
423 def remove_edges_scalar(quantity,
424                         x_start=2, x_end=None,
425                         y_start=0, y_end=-2,
426                         z_start=0, z_end=-1):
427     r"""
428     Returns vector field with edges removed.
429
430     Notes
431     -----
432     Usually used to remove nans from interpolation, if parts of the grid were outside
433     of the convex hull of the measurement points
434     """
435     quantity = quantity[y_start:y_end, x_start:x_end, z_start:z_end]
436     return quantity
437
```

```
438
439 def boxcar_filter(quantity, width, interpolate_nan=False):
440     r"""
441     Filter scalar field with boxcar of given width.
442     """
443     quantity = np.array(quantity)
444     nan_indexes = np.where(np.isnan(quantity))
445     quantity[nan_indexes] = 0
446     boxcar = np.ones((width, width, width))
447     boxcar /= boxcar.sum()
448     if interpolate_nan:
449         return convolve_fft(quantity, boxcar,
450                             boundary='wrap',
451                             interpolate_nan=True,
452                             fftn=scipy.fftpack.fft,
453                             ifftn=scipy.fftpack.ifft)
454     else:
455         return fftconvolve(quantity, boxcar, mode='same')
456
457
458 def grad_ion_velocity(mach, grad_mach, te, grad_te):
459     r"""
460     Return grad of ion velocity.
461
462     Notes
463     -----
464     The grad of ion velocity depends on the grad of mach number and temperature.
465     """
466     factor = q_e/m_i
467     term1 = np.sqrt(te*factor)*(grad_mach)
468     term2 = mach*factor*grad_te/(2.*np.sqrt(factor*te))
469     return term1 + term2
470
471
472
473
474 if __name__ == '__main__':
475     args = parse_args()
476     main(args)
```

D.7.5 calculate_helicity.py

```
1  #! /Users/vonderlinden2/anaconda/bin/python
2  # -*- coding: utf-8 -*-
3  """
4  Created on Mon Aug  1 18:07:14 2016
5
6  @author: Jens von der Linden
7
8  Calculate gauge dependent and relative helicities.
9  Read field quantities from rectilinear vtk files.
10 Write out helicity to ASCII files.
11 """
12 import argparse
```

```
13  import numpy as np
14  from scipy.constants import mu_0
15  from scipy.constants import elementary_charge as q_e
16  from scipy.constants import proton_mass as m_i
17  from scipy.integrate import trapz
18  from datetime import date
19  from datetime import datetime
20  import visit_writer
21  import os
22  from vector_comparison import read_rectilinear_vtk as rrv
23  from write_to_vtk import structured_3d_vtk as struc_3d
24  import write_to_vtk.helicities as hel
25
26
27  def main(args):
28      r"""
29      Calculate gauge dependent and relative helcities.
30      """
31      just_magnetic = args.just_magnetic
32      now = datetime.now().strftime("%Y-%m-%d-%H-%M")
33      out_dir = '../output/' + args.output_prefix + '/' + now
34      try:
35          os.makedirs(out_dir)
36      except:
37          pass
38      in_dir = args.input_path + args.input_date + '/'
39      in_file = args.input_file_text
40      input_path = in_dir + in_file
41      xlow, xhigh = args.to_cut[:2]
42      ylow, yhigh = args.to_cut[2:4]
43      zlow, zhigh = args.to_cut[4:6]
44
45      if not just_magnetic:
46          fields_to_read = ('B_x', 'B_y', 'B_z',
47                            'B_ref_x', 'B_ref_y', 'B_ref_z',
48                            'A_x', 'A_y', 'A_z',
49                            'A_ref_x', 'A_ref_y', 'A_ref_z',
50                            'u_i_x_plus', 'u_i_y', 'u_i_z',
51                            'u_i_ref_x', 'u_i_ref_y', 'u_i_ref_z',
52                            'w_i_raw_x_plus', 'w_i_raw_y_plus', 'w_i_raw_z_plus',
53                            'w_i_raw_ref_x', 'w_i_raw_ref_y', 'w_i_raw_ref_z',
54                            'w_i_x_plus', 'w_i_y_plus', 'w_i_z_plus',
55                            'w_i_ref_x', 'w_i_ref_y', 'w_i_ref_z', 'n')
56      else:
57          fields_to_read = ('B_x', 'B_y', 'B_z',
58                            'B_ref_x', 'B_ref_y', 'B_ref_z',
59                            'A_x', 'A_y', 'A_z',
60                            'A_ref_x', 'A_ref_y', 'A_ref_z')
61
62
63      for time_point in xrange(args.time_steps):
64          print time_point
65          input_file = (input_path +
66                        str(time_point).zfill(4) +
67                        '.vtk')
68          fields = {}
69          for field in fields_to_read:
```

```
70              read = rrv.read_scalar(input_file, field)
71              fields[field] = read[1][ylow:yhigh, xlow:xhigh, zlow:zhigh]
72          mesh = read[0]
73          mesh[0] = mesh[0][ylow:yhigh, xlow:xhigh, zlow:zhigh]
74          mesh[1] = mesh[1][ylow:yhigh, xlow:xhigh, zlow:zhigh]
75          mesh[2] = mesh[2][ylow:yhigh, xlow:xhigh, zlow:zhigh]
76
77          dx = mesh[0][0, 1, 0] - mesh[0][0, 0, 0]
78          dy = mesh[1][1, 0, 0] - mesh[1][0, 0, 0]
79          dz = mesh[2][0, 0, 1] - mesh[2][0, 0, 0]
80
81          b_field = (fields['B_x'],
82                     fields['B_y'],
83                     fields['B_z'])
84          b_field_ref = (fields['B_ref_x'],
85                         fields['B_ref_y'],
86                         fields['B_ref_z'])
87          vector_potential = (fields['A_x'],
88                              fields['A_y'],
89                              fields['A_z'])
90          vector_potential_ref = (fields['A_ref_x'],
91                                  fields['A_ref_y'],
92                                  fields['A_ref_z'])
93
94          if not just_magnetic:
95              ion_velocity = (fields['u_i_x_plus'],
96                              fields['u_i_y'],
97                              fields['u_i_z'])
98              ion_velocity_ref = (fields['u_i_ref_x'],
99                                  fields['u_i_ref_y'],
100                                 fields['u_i_ref_z'])
101             ion_vorticity = (fields['w_i_x_plus'],
102                              fields['w_i_y_plus'],
103                              fields['w_i_z_plus'])
104             ion_vorticity_raw = (fields['w_i_raw_x_plus'],
105                                  fields['w_i_raw_y_plus'],
106                                  fields['w_i_raw_z_plus'])
107             ion_vorticity_ref = (fields['w_i_ref_x'],
108                                  fields['w_i_ref_y'],
109                                  fields['w_i_ref_z'])
110             ion_vorticity_raw_ref =(fields['w_i_raw_ref_x'],
111                                     fields['w_i_raw_ref_y'],
112                                     fields['w_i_raw_ref_z'])
113             n = fields['n']
114
115             kin_helicity = hel.kinetic_helicity(ion_velocity,
116                                                 ion_vorticity,
117                                                 dx, dy, dz)
118             kin_helicity_raw_vorticity = hel.kinetic_helicity(ion_velocity,
119                                                               ion_vorticity_raw,
120                                                               dx, dy, dz)
121             cross_helicity = hel.cross_helicity(ion_velocity,
122                                                 b_field,
123                                                 dx, dy, dz)
124             rel_kin_helicity = hel.rel_kin_helicity(ion_velocity,
125                                                     ion_velocity_ref,
126                                                     ion_vorticity,
```

```
                                        ion_vorticity_ref,
                                        dx, dy, dz)
    rel_cross_helicity = hel.rel_cross_helicity(ion_velocity,
                                            ion_velocity_ref,
                                            b_field,
                                            b_field_ref,
                                            dx, dy, dz)
    rel_kin_helicity_raw_vorticity = hel.rel_kin_helicity(ion_velocity,
                                                    ion_velocity_ref,
                                                    ion_vorticity_raw,
                                                    ion_vorticity_raw_ref,
                                                    dx, dy, dz)
    rel_cross_helicity_raw_vorticity = hel.rel_cross_helicity(ion_velocity,
                                                        ion_velocity_ref,
                                                        b_field,
                                                        b_field_ref,
                                                        dx, dy, dz)

    ## helicities with density dependence
    ##
    kin_helicity_n = hel.kinetic_helicity(ion_velocity,
                                        ion_vorticity,
                                        dx, dy, dz, density=n)
    kin_helicity_raw_vorticity_n = hel.kinetic_helicity(ion_velocity,
                                                    ion_vorticity_raw,
                                                    dx, dy, dz,
                                                    density=n)
    cross_helicity_n = hel.cross_helicity(ion_velocity,
                                        b_field,
                                        dx, dy, dz,
                                        density=n)
    rel_kin_helicity_n = hel.rel_kin_helicity(ion_velocity,
                                            ion_velocity_ref,
                                            ion_vorticity,
                                            ion_vorticity_ref,
                                            dx, dy, dz, density=n)
    rel_cross_helicity_n = hel.rel_cross_helicity(ion_velocity,
                                                ion_velocity_ref,
                                                b_field,
                                                b_field_ref,
                                                dx, dy, dz, density=n)
    rel_kin_helicity_raw_vorticity_n = hel.rel_kin_helicity(ion_velocity,
                                                        ion_velocity_ref,
                                                        ion_vorticity_raw,
                                                        ion_vorticity_raw_ref,
                                                        dx, dy, dz, density=n)
    rel_cross_helicity_raw_vorticity_n = hel.rel_cross_helicity(ion_velocity,
                                                            ion_velocity_ref,
                                                            b_field,
                                                            b_field_ref,
                                                            dx, dy, dz,
                                                            density=n)
    mag_helicity_n = hel.magnetic_helicity(vector_potential,
                                        b_field,
                                        dx, dy, dz,
                                        density=n)

```

```
184             rel_mag_helicity_n = hel.rel_mag_helicity(vector_potential,
185                                                       vector_potential_ref,
186                                                       b_field,
187                                                       b_field_ref,
188                                                       dx, dy, dz,
189                                                       density=n)
190
191
192
193         mag_helicity = hel.magnetic_helicity(vector_potential,
194                                              b_field,
195                                              dx, dy, dz)
196
197         rel_mag_helicity = hel.rel_mag_helicity(vector_potential,
198                                                 vector_potential_ref,
199                                                 b_field,
200                                                 b_field_ref,
201                                                 dx, dy, dz)
202         if not just_magnetic:
203             with open(out_dir +
204                       "/kinetic_helicity.txt", "a") as myfile:
205                 myfile.write(str(kin_helicity) + '\n')
206             with open(out_dir +
207                       "/kinetic_helicity_raw_vorticity.txt",
208                       "a") as myfile:
209                 myfile.write(str(kin_helicity_raw_vorticity) + '\n')
210             with open(out_dir +
211                       "/cross_helicity.txt", "a") as myfile:
212                 myfile.write(str(cross_helicity) + '\n')
213             with open(out_dir +
214                       "/relative_kinetic_helicity.txt", "a") as myfile:
215                 myfile.write(str(rel_kin_helicity) + '\n')
216             with open(out_dir +
217                       "/relative_cross_helicity.txt", "a") as myfile:
218                 myfile.write(str(rel_cross_helicity) + '\n')
219             with open(out_dir +
220                       "/relative_kinetic_helicity_raw_vorticity.txt",
221                       "a") as myfile:
222                 myfile.write(str(rel_kin_helicity_raw_vorticity) + '\n')
223             with open(out_dir +
224                       "/relative_cross_helicity_raw_vorticity.txt",
225                       "a") as myfile:
226                 myfile.write(str(rel_cross_helicity_raw_vorticity) + '\n')
227
228             ## with density dependence
229             ##
230             with open(out_dir +
231                       "/kinetic_helicity_n_dependence.txt", "a") as myfile:
232                 myfile.write(str(kin_helicity_n) + '\n')
233             with open(out_dir +
234                       "/kinetic_helicity_raw_vorticity_n_dependence.txt",
235                       "a") as myfile:
236                 myfile.write(str(kin_helicity_raw_vorticity_n) + '\n')
237             with open(out_dir +
238                       "/cross_helicity_n_dependence.txt", "a") as myfile:
239                 myfile.write(str(cross_helicity_n) + '\n')
240             with open(out_dir +
```

```
241                     "/relative_kinetic_helicity_n_dependence.txt", "a") as myfile:
242                 myfile.write(str(rel_kin_helicity_n) + '\n')
243             with open(out_dir +
244                     "/relative_cross_helicity_n_dependence.txt", "a") as myfile:
245                 myfile.write(str(rel_cross_helicity_n) + '\n')
246             with open(out_dir +
247                     "/relative_kinetic_helicity_raw_vorticity_n_dependence.txt",
248                     "a") as myfile:
249                 myfile.write(str(rel_kin_helicity_raw_vorticity_n) + '\n')
250             with open(out_dir +
251                     "/relative_cross_helicity_raw_vorticity_n_dependence.txt",
252                     "a") as myfile:
253                 myfile.write(str(rel_cross_helicity_raw_vorticity_n) + '\n')
254             with open(out_dir +
255                     "/magnetic_helicity_n_dependence.txt", "a") as myfile:
256                 myfile.write(str(mag_helicity_n) + '\n')
257             with open(out_dir +
258                     "/relative_magnetic_helicity_n_dependence.txt", "a") as myfile:
259                 myfile.write(str(rel_mag_helicity_n) + '\n')
260
261
262         with open(out_dir +
263                 "/magnetic_helicity.txt", "a") as myfile:
264             myfile.write(str(mag_helicity) + '\n')
265         with open(out_dir +
266                 "/relative_magnetic_helicity.txt", "a") as myfile:
267             myfile.write(str(rel_mag_helicity) + '\n')
268
269 def parse_args():
270     r"""
271     """
272     parser = argparse.ArgumentParser(description='Calculate helicity')
273     parser.add_argument('--input_path',
274                         help='path to input files',
275                         default='../output/canonical_quantities/')
276     parser.add_argument('--input_date',
277                         help='time stamp of input files',
278                         default='2017-04-04-16-33')
279     parser.add_argument('--input_file_text',
280                         help='input file name',
281                         default='canonical_quantities')
282     parser.add_argument('--output_prefix',
283                         help='prefix of output files',
284                         default='helicity')
285     parser.add_argument('--time_steps',
286                         help='number of time steps',
287                         type=int,
288                         default=250)
289     parser.add_argument('--to_cut',
290                         help=("number of indicies to cut on"
291                               "each side to remove NaNs e.g."
292                               "xlow xhigh ylow yhigh zlow zhigh"),
293                         nargs=6,
294                         default=['0', 'None', '0', 'None', '0', 'None'])
295     parser.add_argument('--just_magnetic',
296                         help='only calculate magnetic quantities',
297                         action='store_true',
```

```
                        default=False)
    args = parser.parse_args()
    to_cut_list = []
    for to_cut in args.to_cut:
        if to_cut == 'None':
            to_cut_list.append(None)
        else:
            to_cut_list.append(int(to_cut))
    args.to_cut = to_cut_list
    return args


if __name__ == '__main__':
    args = parse_args()
    main(args)
```

D.7.6 vis_canonical_fluxtubes.py

```
#! /home/jensv/anaconda/bin/python
# -*- coding: utf-8 -*-
"""
Created on Fri Aug 19 14:38:10 2016

@author: Jens von der Linden

Plot frames of canonical flux tube animations.
Plot options include ion, electron canonical flux tubes,
temperature and density isosurfaces, current contours
in an x-y plane.
"""

from datetime import datetime as date
import numpy as np
import os
from scipy.constants import elementary_charge, proton_mass
from glob import glob
import sys
visit_path1 = "/home/jensv/visit/visit2_10_3.linux-x86_64/2.10.3/linux-x86_64/lib/site-packages"
visit_path2 = "/home/jensv/visit/visit2_10_3.linux-x86_64/bin/"
sys.path.append(visit_path1)
sys.path.append(visit_path2)
os.environ["PATH"] += os.pathsep + visit_path1
os.environ["PATH"] += os.pathsep + visit_path2
import visit
import argparse

tan = (209, 178, 111, 255)
olive = (110, 117, 14, 255)
dim_grey =(105, 105, 105, 255)
black = (0, 0, 0, 255)
dark_grey = (169, 169, 169, 255)
red = (255, 0, 0, 255)
dark_red = (84, 0, 0, 255)
green = (0, 154, 0, 255)
```

```
navy = (0, 0, 128, 255)
aqua = (0, 255, 255, 255)

def define_expressions(visit):
    r"""
    Define Visit expressions.
    """
    visit.DefineVectorExpression("B", "{B_x, B_y, B_z}")
    visit.DefineVectorExpression("B_norm", "{B_norm_x, B_norm_y, "
                                 "B_norm_z}")
    visit.DefineVectorExpression("B_perp", "{B_x, B_y, 0}")
    visit.DefineVectorExpression("B_para", "{0, 0, B_z}")
    visit.DefineScalarExpression("B_para_scalar", "B_z")

    visit.DefineVectorExpression("Omega_e_times_density", "B*n")

    visit.DefineVectorExpression("A_vacuum", "{A_vacuum_x, A_vacuum_y, 0}")
    visit.DefineVectorExpression("A", "{A_x, A_y, A_z}")

    visit.DefineVectorExpression("J_smooth", "{j_x, j_y, j_z}")
    visit.DefineScalarExpression("J_smooth_mag", "sqrt(j_x^2 + j_y^2 + j_z^2)")
    visit.DefineVectorExpression("J_smooth_perp", "{j_x, j_y, 0}")
    visit.DefineVectorExpression("J_smooth_para", "{0, 0, j_z}")
    visit.DefineVectorExpression("J_smooth_para_mag", "j_z")

    visit.DefineVectorExpression("J_raw", "{j_raw_x, j_raw_y, j_raw_z}")
    visit.DefineScalarExpression("J_raw_mag", "sqrt(j_raw_x^2 +" +
                                              "j_raw_y^2 +" +
                                              "j_raw_z^2)")
    visit.DefineVectorExpression("J_raw_perp", "{j_raw_x, j_raw_y, 0}")
    visit.DefineVectorExpression("J_raw_para", "{0, 0, j_raw_z}")

    visit.DefineScalarExpression("divergence_B", "divergence(B)")
    visit.DefineScalarExpression("divergence_Omega_i_raw_plus",
                                 "divergence(Omega_i_raw_plus)")
    visit.DefineScalarExpression("divergence_Omega_i_plus",
                                 "divergence(Omega_i_plus)")

    visit.DefineVectorExpression("J_raw_filtered_by_Te",
                                 "J_raw * Te_raw_normalized")
    visit.DefineVectorExpression("J_raw_filtered_by_Te^2",
                                 "J_raw * Te_raw_normalized^2")
    visit.DefineVectorExpression("J_raw_filtered_by_Te^3",
                                 "J_raw * Te_raw_normalized^3")
    visit.DefineVectorExpression("J_raw_filtered_by_Te^4",
                                 "J_raw * Te_raw_normalized^4")
    visit.DefineVectorExpression("J_raw_filtered_by_Te_smooth",
                                 "J_raw * Te_smooth_normalized")
    visit.DefineVectorExpression("J_raw_filtered_by_Te_smooth^2",
                                 "J_raw * Te_smooth_normalized^2")
    visit.DefineVectorExpression("J_raw_filtered_by_Te_smooth^3",
                                 "J_raw * Te_smooth_normalized^3")
    visit.DefineVectorExpression("J_raw_filtered_by_Te_smooth^4",
                                 "J_raw * Te_smooth_normalized^4")

    visit.DefineVectorExpression("u_i_plus", "{u_i_x_plus, u_i_y, u_i_z}")
    visit.DefineVectorExpression("u_i_plus_perp", "{dot(u_i_plus, {1, 0, 0}), dot(u_i_plus, "
```

```
94                                  "{0, 1, 0}), 0}")
95          visit.DefineVectorExpression("u_i_plus_para", "{0, 0, dot(u_i_plus, {0, 0, 1})}")
96
97          visit.DefineVectorExpression("omega_i_plus", "{w_i_x_plus, w_i_y_plus, w_i_z_plus}")
98          visit.DefineVectorExpression("omega_i_plus_perp", "{dot(omega_i_plus, {1, 0, 0}),"
99                                                  "dot(omega_i_plus, {0, 1, 0}), 0})")
100         visit.DefineVectorExpression("omega_i_plus_para", "{0, 0, dot(omega_i_plus, "
101                                 "{0, 0, 1})}")
102
103         visit.DefineVectorExpression("omega_i_raw_plus",
104                                 "{w_i_raw_x_plus, w_i_raw_y_plus, w_i_raw_z_plus}")
105         visit.DefineVectorExpression("omega_i_raw_plus_perp", "{dot(omega_i_raw_plus, {1, 0, 0}),"
106                                 "dot(omega_i_raw_plus, {0, 1, 0}), 0})")
107         visit.DefineVectorExpression("omega_i_raw_plus_para", "{0, 0, dot(omega_i_raw_plus, "
108                                 "{0, 0, 1})}")
109
110         visit.DefineVectorExpression("Omega_i_plus",
111                                 str(elementary_charge) + "*B +" + str(proton_mass) +
112                                 "*omega_i_plus")
113         visit.DefineVectorExpression("Omega_i_plus_perp", "{dot(Omega_i_plus, {1, 0, 0}), "
114                                 "dot(Omega_i_plus, {0, 1, 0}), 0}")
115         visit.DefineVectorExpression("Omega_i_plus_para", "{0, 0, dot(Omega_i_plus, "
116                                 "{0, 0, 1})}")
117         visit.DefineScalarExpression("Omega_i_plus_para_scalar",
118                                 "dot(Omega_i_plus, {0, 0, 1})")
119
120         visit.DefineVectorExpression("Omega_i_raw_plus",
121                                 str(elementary_charge) + "*B +" + str(proton_mass) +
122                                 "*omega_i_raw_plus")
123         visit.DefineVectorExpression("Omega_i_raw_plus_perp", "{dot(Omega_i_raw_plus, {1, 0, 0}), "
124                                 "dot(Omega_i_raw_plus, {0, 1, 0}), 0}")
125         visit.DefineVectorExpression("Omega_i_raw_plus_para", "{0, 0, dot(Omega_i_raw_plus, "
126                                 "{0, 0, 1})}")
127         visit.DefineScalarExpression("Omega_i_raw_plus_para_scalar",
128                                 "dot(Omega_i_raw_plus, {0, 0, 1})")
129
130         visit.DefineVectorExpression("Omega_i_plus_density_dependence",
131                                 "n*(%e *B + %e *omega_i_plus) +"
132                                 "cross(gradient(n), %e *A + %e * u_i_plus)"
133                                 % (elementary_charge, proton_mass,
134                                    elementary_charge, proton_mass))
135         visit.DefineVectorExpression("Omega_i_plus_density_dependence_perp",
136                                 "{dot(Omega_i_plus_density_dependence, {1, 0, 0}), "
137                                 "dot(Omega_i_plus_density_dependence, {0, 1, 0}), 0}")
138         visit.DefineVectorExpression("Omega_i_plus_density_dependence_para",
139                                 "{0, 0, dot(Omega_i_plus_density_dependence, "
140                                 "{0, 0, 1})}")
141         visit.DefineScalarExpression("Omega_i_plus_density_dependence_para_scalar",
142                                 "dot(Omega_i_plus_density_dependence, {0, 0, 1})")
143
144         visit.DefineVectorExpression("Omega_i_raw_plus_density_dependence",
145                                 "n*(%e *B + %e *omega_i_raw_plus) +"
146                                 " cross(gradient(n), %e *A + %e * u_i_plus)"
147                                 % (elementary_charge, proton_mass,
148                                    elementary_charge, proton_mass))
149         visit.DefineVectorExpression("Omega_i_raw_plus_density_dependence_perp",
150                                 "{dot(Omega_i_raw_plus_density_dependence, {1, 0, 0}), "
```

```
151                                 "dot(Omega_i_raw_plus_density_dependence, {0, 1, 0}), 0}")
152         visit.DefineVectorExpression("Omega_i_raw_plus_density_dependence_para",
153                                 "{0, 0, dot(Omega_i_raw_plus_density_dependence, "
154                                 "{0, 0, 1})}")
155         visit.DefineScalarExpression("Omega_i_raw_plus_density_dependence_para_scalar",
156                                 "dot(Omega_i_raw_plus_density_dependence, {0, 0, 1})")
157
158         visit.DefineVectorExpression("Omega_i_plus_times_density",
159                                 "n*(%e *B + %e *omega_i_plus)" %
160                                 (elementary_charge, proton_mass))
161         visit.DefineVectorExpression("Omega_i_plus_times_density_perp",
162                                 "{dot(Omega_i_plus_times_density, {1, 0, 0}), "
163                                 "dot(Omega_i_plus_times_density, {0, 1, 0}), 0}")
164         visit.DefineVectorExpression("Omega_i_plus_times_density_para",
165                                 "{0, 0, dot(Omega_i_plus_density_dependence, "
166                                 "{0, 0, 1})}")
167         visit.DefineScalarExpression("Omega_i_plus_times_density_para_scalar",
168                                 "dot(Omega_i_plus_times_density, {0, 0, 1})")
169
170         visit.DefineVectorExpression("Omega_i_raw_plus_times_density",
171                                 "n*(%e *B + %e *omega_i_raw_plus)" %
172                                 (elementary_charge, proton_mass))
173         visit.DefineVectorExpression("Omega_i_raw_plus_times_density_perp",
174                                 "{dot(Omega_i_raw_plus_density_dependence, {1, 0, 0}), "
175                                 "dot(Omega_i_raw_plus_times_density, {0, 1, 0}), 0}")
176         visit.DefineVectorExpression("Omega_i_raw_plus_times_density_para",
177                                 "{0, 0, dot(Omega_i_raw_plus_times_density, "
178                                 "{0, 0, 1})}")
179         visit.DefineScalarExpression("Omega_i_raw_plus_times_density_para_scalar",
180                                 "dot(Omega_i_raw_plus_times_density, {0, 0, 1})")
181
182
183         ## Omega_e density dependence
184         ##
185         visit.DefineVectorExpression("Omega_e_density_dependence",
186                                 "n*B + cross(gradient(n), A)")
187         visit.DefineVectorExpression("Omega_e_density_dependence_perp",
188                                 "{dot(Omega_e_density_dependence, {1, 0, 0}), "
189                                 "dot(Omega_e_density_dependence, {0, 1, 0}), 0}")
190         visit.DefineVectorExpression("Omega_e_density_dependence_para",
191                                 "{0, 0, dot(Omega_e_plus_density_dependence, "
192                                 "{0, 0, 1})}")
193         visit.DefineScalarExpression("Omega_e_density_dependence_para_scalar",
194                                 "dot(Omega_e_density_dependence, {0, 0, 1})")
195
196
197
198         ## u_i_x(t) = u_i_y(t MINUS tau*0.25)
199         ##
200         visit.DefineVectorExpression("u_i_minus",
201                                 "{u_i_x_minus, u_i_y, u_i_z}")
202         visit.DefineVectorExpression("u_i_minus_perp",
203                                 "{dot(u_i_minus, {1, 0, 0}), dot(u_i_minus, "
204                                 "{0, 1, 0}), 0}")
205         visit.DefineVectorExpression("u_i_minus_para", "{0, 0, dot(u_i_minus, {0, 0, 1})}")
206         visit.DefineScalarExpression("u_i_minus_para_scalar", "dot(u_i_minus, {0, 0, 1})")
207
```

```
208     visit.DefineVectorExpression("omega_i_minus", "{w_i_minus_x, w_i_minus_y, w_i_minus_z}")
209     visit.DefineVectorExpression("omega_i_minus_perp", "{dot(omega_i_minus, {1, 0, 0}),"
210                                                        "dot(omega_i_minus, {0, 1, 0}), 0})")
211     visit.DefineVectorExpression("omega_i_minus_para", "{0, 0, dot(omega_i_minus, "
212                                  "{0, 0, 1})}")
213     visit.DefineScalarExpression("omega_i_minus_para_scalar", "dot(omega_i_minus, "
214                                  "{0, 0, 1})")
215
216     visit.DefineVectorExpression("omega_i_raw_minus",
217                                  "{w_i_raw_x_minus, w_i_raw_y_minus, w_i_raw_z_minus}")
218     visit.DefineVectorExpression("omega_i_minus_raw_perp",
219                                  "{dot(omega_i_raw_minus, {1, 0, 0}),"
220                                  "dot(omega_i_raw_minus, {0, 1, 0}), 0})")
221     visit.DefineVectorExpression("omega_i_minus_raw_para",
222                                  "{0, 0, dot(omega_i_raw_minus, "
223                                  "{0, 0, 1})}")
224     visit.DefineScalarExpression("omega_i_raw_minus_para_scalar",
225                                  "dot(omega_i_raw_minus, "
226                                  "{0, 0, 1})")
227
228     visit.DefineVectorExpression("Omega_i_minus", str(elementary_charge) +
229                                  "*B +" + str(proton_mass) +
230                                  "*omega_i_minus")
231     visit.DefineVectorExpression("Omega_i_minus_perp", "{dot(Omega_i_minus, {1, 0, 0}), "
232                                  "dot(Omega_i_minus, {0, 1, 0}), 0}")
233     visit.DefineVectorExpression("Omega_i_minus_para", "{0, 0, dot(Omega_i_minus, "
234                                  "{0, 0, 1})}")
235     visit.DefineScalarExpression("Omega_i_minus_para_scalar",
236                                  "dot(Omega_i_minus, {0, 0, 1})")
237
238     visit.DefineVectorExpression("Omega_i_raw_minus",
239                                  str(elementary_charge) + "*B +" + str(proton_mass) +
240                                  "*omega_i_raw_minus")
241     visit.DefineVectorExpression("Omega_i_raw_minus_perp", "{dot(Omega_i_raw_minus, {1, 0, 0}), "
242                                  "dot(Omega_i_raw_minus, {0, 1, 0}), 0}")
243     visit.DefineVectorExpression("Omega_i_raw_minus_para", "{0, 0, dot(Omega_i_raw_minus, "
244                                  "{0, 0, 1})}")
245     visit.DefineScalarExpression("Omega_i_raw_minus_para_scalar",
246                                  "dot(Omega_i_raw_minus, {0, 0, 1})")
247
248     visit.DefineVectorExpression("Omega_i_minus_density_dependence",
249                                  "n*(%e *B + %e *omega_i_minus) +"
250                                  " cross(gradient(n), %e *A + %e * u_i_minus)" %
251                                  (elementary_charge, proton_mass,
252                                   elementary_charge, proton_mass))
253     visit.DefineVectorExpression("Omega_i_minus_density_dependence_perp",
254                                  "{dot(Omega_i_minus_density_dependence, {1, 0, 0}), "
255                                  "dot(Omega_i_minus_density_dependence, {0, 1, 0}), 0}")
256     visit.DefineVectorExpression("Omega_i_minus_density_dependence_para",
257                                  "{0, 0, dot(Omega_i_minus_density_dependence, "
258                                  "{0, 0, 1})}")
259     visit.DefineScalarExpression("Omega_i_minus_density_dependence_para_scalar",
260                                  "dot(Omega_i_minus_density_dependence, {0, 0, 1})")
261
262     visit.DefineVectorExpression("Omega_i_raw_minus_density_dependence",
263                                  "n*(%e *B + %e *omega_i_raw_minus) +"
264                                  " cross(gradient(n), %e *A + %e * u_i_minus)" %
```

```
265                                 (elementary_charge, proton_mass,
266                                  elementary_charge, proton_mass))
267         visit.DefineVectorExpression("Omega_i_raw_minus_density_dependence_perp",
268                                 "{dot(Omega_i_raw_minus_density_dependence, {1, 0, 0}), "
269                                 "dot(Omega_i_raw_minus_density_dependence, {0, 1, 0}), 0}")
270         visit.DefineVectorExpression("Omega_i_raw_minus_density_dependence_para",
271                                 "{0, 0, dot(Omega_i_raw_minus_density_dependence, "
272                                 "{0, 0, 1})}")
273         visit.DefineScalarExpression("Omega_i_raw_minus_density_dependence_para_scalar",
274                                 "dot(Omega_i_raw_minus_density_dependence, {0, 0, 1})")
275
276         visit.DefineVectorExpression("Omega_i_minus_times_density",
277                                 "n*(%e *B + %e *omega_i_minus)" %
278                                 (elementary_charge, proton_mass))
279         visit.DefineVectorExpression("Omega_i_minus_times_density_perp",
280                                 "{dot(Omega_i_plus_times_density, {1, 0, 0}), "
281                                 "dot(Omega_i_minus_times_density, {0, 1, 0}), 0}")
282         visit.DefineVectorExpression("Omega_i_minus_times_density_para",
283                                 "{0, 0, dot(Omega_i_plus_density_dependence, "
284                                 "{0, 0, 1})}")
285         visit.DefineScalarExpression("Omega_i_minus_times_density_para_scalar",
286                                 "dot(Omega_i_minus_times_density, {0, 0, 1})")
287
288         visit.DefineVectorExpression("Omega_i_raw_minus_times_density",
289                                 "n*(%e *B + %e *omega_i_raw_minus)" %
290                                 (elementary_charge, proton_mass))
291         visit.DefineVectorExpression("Omega_i_raw_minus_times_density_perp",
292                                 "{dot(Omega_i_raw_minus_density_dependence, {1, 0, 0}), "
293                                 "dot(Omega_i_raw_minus_times_density, {0, 1, 0}), 0}")
294         visit.DefineVectorExpression("Omega_i_raw_minus_times_density_para",
295                                 "{0, 0, dot(Omega_i_raw_minus_times_density, "
296                                 "{0, 0, 1})}")
297         visit.DefineScalarExpression("Omega_i_raw_minus_times_density_para_scalar",
298                                 "dot(Omega_i_raw_minus_times_density, {0, 0, 1})")
299
300         ## Canonical momentum fields
301         visit.DefineVectorExpression("P_i", "%e*A + %e*u_i_plus" %
302                                 (elementary_charge, proton_mass))
303         visit.DefineVectorExpression("P_i_times_density", "n*P_i")
304
305         ## Reference fields
306         visit.DefineVectorExpression("B_ref", "{B_ref_x, B_ref_y, B_ref_z}")
307         visit.DefineVectorExpression("A_ref", "{A_ref_x, A_ref_y, A_ref_z}")
308         visit.DefineVectorExpression("u_i_ref",
309                                 "{u_i_ref_x, u_i_ref_y, u_i_ref_z}")
310         visit.DefineVectorExpression("omega_i_ref",
311                                 "{omega_i_ref_x, omega_i_ref_y, omega_i_ref_z}")
312         #visit.DefineVectorExpression("u_i_ref_raw_vort",
313         #                           "{u_i_raw_ref_x, u_i_raw_ref_y, u_i_raw_ref_z}")
314         visit.DefineVectorExpression("omega_i_ref_raw",
315                                 "{w_i_raw_ref_x, w_i_raw_ref_y, w_i_raw_ref_z}")
316         visit.DefineVectorExpression("P_i_ref", "%e*A_ref + %e*u_i_ref" %
317                                 (elementary_charge, proton_mass))
318         visit.DefineVectorExpression("P_i_ref_times_density", "n*P_i_ref")
319         visit.DefineVectorExpression("P_i_ref_raw_vort", "%e*A_ref + %e*u_i_ref_raw_vort" %
320                                 (elementary_charge, proton_mass))
321         visit.DefineVectorExpression("P_i_ref_raw_vort_times_density", "n*P_i_ref_raw_vort")
```

```
322     visit.DefineVectorExpression("Omega_i_ref", "%e*B_ref + %e*omega_i_ref_raw" %
323                                  (elementary_charge, proton_mass))
324     visit.DefineVectorExpression("Omega_i_ref_times_density", "n*Omega_i_ref")
325     visit.DefineVectorExpression("Omega_i_ref_density_dependence",
326                                  "n*(%e *B_ref + %e *omega_i_ref) +"
327                                  "cross(gradient(n), %e *A_ref + %e * u_i_ref)"
328                                  % (elementary_charge,proton_mass,
329                                     elementary_charge, proton_mass))
330     visit.DefineVectorExpression("Omega_i_ref_raw_vort",
331                                  "%e*B_ref + %e*omega_i_ref_raw" %
332                                  (elementary_charge, proton_mass))
333     visit.DefineVectorExpression("Omega_i_ref_raw_vort_time_density",
334                                  "n*Omega_i_ref_raw_vort")
335     visit.DefineVectorExpression("Omega_i_ref_raw_vort_density_dependence",
336                                  "n*(%e *B_ref + %e *omega_i_ref_raw) +"
337                                  "cross(gradient(n), %e *A_ref + %e * u_i_ref_raw_vort)"
338                                  % (elementary_charge,proton_mass,
339                                     elementary_charge, proton_mass))
340
341
342     ## Relative fields
343     visit.DefineVectorExpression("B_rel", "B + B_ref")
344     visit.DefineVectorExpression("B_rel_minus", "B - B_ref")
345     visit.DefineVectorExpression("A_rel", "A - A_ref")
346     visit.DefineVectorExpression("u_i_rel", "u_i_plus - u_i_ref")
347     visit.DefineVectorExpression("u_i_rel_plus", "u_i_plus + u_i_ref")
348     #visit.DefineVectorExpression("u_i_rel_raw_vort", "u_i_plus - u_i_ref_raw_vort")
349     #visit.DefineVectorExpression("u_i_rel_plus_raw_vort", "u_i_plus + u_i_ref_raw_vort")
350     visit.DefineVectorExpression("omega_i_rel", "omega_i_plus + omega_i_ref")
351     visit.DefineVectorExpression("omega_i_rel_raw", "omega_i_raw_plus + omega_i_ref_raw")
352     visit.DefineVectorExpression("P_i_rel", "P_i - P_i_ref")
353     visit.DefineVectorExpression("P_i_rel_times_density",
354                                  "P_i_times_density - P_i_ref_times_density")
355     visit.DefineVectorExpression("P_i_rel_raw_vort", "P_i - P_i_ref_raw_vort")
356     visit.DefineVectorExpression("P_i_rel_raw_vort_times_density",
357                                  "P_i_rel_raw_vort_times_density - P_i_ref_raw_vort_times_density")
358     visit.DefineVectorExpression("Omega_i_rel", "Omega_i_plus + Omega_i_ref")
359     visit.DefineVectorExpression("Omega_i_rel_times_density",
360                                  "Omega_i_plus_times_density + Omega_i_ref_times_density")
361     visit.DefineVectorExpression("Omega_i_rel_density_dependence",
362                                  "Omega_i_plus_density_dependence + Omega_i_ref_density_dependence")
363     visit.DefineVectorExpression("Omega_i_raw_rel",
364                                  "Omega_i_raw_plus + Omega_i_ref_raw_vort")
365     visit.DefineVectorExpression("Omega_i_raw_rel_times_density",
366                                  "Omega_i_raw_plus_times_density +"
367                                  "Omega_i_ref_raw_vort_times_density")
368     visit.DefineVectorExpression("Omega_i_ref_raw_density_dependence",
369                                  "Omega_i_raw_plus_density_dependence +"
370                                  "Omega_i_ref_raw_vort_density_dependence")
371
372     ## Dynamic fields
373     visit.DefineVectorExpression("B_dynamic", "{B_dynamic_x, B_dynamic_y, B_dynamic_z}")
374     visit.DefineVectorExpression("A_dynamic", "{A_dynamic_x, A_dynamic_y, A_dynamic_z}")
375     visit.DefineVectorExpression("B_dynamic_ref", "{B_dynamic_ref_x,"
376                                                   "B_dynamic_ref_y, B_dynamic_ref_z}")
377     visit.DefineVectorExpression("A_dynamic_ref", "{A_dynamic_ref_x,"
378                                                   "A_dynamic_ref_y, A_dynamic_ref_z}")
```

```
379
380     ## Helicity density
381     visit.DefineScalarExpression("mag_helicity_density", "dot(A, B)")
382     visit.DefineScalarExpression("mag_ref_helicity_density", "dot(A_ref, B_ref)")
383     visit.DefineScalarExpression("mag_rel_helicity_density", "dot(A-A_ref, B+B_ref)")
384     visit.DefineScalarExpression("mag_dynamic_helicity_density", "dot(A_dynamic, B_dynamic)")
385     visit.DefineScalarExpression("mag_dynamic_ref_helicity_density",
386                                  "dot(A_dynamic_ref, B_dynamic_ref)")
387     visit.DefineScalarExpression("mag_dynamic_rel_helicity_density",
388                                  "dot(A_dynamic - A_dynamic_ref, B_dynamic + B_dynamic_ref)")
389     visit.DefineScalarExpression("kin_helicity_density",
390                                  "dot(u_i_plus, omega_i_raw_plus)")
391     visit.DefineScalarExpression("kin_ref_helicity_density",
392                                  "dot(u_i_ref_raw, omega_i_ref_raw)")
393     visit.DefineScalarExpression("kin_rel_helicity_density",
394                                  "dot(u_i_plus - u_i_ref_raw, omega_i_raw_plus + omega_i_ref_raw)")
395     visit.DefineScalarExpression("cross_helicity_density",
396                                  "2.*dot(B, u_i_plus)")
397     visit.DefineScalarExpression("cross_ref_helicity_density",
398                                  "2.*dot(B_ref, u_i_ref)")
399     visit.DefineScalarExpression("cross_rel_helicity_density",
400                                  "(dot(u_i_plus - u_i_ref, B + B_ref)"
401                                  "+ dot(u_i_plus + u_i_ref, B - B_ref))")
402
403
404 def normalize_scalar(visit, scalar_name,
405                      normalized_scalar_name):
406     r"""
407     Determine max of scalar.
408     """
409     visit.AddPlot("Pseudocolor", scalar_name)
410     visit.DrawPlots()
411     visit.Query("Max")
412     max = visit.GetQueryOutputValue()
413     visit.DefineScalarExpression(normalized_scalar_name,
414                                  "%s - %g" % (scalar_name, max))
415     visit.DeleteActivePlots()
416
417
418 def launch_points_inner_outer(center, plane=0.249,
419                               radius_inner=0.001, radius_outer=0.005,
420                               num_inner=80, num_outer=60,
421                               return_cut_point=False):
422     r"""
423     Calculate points on a circle outline for a given center point.
424     """
425     thetas = circle_with_cut_thetas(num_outer)
426     points_outer = launch_points(center, thetas, radius=radius_outer,
427                                  plane=plane)
428
429     thetas = full_circle_thetas(num_inner)
430     points_inner = launch_points(center, thetas, radius=radius_inner,
431                                  plane=plane)
432
433     cut_point_x = points_outer[(num_outer-1)*3]
434     cut_point_y = points_outer[(num_outer-1)*3+1]
435     cut_point = [cut_point_x, cut_point_y]
```

```
436     if return_cut_point:
437         return points_outer, points_inner, cut_point
438     else:
439         return points_outer, points_inner
440
441
442 def full_circle_thetas(num_points):
443     r"""
444     Return a linear space of angles.
445     """
446     thetas = np.linspace(0, 2.*np.pi, num_points)
447     return thetas
448
449
450 def circle_with_cut_thetas(num_points):
451     r"""
452     Return a linear space of angles with a cut from 3/4pi to 5/4pi.
453     """
454     thetas = np.linspace(0, 3./4.*np.pi, num_points)
455     thetas = np.concatenate((thetas, np.linspace(5./4.*np.pi, 2.*np.pi,
456                                                  num_points)))
457     return thetas
458
459
460 def launch_points(center, thetas, radius=0.003,
461                   plane=0.249):
462     r"""
463     Return launch points for field lines.
464     """
465     x_points = radius * np.cos(thetas) + center[0]
466     y_points = radius * np.sin(thetas) + center[1]
467     z_points = plane * np.ones(x_points.size)
468     points = np.empty((x_points.size + y_points.size + z_points.size))
469     points[0::3] = x_points
470     points[1::3] = y_points
471     points[2::3] = z_points
472     points = tuple(points)
473     return points
474
475
476 def setup_scalar_isosurface(visit, quantity,
477                             colortable="PuRd", max_val=1., min_val=0.9):
478     r"""
479     Setup iso_surface. Works best if quantity is plane normalized.
480     """
481     visit.AddPlot("Pseudocolor", quantity, 1, 0)
482     PseudocolorAtts = visit.PseudocolorAttributes()
483     PseudocolorAtts.colorTableName = colortable
484     PseudocolorAtts.opacityType = PseudocolorAtts.Constant
485     PseudocolorAtts.opacity = 0.25
486     PseudocolorAtts.smoothingLevel = 0
487     PseudocolorAtts.legendFlag = 0
488     visit.SetPlotOptions(PseudocolorAtts)
489
490     visit.AddOperator("Isosurface", 0)
491     IsosurfaceAtts = visit.IsosurfaceAttributes()
492
```

```
493         IsosurfaceAtts.contourNLevels = 5
494         IsosurfaceAtts.contourValue = ()
495         IsosurfaceAtts.contourPercent = ()
496         IsosurfaceAtts.contourMethod = IsosurfaceAtts.Level
497         IsosurfaceAtts.minFlag = 1
498         IsosurfaceAtts.min = min_val
499         IsosurfaceAtts.maxFlag = 1
500         IsosurfaceAtts.max = max_val
501         visit.SetOperatorOptions(IsosurfaceAtts, 0)
502         return PseudocolorAtts, IsosurfaceAtts
503
504     def setup_current_pseudocolor(visit, current_to_use,
505                                   colortable="PRGn_Stepped", max_val=1e6,
506                                   min_val=-1e6, invert=True, horizontal=True):
507         r"""
508         Setup pseudocolor current plot.
509         """
510         visit.AddPlot("Pseudocolor", current_to_use, 1, 0)
511         PseudocolorAtts = visit.PseudocolorAttributes()
512         PseudocolorAtts.scaling = PseudocolorAtts.Linear
513         PseudocolorAtts.limitsMode = PseudocolorAtts.OriginalData
514         PseudocolorAtts.colorTableName = colortable
515         PseudocolorAtts.invertColorTable = invert
516
517         if max_val:
518             PseudocolorAtts.maxFlag = 1
519             PseudocolorAtts.max = max_val
520         if min_val:
521             PseudocolorAtts.minFlag = 1
522             PseudocolorAtts.min = min_val
523
524         visit.SetPlotOptions(PseudocolorAtts)
525         visit.AddOperator("Slice", 0)
526         SliceAtts = visit.SliceAttributes()
527         SliceAtts.originType = SliceAtts.Intercept
528         SliceAtts.originIntercept = 0.249
529         SliceAtts.axisType = SliceAtts.ZAxis
530         SliceAtts.project2d = 0
531         visit.SetOperatorOptions(SliceAtts, 0)
532
533         colorbar = visit.GetAnnotationObject('Plot0000')
534         colorbar.SetDrawMinMax(0)
535
536         if horizontal:
537             colorbar.SetOrientation("HorizontalBottom")
538             colorbar.SetFontHeight(0.017)
539             colorbar.SetNumberFormat('%#3.1g')
540             colorbar.SetManagePosition(0)
541             colorbar.SetPosition((0.05, 0.99))
542         return PseudocolorAtts, SliceAtts
543
544
545     def setup_massless_electron_canonical_flux_tubes(visit, points_outer,
546                                                      points_inner):
547         r"""
548         Setup two massless electron canonical flux tubes i.e. magnetic flux tubes.
549         Intended to be inner and outer flux tubes.
```

```
550         """
551         visit.AddPlot("Streamline", "Omega_e_times_density", 1, 0)
552         StreamlineAtts_outer = visit.StreamlineAttributes()
553         StreamlineAtts_outer.sourceType = StreamlineAtts_outer.SpecifiedPointList
554         StreamlineAtts_outer.SetPointList(points_outer)
555         StreamlineAtts_outer.coloringMethod = StreamlineAtts_outer.Solid
556         StreamlineAtts_outer.colorTableName = "Default"
557         StreamlineAtts_outer.singleColor = (255, 0, 0, 255)
558         StreamlineAtts_outer.legendFlag = 0
559         visit.SetPlotOptions(StreamlineAtts_outer)
560 
561         visit.AddPlot("Streamline", "Omega_e_times_density", 1, 0)
562         StreamlineAtts_inner = visit.StreamlineAttributes()
563         StreamlineAtts_inner.sourceType = StreamlineAtts_inner.SpecifiedPointList
564         StreamlineAtts_inner.SetPointList(points_inner)
565         StreamlineAtts_inner.coloringMethod = StreamlineAtts_inner.Solid
566         StreamlineAtts_inner.colorTableName = "Default"
567         StreamlineAtts_inner.singleColor = (190, 64, 0, 255)
568         StreamlineAtts_inner.legendFlag = 0
569         visit.SetPlotOptions(StreamlineAtts_inner)
570 
571         return StreamlineAtts_outer, StreamlineAtts_inner
572 
573 
574     def setup_outer_inner_ion_canonical_flux_tubes(visit, quantity, points_outer,
575                                                    points_inner,
576                                                    outer_color=dark_grey,
577                                                    inner_color=black):
578         r"""
579         Setup two ion canonical flux tubes.
580         Inteteded to be inner and outer flux tubes.
581         """
582         visit.AddPlot("Streamline", quantity, 1, 0)
583         StreamlineAtts_outer = visit.StreamlineAttributes()
584         StreamlineAtts_outer.sourceType = StreamlineAtts_outer.SpecifiedPointList
585         StreamlineAtts_outer.SetPointList(points_outer)
586         StreamlineAtts_outer.coloringMethod = StreamlineAtts_outer.Solid
587         StreamlineAtts_outer.colorTableName = "Default"
588         StreamlineAtts_outer.singleColor = outer_color
589         StreamlineAtts_outer.legendFlag = 0
590         visit.SetPlotOptions(StreamlineAtts_outer)
591 
592         visit.AddPlot("Streamline", quantity, 1, 0)
593         StreamlineAtts_inner = visit.StreamlineAttributes()
594         StreamlineAtts_inner.sourceType = StreamlineAtts_inner.SpecifiedPointList
595         StreamlineAtts_inner.SetPointList(points_inner)
596         StreamlineAtts_inner.coloringMethod = StreamlineAtts_inner.Solid
597         StreamlineAtts_inner.colorTableName = "Default"
598         StreamlineAtts_inner.singleColor = inner_color
599         StreamlineAtts_inner.legendFlag = 0
600         visit.SetPlotOptions(StreamlineAtts_inner)
601 
602         return StreamlineAtts_outer, StreamlineAtts_inner
603 
604 
605     def setup_forward_backward_ion_canonical_flux_tubes(visit, points_foward,
606                                                         points_backward,
```

```
607                                                 forward_color=tan,
608                                                 backward_color=olive):
609     r"""
610     Setup two ion canonical flux tubes, one integrating in the forward
611     direction, one integrating in the backward direction.
612     """
613     visit.AddPlot("Streamline", "Omega_i", 1, 0)
614     StreamlineAtts_forward = visit.StreamlineAttributes()
615     StreamlineAtts_forward.sourceType = StreamlineAtts_forward.SpecifiedPointList
616     StreamlineAtts_forward.SetPointList(points_foward)
617     StreamlineAtts_forward.coloringMethod = StreamlineAtts_forward.Solid
618     StreamlineAtts_forward.colorTableName = "Default"
619     StreamlineAtts_forward.singleColor = forward_color
620     StreamlineAtts_forward.integrationDirection = StreamlineAtts_forward.Forward
621     StreamlineAtts_forward.legendFlag = 0
622     visit.SetPlotOptions(StreamlineAtts_forward)
623
624     visit.AddPlot("Streamline", "Omega_i", 1, 0)
625     StreamlineAtts_backward = visit.StreamlineAttributes()
626     StreamlineAtts_backward.sourceType = StreamlineAtts_backward.SpecifiedPointList
627     StreamlineAtts_backward.SetPointList(points_backward)
628     StreamlineAtts_backward.coloringMethod = StreamlineAtts_backward.Solid
629     StreamlineAtts_backward.colorTableName = "Default"
630     StreamlineAtts_backward.singleColor = backward_color
631     StreamlineAtts_backward.integrationDirection = StreamlineAtts_backward.Backward
632     StreamlineAtts_backward.legendFlag = 0
633     visit.SetPlotOptions(StreamlineAtts_backward)
634
635     return StreamlineAtts_forward, StreamlineAtts_backward
636
637
638 def setup_backward_and_B_stream(visit, name, launch_points,
639                                 B_launch_points, color=green, B_color=red):
640     r"""
641     Setup fiedlines for a magnetic flux tube and for a backward integrated quantity.
642     """
643     visit.AddPlot("Streamline", 'B', 1, 0)
644     StreamlineAtts_B = visit.StreamlineAttributes()
645     StreamlineAtts_B.sourceType = StreamlineAtts_B.SpecifiedPointList
646     StreamlineAtts_B.SetPointList(B_launch_points)
647     StreamlineAtts_B.coloringMethod = StreamlineAtts_B.Solid
648     StreamlineAtts_B.colorTableName = "Default"
649     StreamlineAtts_B.singleColor = B_color
650     StreamlineAtts_B.integrationDirection = StreamlineAtts_B.Forward
651     StreamlineAtts_B.legendFlag = 0
652     visit.SetPlotOptions(StreamlineAtts_B)
653
654
655     visit.AddPlot("Streamline", name, 1, 0)
656     StreamlineAtts_backward = visit.StreamlineAttributes()
657     StreamlineAtts_backward.sourceType = StreamlineAtts_backward.SpecifiedPointList
658     StreamlineAtts_backward.SetPointList(launch_points)
659     StreamlineAtts_backward.coloringMethod = StreamlineAtts_backward.Solid
660     StreamlineAtts_backward.colorTableName = "Default"
661     StreamlineAtts_backward.singleColor = color
662     StreamlineAtts_backward.integrationDirection = StreamlineAtts_backward.Backward
663     StreamlineAtts_backward.legendFlag = 0
```

```
664     visit.SetPlotOptions(StreamlineAtts_backward)
665
666     return StreamlineAtts_B, StreamlineAtts_backward
667
668
669 def setup_field_line(visit, quantity,
670                      launch_point=(0.01, 0.01), launch_z=0.249,
671                      color=dark_red):
672     r"""
673     Setup single field line plot to better see twistedness.
674     """
675     visit.AddPlot("Streamline", quantity, 1, 0)
676     StreamlineAtts_line = visit.StreamlineAttributes()
677     StreamlineAtts_line.sourceType = StreamlineAtts_line.SpecifiedPoint
678     StreamlineAtts_line.pointSource = (launch_point[0], launch_point[1], launch_z)
679     StreamlineAtts_line.coloringMethod = StreamlineAtts_line.Solid
680     StreamlineAtts_line.singleColor = color
681     StreamlineAtts_line.legendFlag = 0
682     StreamlineAtts_line.showSeeds = 0
683     StreamlineAtts_line.lineWidth = 8
684     visit.SetPlotOptions(StreamlineAtts_line)
685     return StreamlineAtts_line
686
687
688 def setup_annotations(visit, time_scale=1, time_offset=0):
689     r"""
690     Setup Annotations: scale tick font size, label font size,
691     hide unecessary text.
692     """
693     AnnotationAtts = visit.AnnotationAttributes()
694     AnnotationAtts.axes3D.autoSetScaling = 0
695     AnnotationAtts.axes3D.xAxis.title.visible = 0
696     AnnotationAtts.axes3D.yAxis.title.visible = 0
697     AnnotationAtts.axes3D.zAxis.title.visible = 0
698     AnnotationAtts.axes3D.xAxis.label.font.scale = 3
699     AnnotationAtts.axes3D.xAxis.label.scaling = -2
700     AnnotationAtts.axes3D.yAxis.label.font.scale = 3
701     AnnotationAtts.axes3D.yAxis.label.scaling = -2
702     AnnotationAtts.axes3D.zAxis.label.font.scale = 3
703     AnnotationAtts.axes3D.zAxis.label.scaling = -2
704     AnnotationAtts.userInfoFlag = 0
705     AnnotationAtts.databaseInfoFlag = 0
706     AnnotationAtts.databaseInfoTimeScale = time_scale
707     AnnotationAtts.databaseInfoTimeOffset = time_offset
708     visit.SetAnnotationAttributes(AnnotationAtts)
709     return AnnotationAtts
710
711
712 def set_default_view(visit):
713     r"""
714     Set default view for viewing fluxtubes.
715     If view needs to be modified it is best to align visit with gui and save
716     parameters from visit.GetView3D().
717     """
718     view = visit.GetView3D()
719     view.SetViewNormal((-0.731293, 0.40847, 0.546227))
720     view.SetFocus((0.00202222, 0.000976744, 0.331997))
```

```
721         view.SetViewUp((0.322268, 0.91274, -0.251095))
722         view.SetViewAngle(30)
723         view.SetParallelScale(0.088383)
724         view.SetNearPlane(-0.176766)
725         view.SetImagePan((0, 0))
726         view.SetImageZoom(1.5)
727         view.SetPerspective(1)
728         view.SetEyeAngle(2)
729         view.SetCenterOfRotationSet(0)
730         view.SetCenterOfRotation((0.00202222, 0.000976744, 0.331997))
731         view.SetAxis3DScaleFlag(0)
732         view.SetAxis3DScales((1, 1, 1))
733         view.SetShear((0, 0, 1))
734         view.SetWindowValid(0)
735         visit.SetView3D(view)
736
737
738     def set_default_lighting(visit):
739         r"""
740         Set lightening to light up contour plot in z=0.249 plane
741         when default view is used.
742         """
743         light0 = visit.LightAttributes()
744         light0.enabledFlag = 1
745         light0.type = light0.Camera
746         light0.direction = (-0.5, 0, -0.5)
747         light0.color = (255, 255, 255, 255)
748         light0.brightness = 1
749         visit.SetLight(0, light0)
750
751
752     def set_default_view_thesis(visit):
753         r"""
754         Set default view for viewing fluxtubes.
755         If view needs to be modified it is best to align visit with gui and save
756         parameters from visit.GetView3D().
757         """
758         View3DAtts = visit.View3DAttributes()
759         View3DAtts.viewNormal = (-0.652048, 0.487146, 0.580966)
760         View3DAtts.focus = (0.00151111, 0.0045, 0.331997)
761         View3DAtts.viewUp = (0.365672, 0.873317, -0.321872)
762         View3DAtts.viewAngle = 30
763         View3DAtts.parallelScale = 0.0883679
764         View3DAtts.nearPlane = -0.176736
765         View3DAtts.farPlane = 0.176736
766         View3DAtts.imagePan = (-0.0429026, -0.00832011)
767         View3DAtts.imageZoom = 0.95
768         View3DAtts.perspective = 1
769         View3DAtts.eyeAngle = 2
770         View3DAtts.centerOfRotationSet = 0
771         View3DAtts.centerOfRotation = (0.00151111, 0.0045, 0.331997)
772         View3DAtts.axis3DScaleFlag = 0
773         View3DAtts.axis3DScales = (1, 1, 1)
774         View3DAtts.shear = (0, 0, 1)
775         View3DAtts.windowValid = 0
776         visit.SetView3D(View3DAtts)
777
```

```
778
779 def set_default_view_lower_angle(visit):
780     r"""
781     Set view with lower angle for viewing fluxtubes.
782     If view needs to be modified it is best to align visit with gui and save
783     parameters from visit.GetView3D().
784     """
785     view = visit.GetView3D()
786     view.setViewNormal((-0.776189, 0.193398, 0.600106))
787     view.setFocus((0.00202222, 0.000976744, 0.331997))
788     view.setViewUp((0.138771, 0.980856, -0.136615))
789     view.setViewAngle(30)
790     view.setParallelScale(0.088383)
791     view.setNearPlane(-0.176766)
792     view.setFarPlane(0.175437)
793     view.setImagePan((0, 0))
794     view.setImageZoom(1)
795     view.setPerspective(1)
796     view.setEyeAngle(2)
797     view.setCenterOfRotationSet(0)
798     view.setCenterOfRotation((0.00202222, 0.000976744, 0.331997))
799     view.setAxis3DScaleFlag(0)
800     view.setAxis3DScales((1, 1, 1))
801     view.setShear((0, 0, 1))
802     view.setWindowValid(0)
803     visit.SetView3D(view)
804
805
806 def set_positive_x_view(visit):
807     r"""
808     Set view along positive x for viewing fluxtubes.
809     If view needs to be modified it is best to align visit with gui and save
810     parameters from visit.GetView3D().
811     """
812     view = visit.GetView3D()
813     view.SetViewNormal((-0.00997631, 0.0600385, 0.0335938))
814     view.SetFocus((0.00202222, -0.00202703, 0.331997))
815     view.SetViewUp((0.0598395, 0.998184, 0.00689852))
816     view.SetViewAngle(30)
817     view.SetParallelScale(0.0877186)
818     view.SetNearPlane(-0.175437)
819     view.SetImagePan((0, 0))
820     view.SetImageZoom(1)
821     view.SetPerspective(1)
822     view.SetEyeAngle(2)
823     view.SetCenterOfRotationSet(0)
824     view.SetCenterOfRotation((0.00202222, -0.00202703, 0.331997))
825     view.SetAxis3DScaleFlag(0)
826     view.SetAxis3DScales((1, 1, 1))
827     view.SetShear((0, 0, 1))
828     view.SetWindowValid(1)
829     visit.SetView3D(view)
830
831
832 def set_postive_z_view(visit):
833     r"""
834     Set view along positive z for viewing fluxtubes.
```

```
835         If view needs to be modified it is best to align visit with gui and save
836         parameters from visit.GetView3D().
837         """
838         view = visit.GetView3D()
839         view.SetViewNormal((0.00944856, 0.0379894, 0.999233))
840         view.SetFocus((0.00202222, -0.00202703, 0.331997))
841         view.SetViewUp((-0.00367716, 0.9999274, 0.0037961))
842         view.SetViewAngle(30)
843         view.SetParallelScale(0.0877186)
844         view.SetNearPlane(-0.175437)
845         view.SetImagePan((0, 0))
846         view.SetImageZoom(2.14359)
847         view.SetPerspective(1)
848         view.SetEyeAngle(2)
849         view.SetCenterOfRotationSet(0)
850         view.SetCenterOfRotation((0.00202222, -0.00202703, 0.331997))
851         view.SetAxis3DScaleFlag(0)
852         view.SetAxis3DScales((1, 1, 1))
853         view.SetShear((0, 0, 1))
854         view.SetWindowValid(1)
855         visit.SetView3D(view)
856
857
858     def set_negative_z_view(visit):
859         r"""
860         Set view along negative z for viewing fluxtubes.
861         If view needs to be modified it is best to align visit with gui and save
862         parameters from visit.GetView3D().
863         """
864         view = visit.GetView3D()
865         view.SetViewNormal((-0.00894299, -0.00985814, 0.999911))
866         view.SetFocus((0.00202222, 0.000976744, 0.331997))
867         view.SetViewUp((0.00367716, 0.999944, 0.00989136))
868         view.SetViewAngle(30)
869         view.SetParallelScale(0.0877186)
870         view.SetNearPlane(-0.175437)
871         view.SetImagePan((0, 0))
872         view.SetImageZoom(2.14359)
873         view.SetPerspective(1)
874         view.SetEyeAngle(2)
875         view.SetCenterOfRotationSet(0)
876         view.SetCenterOfRotation((0.00202222, -0.00202703, 0.331997))
877         view.SetAxis3DScaleFlag(0)
878         view.SetAxis3DScales((1, 1, 1))
879         view.SetShear((0, 0, 1))
880         view.SetWindowValid(1)
881         visit.SetView3D(view)
882
883
884     def determine_j_mag_extrema(database_path, plane_num=0):
885         r"""
886         Determine extrema over time of current across all shots.
887         Can be sued to set min and max values for colormaps.
888         """
889         numpy_archives =  glob(database_path + '*.npz')
890         data = np.load(numpy_archives[0])
891         j_x = data['j_x'][:, :, plane_num]
```

```
892         j_y = data['j_y'][:, :, plane_num]
893         j_z = data['j_z'][:, :, plane_num]
894         j_mag = np.sqrt(j_x**2. + j_y**2. + j_z**2.)
895         j_mag_max = np.nanmax(j_mag)
896         j_mag_min = np.nanmin(j_mag)
897         for archive in numpy_archives[1:]:
898             data = np.load(archive)
899             j_x = data['j_x'][:, :, plane_num]
900             j_y = data['j_y'][:, :, plane_num]
901             j_z = data['j_z'][:, :, plane_num]
902             j_mag = np.sqrt(j_x**2. + j_y**2. + j_z**2.)
903             j_mag_max = np.nanmax(j_mag) if np.nanmax(j_mag) > j_mag_max else j_mag_max
904             j_mag_min = np.nanmin(j_mag) if np.nanmin(j_mag) < j_mag_min else j_mag_min
905         return j_mag_max, j_mag_min
906 
907 
908     def set_save_settings(visit):
909         r"""
910         Set and return save_atts.
911         """
912         save_atts = visit.SaveWindowAttributes()
913         save_atts.format = save_atts.PNG
914         save_atts.height = 1080
915         save_atts.width = 1920
916         save_atts.family = 0
917         visit.SetSaveWindowAttributes(save_atts)
918         return save_atts
919 
920 
921     def setup_slider(visit):
922         r"""
923         Add a time slider with us text label.
924         """
925         slider = visit.CreateAnnotationObject("TimeSlider")
926         slider.SetText("$time us")
927         slider.SetRounded(0)
928         slider.SetVisible(1)
929 
930 
931     def main():
932         r"""
933         Plot frames of canonical flux tube animations.
934         """
935         args = parse_args()
936         database_prefix = args.database_prefix + args.database_date
937         visit.Launch()
938         today = date.now().strftime('%Y-%m-%d-%H-%M')
939         out_dir = '../output/canonical_flux_tubes/' + today
940         try:
941            os.makedirs(out_dir)
942         except:
943             pass
944 
945 
946         if args.interactive_session:
947            visit.OpenDatabase(database_prefix + args.database_postfix + "*.vtk database")
948            define_expressions(visit)
```

```
949             visit.OpenGUI()
950             return
951
952         output_path = out_dir + '/' + args.output_prefix
953         print 'data_path', database_prefix + args.database_postfix
954         visit.OpenDatabase(database_prefix + args.database_postfix + "*.vtk database")
955         define_expressions(visit)
956
957         field_nulls = np.loadtxt(args.field_nulls)
958         field_nulls = np.roll(field_nulls, args.time_shift, axis=0)
959         time_points = np.arange(args.start_time_point, args.end_time_point)
960         #time_points = np.roll(time_points, args.time_shift, axis=0)
961
962         AnnotationAtts = setup_annotations(visit, time_scale=args.time_scale,
963                                            time_offset=0)
964
965         plot_count = 0
966
967         if args.current_plane:
968             PseudocolorAtts, SliceAtts = setup_current_pseudocolor(visit,
969                                                                    args.current_to_use,
970                                                                    max_val=args.current_max,
971                                                                    min_val=args.current_min)
972             plot_count += 1
973         if args.temperature_tubes:
974             setup_scalar_isosurface(visit, "Te_plane_normalized", colortable="PuRd")
975             plot_count += 1
976         if args.density_tubes:
977             setup_scalar_isosurface(visit, "n_plane_normalized", colortable="Greys")
978             plot_count += 1
979
980         if args.stationary_tube:
981             (points_outer, points_inner,
982              cut_point) = launch_points_inner_outer(field_nulls[0],
983                                                    return_cut_point=True)
984         else:
985             (points_outer, points_inner,
986              cut_point) = launch_points_inner_outer(args.stationary_center,
987                                                    return_cut_point=True)
988         if args.ion:
989             (StreamlineAtts_ion_outer,
990              StreamlineAtts_ion_inner) = setup_outer_inner_ion_canonical_flux_tubes(visit,
991                                                                                     args.omega_to_use,
992                                                                                     points_outer,
993                                                                                     points_inner)
994             plot_count += 2
995         if args.electron:
996             stream_line_func = setup_massless_electron_canonical_flux_tubes
997             (StreamlineAtts_electron_outer,
998              StreamlineAtts_electron_inner) = setup_massless_electron_canonical_flux_tubes(visit,
999                                                                                            points_outer,
1000                                                                                           points_inner)
1001            plot_count += 2
1002            if not args.ion:
1003                cut_point[1] += 0.0005
1004                FieldLineAtts = setup_field_line(visit, 'Omega_e_times_density',
1005                                                 launch_point=cut_point)
```

```
1006            plot_count += 1
1007
1008    if args.velocity:
1009        (velocity_stream_1,
1010         velocity_stream_2) = setup_outer_inner_ion_canonical_flux_tubes(visit,
1011                                                                         'u_i_plus',
1012                                                                         points_inner,
1013                                                                         points_outer,
1014                                                                         outer_color=aqua,
1015                                                                         inner_color=navy)
1016        plot_count += 2
1017
1018    if args.view == 'default':
1019        if not args.for_wide_screen:
1020            set_default_view_thesis(visit)
1021        else:
1022            set_default_view(visit)
1023    elif args.view == 'default_lower_angle':
1024        set_default_view_lower_angle(visit)
1025    elif args.view == 'positive_z':
1026        set_postive_z_view(visit)
1027    elif args.view == 'negative_z':
1028        set_negative_z_view(visit)
1029    elif args.view == 'positive_x':
1030        set_positive_x_view(visit)
1031    set_default_lighting(visit)
1032    setup_slider(visit)
1033
1034    if args.double_stream:
1035        stream_launch_point = (field_nulls[args.start_time_point][0] + args.x_offset,
1036                               field_nulls[args.start_time_point][1] + args.y_offset)
1037        setup_field_line(visit, args.double_stream,
1038                         launch_point=(stream_launch_point[0] + 0.001,
1039                                       stream_launch_point[1] + 0.))
1040        setup_field_line(visit, args.double_stream,
1041                         launch_point=(stream_launch_point[0] + 0.,
1042                                       stream_launch_point[1] + 0.001))
1043        setup_field_line(visit, args.double_stream,
1044                         launch_point=(stream_launch_point[0] - 0.001,
1045                                       stream_launch_point[1] + 0.))
1046        setup_field_line(visit, args.double_stream,
1047                         launch_point=(stream_launch_point[0] + 0.,
1048                                       stream_launch_point[1] - 0.001))
1049
1050        setup_field_line(visit, args.double_stream,
1051                         launch_point=(stream_launch_point[0] + 0.005,
1052                                       stream_launch_point[1] + 0.))
1053        setup_field_line(visit, args.double_stream,
1054                         launch_point=(stream_launch_point[0] + 0.,
1055                                       stream_launch_point[1] + 0.005))
1056        setup_field_line(visit, args.double_stream,
1057                         launch_point=(stream_launch_point[0] - 0.005,
1058                                       stream_launch_point[1] + 0.))
1059        setup_field_line(visit, args.double_stream,
1060                         launch_point=(stream_launch_point[0] + 0.,
1061                                       stream_launch_point[1] - 0.005))
1062
```

```
1063            setup_field_line(visit, args.double_stream,
1064                             launch_point=stream_launch_point)
1065
1066            setup_field_line(visit, 'B',
1067                             launch_point=stream_launch_point, color=red)
1068
1069
1070        visit.DrawPlots()
1071        save_atts = set_save_settings(visit)
1072        ending = '.png'
1073        visit.SetTimeSliderState(time_points[0])
1074        if not args.for_wide_screen:
1075            visit.ResizeWindow(1, 960, 1000)
1076        else:
1077            visit.ResizeWindow(1, 1920, 1080)
1078
1079        if args.wait_for_manual_settings:
1080            visit.OpenGUI()
1081            comment = raw_input()
1082
1083
1084        for index, time_point in enumerate(time_points):
1085            print time_point
1086            plot_number = 0
1087            save_atts.fileName = output_path + str(index + args.file_number_offset).zfill(4) + ending
1088            visit.SetSaveWindowAttributes(save_atts)
1089
1090            if args.current_plane:
1091                plot_number += 1
1092            if args.temperature_tubes:
1093                plot_number += 1
1094            if args.density_tubes:
1095                plot_number += 1
1096
1097            (points_outer,
1098             points_inner,
1099             cut_point) = launch_points_inner_outer(field_nulls[index],
1100                                                    return_cut_point=True)
1101            if args.stationary_tube:
1102                (points_outer,
1103                 points_inner,
1104                 cut_point) = launch_points_inner_outer(args.stationary_center,
1105                                                        return_cut_point=True)
1106
1107
1108            if args.ion:
1109                visit.SetActivePlots(plot_number)
1110                StreamlineAtts_ion_outer.SetPointList(points_outer)
1111                visit.SetPlotOptions(StreamlineAtts_ion_outer)
1112                plot_number += 1
1113
1114                visit.SetActivePlots(plot_number)
1115                StreamlineAtts_ion_inner.SetPointList(points_inner)
1116                visit.SetPlotOptions(StreamlineAtts_ion_inner)
1117                plot_number += 1
1118
1119            if args.electron:
```

```
            visit.SetActivePlots(plot_number)
            StreamlineAtts_electron_outer.SetPointList(points_outer)
            visit.SetPlotOptions(StreamlineAtts_electron_outer)
            plot_number += 1

            visit.SetActivePlots(plot_number)
            StreamlineAtts_electron_inner.SetPointList(points_inner)
            visit.SetPlotOptions(StreamlineAtts_electron_inner)
            plot_number +=1

            if not args.ion:
                cut_point[1] += 0.0005
                visit.SetActivePlots(plot_number)
                FieldLineAtts.SetPointSource(cut_point[0],
                                             cut_point[1],
                                             0.249)
                visit.SetPlotOptions(FieldLineAtts)
                plot_number += 1

        if args.velocity:
            visit.SetActivePlots(plot_number)
            velocity_stream_1.SetPointList(points_outer)
            visit.SetPlotOptions(velocity_stream_1)
            plot_number +=1

            visit.SetActivePlots(plot_number)
            velocity_stream_2.SetPointList(points_inner)
            visit.SetPlotOptions(velocity_stream_2)
            plot_number +=1

        visit.SetTimeSliderState(time_point)

        name = visit.SaveWindow()


def parse_args():
    parser = argparse.ArgumentParser(description="Generate time step plots of canonical flux tubes.")
    parser.add_argument('--database_prefix', help='path to visit database i.e. vtk files',
                        default='/home/jensv/rsx/jens_analysis/output/canonical_quantities/')
    parser.add_argument('--database_postfix', help='path to visit database i.e. vtk files',
                        default='/canonical_quantities')
    parser.add_argument('database_date', help='date of data run YYYY-MM-DD-mm-ss')
    parser.add_argument('--output_prefix', help='output_file_prefix',
                        default='canonical_flux_tubes_')
    parser.add_argument('--current_min', help='minimum for current color map', default=-3.5e5)
    parser.add_argument('--current_max', help='maximum for current color map', default=3.5e5)
    parser.add_argument('--start_time_point', help='time point of first output frame', type=int, default=0)
    parser.add_argument('--end_time_point', help='time point of last output frame', type=int, default=250)
    parser.add_argument('--field_nulls', help='path to file listing field_nulls (launching centers)',
                        default='/home/jensv/rsx/jens_analysis/output/field_nulls/2017-05-05/averaged_nulls.txt')
    parser.add_argument('--time_scale', help='time scale of time steps', default=0.068)
    parser.add_argument('--current_plane', help='plot temperature contours',
                        action='store_true', default=False)
    parser.add_argument('--temperature_tubes', help='plot temperature isosurfaces',
                        action='store_true', default=False)
    parser.add_argument('--density_tubes', help='plot density isosurfaces',
                        action='store_true', default=False)
```

```
1177      parser.add_argument('--electron', help='plot canonical electron flux tubes',
1178                          action='store_true', default=False)
1179      parser.add_argument('--ion', help='plot canonical ion flux tubes', action='store_true', default=False)
1180      parser.add_argument('--current',
1181                          help='plot thin current flux tube surrounded by electron / magnetic flux tube',
1182                          action='store_true', default=False)
1183      parser.add_argument('--interactive_session', action='store_true', default=False)
1184      parser.add_argument('--current_to_use', default='j_z')
1185      parser.add_argument('--omega_to_use', default='Omega_i_raw_plus_times_density')
1186      parser.add_argument('--view', help='pre-configured_views: default, default_lower_angle, positive_z, negative_z, positive_x', default
              ='default')
1187      parser.add_argument('--wait_for_manual_settings',
1188                          help='flag makes program wait for input before rendering time series.',
1189                          default=False, action='store_true')
1190      parser.add_argument('--double_stream', help='plot canonical streamline and magnetic of given variable', default=None)
1191      parser.add_argument('--x_offset', help='x offset of single streamline', default=0, type=int)
1192      parser.add_argument('--y_offset', help='y offset of single streamline', default=0, type=int)
1193      parser.add_argument('--file_number_offset', help='offset in file numbering', default=0, type=int)
1194      parser.add_argument('--turn_off_density_start', help='time step at which to start turning off density cloud.', type=int, default=
              None)
1195      parser.add_argument('--turn_off_density_end', help='time step at which to end turning off density cloud', type=int, default=None)
1196      parser.add_argument('--velocity', action='store_true', default=False)
1197      parser.add_argument('--stationary_tube', help="flag to hold flux tube launch point"
1198                          "stationary",
1199                          action='store_true', default=False)
1200      parser.add_argument('--stationary_center',
1201                          help='launch_point of stationary tube',
1202                          nargs=2,
1203                          type=float,
1204                          default = [0, 0])
1205      parser.add_argument('--time_shift',
1206                          help='shift gyration phase',
1207                          type=int, default=125)
1208      parser.add_argument('--for_wide_screen',
1209                          help='set resolution and view settings for wide screen animations',
1210                          action='store_true', default=False)
1211      args = parser.parse_args()
1212      return args
1213
1214  if __name__ == '__main__':
1215      main()
```

VITA

Jens von der Linden was born in Mt. Lebanon, PA. At age 6 he moved with his family to Germany. In 2005, he earned his Abitur at Gymnasium Gerresheim in Düesseldorf, Germany. He returned to the USA for his undergraduate studies. As an undergraduate he worked in the Devlin experimental cosmology group, the accelerator physics group at Oak Ridge National Laboratory, and the DIII-D experiment at General Atomics. He earned a Bachelor of Arts with a major in Physics and a minor in Mathematics from the University of Pennsylvania in 2009. During his graduate studies, he volunteered as a science communication fellow at the Pacific Science Center and as an instructor at Software Carpentry. He spent a year conducting part of his dissertation research at Lawrence Livermore National Laboratory where he was supported by the Department of Energy Office of Science Graduate Student Research Program. He earned a Doctor of Philosophy from the William E. Boeing Department of Aeronautics & Astronautics at the University of Washington in 2017.