# GPU-based Parallel Computation of Integral Properties of Volumetrically Digitized Objects

Mete Yurtoglu

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

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Duane Storti, Chair

Mark Ganter

James Riley

Program Authorized to Offer Degree:
Department of Mechanical Engineering

University of Washington

**Abstract**

GPU-based
Parallel Computation of Integral Properties
of Volumetrically Digitized Objects

Mete Yurtoglu

Chair of the Supervisory Committee:
Professor Duane Storti
Department of Mechanical Engineering

Integral properties of objects that have been created digitally or that are volumetrically digitized have an important role in many different applications. There are a few different approaches for the computation of integral properties.

In this dissertation, direct integration methods, using a convergence proof that exists for twice differentiable functions, and basing the computations on numerical derivative operations that have been obtained by discretization of integral formulas have been proposed. We present the theoretical formulations for direct methods of computing integral properties, including surface, volume, and line integrals, based on grid data which represents an object.

We show that while direct integration methods offer advantages by removing the need for random number generation or parametrization of data, they are still computationally intensive like the traditional methods such as Monte-Carlo computations and other integration methods that rely on parametrization of the given grid representation. While the computations are intensive they are parallelizable, and we explore overcoming the computational challenge by implementing parallel algorithms using general purpose graphics processing unit (GPGPU) computing techniques with the help of the parallel programming platform CUDA created by Nvidia.

We present detailed accuracy analysis between the traditional and direct integration methods and detailed performance comparisons between traditional methods, serial implementation of direct methods, and parallel implementations of direct methods. In addition to the introduction and analysis of direct integration methods, we explore an application of direct integration. Motivated by research of alternative methods to Finite Elements and Boundary Elements methods, we apply our direct integration approach to the integral equation that is being used to solve a boundary value problem.

The areas of application for direct integration method extend from 3D printing to medical applications and CAD software. Considering the application areas of these methods, one of the most crucial aspects is performance. Our tests suggest that using GPGPU techniques on direct integration methods reduces the computation times to levels which allow live visualization for the aforementioned application areas.

# TABLE OF CONTENTS

# LIST OF FIGURES

# Chapter 1

# INTRODUCTION

Integral properties of objects have an important role in many different applications. For the objects that exist in the real world, there has been numerous studies for developing instruments and techniques in order to be able to measure the properties [1, 2]. In measuring properties, there are two main challenges, first, successfully digitizing the object, then, making accurate measurements in obtaining the desired properties. This study is focused in the second part, where most accurate measurements are aimed to be obtained using high performance computing techniques. While real world objects are a good motivation and there are many needs for obtaining integral properties of these objects, this study is aimed to also apply the same techniques to objects existing solely in digital world as well. In order to distinguish these two type of objects, using Lindblad and Nyström's terminology [3], we call objects that exist solely in the digital world, "digital objects", whereas the ones that are volumetrically digitized from a real world object "digitized objects". Since our computations are carried out on both of these objects, and the algorithms that are used in computing the integral properties do not distinguish between digital and digitized objects, here we group both of these class of objects under a single term, Digital Grid Representation (DG-Rep). Our interest lies on carrying out integrals that compute object properties such as surface area, moment of inertia, and volume are defined over the object in $R^3$ [4] on DG-Reps. In order to carry out these integrals, there are multiple approaches including statistical methods, polygonization, and direct integration [5]. In this dissertation, direct methods that leverage function values on a grid is proposed. We present the theoretical formulations for direct methods of computing integral properties, including surface, volume, and line integrals, based on grid data. The computations are intensive but parallelizable, and we will

explore overcoming the computational challenge by implementing parallel algorithms using general purpose graphics processing unit (GPGPU) computing techniques.

### *1.1 Direct integration*

There are many ways to obtain integral properties of object representations. While most of the existing literature relies on traditional methods such as Monte-Carlo computations which require the intermediate step of random number generation which affect the deterministic nature of the computations or meshing which requires intermediate step of polygon generation that may affect the reliable nature of the computation as discussed in the next chapter, the approach studied in this thesis aims to obtain information from the DG-Rep directly. Using the convergence proof that exists for twice differentiable functions that has been stated by Resnikoff and Wells as Theorems 11.7 & 11.8 in **Wavelet Analysis** [6], and basing the computations on numerical derivative operations that have been obtained by discretization of integral formulas that have been obtained with the help of calculus theorems such as Divergence theorem, we propose a method that we name direct integration methods.

This study aims to show that, with direct methods; surface integrals, volume integrals, and line integrals can be used in computing integral properties such as surface area, volume, centroid, moment of inertia, intersection length of 3D objects, etc. Direct methods are computationally intensive, and that creates the need for a method to improve performance of these computations. Once the computational intensity is overcome by employing high performance computing (HPC) techniques with GPGPU, it can be shown that direct integration yields reliable results that confirm the results have been obtained from other reliable methods.

In this dissertation, the integrals are evaluated in $\mathbb{R}^3$. Thus, the volume integral is evaluated over regions of codimension 0, surface integral is on surfaces of codimension 1, and line integral over regions of codimension 2. The reason we assume that we are in $\mathbb{R}^3$ not only lies on the fact that that is the space we perceive, but also because of computational reasons related to GPGPU, explained below.

### *1.2 Parallel computation*

In addition to developing the theory and studying the convergence behavior of direct integration methods, another aim of this study is to create high-performance solvers to enable the practical use of these. Parallelism, when applied to a suitable problem, is proven to be an effective method for speeding-up solvers.

All modern computers are able to process information in parallel. As of early 2000s, major processor manufacturers and architectures saw the diminishing gains in single-processor performance growth and started concentrating on multicore configurations [7]. While Moore's law which states that the number of transistors that can be mounted on a chip grows exponentially still holds, the processor clock rates came to a halt in mid 2000s due to the thermal/power limits [8, 9]. Figures 1.1 and 1.2 show these trends clearly.

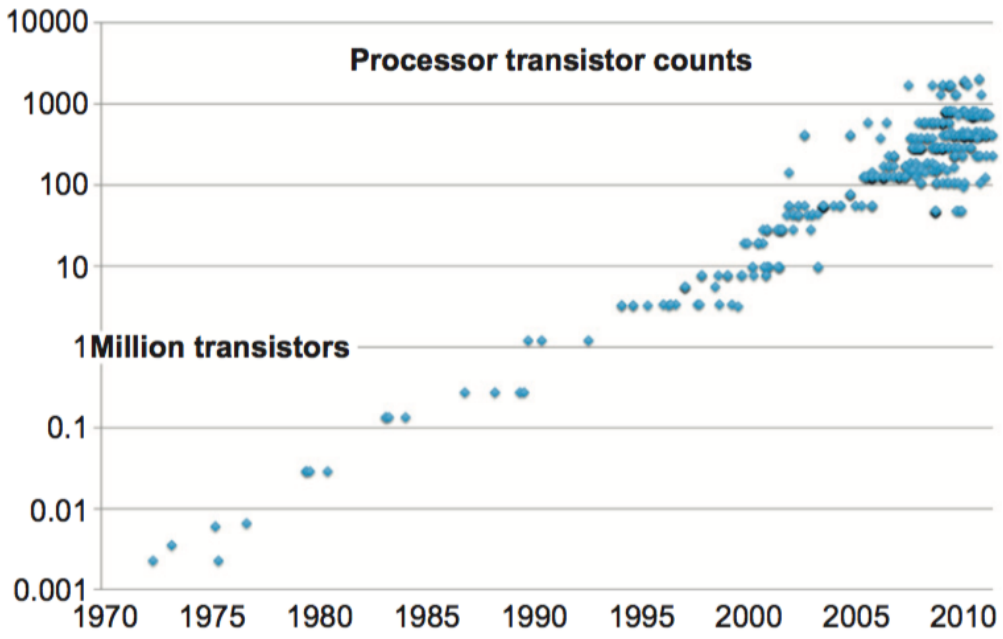



Figure 1.1: Processor transistor counts (log scale) vs time (in years). It shows that a linear fit can be made on this logarithmic plot which indicates that Moore's law still holds for processor transistor counts [9].

As multicore systems become more and more popular, the answer to diminishing gains

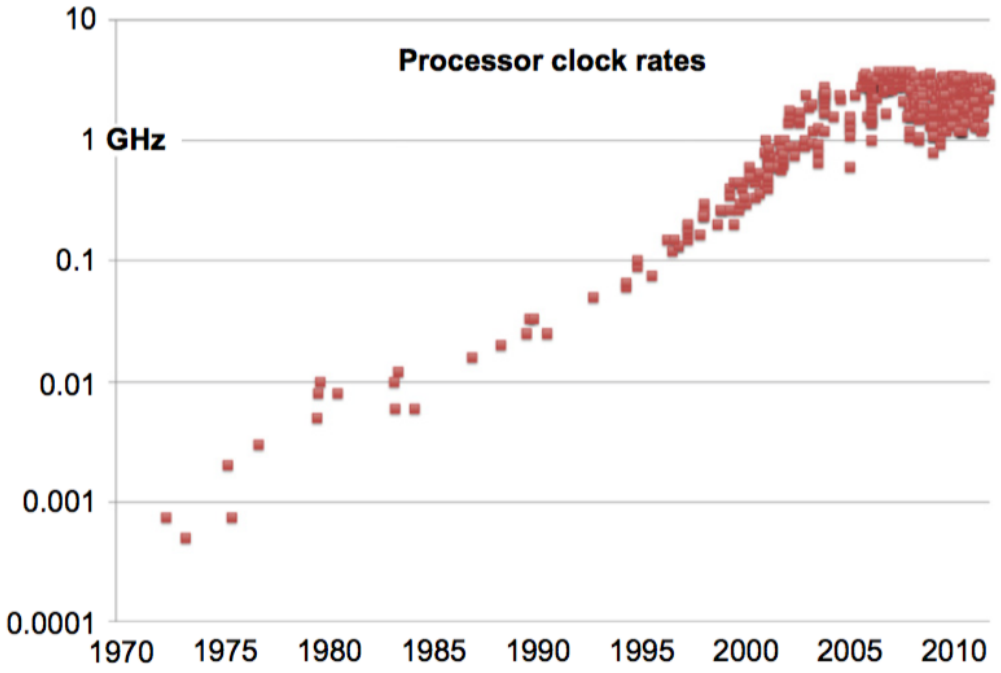



Figure 1.2: Processor clock rates (log scale) vs time (in years) show that processor clock rate improvements plateaued in mid 2000s[9].

in performance comes from the software-side. In high-performance computing (HPC), modern approaches are almost always based on parallel algorithms. While the processor clock rates may now be stale, parallelization of problems and efficient usage of multicore architectures still lead to significant performance gains in HPC problems. Currently, there are many ways to achieve parallel problem solving capabilities on consumer machines. While CPUs become multicore and nowadays it is possible to find 2, 4, or 8 core processors on high-performance systems mainly marketed toward the gaming market, another gaming-related product, namely, the graphics processing unit (GPU), provides even greater parallelism opportunities. Due to their task of graphics processing, which requires a design aiming for higher parallel throughput rather than for reducing latency, GPUs were well positioned for parallel computation [10]. Figure 1.3 from Nvidia's CUDA C programming guide [11] shows this fundamental difference in the hardware design between CPUs and GPUs.


Figure 1.3: While the CPU minimizes latency by dedicating lots of DRAM and Cache space (shown in orange) to storing data where it can be quickly accessed, GPU is designed to have lots of computing units (in green) aimed at producing high throughput[11, 12].

While initially there was no programming environment/language to exploit the computation capabilities of GPUs easily, people started attempting solving numerical problems using graphics APIs such as OpenGL and Direct3D. This approach of solving general numerical problems on the GPU lead to the term GPGPU, which stands for general-purpose GPU programming. The milestone in the development of GPGPU environments was the introduction

of CUDA in 2007 by NVIDIA [10]. Now that it has been a decade since the introduction of this environment, it has matured considerably and become easier to use, adding new capabilities and many libraries to the CUDA ecosystem. It should also be noted that CUDA is not the only GPGPU programming environment. Other programming models such as OpenCL by Khronos Group, OpenACC by OpenACC.org., and C++AMP by Microsoft are also used by people working in the HPC [13]. For this work, NVIDIA's CUDA was chosen because of its maturity and ease of use.

NVIDIA GPUs can be programmed using CUDA through many different languages thanks to its thriving ecosystem. Languages that can be used include C/C++, Fortran, Python, Java, and .NET family. Among these, for the purpose of this work, C/C++ has been selected. The factors motivating this choice include:

- It is CUDA Toolkit's native language.
- Existing code related to this project in the Solheim lab mostly used C/C++.
- It was also used in the serial code written in this study.

Serial code written in C/C++ can readily be parallelized using CUDA C/C++. Significant performance gains are often possible, but not all problems are easily parallelizable.

Direct methods, while offering advantages by removing the need for parametrization of data, are intensive but parallelizable through map, stencil, and reduction operations. Shown in Figures 1.4 to 1.6, these operations are highly parallelizable.

As stated in the beginning of this section, in addition to developing the theory and studying the convergence behavior of direct integration methods, another aim of this study is to create high-performance numerical integrators to enable the practical use of these. In the following chapters, after the theory and implementation is discussed for each direct integration approach, a performance analysis section is provided where the CPU performance is compared to the GPU performance. These comparisons are done on a Linux system with Ubuntu 16.04 LTS installed that has a quad-core 3.00 GHz CPU (Intel Core i5-3330) with

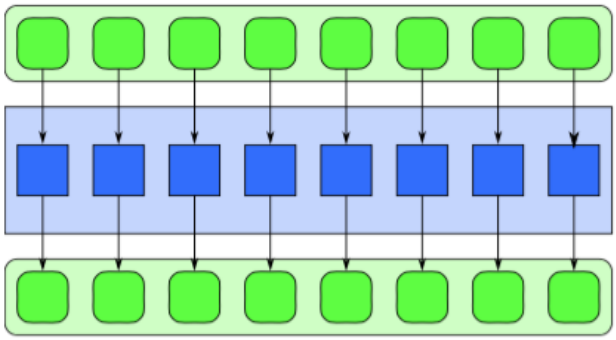


Figure 1.4: Map pattern, where a function is applied to each element in a collection, independent of its neighbors [9]. Here the green squares with rounded corners represent data whereas the blue squares represent function(s) applied to data.

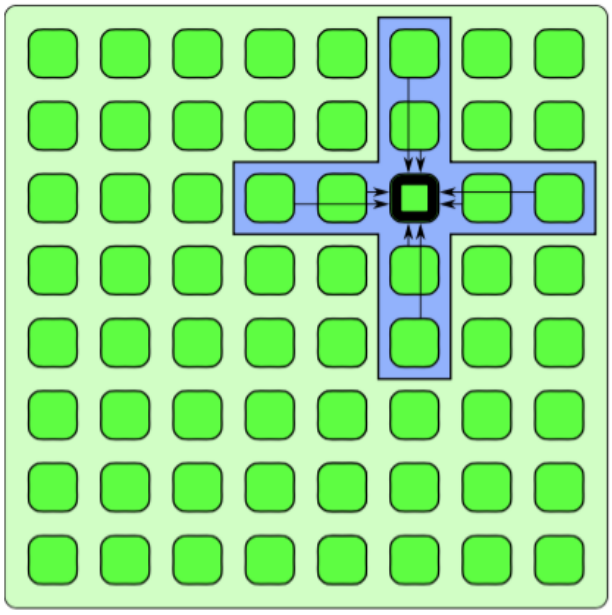


Figure 1.5: Stencil pattern, where a function is applied to an element and its neighbors with some radius (1 in this case) for each element in the output [9].

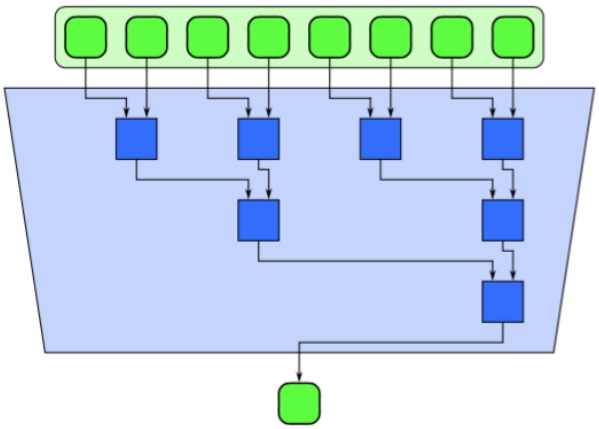


Figure 1.6: Reduction pattern combining green data with blue combiner functions parallely into a single result. Given the combining function associative, this pattern will compute the result efficiently in parallel [9].

32 GB of 1866 MHz DDR3 SDRAM and an NVIDIA GeForce GTX 1080 GPU (Pascal architecture) with 8 GB of GDDR5 memory and 2560 CUDA cores. Pascal architecture is an important technology that allows high performance double precision computing. Unlike previous CUDA architectures, double precision is now much faster compared to before and the performance ratio between single precision and high precision is only 2:1 [14]. Additionally, double precision atomic operations are also supported in Pascal architecture (as of CUDA 8), which is critical for some of the CUDA code provided in this study to be able to run successfully.

As mentioned in the previous section, the integrals in this work are evaluated in $\mathbb{R}^3$. In addition to $\mathbb{R}^3$ being the space we perceive naturally, CUDA implements computational grids of dimension up to 3. This makes CUDA work nicely with problems set in $\mathbb{R}^3$.

# Chapter 2

# SURFACE INTEGRATION OF DIGITIZED OBJECTS

This chapter begins with a brief discussion motivating the need for surface integration in digitized objects. Next, theory of direct surface integration of objects represented by a function is presented. The integration formula is then discretized and becomes also applicable to grids of data obtained either from sampling of functions or from digitization of objects through segmentation. Once the discrete formulation is obtained, the implementation is shown and discussed in the proceeding section. Finally, the Chapter concludes with the comparison and discussion of the results.

As mentioned in Section 1.1, Resnikoff and Wells' convergence proof [6] and Storti's previous work on surface integration [5] are the starting points and inspiration for this study. When it comes to surface integral properties of digitized objects, the most well-known approach is to first polygonize the object surface using a computer graphics algorithm such as marching cubes [15], its improved variants such as Marching Tetrahedra and Dual Contouring [16, 17, 18], and simplicial pivoting [19] and to obtain a polygonal mesh (usually triangles).

These triangle areas are then computed and summed in the hope of obtaining a close approximation to the surface area of the actual object. However, it has been shown in late 19th century that when a cylinder area is approximated by forming polyhedra such as triangles, depending on the ratio of the element numbers, the surface area can be calculated as any value between approximately the actual value of the cylinder and infinity [20, 21, 22]. Figures 2.1 and 2.2 created with the help of the Lantern Paradox Mathematica demonstration by Beck and Hafner [23] illustrate the problems associated with polygonization methods and provide motivation for alternative approaches. In this study we hypothesize that direct

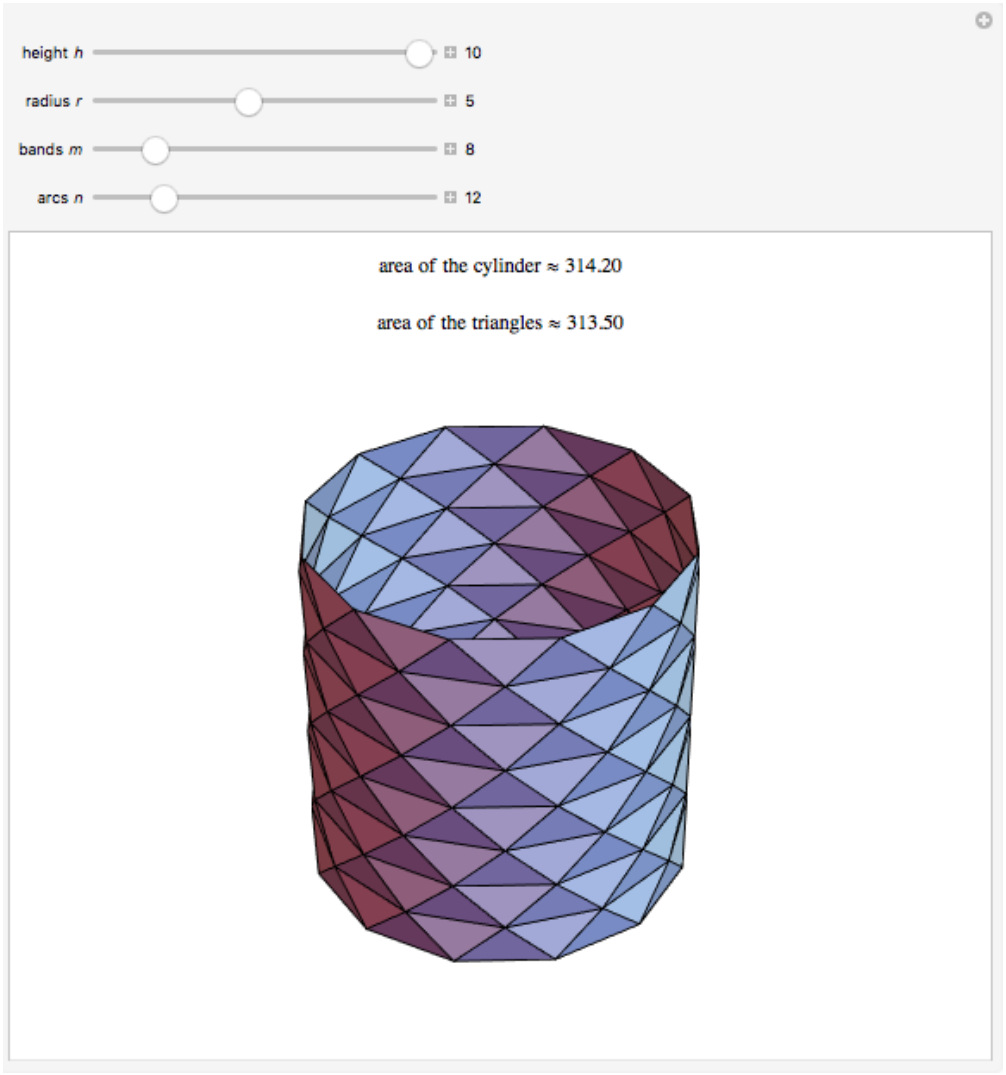



Figure 2.1: Lantern with approximate area as sum of triangles.

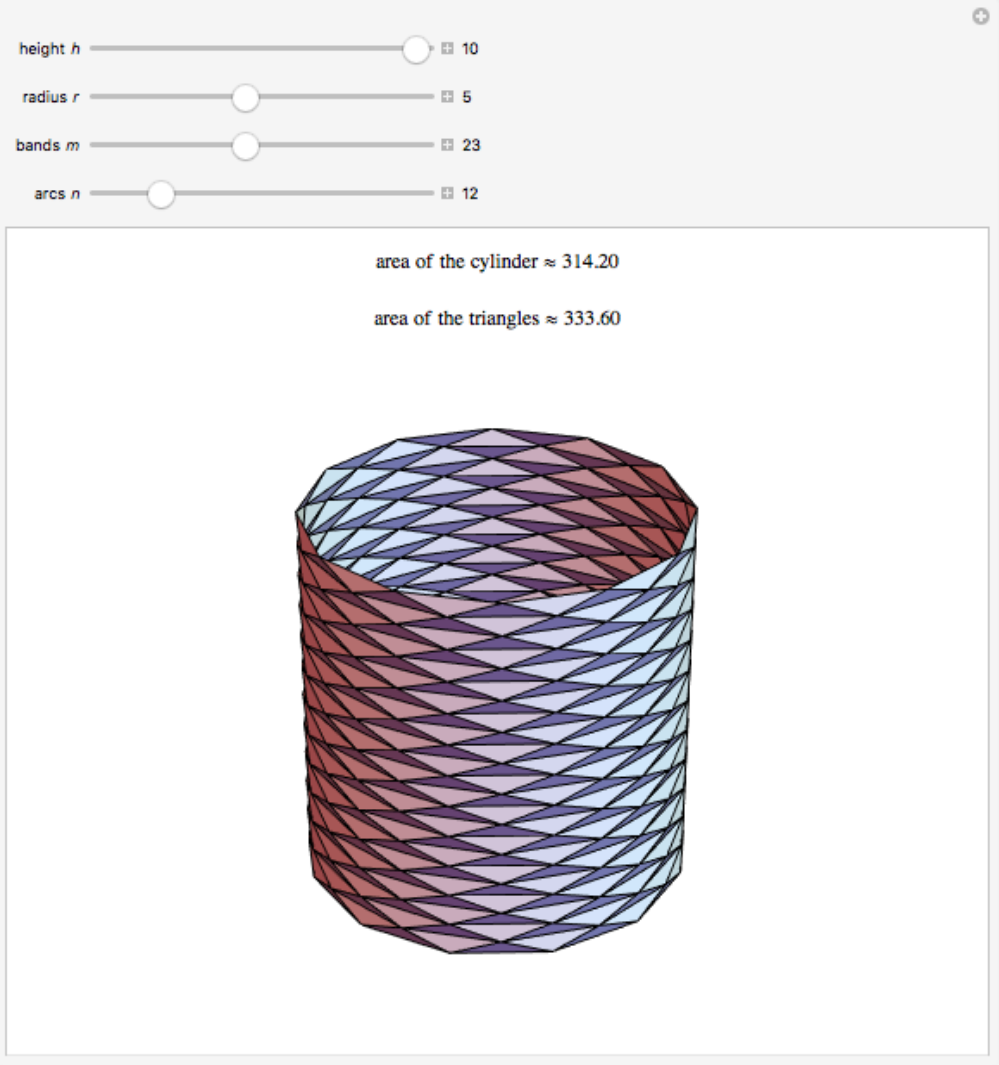



Figure 2.2: Compared to Figure 2.1, the area is diverging although the number of bands has increased and arcs kept the same.

surface integration is a solution to this problem. In the next section, the direct formula is derived and then discretized.

### *2.1 Theory*

In this section, the discrete surface integration formula is derived by closely following the derivation steps shown in [5].

Let $f(\boldsymbol{r})$ be an implicit function representing the region $\Omega \subset \mathbb{R}^3$ with boundary $\partial\Omega$, where $\boldsymbol{r}$ is the position vector. In our sign convention, we chose to assign the negative values to the interior points, i.e., $f(\boldsymbol{r}) < 0$ inside the object. For this sign convention, the outward unit normal, $\boldsymbol{n}$ is related to the implicit function, $f(\boldsymbol{r})$ by $\boldsymbol{n} = \frac{\nabla f}{|\nabla f|}$. The occupancy function is then given by

$$\chi(\boldsymbol{r}) = [1 - sgn(f(\boldsymbol{r}))]/2 = \begin{cases} 1, & f < 0 \\ \frac{1}{2}, & f = 0 \\ 0, & f > 0 \end{cases} \tag{2.1}$$

where $sgn$ is the sign (or signum) function. The integral of a scalar function, $g(\boldsymbol{r})$, over a surface is then given by

$$A(\partial\Omega) = \int_{\partial\Omega} g(\boldsymbol{r}) ds \tag{2.2}$$

which can also be written using the definition of the unit normal vector, $\boldsymbol{n} \cdot \boldsymbol{n} = 1$ as

$$A(\partial\Omega) = \int_{\partial\Omega} g(\boldsymbol{r}) \boldsymbol{n} \cdot \boldsymbol{n}\, ds \tag{2.3}$$

where $\boldsymbol{n}ds$ is oriented surface element. Applying Divergence Theorem [24]

$$\int_V (\nabla \cdot \boldsymbol{F}) dv = \int_{\partial V} \boldsymbol{F} \cdot \boldsymbol{n} ds \tag{2.4}$$

where $dv$ on the left-hand side is volume element, to Equation 2.3 we obtain the volume integral

$$A(\partial\Omega) = \int_{\Omega} \nabla \cdot g(\boldsymbol{r}) \boldsymbol{n}\, dv \tag{2.5}$$

Extending the integration in Equation 2.5 to $\mathbb{R}^3$ using the occupancy function, $\chi(\boldsymbol{r})$ from Equation 2.1, we get

$$A(\partial\Omega) = \int_{\mathbb{R}^3} \chi \nabla \cdot g(\boldsymbol{r})\boldsymbol{n}\, dv \tag{2.6}$$

Substituting the product rule for the divergence of a product of a scalar $\chi$ and a vector $\boldsymbol{F}$

$$\nabla \cdot (\chi \boldsymbol{F}) = \nabla\chi \cdot \boldsymbol{F} + \chi(\nabla \cdot \boldsymbol{F}) \tag{2.7}$$

into Equation 2.6, we obtain

$$A(\partial\Omega) = \int_{\mathbb{R}^3} \nabla \cdot (\chi\boldsymbol{n})g(\boldsymbol{r})\, dv - \int_{\mathbb{R}^3} (\nabla\chi \cdot \boldsymbol{n})g(\boldsymbol{r})\, dv \tag{2.8}$$

Applying the Divergence Theorem on the right-hand side, the first term vanishes since the occupancy is zero at $\partial\mathbb{R}^3$

$$\begin{aligned} A(\partial\Omega) &= \cancel{\int_{\partial\mathbb{R}^3} \chi\boldsymbol{n} \cdot (g(\boldsymbol{r})\boldsymbol{n})\, ds} - \int_{\mathbb{R}^3} (\nabla\chi \cdot \boldsymbol{n})g(\boldsymbol{r})\, dv \\ &= -\int_{\mathbb{R}^3} (\nabla\chi \cdot \boldsymbol{n})g(\boldsymbol{r})\, dv \end{aligned} \tag{2.9}$$

Substituting the definitions for $\chi$ and $\boldsymbol{n}$ yields

$$A(\partial\Omega) = \frac{1}{2}\int_{\mathbb{R}^3} \nabla(sgn(f)) \cdot \frac{\nabla f}{|\nabla f|} g(\boldsymbol{r})\, dv \tag{2.10}$$

and discretization on a regular grid with spacing $\Delta$ produces

$$A(\partial\Omega) = \frac{\Delta^3}{2}\sum_{i,j,k} \frac{\left[\frac{\partial}{\partial x}sgn(f)\right]_{i,j,k}\left[\frac{\partial f}{\partial x}\right]_{i,j,k} + \left[\frac{\partial}{\partial y}sgn(f)\right]_{i,j,k}\left[\frac{\partial f}{\partial y}\right]_{i,j,k} + \left[\frac{\partial}{\partial z}sgn(f)\right]_{i,j,k}\left[\frac{\partial f}{\partial z}\right]_{i,j,k}}{\sqrt{\left[\frac{\partial f}{\partial x}\right]^2_{i,j,k} + \left[\frac{\partial f}{\partial y}\right]^2_{i,j,k} + \left[\frac{\partial f}{\partial z}\right]^2_{i,j,k}}} g_{i,j,k} \tag{2.11}$$

where, the derivatives are evaluated as

$$\left[\frac{\partial f}{\partial x}\right]_{i,j,k} = \frac{f_{i+1,j,k} - f_{i-1,j,k}}{2\Delta} \tag{2.12}$$

The coefficients in this derivative estimate, which coincides with a standard second order accurate centered finite-difference estimate, are in fact the connection coefficients for computing the derivative of a function described by Daubechies wavelets [25, 26]. The derivative

estimate therefore is a valid estimate for derivatives of Daubechies wavelet functions and allows us to invoke the convergence proof in Resnikoff and Wells for twice differentiable functions that define bounded regions of finite perimeter with finite number of nonsmooth points [6].

The discrete formula in Equation 2.11 is applicable to grids of data obtained either from sampling of functions or from digitization of objects through segmentation of volumetric scan data and provides the basis for reliable computation of surface integral properties.

## *2.2 Implementation*

Starting with Equations 2.11 and 2.12, in order to generalize the problem, it is assumed that a grid of values is provided that is either the result of sampling of a function or result of segmentation, with the given spacing, $\Delta$. Another assumption made here for convenience is about grid sizing. The grid is assumed to be in 3D and with the same dimension in all three directions. With these, the function that is written in C++ and shown in Listing 2.1 implements the equation derived in Section 2.1. The function, named `surfArea`, accepts a grid of `double` numbers, named `grid`, another grid of double numbers for the derivative stencils, named `stencil`, and dimensional parameters related to problem size and spacing. The function returns a `double`, which is the result of the integration.

Listing 2.1: Implementation of Surface Integral function

```
1 double surfArea(const double *grid, const double *stencil, int rad,
2   const double3 &lim, double delta, const dim3 &dim) {
3   double areaSum = 0.0;
4   for (int s = 0; s < dim.z; s++) {
5     for (int r = 0; r < dim.y; r++) {
6       for (int c = 0; c < dim.x; c++) {
7         int3 pt = make_int3(c, r, s);
8         double3 df = make_double3(
9           xDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
```

```
10            yDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
11            zDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0));
12          double3 dchi = make_double3(
13            xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
14            yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
15            zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
16          double denom = computeMagnitude(df);
17          if (denom < EPS) continue;
18          areaSum += dotProduct(df, dchi) / denom;
19        }
20      }
21    }
22    return areaSum / 2.0 * std::pow(delta, 3);
23  }
```

## 2.3 Results and Discussion

### 2.3.1 Torus surface area

Using the C++ code presented in Appendix A in Listing A.1, the surface integral function presented in Listing 2.1 has been applied to a torus that has been implicitly defined.

A torus as shown in Figure 2.3 can be implicitly defined [27] as

$$f(x, y, z) = \left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 - r^2 \tag{2.13}$$

where $R$ is the major radius (distance from the center of the tube to the center of the torus) and $r$ is the minor radius (radius of the tube). For a torus with given major and minor radii, the surface is area is given by

$$A = 4\pi^2 Rr \tag{2.14}$$

A sequence of numerical experiments was performed for a range of values of $R/\Delta$ and different stencil widths corresponding to different wavelet genera as given in Resnikoff and

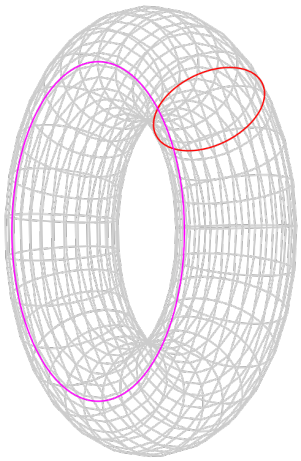


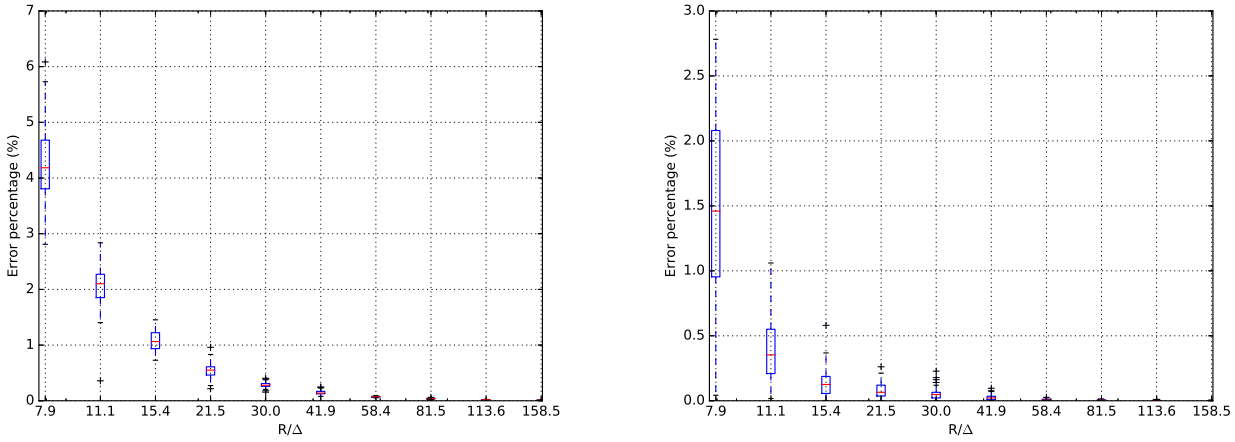
Figure 2.3: Torus with major radius shown in purple (circle on the left) and minor radius shown in red (circle on the right) [27].

Wells [6]. To produce results that do not depend on fortunate grid alignments, 250 trials of surface area computation are performed with random rigid body transformations and the statistics (mean, standard deviation, and confidence interval) of the results are shown in Figures 2.4 and 2.5 for the first four genera.

Curve fitting on log-log plots in Figures 2.6 to 2.9 produces the estimates of convergence rates shown in Table 2.1.

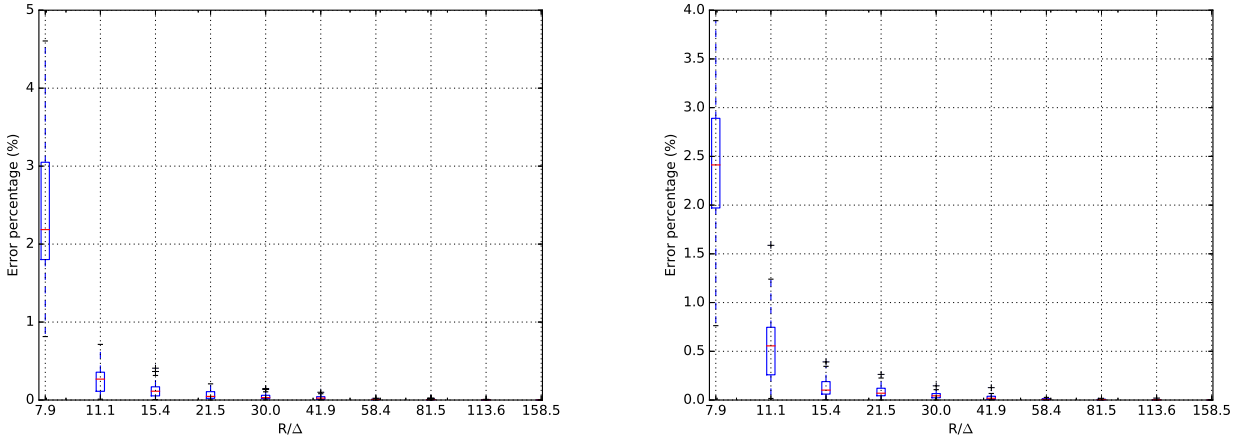
### *2.3.2 Toroidal shell moment of inertia*

In addition to being able to calculate to surface area directly with this method, starting from Equation 2.2 with a function $g$ instead of 1 as the integrand, other surface integral properties of the object can be computed. As an example, continuing with the torus implicitly defined in Section 2.3.1, Equation 2.13, in this section, direct integration is applied for the computation of another fundamental property. Here, we compute the moment of inertia of our torus (which is actually a thin toroidal shell). Since the moment of inertia of a thin toroidal shell about z-axis (axis through the center) is




(a) (b)

Figure 2.4: Error percentage vs. $R/\Delta$ for torus surface area for stencil genus 1 (a) and 2 (b).

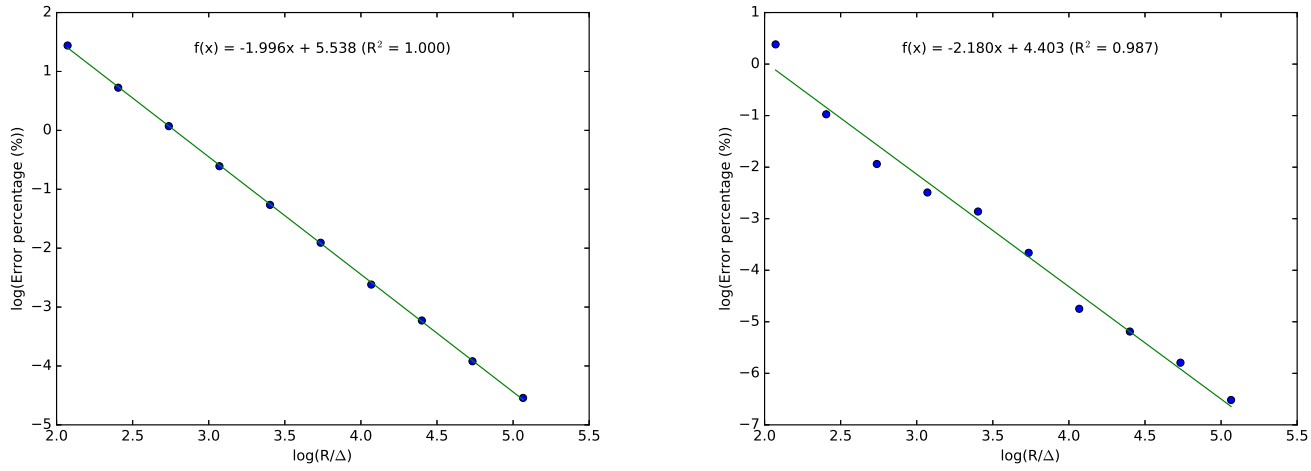



(a) (b)

Figure 2.5: Error percentage vs. $R/\Delta$ for torus surface area for stencil genus 3 (a) and 4 (b).

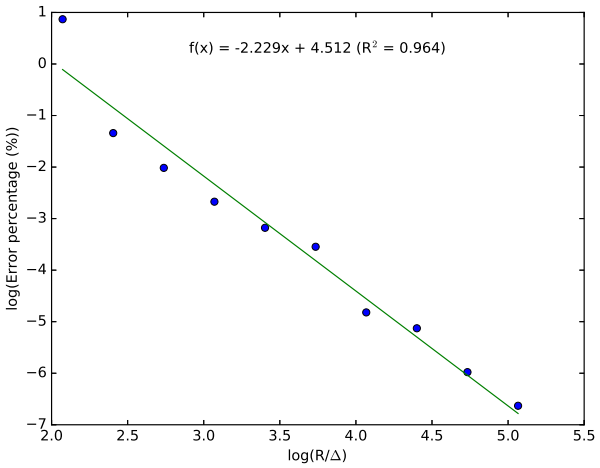
| Genus | Convergence Exponent |
|---|---|
| 1 | -1.99 |
| 2 | -2.18 |
| 3 | -2.23 |
| 4 | -2.29 |
| 5 | -2.36 |
| 6 | -2.42 |
| 7 | -2.46 |

Table 2.1: Gradual improvements in the convergence exponent from -1.99 for genus 1 to -2.46 for genus 7 at the expense of stencil radii increasing from 1 to 12.

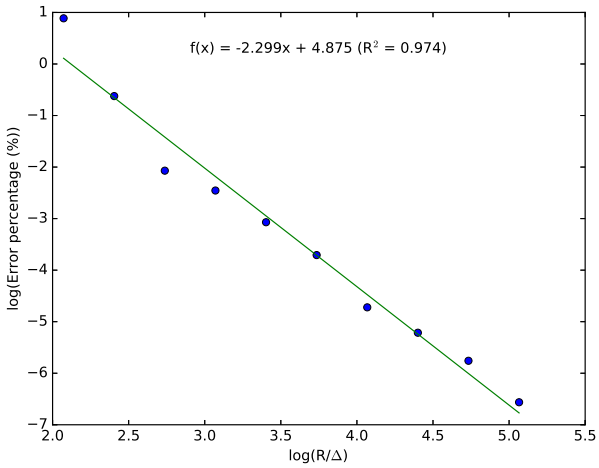



(a) (b)

Figure 2.6: Loglog curve fitting plots of Error percentage vs. $R/\Delta$ for torus surface area for stencil genus 1 (a) and 2 (b).

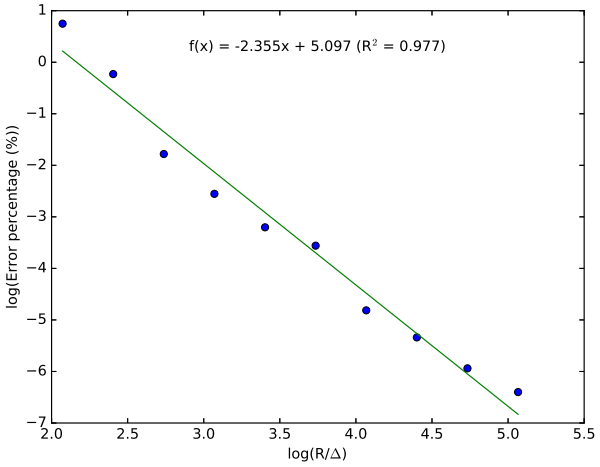



(a)

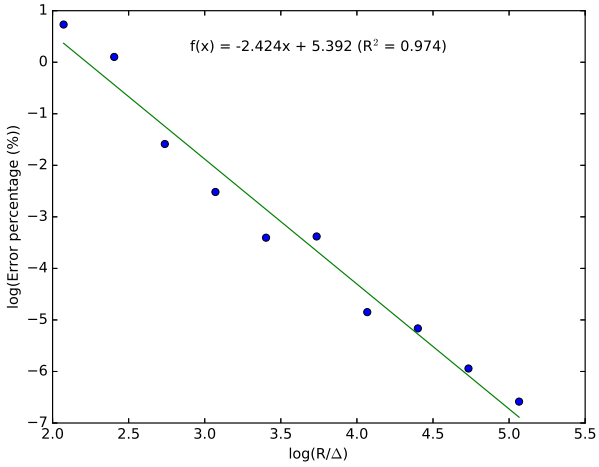



(b)

Figure 2.7: Loglog curve fitting plots of Error percentage vs. $R/\Delta$ for torus surface area for stencil genus 3 (a) and 4 (b).




(a)




(b)

Figure 2.8: Loglog curve fitting plots of Error percentage vs. $R/\Delta$ for torus surface area for stencil genus 5 (a) and 6 (b).

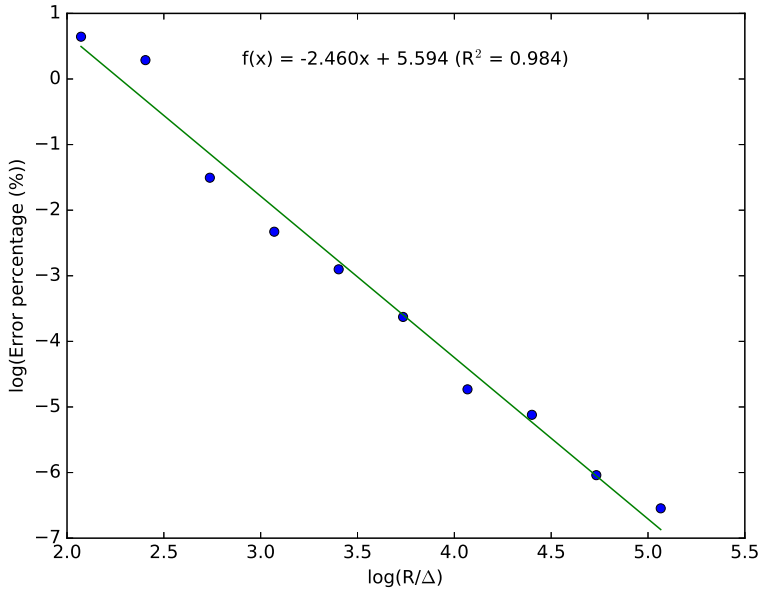



Figure 2.9: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for torus surface area for stencil genus 7.

$$I_z = \frac{1}{2}m\Big(2R^2 + 3r^2\Big) \tag{2.15}$$

where $m$ is the mass, $R$ is the major radius, and $r$ is the minor radius. Assuming unit mass per area, combining the area formula from Equation 2.14, the moment of inertia formula becomes

$$I_z = 2\pi^2 rR\Big(2R^2 + 3r^2\Big). \tag{2.16}$$

Taking this as our ground truth and this time using $g(\boldsymbol{r}) = x^2 + y^2$ in Equation 2.2 instead of the implicit $g(\boldsymbol{r}) = 1$ integrand, the resulting discrete formula is

$$I(\partial\Omega) = \frac{\Delta^3}{2}\sum_{i,j,k}\left(\frac{\big[\frac{\partial}{\partial x}sgn(f)\big]_{i,j,k}\big[\frac{\partial f}{\partial x}\big]_{i,j,k} + \big[\frac{\partial}{\partial y}sgn(f)\big]_{i,j,k}\big[\frac{\partial f}{\partial y}\big]_{i,j,k} + \big[\frac{\partial}{\partial z}sgn(f)\big]_{i,j,k}\big[\frac{\partial f}{\partial z}\big]_{i,j,k}}{\sqrt{\big[\frac{\partial f}{\partial x}\big]^2_{i,j,k} + \big[\frac{\partial f}{\partial y}\big]^2_{i,j,k} + \big[\frac{\partial f}{\partial z}\big]^2_{i,j,k}}}(x^2_{i,j,k}+y^2_{i,j,k})\right) \tag{2.17}$$

Since convergence trends across different genera was already studied in the previous section, this time, the moment of inertia has been calculated for each $R/\Delta$ value for each genus and the results are combined into a single plot shown in Figure 2.10. This figure is obtained by implementing Equation 2.17 for computing the moment of inertia around the z-axis for a toroidal shell, Listing A.2 in Appendix A provides the code used in obtaining the results. While the general program flow is the same as Listing A.1 given in Appendix A, the moment of inertia function is given in Listing 2.2.

Listing 2.2: Implementation of the moment of inertia function

```
double momentOfInertia(const double *grid, const double *stencil, int rad,
  const double3 &lim, double delta, const dim3 &dim) {
  double moiSum = 0.0;
  for (int s = 0; s < dim.z; s++) {
    for (int r = 0; r < dim.y; r++) {
      for (int c = 0; c < dim.x; c++) {
        int3 pt = make_int3(c, r, s);
        double3 df = make_double3(
          xDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
          yDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
          zDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0));
        double3 dchi = make_double3(
          xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
        double denom = computeMagnitude(df);
        if (denom < EPS) continue;
        double3 xyz = crs2xyz(pt, delta, lim);
        moiSum += ((xyz.x * xyz.x) + (xyz.y * xyz.y)) * dotProduct(df, dchi) /
            denom;
      }
    }
```

```
22   }
23   return moiSum / 2.0 * std::pow(delta, 3);
24 }
```

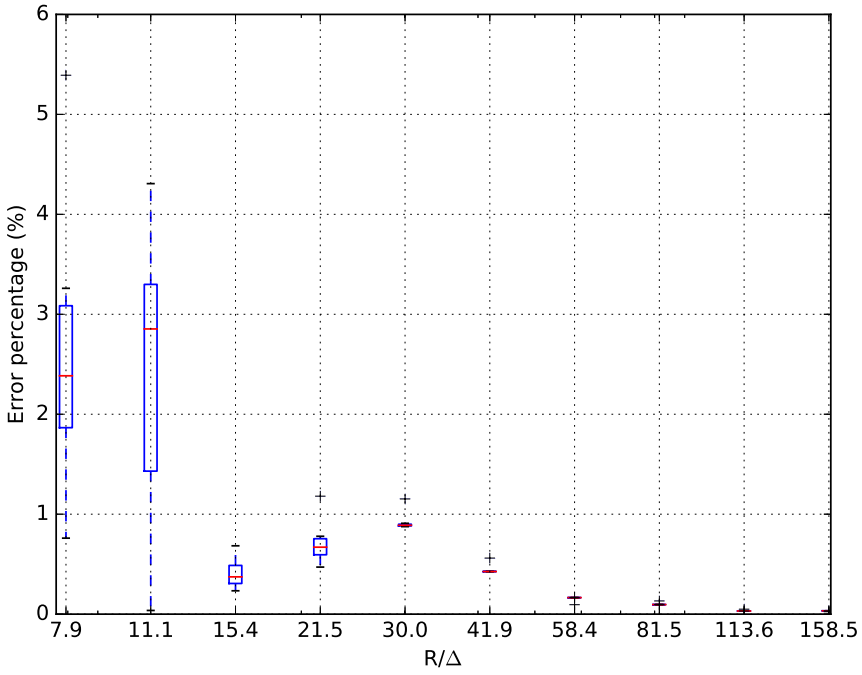



Figure 2.10: Boxplot of Error percentage vs. $R/\Delta$ for torus moment of inertia, $I_{zz}$.

### *2.3.3 Flame surface computation*

As the final example of this chapter, a real-life example is being presented. Understanding the extinction and reignition behaviors of turbulent nonpremixed combustion is important, since it has practical significance [28]. In doing this, flame surface in different configurations is studied. This is a problem that needs to be solved repetitively for all the different configurations. While triangulation and triangle area summation is also applicable, because of its aforementioned shortcomings and performance reasons, direct integration is applicable and benficial.

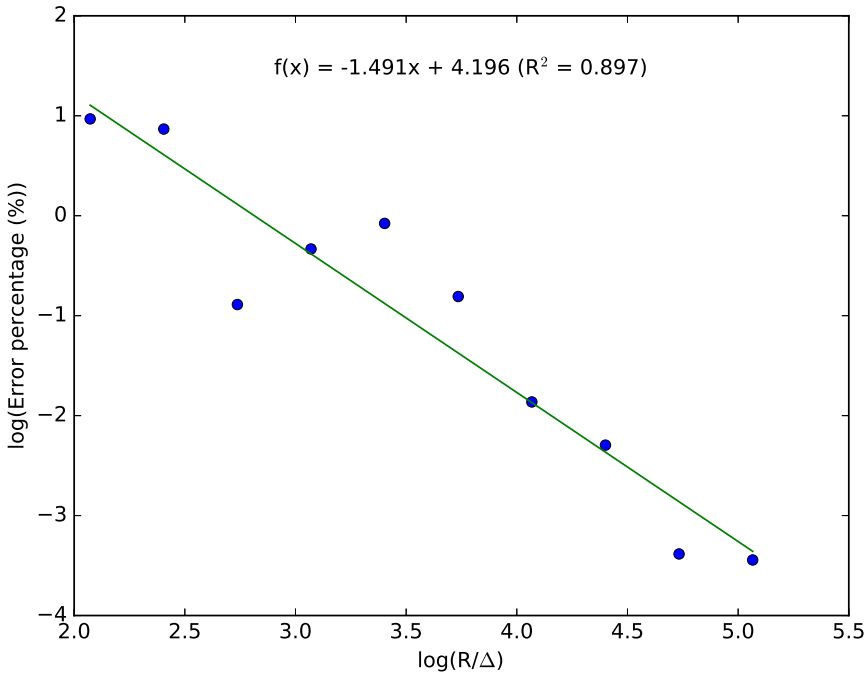



Figure 2.11: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for hollow torus moment of inertia, $I_z$

In the analysis done by Wang, an isosurface using DNS (direct numerical simulation) volumetric mixture fraction data is rendered at an isovalue $Z_{iso}$ using marching cubes algorithm. The algorithm triangulates the given isosurface, and the sum of the triangles gives the isosurface area. Such an isosurface created using this method with Matlab code provided by Brandon Blakeley and Dr. James Riley is shown in Figure 2.12 at the specified isovalue $Z_{iso} = 0.35$. For this selection of isovalue, the sum of rendered triangle areas was found to be 144.61.

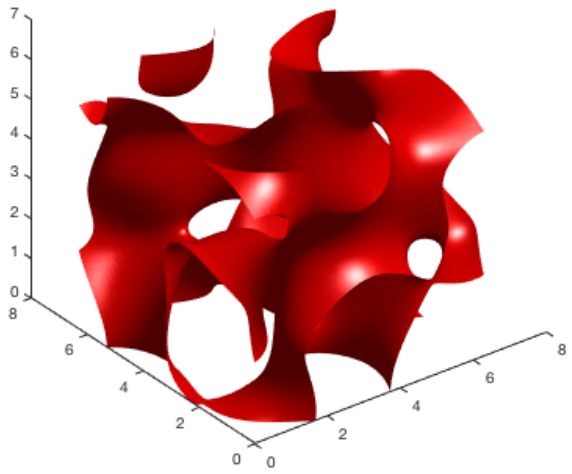



Figure 2.12: Isosurface plot rendering of DNS data using marching cubes at $Z_{iso} = 0.35$.

With the same data, and the same isovalue, direct integration method is also used. The code provided in Appendix Listings A.3, A.4, and A.5 show the code that has been used in obtaining the direct integration results for the same dataset. The result of the computation was found as 145.77, which is 0.7% higher than the triangle summation method.

## *2.4 GPU Implementation and Performance Analysis*

Direct surface integration, developed, implemented and analyzed in Sections 2.1 to 2.3, while offering advantages by removing the need for parametrization of data, is computationally intensive. In this Section, we discuss how each of the test cases solved in the previous section can be parallelized. Using CUDA, we parallelize each problem and then analyze GPU implementations' performance.

### *2.4.1 Torus surface area computation on the GPU*

Starting with Listing 2.1, we implement a kernel function (a function that is executed on the device) named `surfAreaKernel`. The full implementation of the `*.cu` file (CUDA uses `*.cu` extension for code written using CUDA C/C++), is provided in Listing A.6.

The main idea is to create a wrapper function which has the same interface as the serial implementation and is called exactly the same way from the `main.cpp` file. This `kernel.cu` file contains mainly two functions, one that is called from the host (CPU side) and does the necessary CUDA calls to do the required memory transfers and the kernel call, and the other that is the kernel itself. In addition to this, there is code that is used as helper functions for taking derivatives and other operations.

The function, `surfArea` of which the implementation is shown in Listing 2.3, does the interfacing job between `main.cpp` and the kernel function. Between lines 9 and 22, there are a few calls that start with `cuda`. These functions belong to CUDA runtime API, and they care of the memory management aspect of our computations. They do the necessary memory allocations on the GPU for the computations to be successfully carried out. Similarly lines 32 to 37 also do some memory operations needed for transferring the results back to the host side and cleaning up the GPU memory. Finally, the result is returned on line 38. It should also be noted that on line 26, there is a variable called `smSz` that is being set. This is the setting for the shared memory size. Shared memory is an important GPU optimization that allows to increase the memory access speed by bringing global data into the block scope and

allows accessing of data in the shared memory from the members of the same block. That is why, in line 26, when calculating the size of the shared memory, block size is being used. In more detail, we allocate enough memory for the whole thread block and a few more layers on each side of the block depending on the radius of the stencil that is being applied. In addition to that, the stencil and the block sum of the area are also needed to be transferred to the shared memory, and appropriate sized chunks are also reserved here for those.

Listing 2.3: Wrapper function that calls the surface area kernel function

```
1  double surfArea(const double *grid, const double *stencil, int rad,
2    const double3 &lim, double delta, const dim3 &dim) {
3    const int problemSize = dim.x * dim.y * dim.z;
4    double areaSum = 0.0;
5    double *d_areaSum = 0, *d_grid = 0, *d_stencil;
6    dim3 *d_dim = 0;
7    double3 *d_lim = 0;
8
9    cudaMalloc(&d_areaSum, sizeof(double));
10   cudaMemset(d_areaSum, 0, sizeof(double));
11
12   cudaMalloc(&d_grid, problemSize * sizeof(double));
13   cudaMemcpy(d_grid, grid, problemSize * sizeof(double),
        cudaMemcpyHostToDevice);
14
15   cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
16   cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
        cudaMemcpyHostToDevice);
17
18   cudaMalloc(&d_dim, sizeof(dim3));
19   cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
20
21   cudaMalloc(&d_lim, sizeof(double3));
```

```
22   cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
23
24   const dim3 blockSize(TX, TY, TZ);
25   const dim3 gridSize(divUp(dim.x, TX), divUp(dim.y, TY), divUp(dim.z, TZ));
26   const size_t smSz = ((TX + 2 * rad) * (TY + 2 * rad) * (TZ + 2 * rad) +
27                        2 * rad + 1 + 1) * sizeof(double);
28
29   surfAreaKernel<<<gridSize, blockSize, smSz>>>(d_areaSum, d_grid, d_stencil,
30     rad, d_lim, delta, d_dim);
31
32   cudaMemcpy(&areaSum, d_areaSum, sizeof(double), cudaMemcpyDeviceToHost);
33   cudaFree(d_areaSum);
34   cudaFree(d_grid);
35   cudaFree(d_stencil);
36   cudaFree(d_dim);
37   cudaFree(d_lim);
38   return std::pow(delta, 3) / 2.0 * areaSum;
39 }
```

Line 29 in Listing 2.3 calls the kernel function which, in addition to regular C/C++ style arguments, takes three more arguments between the <<< >>> symbols to specify the execution configuration parameters. `gridSize` specifies the number of blocks in the computational grid, `blockSize` specifies the number of threads per block, and `smSz` specifies the amount of storage allocated to each block for shared memory. Grid and block size are of a CUDA-specific data type `dim3`, which has three components corresponding to the three coordinate directions in $\mathbb{R}^3$. The kernel function itself is given in Listing 2.4. The first line contains `__global__`, which is a CUDA qualifier that indicates that the function is called from the host and executed on the device (the GPU side).

Lines from 3 to 12 set up the current thread index both globally and in the scope of the shared block. Lines 16 to 18 set up the shared memory block dimensions. Once the shared block memory is set up properly, lines 21 to 26 set up the shared stencil and the zeroes the

block sum values once for every shared block. Lines 28 to 51 set up the global to shared memory transfer including the halo cells needed for the finite difference operations on the shared block. Line 53 is a barrier function that is being used to synchronize all the threads to the same point within the same block. This is important, because before starting stencil operations, one must make sure that all the necessary memory transfer is complete.

Listing 2.4: Surface area kernel that does the surface area computation on the GPU

```
1 __global__
2 void surfAreaKernel(double *areaSum, const double *grid, const double *stencil
    ,
3   int rad, const double3 *lim, double delta, const dim3 *dim) {
4   const int c = threadIdx.x + blockDim.x * blockIdx.x;
5   const int r = threadIdx.y + blockDim.y * blockIdx.y;
6   const int s = threadIdx.z + blockDim.z * blockIdx.z;
7   if (c >= dim->x || r >= dim->y || s >= dim->z) return;
8   const int3 pt_crs = make_int3(c, r, s);
9   const int i = flatten(pt_crs, *dim);
10  const int s_c = threadIdx.x + rad;
11  const int s_r = threadIdx.y + rad;
12  const int s_s = threadIdx.z + rad;
13  const dim3 sBlockSz = dim3(blockDim.x + 2 * rad, blockDim.y + 2 * rad,
14    blockDim.z + 2 * rad);
15  const int s_i = flatten(make_int3(s_c, s_r, s_s), sBlockSz);
16  extern __shared__ double s_block[];
17  double *s_stencil = s_block + sBlockSz.x * sBlockSz.y * sBlockSz.z;
18  double *s_blockSum = s_stencil + 2 * rad + 1;
19
20  // set to 0 once per shared block
21  if (s_c == rad && s_r == rad && s_s == rad) {
22    for (int k = 0; k < 2 * rad + 1; k++) {
23      s_stencil[k] = stencil[k];
24    }
```

```
25      *s_blockSum = 0.0;
26    }
27
28    // Regular cells
29    s_block[s_i] = grid[i];
30
31    // Halo cells
32    if (threadIdx.x < rad) {
33      s_block[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
34        grid[flatten(make_int3(c - rad, r, s), *dim)];
35      s_block[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
36        grid[flatten(make_int3(c + blockDim.x, r, s), *dim)];
37    }
38
39    if (threadIdx.y < rad) {
40      s_block[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
41        grid[flatten(make_int3(c, r - rad, s), *dim)];
42      s_block[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
43        grid[flatten(make_int3(c, r + blockDim.y, s), *dim)];
44    }
45
46    if (threadIdx.z < rad) {
47      s_block[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
48        grid[flatten(make_int3(c, r, s - rad), *dim)];
49      s_block[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
50        grid[flatten(make_int3(c, r, s + blockDim.z), *dim)];
51    }
52
53    __syncthreads();
54
55    if ((c >= dim->x - rad) || (r >= dim->y - rad) || (s >= dim->z - rad) ||
56        c <= (rad - 1) || r <= (rad - 1) || s <= (rad - 1)) return;
57
```

```
58   const int3 s_pt = make_int3(s_c, s_r, s_s);
59
60   double3 df = make_double3(
61     xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
62     yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
63     zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0));
64
65   double3 dchi = make_double3(
66     xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
67     yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
68     zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0));
69
70   double denom = computeMagnitude(df);
71
72   if (denom > EPS) atomicAdd(s_blockSum, dotProduct(df, dchi) / denom);
73
74   __syncthreads();
75
76   // add only once per shared block
77   if (s_c == rad && s_r == rad && s_s == rad) {
78     atomicAdd(areaSum, *s_blockSum);
79   }
80 }
```

Once the threads are synchronized, the derivative operations are done and the results are added to the local shared result. After another synchronization, only once per block, the result of the block is written to the global result, `areaSum`. The usage of `atomicAdd()` is also noteworthy. Thanks to the atomic addition operation, race conditions are prevented and it is made sure that every thread accesses the sum variables at different times, without stepping on each other.

One thing to be remarked is about the limitation that arises from the usage of shared memory. While it is an important optimization feature, shared memory is limited (48 kB).

Since each double is 8 bytes, the number of maximum doubles that can fit in shared memory is 6,000. Under some conditions (when the stencil radius is bigger than 3), this limitation prevents the CUDA implementation from executing successfully. Thus, for the purpose of timing runs being done here, the first three genera are plotted.

Using double precision on both the serial and parallel implementations, the results are identical. Thus, there is no need for plotting the difference between the serial results vs parallel results. However, timing-wise, GPU enables huge gains. Figures 2.13 and 2.14 show the performance gain obtained with CUDA for the first three genera. At the heaviest computational loading of $\log(R/\Delta) = 5$, CPU takes about 35,000 ms whereas the GPU takes about 500 ms with the kernel itself taking about 300 ms (the rest is spent on memory transfers). This shows a speedup on the order of 100x. More importantly, about 120x speedup created by the kernel, is critical for live applications. In live applications where a shape is manipulated and this is displayed live, keeping data on the GPU without memory transfers, the computation will take about 300 ms at the highest settings, which will allow for almost live visualization. This shows the effectivity and suitability of GPGPU for the discussed type of computations.

### *2.4.2 Torus moment of inertia on the GPU*

Similar to the steps followed in the previous subsection, the only big change is the kernel function that is defined in the `*.cu` file. Listing 2.5 shows the kernel function used for the calculation of moment of inertia.

Listing 2.5: Moment of intertia kernel that runs on the GPU

```
1 __global__
2 void moiKernel(double *moiSum, const double *grid, const double *stencil,
3   int rad, const double3 *lim, double delta, const dim3 *dim) {
4   const int c = threadIdx.x + blockDim.x * blockIdx.x;
5   const int r = threadIdx.y + blockDim.y * blockIdx.y;
```

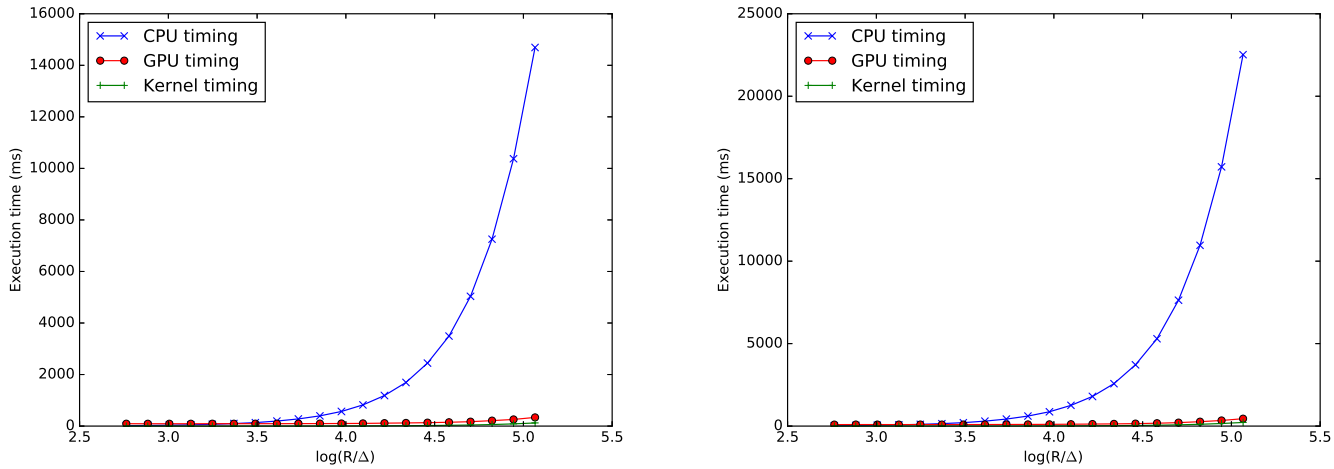



Figure 2.13: Timing comparison for torus surface area computation in wavelet genus 1 (a) and 2 (b).

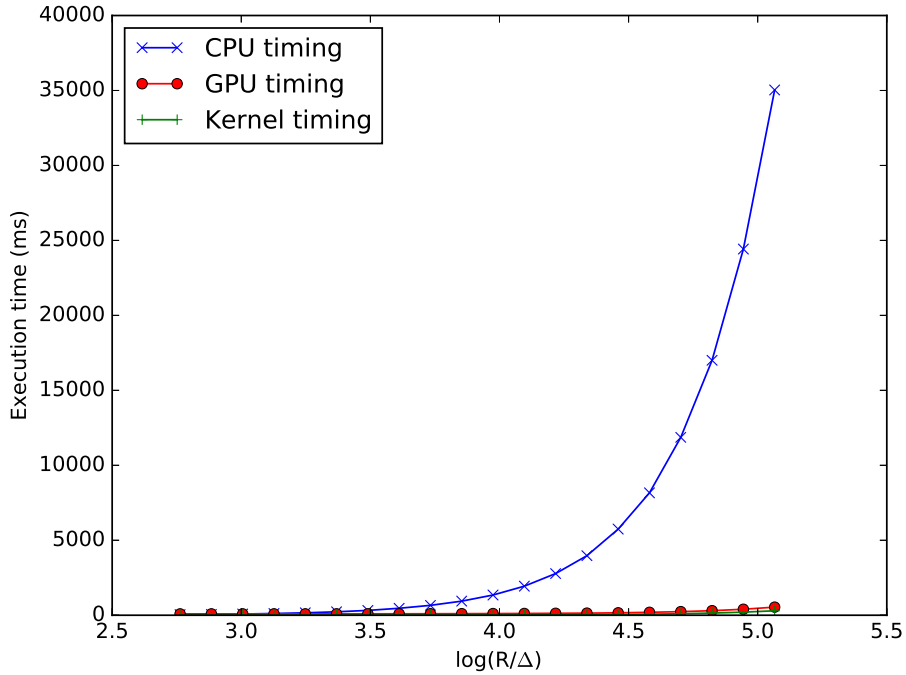



Figure 2.14: Timing comparison for torus surface area computation in wavelet genus 3.

```
6  const int s = threadIdx.z + blockDim.z * blockIdx.z;
7  if (c >= dim->x || r >= dim->y || s >= dim->z) return;
8  const int3 pt_crs = make_int3(c, r, s);
9  const int i = flatten(pt_crs, *dim);
10 const int s_c = threadIdx.x + rad;
11 const int s_r = threadIdx.y + rad;
12 const int s_s = threadIdx.z + rad;
13 const dim3 sBlockSz = dim3(blockDim.x + 2 * rad, blockDim.y + 2 * rad,
14   blockDim.z + 2 * rad);
15 const int s_i = flatten(make_int3(s_c, s_r, s_s), sBlockSz);
16 extern __shared__ double s_block[];
17 double *s_stencil = s_block + sBlockSz.x * sBlockSz.y * sBlockSz.z;
18 double *s_blockSum = s_stencil + 2 * rad + 1;
19
20 // set to 0 once per shared block
21 if (s_c == rad && s_r == rad && s_s == rad) {
22   for (int k = 0; k < 2 * rad + 1; k++) {
23     s_stencil[k] = stencil[k];
24   }
25   *s_blockSum = 0.0;
26 }
27
28 // Regular cells
29 s_block[s_i] = grid[i];
30
31 // Halo cells
32 if (threadIdx.x < rad) {
33   s_block[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
34     grid[flatten(make_int3(c - rad, r, s), *dim)];
35   s_block[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
36     grid[flatten(make_int3(c + blockDim.x, r, s), *dim)];
37 }
38
```

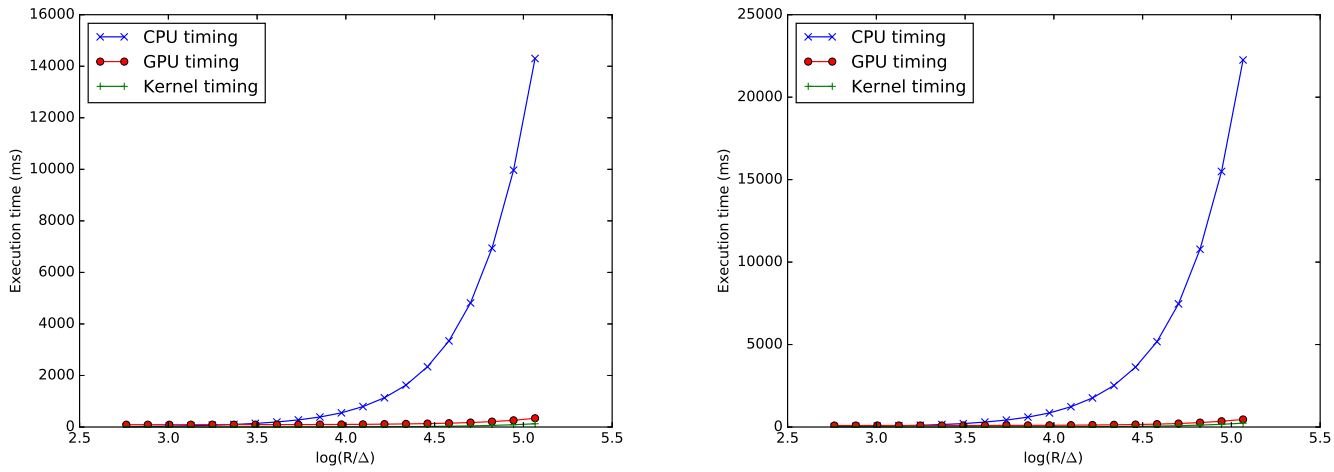
```
39  if (threadIdx.y < rad) {
40    s_block[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
41      grid[flatten(make_int3(c, r - rad, s), *dim)];
42    s_block[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
43      grid[flatten(make_int3(c, r + blockDim.y, s), *dim)];
44  }
45
46  if (threadIdx.z < rad) {
47    s_block[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
48      grid[flatten(make_int3(c, r, s - rad), *dim)];
49    s_block[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
50      grid[flatten(make_int3(c, r, s + blockDim.z), *dim)];
51  }
52
53  __syncthreads();
54
55  if ((c >= dim->x - rad) || (r >= dim->y - rad) || (s >= dim->z - rad) ||
56      c <= (rad - 1) || r <= (rad - 1) || s <= (rad - 1)) return;
57
58  const int3 s_pt = make_int3(s_c, s_r, s_s);
59
60  double3 df = make_double3(
61    xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
62    yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
63    zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0));
64
65  double3 dchi = make_double3(
66    xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
67    yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
68    zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0));
69
70  double denom = computeMagnitude(df);
71
```

```
72   if (denom > EPS) {
73     double3 xyz = crs2xyz(pt_crs, delta, *lim);
74     double res = ((xyz.x * xyz.x) + (xyz.y * xyz.y)) * dotProduct(df, dchi);
75     atomicAdd(s_blockSum, res / denom);
76   }
77
78   __syncthreads();
79
80   // add only once per shared block
81   if (s_c == rad && s_r == rad && s_s == rad) {
82     atomicAdd(moiSum, *s_blockSum);
83   }
84 }
```

Again similar to the previous subsection, the error results are identical to the serial version and the difference between the two is timings. CPU vs. GPU timing plots are given in Figures 2.15 and 2.16. These plots again exhibit a very similar trend to the previous subsection and depending on the $\log(R/\Delta)$ value, show that CUDA gives a performance boost of up to about 100 times.

### *2.4.3 Flame surface computation on the GPU*

The final CUDA implementation of this chapter involves the flame surface computation. In contrast to the previous two problems where the data was obtained through an implicit function, this data set is the result of direct numerical simulation (DNS). The implicit functions were constructed so that the surface level set was separated from the grid boundary by more than the stencil radius. There is no such control over the DNS data, and the surface level set intersects the grid boundary. The DNS data set should be treated as periodic in each coordinate direction, and the current shared memory implementation is not designed to handle periodic data sets. Therefore, we employ a straightforward global memory implementation in this case. Listing 2.6 shows the full implementation of the kernel function and the




(a) (b)

Figure 2.15: Timing comparison for torus moment of inertia computation in wavelet genus 1 (a) and 2 (b).

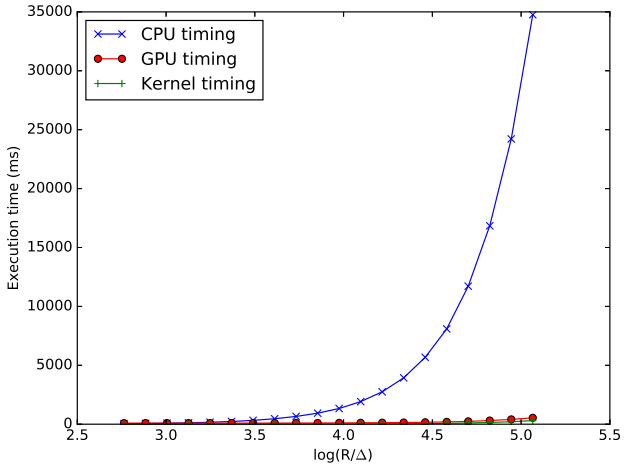



Figure 2.16: Timing comparison for torus moment of inertia computation in wavelet genus 3.

| | CPU | GPU | Kernel |
|---|---|---|---|
| Mean Execution Time (ms) | 3727.7 | 141.6 | 29.8 |
| Standard Deviation (ms) | 19.3 | 4.1 | 1.8 |
| Median Execution Time (ms) | 3722.2 | 140.8 | 28.7 |

Table 2.2: Flame surface area computation execution times.

wrapper function in the `kernel.cu` file. Notice how short the kernel function is compared to the previous shared memory implementations. This time, since we cannot change the grid spacing as it is real-life data with fixed grid size, instead of plotting Execution time vs. $\log(R/\Delta)$, the test has been run with the same code 100 times and the timings are analyzed statistically. Results are shown in Table 2.2.

The performance analysis given in Table 2.1 show that GPU-based parallel implementation shows significant performance gains on the order of 25x. If further performance gain is desired, texture memory would be a candidate basis for accelerated implementations [11].

Listing 2.6: Flame surface computation on the GPU

```
1 __global__
2 void surfAreaKernel(double *areaSum, const double *grid, const
3   double *stencil, int rad, const double3 *lim, double delta,
4   const dim3 *dim) {
5   const int c = threadIdx.x + blockDim.x * blockIdx.x;
6   const int r = threadIdx.y + blockDim.y * blockIdx.y;
7   const int s = threadIdx.z + blockDim.z * blockIdx.z;
8   if (c >= dim->x || r >= dim->y || s >= dim->z) return;
9   const int3 pt = make_int3(c, r, s);
10
11  double3 df = make_double3(
12    xDeriv(grid, stencil, rad, pt, delta, *dim, id, 0.0),
```

```
13      yDeriv(grid, stencil, rad, pt, delta, *dim, id, 0.0),
14      zDeriv(grid, stencil, rad, pt, delta, *dim, id, 0.0));
15
16    double3 dchi = make_double3(
17      xDeriv(grid, stencil, rad, pt, delta, *dim, sgn, 0.0),
18      yDeriv(grid, stencil, rad, pt, delta, *dim, sgn, 0.0),
19      zDeriv(grid, stencil, rad, pt, delta, *dim, sgn, 0.0));
20
21    double denom = computeMagnitude(df);
22    if (denom > EPS) atomicAdd(areaSum, dotProduct(df, dchi) / denom);
23  }
24
25  double surfArea(const double *grid, const double *stencil, int rad,
26    const double3 &lim, double delta, const dim3 &dim) {
27    const int problemSize = dim.x * dim.y * dim.z;
28    double areaSum = 0.0;
29    double *d_areaSum = 0, *d_grid = 0, *d_stencil;
30    dim3 *d_dim = 0;
31    double3 *d_lim = 0;
32
33    cudaMalloc(&d_areaSum, sizeof(double));
34    cudaMemset(d_areaSum, 0, sizeof(double));
35
36    cudaMalloc(&d_grid, problemSize * sizeof(double));
37    cudaMemcpy(d_grid, grid, problemSize * sizeof(double),
38      cudaMemcpyHostToDevice);
39
40    cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
41    cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
42      cudaMemcpyHostToDevice);
43
44    cudaMalloc(&d_dim, sizeof(dim3));
45    cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
```

```
46
47   cudaMalloc(&d_lim, sizeof(double3));
48   cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
49
50   const dim3 blockSize(TX, TY, TZ);
51   const dim3 gridSize(divUp(dim.x, TX), divUp(dim.y, TY), divUp(dim.z, TZ));
52
53   surfAreaKernel<<<gridSize, blockSize>>>(d_areaSum, d_grid, d_stencil, rad,
54     d_lim, delta, d_dim);
55
56   cudaMemcpy(&areaSum, d_areaSum, sizeof(double), cudaMemcpyDeviceToHost);
57   cudaFree(d_areaSum);
58   cudaFree(d_grid);
59   cudaFree(d_stencil);
60   cudaFree(d_dim);
61   cudaFree(d_lim);
62   return std::pow(delta, 3) / 2.0 * areaSum;
63 }
```

# Chapter 3

# VOLUME INTEGRATION OF DIGITIZED OBJECTS

Having considered integrals on surfaces of 3D solids (on surfaces of codimension 1) in Chapter 2, we now consider direct, grid-based approaches to computing volume integrals over regions of codimension 0. (Line integrals over regions of codimension 2 will be considered below in the next chapter.)

In volume integration, current practice typically involves Monte Carlo methods [29] or classic voxel or octree methods that accumulate contributions from elements [30, 31]. Looking at the classically used approaches, the choice of method depends on the given conditions. If the boundary is complicated, and the integrand that needs to be calculated does not have sudden peaks, the usual approach is to use Monte Carlo integration. If the boundary is simple and the integrand function is smooth, then the volume integral can be broken down to 1D integrals [32]. The problem with both of these approaches is that they require the bounds of the shape to be explicitly defined. If the the bounds are not explicitly given, but the shape is defined by a distance function or by a function that satisfies the Lipschitz condition [33], cube membership classification with refinement using octree classification is another efficient solution [31].

Here we apply our direct integration approach to volume integrals. As is the case with surface integration, this discrete approach makes the volume integration applicable to grids of data. The approach is once again employs the Divergence theorem and applies for all integrands which can be written as the divergence of a vector function which includes all polynomial integrands. In particular, this approach applies for computing all moment integrals including volume, centroid, and moments of inertia.

### *3.1 Theory*

The volume integral of function $g(\boldsymbol{r})$ over a region $\Omega$ defined by function $f$ sampled on a uniform grid with spacing $\Delta$, is given by

$$V(\Omega) = \int_{\Omega} g(\boldsymbol{r}) dv \tag{3.1}$$

This equation can be put in the form of Equation 2.9 for any $g(\boldsymbol{r})$ that can be expressed as the divergence of a vector potential $\boldsymbol{u} = u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k}$, i.e., $g = \nabla \cdot \boldsymbol{u}$, so that

$$V(\Omega) = \int_{\Omega} (\nabla \cdot \boldsymbol{u}) dv \tag{3.2}$$

Extending the integration to $\mathbb{R}^3$ using the occupancy function from Equation 2.1, we get

$$V(\Omega) = \int_{\mathbb{R}^3} (\chi \nabla \cdot \boldsymbol{u}) dv \tag{3.3}$$

Using the product rule for the gradient for a product of a scalar and a vector just like we did in going from Equation 2.6 to Equation 2.9, we obtain

$$V(\Omega) = -\int_{\mathbb{R}^3} (\nabla \chi \cdot \boldsymbol{u}) dv \tag{3.4}$$

Discretization on the uniform grid gives

$$V(\Omega) = \frac{\Delta^3}{2} \sum_{i,j,k} \left[ u_x \Big[ \frac{\partial}{\partial x} sgn(f) \Big] + u_y \Big[ \frac{\partial}{\partial y} sgn(f) \Big] + u_z \Big[ \frac{\partial}{\partial z} sgn(f) \Big] \right]_{i,j,k} \tag{3.5}$$

where derivatives are to be evaluated with the scheme shown in Equation 2.12.

For the particular case of computing the volume, $g = 1$, and we can choose $\boldsymbol{u}(\boldsymbol{r}) = \frac{1}{3}(x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k})$ so that $\nabla \cdot \boldsymbol{u} = g = 1$ and, we obtain the following discretized expression for the volume

$$V(\Omega) = \frac{\Delta^3}{6} \sum_{i,j,k} \left[ x \Big[ \frac{\partial}{\partial x} sgn(f) \Big] + y \Big[ \frac{\partial}{\partial y} sgn(f) \Big] + z \Big[ \frac{\partial}{\partial z} sgn(f) \Big] \right]_{i,j,k} \tag{3.6}$$

This choice of $\boldsymbol{u}(\boldsymbol{r})$ is not unique and the volume could be computed equally well with other choice of $\boldsymbol{u}$ such as $\boldsymbol{u}(\boldsymbol{r}) = x\boldsymbol{i}$.

### *3.2 Implementation*

With the goal of using our general volume integral to compute the volume of a given DG-Rep, we start with Equation 2.12 and the specific volume integral formula given in Equation 3.6, and following similar arguments from section 2.2, we create a function in C++ that implements the equation derived in Section 3.1. The function, named `volumeInt`, accepts a DG-Rep of `double` numbers, named `grid`, the stencil for the derivative operations, `stencil`, the radius of this stencil, `rad`, the real-world limits of volume that contains the DG-Rep, `lim`, a `double` parameter for spacing named `delta`, and the number of points in the volumetric grid, `dim`. The function returns a `double`, which is the result of the volume integration.

Listing 3.1: Implementation of volume integral function for volume computation

```
double volumeInt(const double *grid, const double *stencil, int rad,
  const double3 &lim, double delta, const dim3 &dim) {
  double volSum = 0.0;
  for (int s = 0; s < dim.z; s++) {
    for (int r = 0; r < dim.y; r++) {
      for (int c = 0; c < dim.x; c++) {
        int3 pt = make_int3(c, r, s);
        double3 dchi = make_double3(
          xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
        double3 xyz = crs2xyz(pt, delta, lim);
        volSum += dotProduct(dchi, xyz) * std::pow(delta, 3);
      }
    }
  }
  return volSum / 6.0;
}
```

## 3.3 Results and Discussion

### 3.3.1 Volume of a solid torus

The volume of a solid torus can be analytically calculated with the formula

$$V = 2\pi^2 r^2 R \tag{3.7}$$

Using the C++ code presented in Appendix A in Listing A.7, the volume integral function presented in Listing 3.1 has been applied to a torus that has been implicitly defined. For comparison purposes, Monte-Carlo computation is also run on the same torus and also provided in the code listing. In order to be able to make a proper comparison between a grid-based method and Monte-Carlo method, instead of grid spacing, the computational error is considered as a function of the number of evaluation points. Figure 3.1 shows the comparison between the two methods. The green dashed line represents the exact value. The blue crosses show sample Monte-Carlo results for numbers of sample points in the range about 7 thousand to approximately 7 million. For smaller number of points, there is large variation in the Monte-Carlo results; the variation decreases for larger number of points, but can be still visible at maximum number of points. The red dots show the results of the direct integration grids with the same range of total number of points. The method is deterministic, and it can be seen that the results converge to the exact value faster. While direct integration method starts to settle around 1 million sample points, Monte-Carlo still shows some jumps at around 1 million and higher number of sample points.

### 3.3.2 Solid Torus moment of inertia

Here, we apply the direct volume integration method to compute something different than the volume itself. For this, we chose moment of inertia for the solid torus. In Section 2.3.2 of Chapter 2, the moment of inertia about the axis passing through the center of a hollow torus (assumed to be $I_z$ here) was computed and its convergence behavior was studied. In this section, we to de the same for a solid torus. In order to be able to compute solid

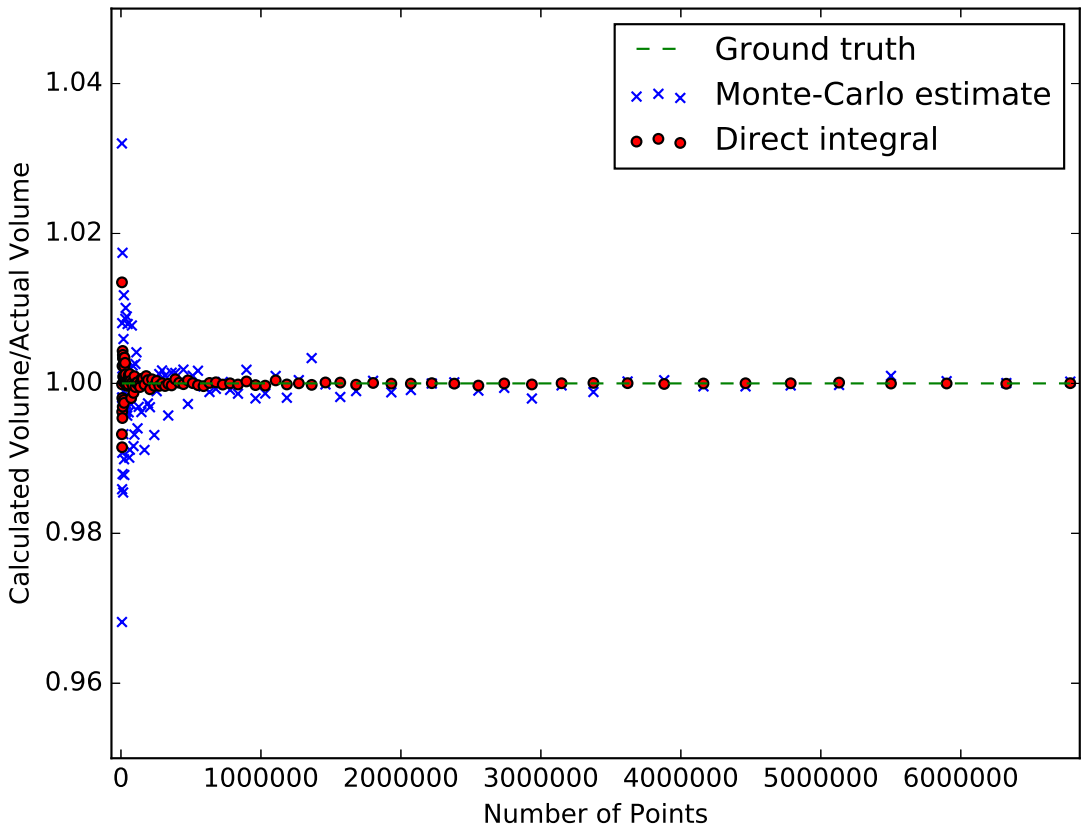



Figure 3.1: Computed Volume / Actual Volume vs. Number of points for a torus. Direct integration settles around 1 million points whereas the Monte-Carlo estimate still displays some bigger jumps.

torus moment of inertia, we start with Equation 3.5, and since $g(\boldsymbol{r})$ from Equation 3.2 is $g(\boldsymbol{r}) = x^2 + y^2$, $\boldsymbol{u}(\boldsymbol{r})$ can be chosen as

$$\boldsymbol{u}(\boldsymbol{r}) = \frac{x^3\boldsymbol{i} + y^3\boldsymbol{j}}{3} \tag{3.8}$$

where $\nabla \cdot \boldsymbol{u} = g = x^2 + y^2$. Using this, Listing 3.1 can be modified as shown in Listing 3.2

Listing 3.2: Implementation of volume integral function for moment of inertia computation

```
double momentOfInertia(const double *grid, const double *stencil, int rad,
  const double3 &lim, double delta, const dim3 &dim) {
  double inertiaSum = 0.0;
  for (int s = 0; s < dim.z; s++) {
    for (int r = 0; r < dim.y; r++) {
      for (int c = 0; c < dim.x; c++) {
        int3 pt = make_int3(c, r, s);
        double3 dchi = make_double3(
          xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
        double3 xyz = crs2xyz(pt, delta, lim);
        // g = x^2 + y^2 => u = (x^3 i + y^3 j) / 3
        xyz.x = xyz.x * xyz.x * xyz.x;
        xyz.y = xyz.y * xyz.y * xyz.y;
        xyz.z = 0;
        inertiaSum += dotProduct(dchi, xyz);
      }
    }
  }
  return inertiaSum / 6.0 * std::pow(delta, 3);
}
```

The moment of inertia about the axis passing through its center for a torus is [34]

$$I_z = \frac{1}{4}m\Big(4R^2 + 3r^2\Big) \tag{3.9}$$

where $m$ is the mass, $R$ is the major radius, and $r$ is the minor radius. Assuming unit mass per volume, combining the volume formula from Equation 3.7, the inertia formula for solid torus becomes

$$I_z = \frac{1}{2}\pi^2 r^2 R\Big(4R^2 + 3r^2\Big). \tag{3.10}$$

Using this formula's result as the exact value, solving the moment of inertia integral for different $R/\Delta$ values, the convergence of the result is plotted in Figure 3.2. Listing A.8 in Appendix A contains the C++ implementation of moment of inertia integration for the solid torus.

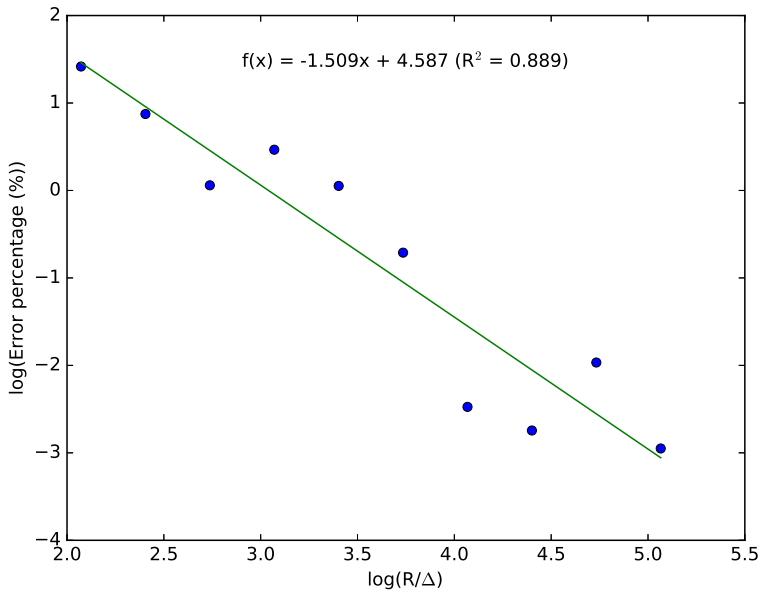



Figure 3.2: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for solid torus moment of inertia, $I_z$

## 3.4 GPU Implementation and Performance Analysis

As it was done in the previous Chapter, here we discuss CUDA implementation and parallel performance of each of the problems that were solved serially.

### *3.4.1 Torus volume computation on the GPU*

In Listing 3.3, we provide the kernel and the wrapper code. While mostly similar to Listing 2.3 this time, the kernel does volume summation as shown in lines 65 and 66, and in the end the sum is processed according to the Equation 3.6 in line 115.

Listing 3.3: Torus volume kernel code

```
1 __global__
2 void volIntKernel(double *volSum, const double *grid, const double *stencil,
3   int rad, const double3 *lim, double delta, const dim3 *dim) {
4   const int c = threadIdx.x + blockDim.x * blockIdx.x;
5   const int r = threadIdx.y + blockDim.y * blockIdx.y;
6   const int s = threadIdx.z + blockDim.z * blockIdx.z;
7   if (c >= dim->x || r >= dim->y || s >= dim->z) return;
8   const int3 pt_crs = make_int3(c, r, s);
9   const int i = flatten(pt_crs, *dim);
10  const int s_c = threadIdx.x + rad;
11  const int s_r = threadIdx.y + rad;
12  const int s_s = threadIdx.z + rad;
13  const dim3 sBlockSz = dim3(blockDim.x + 2 * rad, blockDim.y + 2 * rad,
14    blockDim.z + 2 * rad);
15  const int s_i = flatten(make_int3(s_c, s_r, s_s), sBlockSz);
16  extern __shared__ double s_block[];
17  double *s_stencil = s_block + sBlockSz.x * sBlockSz.y * sBlockSz.z;
18  double *s_blockSum = s_stencil + 2 * rad + 1;
19
20  // set to 0 once per shared block
```

```
21  if (s_c == rad && s_r == rad && s_s == rad) {
22    for (int k = 0; k < 2 * rad + 1; k++) {
23      s_stencil[k] = stencil[k];
24    }
25    *s_blockSum = 0.0;
26  }
27
28  // Regular cells
29  s_block[s_i] = grid[i];
30
31  // Halo cells
32  if (threadIdx.x < rad) {
33    s_block[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
34      grid[flatten(make_int3(c - rad, r, s), *dim)];
35    s_block[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
36      grid[flatten(make_int3(c + blockDim.x, r, s), *dim)];
37  }
38
39  if (threadIdx.y < rad) {
40    s_block[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
41      grid[flatten(make_int3(c, r - rad, s), *dim)];
42    s_block[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
43      grid[flatten(make_int3(c, r + blockDim.y, s), *dim)];
44  }
45
46  if (threadIdx.z < rad) {
47    s_block[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
48      grid[flatten(make_int3(c, r, s - rad), *dim)];
49    s_block[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
50      grid[flatten(make_int3(c, r, s + blockDim.z), *dim)];
51  }
52
53  __syncthreads();
```

```
54
55    if ((c >= dim->x - rad) || (r >= dim->y - rad) || (s >= dim->z - rad) ||
56        c <= (rad - 1) || r <= (rad - 1) || s <= (rad - 1)) return;
57
58    const int3 s_pt = make_int3(s_c, s_r, s_s);
59
60    double3 dchi = make_double3(
61      xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
62      yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
63      zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0));
64
65    double3 xyz = crs2xyz(pt_crs, delta, *lim);
66    atomicAdd(s_blockSum, dotProduct(dchi, xyz));
67
68    __syncthreads();
69
70    // add only once per shared block
71    if (s_c == rad && s_r == rad && s_s == rad) {
72      atomicAdd(volSum, *s_blockSum);
73    }
74  }
75
76  double volumeInt(const double *grid, const double *stencil, int rad,
77    const double3 &lim, double delta, const dim3 &dim) {
78    const int problemSize = dim.x * dim.y * dim.z;
79    double volSum = 0.0;
80    double *d_volSum = 0, *d_grid = 0, *d_stencil;
81    dim3 *d_dim = 0;
82    double3 *d_lim = 0;
83
84    cudaMalloc(&d_volSum, sizeof(double));
85    cudaMemset(d_volSum, 0, sizeof(double));
86
```

```
87   cudaMalloc(&d_grid, problemSize * sizeof(double));
88   cudaMemcpy(d_grid, grid, problemSize * sizeof(double),
89     cudaMemcpyHostToDevice);
90
91   cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
92   cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
93     cudaMemcpyHostToDevice);
94
95   cudaMalloc(&d_dim, sizeof(dim3));
96   cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
97
98   cudaMalloc(&d_lim, sizeof(double3));
99   cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
100
101  const dim3 blockSize(TX, TY, TZ);
102  const dim3 gridSize(divUp(dim.x, TX), divUp(dim.y, TY), divUp(dim.z, TZ));
103  const size_t smSz = ((TX + 2 * rad) * (TY + 2 * rad) * (TZ + 2 * rad) +
104                       2 * rad + 1 + 1) * sizeof(double);
105
106  volIntKernel<<<gridSize, blockSize, smSz>>>(d_volSum, d_grid, d_stencil,
107    rad, d_lim, delta, d_dim);
108
109  cudaMemcpy(&volSum, d_volSum, sizeof(double), cudaMemcpyDeviceToHost);
110  cudaFree(d_volSum);
111  cudaFree(d_grid);
112  cudaFree(d_stencil);
113  cudaFree(d_dim);
114  cudaFree(d_lim);
115  return std::pow(delta, 3) / 6.0 * volSum;
116 }
```

Figures 3.3 and 3.4 show the results of performance tests. It should also be noted that this time, instead of using $R/\Delta$ in the x-axis, the number of points in the grid is used. The

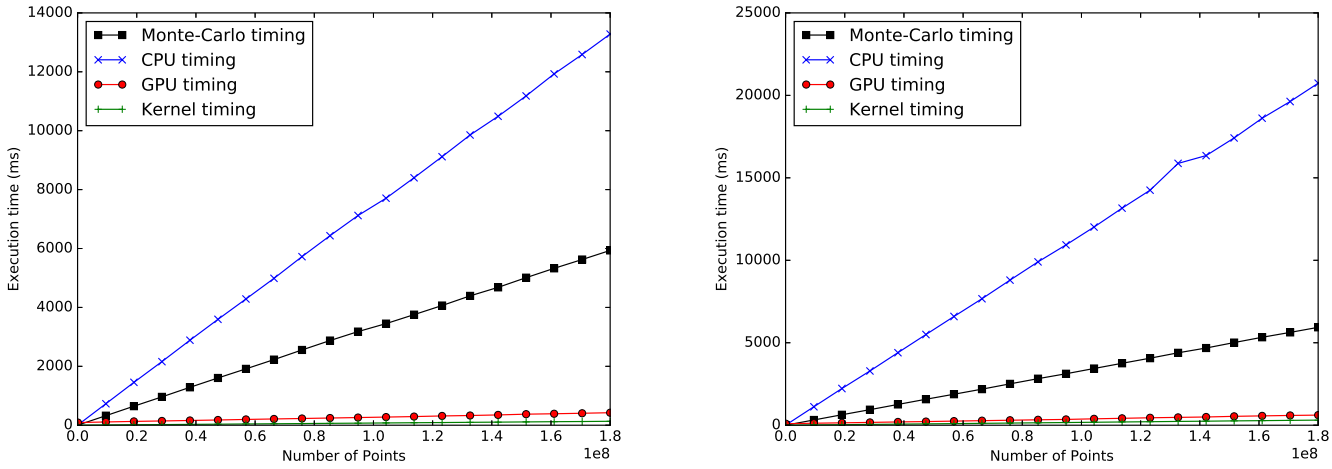



Figure 3.3: Timing comparison for torus volume in wavelet genus 1 (a) and 2 (b).

reason behind this is the inclusion of Monte-Carlo method timings. While CPU and GPU results are identical, Monte-Carlo results are different and usually slightly less accurate as previously seen in Figure 3.1. This can be seen as comparing apples and oranges, however, for timing related discussions, it gives a good idea about the efficiency of the GPU computations. At the highest loading configuration with about 180 million points and Daubechies wavelet genus 3 stencil, direct integration on the CPU takes about 40,000 ms to complete whereas the Monte-Carlo computation on the CPU takes about 7,200 ms, and the direct integration on the GPU takes about 800 ms (kernel time about 500 ms). This shows about 50x speed up compared to the same CPU implementation and about 9x speed up (as well as showing better accuracy and deterministic results) compared to Monte-Carlo computation.

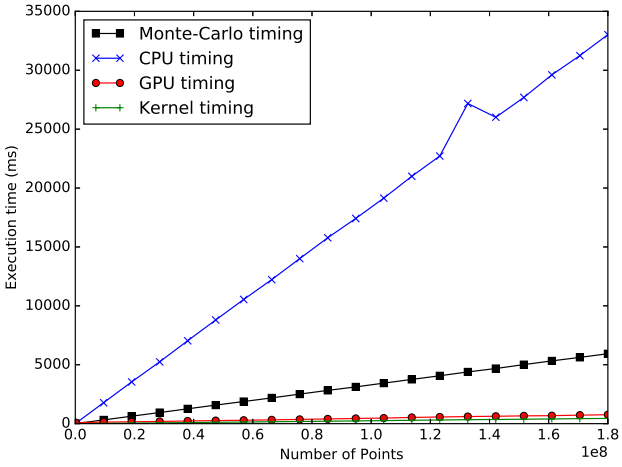


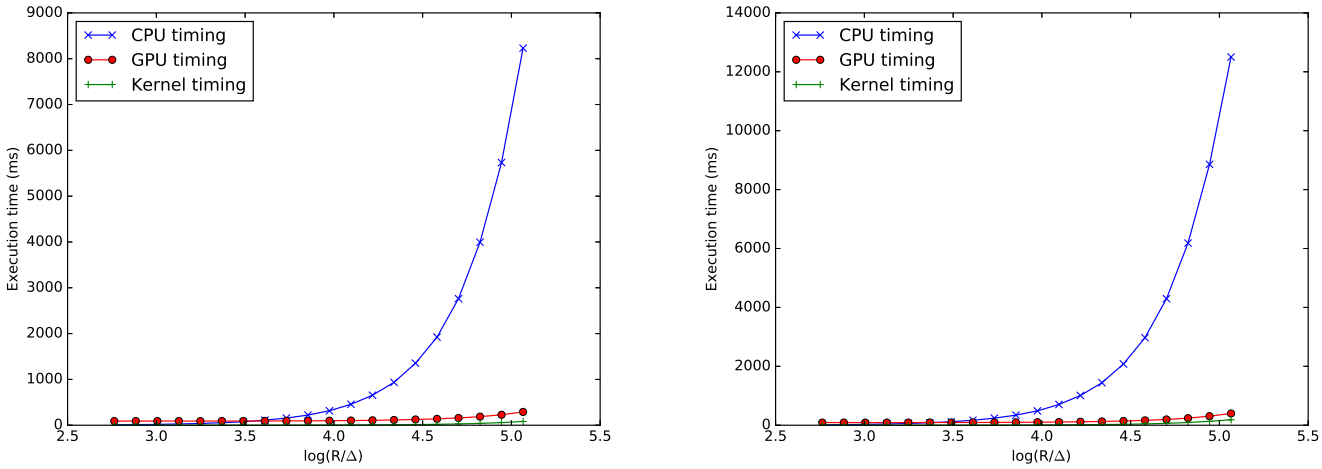

Figure 3.4: Timing comparison for torus volume in wavelet genus 3.

### *3.4.2 Solid torus moment of inertia computation on the GPU*

The wrapper function and most of the kernel (such as memory operations) in the `.cu` file is identical to the ones in Listing 3.3. Listing 3.4 shows the relevant part of the kernel function where the moment of inertia around the z-axis is being computed due to the change in the function $g$ from $g = 1$ to $g = x^2 + y^2$.

Listing 3.4: Solid torus moment of inertia changes from the volume kernel

```
1  double3 dchi = make_double3(
2    xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
3    yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
4    zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0));
5  double3 xyz = crs2xyz(pt_crs, delta, *lim);
6  xyz.z = 0;
7  xyz.x = xyz.x * xyz.x * xyz.x;
8  xyz.y = xyz.y * xyz.y * xyz.y;
9  atomicAdd(s_blockSum, dotProduct(dchi, xyz));
10
```




(a) (b)

Figure 3.5: Timing comparison for solid torus moment of inertia in wavelet genus 1 (a) and 2 (b).

```
11  __syncthreads();
12
13  // add only once per shared block
14  if (s_c == rad && s_r == rad && s_s == rad) {
15    atomicAdd(moiSum, *s_blockSum);
```

Figures 3.5 and 3.6 show the results of performance tests. These plots again show similar results to previous sections and GPU code gives a performance boost of up to about 100 times depending on the $\log(R/\Delta)$ value.

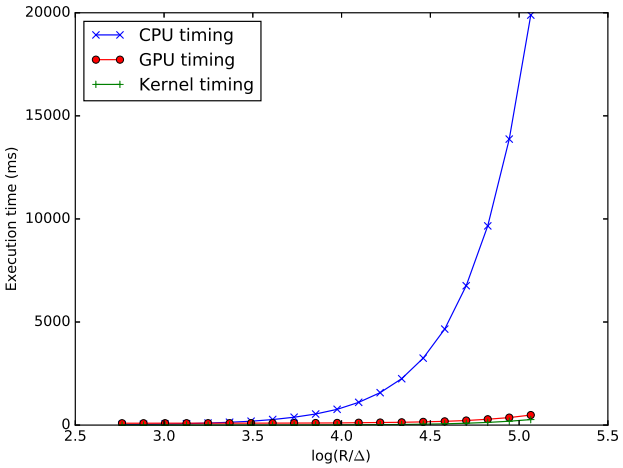



Figure 3.6: Timing comparison for solid torus moment of inertia in wavelet genus 3.

# Chapter 4

# LINE INTEGRATION OF DIGITIZED OBJECTS

Here we consider computation of line integrals (over regions of codimension 2). In contrast to traditional parametrized modeling methods, we address the case in which a parametrization of the path is not given. Instead, two implicit functions (or grids of values for two implicit functions) are given, and the path corresponds to the intersection of the two codimension 1 surfaces defined by the functions. Current approaches involve attempts to create a parametrized path by (1) creating a spline to approximate the intersection of the surfaces (which may not be feasible especially for digitized functions) or (2) polygonizing each implicit surface and attempting to compute the intersection of the surface polygonizations. Given that polygonization is known to be unreliable for computing surface properties, it is definitely not safe to expect that intersections of polygonizations will provide a reliable basis for computation of line integrals.

## *4.1 Theory*

The line integral of a scalar function $g(\boldsymbol{r})$, along a curve which is the intersection of two surfaces, $\partial\Omega_1$ and $\partial\Omega_2$ is given by

$$L(\partial\Omega_1 \cap \partial\Omega_2) = \int_{\partial\Omega_1 \cap \partial\Omega_2} g(\boldsymbol{r}) dl \tag{4.1}$$

which can also be written using the definition of unit tangent vector, $\boldsymbol{t} \cdot \boldsymbol{t} = 1$, where $\boldsymbol{t} = \boldsymbol{n}_1 \times \boldsymbol{n}_2$, as

$$L(\partial\Omega_1 \cap \partial\Omega_2) = \int_{\partial\Omega_1 \cap \partial\Omega_2} g(\boldsymbol{r}) \boldsymbol{t} \cdot \boldsymbol{t}\, dl \tag{4.2}$$

applying the Stokes' Theorem [24] on Equation 4.2, surface integral is obtained as

$$L(\partial\Omega_1 \cap \partial\Omega_2) = \int_{\partial\Omega_1 \cap \Omega_2} (\nabla \times g(\boldsymbol{r}) \boldsymbol{t}) \cdot \boldsymbol{n}_1\, ds \tag{4.3}$$

introducing the occupancy function $\chi_2$, for the volume, $\Omega_2$

$$L(\partial\Omega_1 \cap \partial\Omega_2) = \int_{\partial\Omega_1} \chi_2(\nabla \times g(\boldsymbol{r})\boldsymbol{t}) \cdot \boldsymbol{n}_1 \, ds \tag{4.4}$$

at this step, following similar steps from previous chapters that were used to obtain Equation 2.9, starting from Equation 2.5, the resulting integral is

$$L(\partial\Omega_1 \cap \partial\Omega_2) = -\int_{\mathbb{R}^3} \nabla\chi_1 \cdot (\chi_2 \nabla \times \boldsymbol{t}) g(\boldsymbol{r}) \, dv \tag{4.5}$$

where, it should be noted that by symmetry argument, the occupancy functions can be exchanged. Steps needed for discretization of Equation 4.5 are

$$\chi_1(\boldsymbol{r}) = [1 - sgn(f_1(\boldsymbol{r}))], \tag{4.6}$$

$$\chi_2(\boldsymbol{r}) = [1 - sgn(f_2(\boldsymbol{r}))], \tag{4.7}$$

and

$$\boldsymbol{t} = \boldsymbol{n}_1 \times \boldsymbol{n}_2 = \frac{\nabla f_1 \times \nabla f_2}{|\nabla f_1||\nabla f_2|}. \tag{4.8}$$

Hence

$$\nabla\chi_1(\boldsymbol{r}) = -\frac{1}{2}\left[\frac{\partial}{\partial x} sgn(f_1)\boldsymbol{i} + \frac{\partial}{\partial y} sgn(f_1)\boldsymbol{j} + \frac{\partial}{\partial z} sgn(f_1)\boldsymbol{k}\right], \tag{4.9}$$

and

$$\nabla \times \boldsymbol{t} = \left(\frac{\partial t_z}{\partial y} - \frac{\partial t_y}{\partial z}\right)\boldsymbol{i} + \left(\frac{\partial t_x}{\partial z} - \frac{\partial t_z}{\partial x}\right)\boldsymbol{j} + \left(\frac{\partial t_y}{\partial x} - \frac{\partial t_x}{\partial y}\right)\boldsymbol{k}, \tag{4.10}$$

with

$$\boldsymbol{t} = \frac{\left(\frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial z} - \frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial y}\right)\boldsymbol{i} + \left(\frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial z}\right)\boldsymbol{j} + \left(\frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial y} - \frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial x}\right)\boldsymbol{k}}{\left|\left(\frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial z} - \frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial y}\right)\boldsymbol{i} + \left(\frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial z}\right)\boldsymbol{j} + \left(\frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial y} - \frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial x}\right)\boldsymbol{k}\right|}, \tag{4.11}$$

thus

$$\frac{\partial t_x}{\partial y} = \frac{\partial}{\partial y}\frac{\left(\frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial z} - \frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial y}\right)}{\left|\left(\frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial z} - \frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial y}\right)\boldsymbol{i} + \left(\frac{\partial f_1}{\partial z}\frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial z}\right)\boldsymbol{j} + \left(\frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial y} - \frac{\partial f_1}{\partial y}\frac{\partial f_2}{\partial x}\right)\boldsymbol{k}\right|} \tag{4.12}$$

and so on for the other derivatives. With these derivatives calculated, the line integral can be rewritten as

$$L(\partial\Omega_1 \cap \partial\Omega_2) = \frac{1}{4}\int_{\mathbb{R}^3}\left[\left(\frac{\partial}{\partial x}sgn(f_1)\right)\left(\frac{\partial t_z}{\partial y}-\frac{\partial t_y}{\partial z}\right)+\left(\frac{\partial}{\partial y}sgn(f_1)\right)\left(\frac{\partial t_x}{\partial z}-\frac{\partial t_z}{\partial x}\right)+\left(\frac{\partial}{\partial z}sgn(f_1)\right)\left(\frac{\partial t_y}{\partial x}-\frac{\partial t_x}{\partial y}\right)\right](1-sgn(f_2))g(\boldsymbol{r})dv \tag{4.13}$$

and Equation 4.13 can be discretized as

$$L(\partial\Omega_1 \cap \partial\Omega_2) = \frac{\Delta^3}{4}\sum_{i,j,k}\left[\left(\frac{\partial}{\partial x}sgn(f_1)\right)\left(\frac{\partial t_z}{\partial y}-\frac{\partial t_y}{\partial z}\right)+\left(\frac{\partial}{\partial y}sgn(f_1)\right)\left(\frac{\partial t_x}{\partial z}-\frac{\partial t_z}{\partial x}\right)+\left(\frac{\partial}{\partial z}sgn(f_1)\right)\left(\frac{\partial t_y}{\partial x}-\frac{\partial t_x}{\partial y}\right)\right](1-sgn(f_2))g_{i,j,k}. \tag{4.14}$$

where derivatives can be evaluated with the scheme shown in Equation 2.12.

## *4.2 Implementation*

Starting with Equations 2.12 and 4.14, and following similar arguments from section 2.2, a function is written in C++ that implements the equation derived in Section 4.1. The function, named `lineIntegral`, accepts two grids of `double` numbers, named `f1` and `f2` which are the digitized versions of two different objects. The intersection of the objects is the result of the line integration. Listing 4.1 shows the implementation of this function.

Listing 4.1: Implementation of Line Integral function

```
1 double lineIntegral(const double *f1, const double *f2, const double *stencil,
2  int rad, const double3 &lim, double delta, const dim3 &dim) {
3   int problemSize = dim.x * dim.y * dim.z;
4   double lineIntSum = 0.0;
5   double *tx = new double[problemSize]();
```

```
6  double *ty = new double[problemSize]();
7  double *tz = new double[problemSize]();
8  for (int s = 0; s < dim.z; s++) {
9    for (int r = 0; r < dim.y; r++) {
10     for (int c = 0; c < dim.x; c++) {
11       int3 pt = make_int3(c, r, s);
12       double3 df1 = make_double3(
13         xDeriv(f1, stencil, rad, pt, delta, dim, id, 0.0),
14         yDeriv(f1, stencil, rad, pt, delta, dim, id, 0.0),
15         zDeriv(f1, stencil, rad, pt, delta, dim, id, 0.0));
16
17       double3 df2 = make_double3(
18         xDeriv(f2, stencil, rad, pt, delta, dim, id, 0.0),
19         yDeriv(f2, stencil, rad, pt, delta, dim, id, 0.0),
20         zDeriv(f2, stencil, rad, pt, delta, dim, id, 0.0));
21
22       double3 t_ijk = make_double3(df1.y * df2.z - df1.z * df2.y,
23         df1.z * df2.x - df1.x * df2.z, df1.x * df2.y - df1.y * df2.x);
24
25       double magnitude = computeMagnitude(t_ijk);
26
27       if (magnitude < EPS) continue;
28
29       tx[flatten(pt, dim)] = t_ijk.x / magnitude;
30       ty[flatten(pt, dim)] = t_ijk.y / magnitude;
31       tz[flatten(pt, dim)] = t_ijk.z / magnitude;
32     }
33   }
34 }
35
36 for (int s = 0; s < dim.z; s++) {
37   for (int r = 0; r < dim.y; r++) {
38     for (int c = 0; c < dim.x; c++) {
```

```
39          int3 pt = make_int3(c, r, s);
40          double3 dsgnf = make_double3(
41            xDeriv(f1, stencil, rad, pt, delta, dim, sgn, 0.0),
42            yDeriv(f1, stencil, rad, pt, delta, dim, sgn, 0.0),
43            zDeriv(f1, stencil, rad, pt, delta, dim, sgn, 0.0));
44
45          double mul = 1 - sgn(f2[flatten(pt, dim)], 0.0);
46          if (mul == 0.0) continue;
47
48          double tydx = xDeriv(ty, stencil, rad, pt, delta, dim, id, 0.0);
49          double tzdx = xDeriv(tz, stencil, rad, pt, delta, dim, id, 0.0);
50          double txdy = yDeriv(tx, stencil, rad, pt, delta, dim, id, 0.0);
51          double tzdy = yDeriv(tz, stencil, rad, pt, delta, dim, id, 0.0);
52          double txdz = zDeriv(tx, stencil, rad, pt, delta, dim, id, 0.0);
53          double tydz = zDeriv(ty, stencil, rad, pt, delta, dim, id, 0.0);
54
55          double X = dsgnf.x * (tzdy  - tydz);
56          double Y = dsgnf.y * (txdz  - tzdx);
57          double Z = dsgnf.z * (tydx  - txdy);
58
59          lineIntSum += (X + Y + Z) * mul;
60        }
61      }
62    }
63
64    delete[] tx;
65    delete[] ty;
66    delete[] tz;
67
68    return lineIntSum * std::pow(delta, 3) / 4.0;
69  }
```

## 4.3 Results and Discussion

### *4.3.1 Sphere intersection arc length*

Line integration can be used to measure the length of intersection of objects that are in intersecting configuration. Perhaps one of the most intuitive ways to illustrate this is to measure the intersection length of two intersecting spheres. For this sample case, we construct a problem with a known exact result. We have two spheres of unit radii 15 and 13. The sphere with radius 15 is centered at $\boldsymbol{r}_1 = \{-7, 0, 0\}$ whereas the sphere with radius 13 is centered at $\boldsymbol{r}_2 = \{7, 0, 0\}$. Figure 4.1 shows the configuration with the bigger sphere shaded green and smaller sphere shaded red, whereas Figure 4.2 shows the same configuration's cross section. With the given dimensions and coordinates, the problem is constructed so that the length of the intersection curve between the two spherical surfaces (a circle of radius 12) is $24\pi$. Following a similar numerical approach to torus surface area analysis that was done in Section 2.3.1, here we compare the error percentages of our method compared to the actual solution for different stencil options and for different grid spacings.

Making 250 samplings under different configurations (same intersecting spheres with translations and rotations) in the grid for each $R/\Delta$ and wavelet genus in order to avoid overemphasizing accidental appearance of high accuracy associated with "lucky" grid spacings or alignments, Figures 4.3 and 4.4 show the error percentages for genera 1, 4, and 7.

In addition to boxplots in Figures 4.3 and 4.4, Figures 4.5 and 4.6 show the convergence for the same three derivative wavelet genera. Convergence rate is about the same for each genus, and this suggests that picking higher genus for computations in this case is not critical potentially due to round of errors introduced in many computations arising from higher stencil radii.

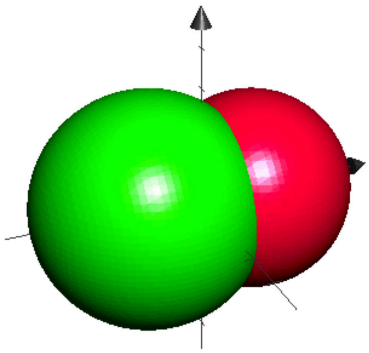


Figure 4.1: Intersecting spheres, and the curve formed by their intersection.

### *4.3.2 Sphere intersection arc length with cutting*

In addition to being able to measure the arc length resulting from intersection of two objects, our method also allows limiting the domain and enables the restriction of the domain under consideration of the problem space. With a slight modification in the derivation, we obtain the line integral function implemented in C++ as shown in Listing 4.2. Here the introduction of new grid, `f3` should be noted. `f3` is the grid that acts as a mask, and it allows the desired cutting/filtering behavior in line length integral calculations.

Listing 4.2: Implementation of Line Integral function with cutting

```
1 double lineIntegral(const double *f1, const double *f2, const double *f3,
2  const double *stencil, int rad, const double3 &lim, double delta,
3  const dim3 &dim) {
4   int problemSize = dim.x * dim.y * dim.z;
5   double lineIntSum = 0.0;
```

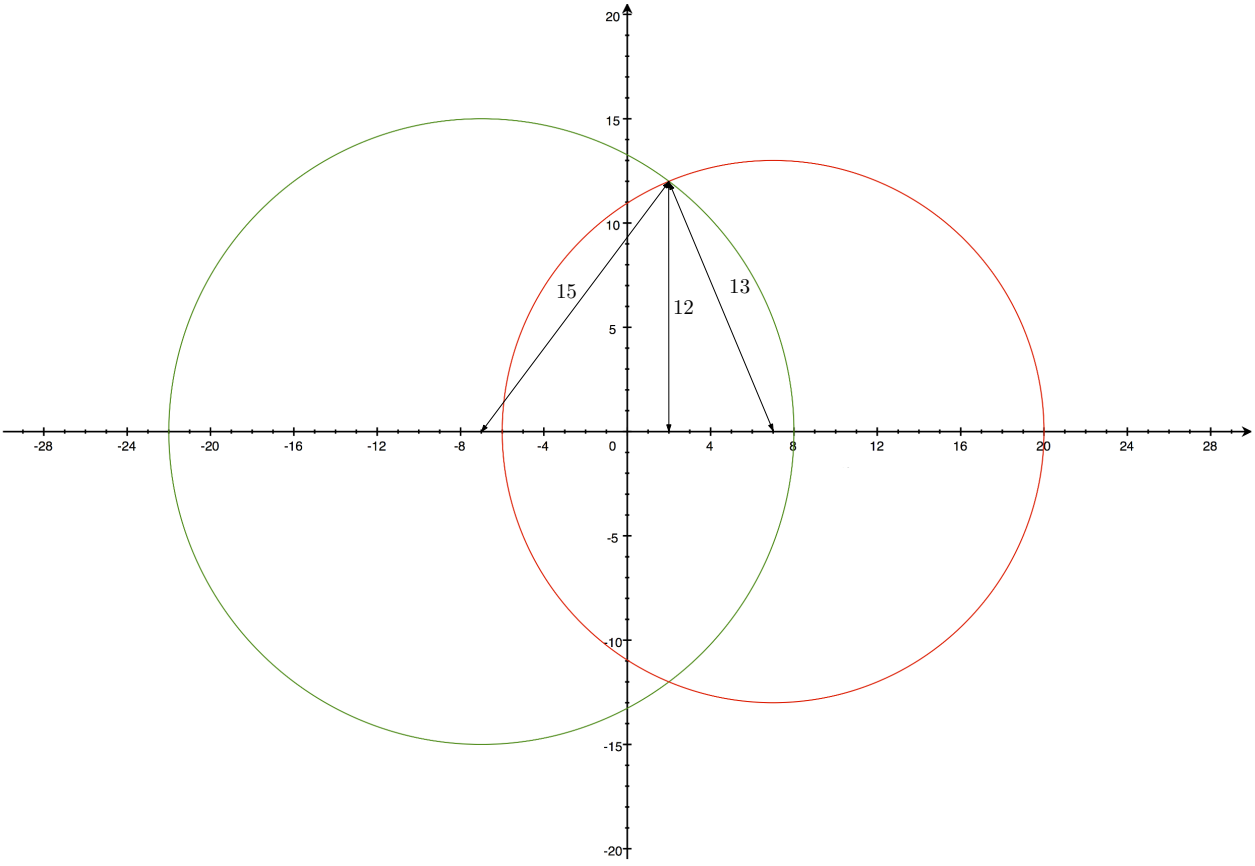
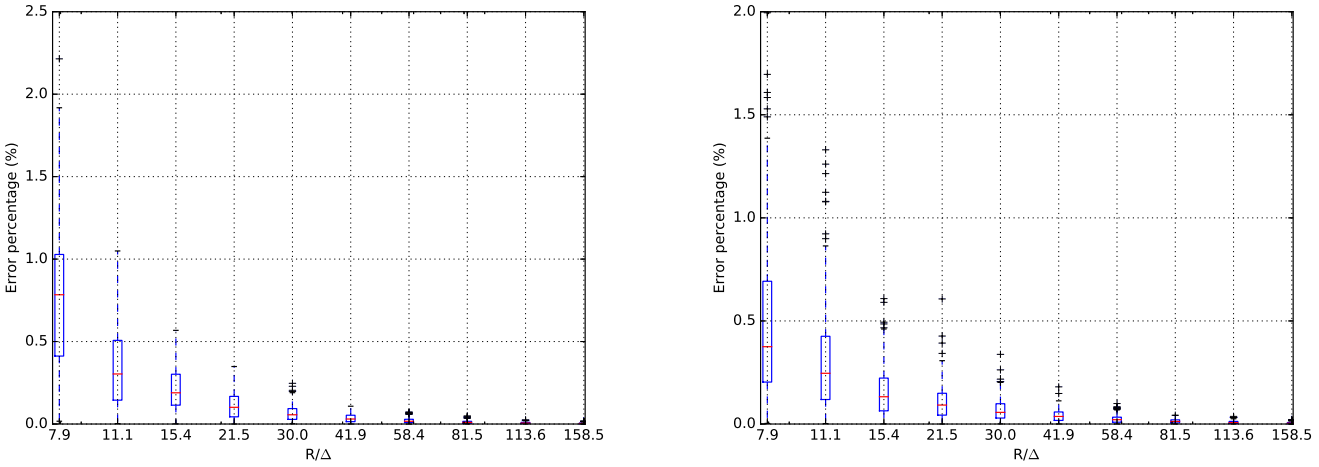



Figure 4.2: Cross section of intersecting spheres and the cross section of the circle with radius 12 created by this intersection.




(a) (b)

Figure 4.3: Computed sphere intersection line length / actual line length vs. Number of points for intersecting spheres for genus 1 (a) and 4 (b).

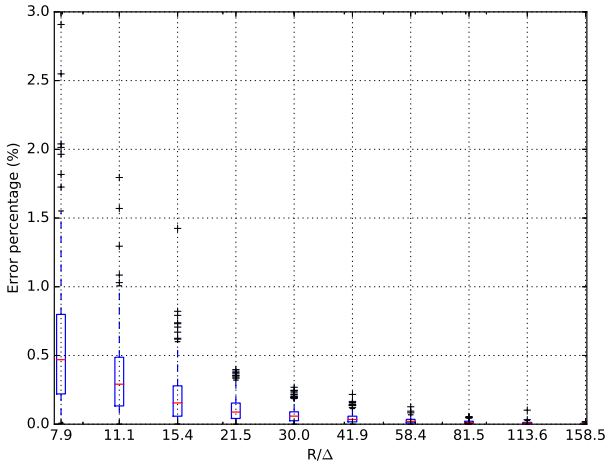
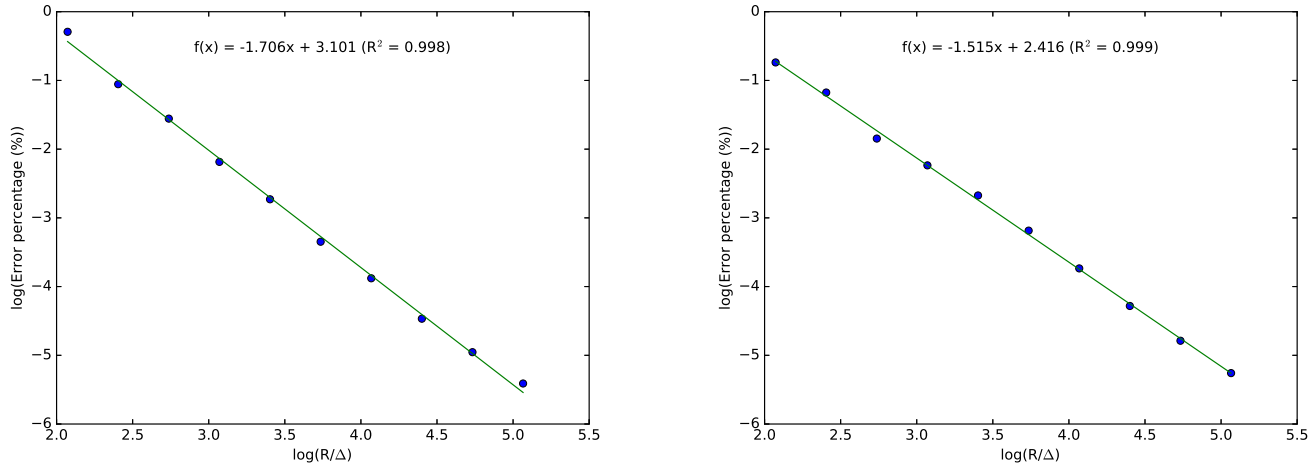



Figure 4.4: Computed sphere intersection line length / actual line length vs. Number of points for intersecting spheres for genus 7.




(a) (b)

Figure 4.5: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for the line length of intersection for intersecting spheres for genus 1 (a) and 4 (b).

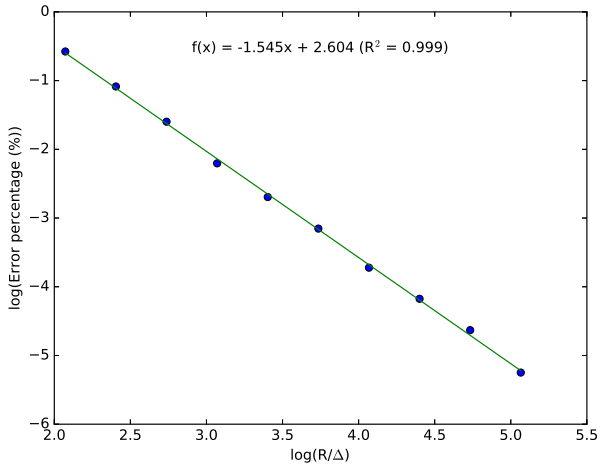
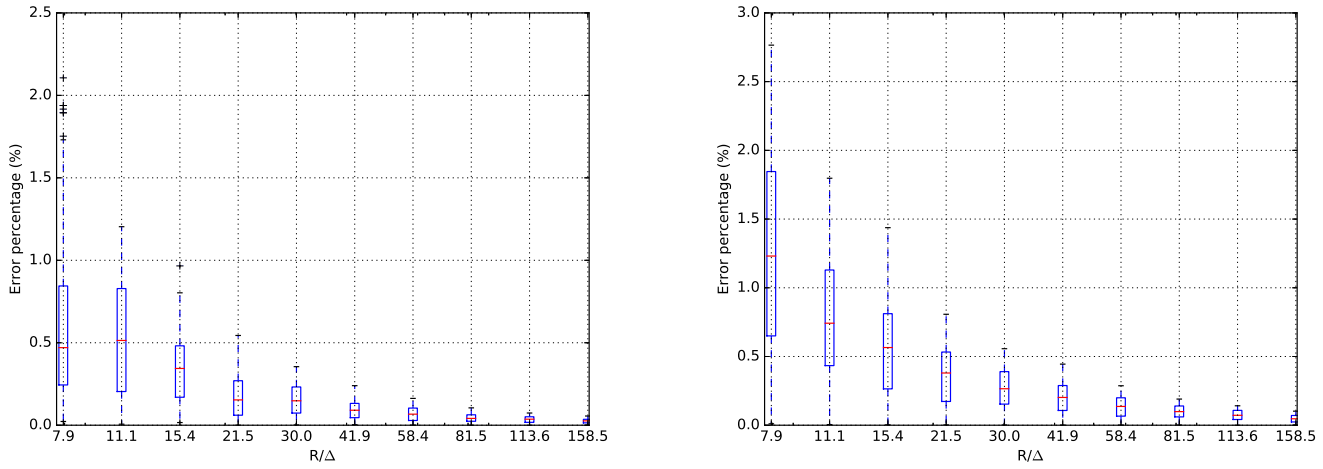



Figure 4.6: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for the line length of intersection for intersecting spheres for genus 7.

```
6  double *tx = new double[problemSize]();
7  double *ty = new double[problemSize]();
8  double *tz = new double[problemSize]();
9  for (int s = 0; s < dim.z; s++) {
10   for (int r = 0; r < dim.y; r++) {
11     for (int c = 0; c < dim.x; c++) {
12       int3 pt = make_int3(c, r, s);
13       double3 df1 = make_double3(
14         xDeriv(f1, stencil, rad, pt, delta, dim, id, 0.0),
15         yDeriv(f1, stencil, rad, pt, delta, dim, id, 0.0),
16         zDeriv(f1, stencil, rad, pt, delta, dim, id, 0.0));
17
18       double3 df2 = make_double3(
19         xDeriv(f2, stencil, rad, pt, delta, dim, id, 0.0),
20         yDeriv(f2, stencil, rad, pt, delta, dim, id, 0.0),
21         zDeriv(f2, stencil, rad, pt, delta, dim, id, 0.0));
22
23       double3 t_ijk = make_double3(df1.y * df2.z - df1.z * df2.y,
24         df1.z * df2.x - df1.x * df2.z, df1.x * df2.y - df1.y * df2.x);
25
26       double magnitude = computeMagnitude(t_ijk);
27
28       if (magnitude < EPS) continue;
29
30       tx[flatten(pt, dim)] = t_ijk.x / magnitude;
31       ty[flatten(pt, dim)] = t_ijk.y / magnitude;
32       tz[flatten(pt, dim)] = t_ijk.z / magnitude;
33     }
34   }
35 }
36
37 for (int s = 0; s < dim.z; s++) {
38   for (int r = 0; r < dim.y; r++) {
```

```
39      for (int c = 0; c < dim.x; c++) {
40        int3 pt = make_int3(c, r, s);
41        double3 dsgnf = make_double3(
42          xDeriv(f1, stencil, rad, pt, delta, dim, sgn, 0.0),
43          yDeriv(f1, stencil, rad, pt, delta, dim, sgn, 0.0),
44          zDeriv(f1, stencil, rad, pt, delta, dim, sgn, 0.0));
45
46        double mul1 = 1 - sgn(f2[flatten(pt, dim)], 0.0);
47        double mul2 = 1 - sgn(f3[flatten(pt, dim)], 0.0);
48        if (mul1 == 0.0 || mul2 == 0.0) continue;
49
50        double tydx = xDeriv(ty, stencil, rad, pt, delta, dim, id, 0.0);
51        double tzdx = xDeriv(tz, stencil, rad, pt, delta, dim, id, 0.0);
52        double txdy = yDeriv(tx, stencil, rad, pt, delta, dim, id, 0.0);
53        double tzdy = yDeriv(tz, stencil, rad, pt, delta, dim, id, 0.0);
54        double txdz = zDeriv(tx, stencil, rad, pt, delta, dim, id, 0.0);
55        double tydz = zDeriv(ty, stencil, rad, pt, delta, dim, id, 0.0);
56
57        double X = dsgnf.x * (tzdy  - tydz);
58        double Y = dsgnf.y * (txdz  - tzdx);
59        double Z = dsgnf.z * (tydx  - txdy);
60
61        lineIntSum += (X + Y + Z) * mul1 * mul2;
62      }
63    }
64  }
65
66  delete[] tx;
67  delete[] ty;
68  delete[] tz;
69
70  return lineIntSum * std::pow(delta, 3) / 8.0;
71 }
```




(a) (b)

Figure 4.7: Computed sphere intersection line length / actual line length vs. Number of points for intersecting spheres for genus 1 (a) and 4 (b) with cutting.

Now that we can cut our domain with a given equation, continuing with the example from Section 4.3.1, this time, we cut our domain with the $xy$-plane at $z = 0$. Following an approach similar to Section 4.3.1 in analyzing errors, however this time running tests only once for each case instead of randomizing, we obtain similar results again.

Just like in Section 4.3.1, making 250 samplings under different configurations for each $R/\Delta$ and wavelet genus, Figures 4.7 and 4.8 show the error percentages for genera 1, 4, and 7.

Figures 4.9 and 4.10 show the convergence for the same three derivative wavelet genera which mostly agree with the results of the previous section. Again, these plots don't show a dramatic convergence improvement with increasing genera, potentially due to round-off errors caused by the increasing number of computations for each derivative operation.

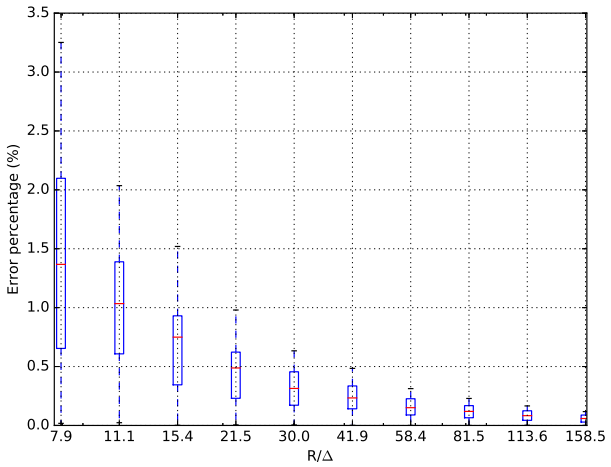
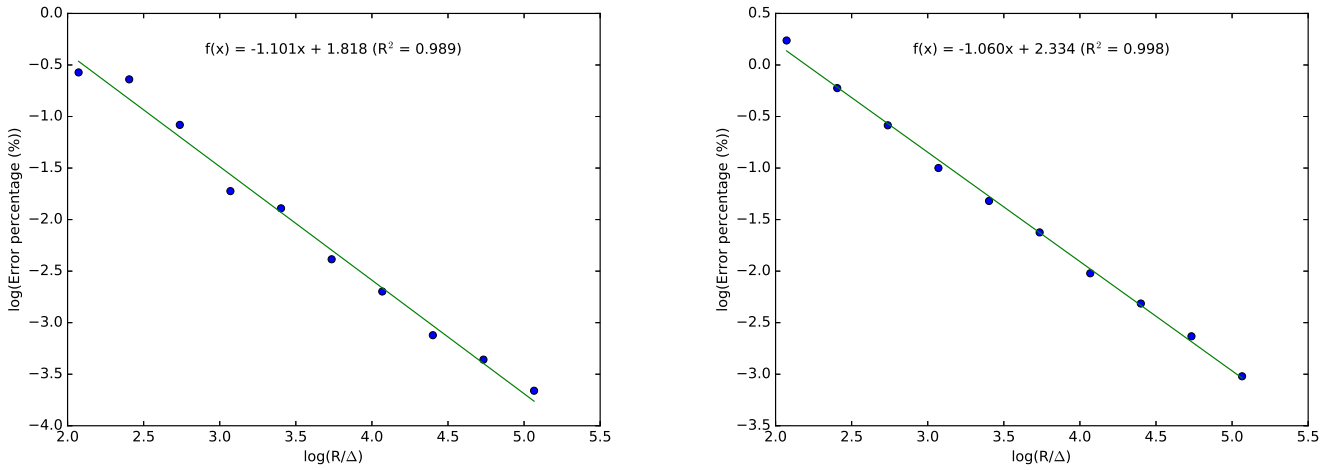



Figure 4.8: Computed sphere intersection line length / actual line length vs. Number of points for intersecting spheres for genus 7 with cutting.




(a) (b)

Figure 4.9: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for the line length of intersection for intersecting spheres for genus 1 (a) and 4 (b) with cutting.

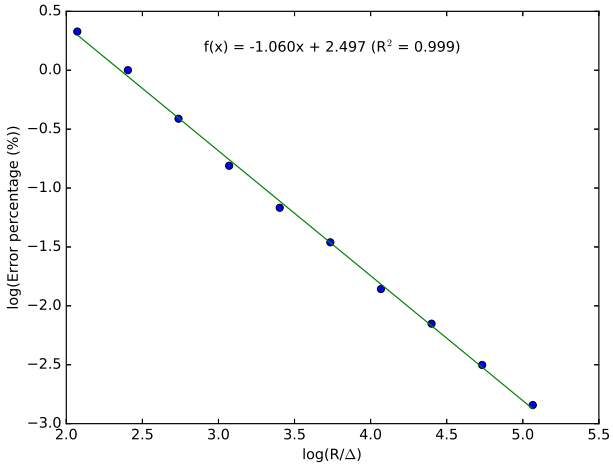



Figure 4.10: Loglog curve fitting plot of Error percentage vs. $R/\Delta$ for the line length of intersection for intersecting spheres (genus 7) with cutting.

### *4.4 GPU Implementation and Performance Analysis*

Here, we discuss CUDA implementation of the sphere intersection line length problem and compare its performance to previously discussed serial implementation. As it can be seen in Listing 4.1, line integral formulation resulted in a considerably complicated serial implementation. Similarly, the GPU implementation is also a little bit involved. Listing 4.3 shows the kernel wrapper function.

Listing 4.3: Line integral GPU implementation - wrapper function

```
1 double lineIntegral(const double *f1, const double *f2, const double *stencil,
2  int rad, const double3 &lim, double delta, const dim3 &dim) {
3   int problemSize = dim.x * dim.y * dim.z;
4   double lineIntSum = 0.0;
5   double *d_lineIntSum = 0, *d_f1 = 0, *d_f2 = 0, *d_stencil = 0;
6   double *d_tx = 0, *d_ty = 0, *d_tz = 0;
7   dim3 *d_dim = 0;
8   double3 *d_lim = 0;
```

```
9
10  cudaMalloc(&d_lineIntSum, sizeof(double));
11  cudaMemset(d_lineIntSum, 0, sizeof(double));
12
13  cudaMalloc(&d_f1, problemSize * sizeof(double));
14  cudaMemcpy(d_f1, f1, problemSize * sizeof(double), cudaMemcpyHostToDevice);
15
16  cudaMalloc(&d_f2, problemSize * sizeof(double));
17  cudaMemcpy(d_f2, f2, problemSize * sizeof(double), cudaMemcpyHostToDevice);
18
19  cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
20  cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
21    cudaMemcpyHostToDevice);
22
23  cudaMalloc(&d_dim, sizeof(dim3));
24  cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
25
26  cudaMalloc(&d_lim, sizeof(double3));
27  cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
28
29  cudaMalloc(&d_tx, problemSize * sizeof(double));
30  cudaMalloc(&d_ty, problemSize * sizeof(double));
31  cudaMalloc(&d_tz, problemSize * sizeof(double));
32
33  const dim3 blockSize(TX, TY, TZ);
34  const dim3 gridSize(divUp(dim.x, TX), divUp(dim.y, TY), divUp(dim.z, TZ));
35  const size_t smSz1 = (2 * (TX + 2 * rad) * (TY + 2 * rad) * (TZ + 2 * rad) +
36                        2 * rad + 1 + 1) * sizeof(double);
37  const size_t smSz2 = (5 * (TX + 2 * rad) * (TY + 2 * rad) * (TZ + 2 * rad) +
38                        2 * rad + 1 + 1) * sizeof(double);
39  tangentKernel<<<gridSize, blockSize, smSz1>>>(d_tx, d_ty, d_tz, d_f1, d_f2,
40    d_stencil, rad, d_lim, delta, d_dim);
41  lineIntegralKernel<<<gridSize, blockSize, smSz2>>>(d_lineIntSum, d_f1, d_f2,
```

```
42      d_tx, d_ty, d_tz, d_stencil, rad, d_lim, delta, d_dim);
43    cudaMemcpy(&lineIntSum, d_lineIntSum, sizeof(double),
44      cudaMemcpyDeviceToHost);
45    cudaFree(d_f1);
46    cudaFree(d_f2);
47    cudaFree(d_lineIntSum);
48    cudaFree(d_stencil);
49    cudaFree(d_tx);
50    cudaFree(d_ty);
51    cudaFree(d_tz);
52    cudaFree(d_lim);
53    cudaFree(d_dim);
54
55    return lineIntSum / 4.0 * std::pow(delta, 3);
56  }
```

As it can be seen in lines 39 and 41, there are now two kernel functions that are being called. First kernel function, provided in Listing 4.4, computes the tangent vectors for each point for which the formula is shown in Equation 4.11.

Listing 4.4: Line integral GPU implementation - tangent kernel

```
1  __global__
2  void tangentKernel(double *tx, double *ty, double *tz, const double *f1,
3    const double *f2, const double *stencil, int rad, const double3 *lim,
4    double delta, const dim3 *dim) {
5    const int c = threadIdx.x + blockDim.x * blockIdx.x;
6    const int r = threadIdx.y + blockDim.y * blockIdx.y;
7    const int s = threadIdx.z + blockDim.z * blockIdx.z;
8    if (c >= dim->x || r >= dim->y || s >= dim->z) return;
9    const int3 pt_crs = make_int3(c, r, s);
10   const int i = flatten(pt_crs, *dim);
11   const int s_c = threadIdx.x + rad;
12   const int s_r = threadIdx.y + rad;
```

```
13  const int s_s = threadIdx.z + rad;
14  const int3 s_pt = make_int3(s_c, s_r, s_s);
15  const dim3 sBlockSz = dim3(blockDim.x + 2 * rad, blockDim.y + 2 * rad,
16    blockDim.z + 2 * rad);
17  const int s_i = flatten(s_pt, sBlockSz);
18
19  extern __shared__ double smem[];
20  double *s_block1 = smem;
21  double *s_block2 = s_block1 + sBlockSz.x * sBlockSz.y * sBlockSz.z;
22  double *s_stencil = s_block1 + 2 * sBlockSz.x * sBlockSz.y * sBlockSz.z;
23
24  // set stencil once per shared block
25  if (s_c == rad && s_r == rad && s_s == rad) {
26    for (int k = 0; k < 2 * rad + 1; k++) {
27      s_stencil[k] = stencil[k];
28    }
29  }
30
31  // Regular cells
32  s_block1[s_i] = f1[i];
33  s_block2[s_i] = f2[i];
34
35  // Halo cells
36  if (threadIdx.x < rad) {
37    s_block1[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
38      f1[flatten(make_int3(c - rad, r, s), *dim)];
39    s_block1[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
40      f1[flatten(make_int3(c + blockDim.x, r, s), *dim)];
41    s_block2[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
42      f2[flatten(make_int3(c - rad, r, s), *dim)];
43    s_block2[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
44      f2[flatten(make_int3(c + blockDim.x, r, s), *dim)];
45  }
```

```
46
47  if (threadIdx.y < rad) {
48    s_block1[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
49      f1[flatten(make_int3(c, r - rad, s), *dim)];
50    s_block1[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
51      f1[flatten(make_int3(c, r + blockDim.y, s), *dim)];
52    s_block2[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
53      f2[flatten(make_int3(c, r - rad, s), *dim)];
54    s_block2[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
55      f2[flatten(make_int3(c, r + blockDim.y, s), *dim)];
56  }
57
58  if (threadIdx.z < rad) {
59    s_block1[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
60      f1[flatten(make_int3(c, r, s - rad), *dim)];
61    s_block1[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
62      f1[flatten(make_int3(c, r, s + blockDim.z), *dim)];
63    s_block2[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
64      f2[flatten(make_int3(c, r, s - rad), *dim)];
65    s_block2[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
66      f2[flatten(make_int3(c, r, s + blockDim.z), *dim)];
67  }
68
69  __syncthreads();
70
71
72  if ((c >= dim->x - rad) || (r >= dim->y - rad) || (s >= dim->z - rad) ||
73      c <= (rad - 1) || r <= (rad - 1) || s <= (rad - 1)) return;
74
75  double3 df1 = make_double3(
76    xDeriv(s_block1, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
77    yDeriv(s_block1, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
78    zDeriv(s_block1, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0));
```

```
79
80   double3 df2 = make_double3(
81     xDeriv(s_block2, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
82     yDeriv(s_block2, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
83     zDeriv(s_block2, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0));
84
85   double3 t_ijk = make_double3(df1.y * df2.z - df1.z * df2.y,
86     df1.z * df2.x - df1.x * df2.z, df1.x * df2.y - df1.y * df2.x);
87
88   double magnitude = computeMagnitude(t_ijk);
89
90   if (magnitude < EPS) return;
91
92   tx[i] = t_ijk.x / magnitude;
93   ty[i] = t_ijk.y / magnitude;
94   tz[i] = t_ijk.z / magnitude;
95 }
```

Second kernel, shown in Listing 4.5, using the vectors computed by the kernel function in Listing 4.4, applies Equations 4.12 and 4.14, and does the line integral computation.

Listing 4.5: Line integral GPU implementation - line integral kernel

```
1 __global__
2 void lineIntegralKernel(double *lineIntegral, const double *const f1,
3   const double *const f2, const double *const tx, const double *const ty,
4   const double *const tz, const double *const stencil, int rad, const double3
        *lim,
5   double delta, const dim3 *dim) {
6   const int c = threadIdx.x + blockDim.x * blockIdx.x;
7   const int r = threadIdx.y + blockDim.y * blockIdx.y;
8   const int s = threadIdx.z + blockDim.z * blockIdx.z;
9   if (c >= dim->x || r >= dim->y || s >= dim->z) return;
10  const int3 pt_crs = make_int3(c, r, s);
```

```
11  const int i = flatten(pt_crs, *dim);
12  const int s_c = threadIdx.x + rad;
13  const int s_r = threadIdx.y + rad;
14  const int s_s = threadIdx.z + rad;
15  const int3 s_pt = make_int3(s_c, s_r, s_s);
16  const dim3 sBlockSz = dim3(blockDim.x + 2 * rad, blockDim.y + 2 * rad,
17    blockDim.z + 2 * rad);
18  const int s_i = flatten(s_pt, sBlockSz);
19
20  extern __shared__ double smem[];
21  double *s_block1 = smem;
22  double *s_block2 = s_block1 + sBlockSz.x * sBlockSz.y * sBlockSz.z;
23  double *s_blockx = s_block1 + 2 * sBlockSz.x * sBlockSz.y * sBlockSz.z;
24  double *s_blocky = s_block1 + 3 * sBlockSz.x * sBlockSz.y * sBlockSz.z;
25  double *s_blockz = s_block1 + 4 * sBlockSz.x * sBlockSz.y * sBlockSz.z;
26  double *s_stencil = s_block1 + 5 * sBlockSz.x * sBlockSz.y * sBlockSz.z;
27
28  // set stencil once per shared block
29  if (s_c == rad && s_r == rad && s_s == rad) {
30    for (int k = 0; k < 2 * rad + 1; k++) {
31      s_stencil[k] = stencil[k];
32    }
33  }
34
35  // Regular cells
36  s_block1[s_i] = f1[i];
37  s_block2[s_i] = f2[i];
38  s_blockx[s_i] = tx[i];
39  s_blocky[s_i] = ty[i];
40  s_blockz[s_i] = tz[i];
41
42  // Halo cells
43  if (threadIdx.x < rad) {
```

```
44    s_block1[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
45      f1[flatten(make_int3(c - rad, r, s), *dim)];
46    s_block1[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
47      f1[flatten(make_int3(c + blockDim.x, r, s), *dim)];
48    s_block2[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
49      f2[flatten(make_int3(c - rad, r, s), *dim)];
50    s_block2[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
51      f2[flatten(make_int3(c + blockDim.x, r, s), *dim)];
52    s_blockx[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
53      tx[flatten(make_int3(c - rad, r, s), *dim)];
54    s_blockx[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
55      tx[flatten(make_int3(c + blockDim.x, r, s), *dim)];
56    s_blocky[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
57      ty[flatten(make_int3(c - rad, r, s), *dim)];
58    s_blocky[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
59      ty[flatten(make_int3(c + blockDim.x, r, s), *dim)];
60    s_blockz[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
61      tz[flatten(make_int3(c - rad, r, s), *dim)];
62    s_blockz[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
63      tz[flatten(make_int3(c + blockDim.x, r, s), *dim)];
64  }
65
66  if (threadIdx.y < rad) {
67    s_block1[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
68      f1[flatten(make_int3(c, r - rad, s), *dim)];
69    s_block1[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
70      f1[flatten(make_int3(c, r + blockDim.y, s), *dim)];
71    s_block2[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
72      f2[flatten(make_int3(c, r - rad, s), *dim)];
73    s_block2[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
74      f2[flatten(make_int3(c, r + blockDim.y, s), *dim)];
75    s_blockx[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
76      tx[flatten(make_int3(c, r - rad, s), *dim)];
```

```
77    s_blockx[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
78      tx[flatten(make_int3(c, r + blockDim.y, s), *dim)];
79    s_blocky[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
80      ty[flatten(make_int3(c, r - rad, s), *dim)];
81    s_blocky[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
82      ty[flatten(make_int3(c, r + blockDim.y, s), *dim)];
83    s_blockz[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
84      tz[flatten(make_int3(c, r - rad, s), *dim)];
85    s_blockz[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
86      tz[flatten(make_int3(c, r + blockDim.y, s), *dim)];
87  }
88
89  if (threadIdx.z < rad) {
90    s_block1[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
91      f1[flatten(make_int3(c, r, s - rad), *dim)];
92    s_block1[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
93      f1[flatten(make_int3(c, r, s + blockDim.z), *dim)];
94    s_block2[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
95      f2[flatten(make_int3(c, r, s - rad), *dim)];
96    s_block2[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
97      f2[flatten(make_int3(c, r, s + blockDim.z), *dim)];
98    s_blockx[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
99      tx[flatten(make_int3(c, r, s - rad), *dim)];
100   s_blockx[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
101     tx[flatten(make_int3(c, r, s + blockDim.z), *dim)];
102   s_blocky[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
103     ty[flatten(make_int3(c, r, s - rad), *dim)];
104   s_blocky[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
105     ty[flatten(make_int3(c, r, s + blockDim.z), *dim)];
106   s_blockz[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
107     tz[flatten(make_int3(c, r, s - rad), *dim)];
108   s_blockz[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
109     tz[flatten(make_int3(c, r, s + blockDim.z), *dim)];
```

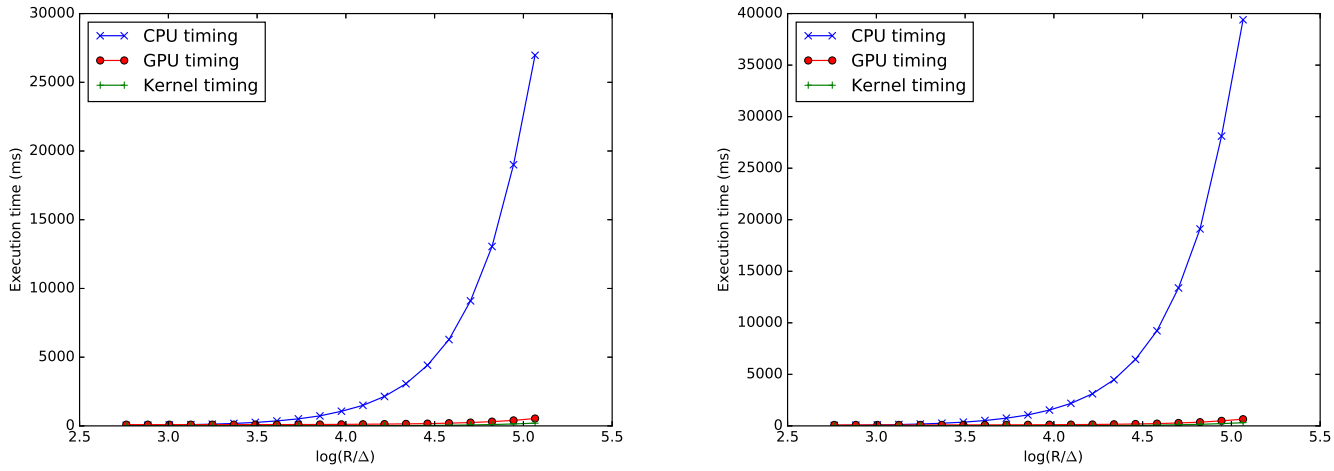
```
110  }
111
112  __syncthreads();
113
114  if ((c >= dim->x - (rad + 1)) || (r >= dim->y - (rad + 1)) ||
115      (s >= dim->z - (rad + 1)) || c <= rad || r <= rad || s <= rad) return;
116
117  double3 dsgnf = make_double3(
118    xDeriv(s_block1, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
119    yDeriv(s_block1, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
120    zDeriv(s_block1, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0));
121
122  double mul = 1.0 - sgn(s_block2[flatten(s_pt, sBlockSz)], 0.0);
123  if (mul == 0.0) return;
124
125  double tydx = xDeriv(s_blocky, s_stencil, rad, s_pt, delta, sBlockSz, id,
       0.0);
126  double tzdx = xDeriv(s_blockz, s_stencil, rad, s_pt, delta, sBlockSz, id,
       0.0);
127  double txdy = yDeriv(s_blockx, s_stencil, rad, s_pt, delta, sBlockSz, id,
       0.0);
128  double tzdy = yDeriv(s_blockz, s_stencil, rad, s_pt, delta, sBlockSz, id,
       0.0);
129  double txdz = zDeriv(s_blockx, s_stencil, rad, s_pt, delta, sBlockSz, id,
       0.0);
130  double tydz = zDeriv(s_blocky, s_stencil, rad, s_pt, delta, sBlockSz, id,
       0.0);
131
132  double X = dsgnf.x * (tzdy  - tydz);
133  double Y = dsgnf.y * (txdz  - tzdx);
134  double Z = dsgnf.z * (tydx  - txdy);
135
136  atomicAdd(lineIntegral, (X + Y + Z) * mul);
```

```
137 }
```

Another important issue is the shared memory. As previously discussed, the 48 kB limit on the shared memory sometimes prevents from being able to impose derivative operators with bigger radii since the shared memory needs to be cached. In this case, since the shared memory usage is heavy, the only two genera that can be run without hitting the 48 kB limit are genus 1 and 2 (derivative stencil with 3 elements and 5 elements).

The serial and the parallel implementations both return the exact same line integral computation result. Thus, only timing plots are provided in order to compare the differences made by the CUDA implementation.

Figure 4.11 shows the difference between CUDA and serial computation performance for the first two genera. At the highest loading around $\log(R/\Delta) = 5$ and for genus 2, the serial implementation takes about 50,000 ms whereas the parallel implementation takes about 760 ms with the kernel taking about 350 ms. This means that the CUDA implementation enables a speedup of almost 100x.




(a) (b)

Figure 4.11: Timing comparison for sphere intersection line length in wavelet genus 1 (a) and 2 (b).

# Chapter 5

# APPLICATION TO BOUNDARY INTEGRALS

## *5.1 Introduction*

In this chapter, we experiment with applying grid-based integration to boundary integral analysis methods.

Boundary integral methods can be used to solve partial differential equations (PDEs) by rephrasing PDEs with boundary conditions as boundary integral equations. The research on boundary integrals that result from application of integral equations to boundary value problems of potential dates back to early 1900s when Fredholm in his 1903 paper demonstrated the existence existence of solutions to such equations [35, 36, 37].

The boundary element method (BEM) is commonly employed as a means of computing numerical approximations of the solution of boundary integral problems [38, 39, 40]. BEM employs numerical integration on a discretization of a parametric description of the boundary to convert the integral equation to a solvable linear algebra problem. The focus of this dissertation is on methods that do not rely on a parametric specification of the boundary, and the most relevant previous work fitting that description involves the Implicit Boundary Integral Method (IBIM) presented by Kublik et al. [41]. IBIM methods employ an approach motivated by level set methods to obtain solvable discretizations of integral formulations of potential problems based on a regular grid of sampled values of the signed distance function that implicitly defines the domain.

Here the direct integration method of integral evaluation is applied to obtain solvable discretizations based on a grid of sampled values of a general implicit function defining the domain; i.e. the method is not restricted to grids of signed distance values.

According to Costabel [36], a boundary element method scheme solution can be broken

down to the following steps:

1. Mathematical model
2. Representation formula
3. Boundary integral equation
4. Boundary elements
5. Discrete equations
6. Solution of the linear system
7. Interpretation

In our approach, the first three steps are the same. However, instead of using boundary elements, we use direct integration to formulate the linear system to be solved.

## *5.2 Theory*

### *5.2.1 Boundary integral formulation for Laplace's equation*

To provide a concrete presentation of how to formulate a boundary integral representation of a PDE with boundary conditions, we present the specifics of the conversion process for the Laplace equation

$$\Delta u = 0 \tag{5.1}$$

Start by considering the product

$$\boldsymbol{F} = u \nabla u^* \tag{5.2}$$

where $u(\boldsymbol{r})$ and $u^*(\boldsymbol{r})$ are two functions. The divergence of the quantity is given by

$$\nabla \cdot \boldsymbol{F} = u \Delta u^* + \nabla u \nabla u^* \tag{5.3}$$

Applying the Divergence theorem

$$\int_V (\nabla \cdot \boldsymbol{F}) dv = \int_{\partial V} \boldsymbol{F} \cdot \boldsymbol{n} ds \tag{5.4}$$

we obtain

$$\int_V (u\Delta u^* + \nabla u \nabla u^*) dv = \int_{\partial V} (u \nabla u^*) n ds \tag{5.5}$$

Exchanging $u$ and $u^*$ gives

$$\int_V (u^* \Delta u + \nabla u^* \nabla u) dv = \int_{\partial V} (u^* \nabla u) n ds \tag{5.6}$$

Subtracting 5.5 from 5.6 we get

$$\int_V (u^* \Delta u - u \Delta u^*) dv = \int_{\partial V} (u^* \nabla u - u \nabla u^*) n ds \tag{5.7}$$

which is also known as Green's Second Identity [38, 42]. Renaming the net inward flux $q = \nabla u n$ and $q^* = \nabla u^* n$, Equation 5.7 becomes

$$\int_V u^* \Delta u dv - \int_V u \Delta u^* dv = \int_{\partial V} u^* q ds - \int_{\partial V} q^* u ds \tag{5.8}$$

With the goal of obtaining a boundary formulation, we seek to eliminate the volume integrals on the left-hand side, and we can do so by appropriately choosing definitions for $u$ and $u^*$ (which have been unspecified to this point). If we identify $u$ as the desired solution of Laplace's equation, then $\Delta u$ is identically zero thoughout the domain and the first volume integral vanishes. The second volume integral is typically handled by identifying $u^*$ as the free-space solution to a point source so that

$$\Delta u^*(\boldsymbol{q}) = -\delta(\boldsymbol{p} - \boldsymbol{q}) \tag{5.9}$$

where $\boldsymbol{p}$ is referred to as a source point and $\boldsymbol{q}$ is referred to as a field point. Since $\delta$ is zero everywhere except at $\boldsymbol{p} = \boldsymbol{q}$. Given the operational properties of the Dirac delta function, the left-hand side of Equation 5.9 evaluates to

$$\int_V u\Delta u^* dv = -\int_V u(\boldsymbol{q})\delta(\boldsymbol{p},\boldsymbol{q})dv = -u(\boldsymbol{p}) \tag{5.10}$$

and the final volume integral reduces to the value of the solution at the source point. Equation 5.8 then becomes

$$c(\boldsymbol{p})u(\boldsymbol{p}) + \int_{\partial V} q^* u ds = \int_{\partial V} u^* q ds \tag{5.11}$$

where $c(\boldsymbol{p}) = 1$. The equation is typically written in this form, because other values of $c(\boldsymbol{p})$ allow the equation to apply for general location of the source point. As we just saw, $c(\boldsymbol{p}) = 1$ corresponds to source point which lies within the domain. A source point located outside the domain would not contribute to the solution, corresponding to $c(\boldsymbol{p}) = 0$. In the limit as the source point approaches a smooth point on the boundary from within the domain, the appropriate value is $c(\boldsymbol{p}) = 1/2$. Having obtained Equation 5.11, we have arrived at a boundary integral formulation of a Laplace equation with Dirichlet boundary conditions.

### *5.2.2 Double layer potential formulation*

Here we pursue a specific boundary integral formulation of Laplace's equation, namely the double layer potential (formulation following Kublik et al. [41]). The double layer potential solution method involves intermediate determination of a dipole source density on the boundary. Once the dipole source density on the boundary has been determined, it is used as the basis to determine solution values at field points throughout the domain via

$$u(\boldsymbol{p}) = \int_{\partial\Omega} \beta(\boldsymbol{q}) \frac{\partial \Phi}{\partial n_{\boldsymbol{q}}^+} ds(\boldsymbol{q}), \tag{5.12}$$

where $\beta$ is the double layer density and $\frac{\partial \Phi}{\partial n_{\boldsymbol{q}}^+}$ is the directional derivative of the fundamental solution along the outward normal, where $\Phi$ is given as [41]

$$\Phi(\boldsymbol{p},\boldsymbol{q}) = \frac{1}{2\pi} \ln|\boldsymbol{p}-\boldsymbol{q}| \tag{5.13}$$

Equation 5.13 applies for 2D problems. In higher dimensions, the fundamental solution is given as

$$\Phi(\boldsymbol{p}, \boldsymbol{q}) = -\frac{1}{n(n-2)\rho_n |\boldsymbol{p} - \boldsymbol{q}|^{n-2}} \tag{5.14}$$

where $n$ is the dimension and $\rho_n$ is the volume of the unit ball.

The integral equation to be solved for the double layer potential is

$$\int_{\partial\Omega} \beta(\boldsymbol{q}) \frac{\partial \Phi(\boldsymbol{p}, \boldsymbol{q})}{\partial n_{\boldsymbol{q}}^+} ds(\boldsymbol{q}) + \frac{1}{2}\beta(\boldsymbol{p}) = b(\boldsymbol{p}) \tag{5.15}$$

where $b(\boldsymbol{p})$ is the Dirichlet boundary. Note that Equation 5.15 is obtained by identifying the value at a boundary point, as specified by the boundary conditions, with the limit of Equation 5.12 as the point p approaches the boundary. We solve this equation for $\beta$ and substitute into Equation 5.12 to obtain the solution values, $u(\boldsymbol{p})$, within the domain.

We apply the direct integration approach to get a set of linear equations corresponding to the discretization of Equation 5.15. For the surface integral term on the left-hand side, we have already made a derivation in Chapter 2, and remembering Equations 2.2 and 2.9 where we had

$$A(\partial\Omega) = \int_{\partial\Omega} g(\boldsymbol{r}) ds = -\int_{\mathbb{R}^3} (\nabla\chi \cdot \boldsymbol{n}) g(\boldsymbol{r})\, dv \tag{5.16}$$

with

$$g(\boldsymbol{q}) = \beta(\boldsymbol{q}) \frac{\partial \Phi(\boldsymbol{p}, \boldsymbol{q})}{\partial n_{\boldsymbol{q}}^+} \tag{5.17}$$

and

$$\frac{\partial \Phi(\boldsymbol{p}, \boldsymbol{q})}{\partial n_{\boldsymbol{q}}^+} = \nabla_{\boldsymbol{q}} \Phi(\boldsymbol{p}, \boldsymbol{q}) \cdot \boldsymbol{n}_{\boldsymbol{q}} \tag{5.18}$$

The integral in Equation 5.15 then becomes

$$-\int_{\mathbb{R}^3} \beta(\boldsymbol{q}) \nabla_{\boldsymbol{q}} \Phi(\boldsymbol{p}, \boldsymbol{q}) \cdot \boldsymbol{n}_{\boldsymbol{q}} (\nabla\chi \cdot \boldsymbol{n}) dv + \frac{1}{2}\beta(\boldsymbol{p}) = b(\boldsymbol{p}) \tag{5.19}$$

Picking a point $\boldsymbol{p} = \boldsymbol{p}_i$, this becomes a summation over $\boldsymbol{q} = \boldsymbol{p}_j$ with each element of the integral in the form

$$\beta(\boldsymbol{p}_j)\nabla_{\boldsymbol{p}_j}\Phi(\boldsymbol{p}_i, \boldsymbol{p}_j) \cdot \left.\frac{\nabla f}{|\nabla f|}\right|_{\boldsymbol{p}_j} \left(\nabla\chi_{\boldsymbol{p}_i}(\boldsymbol{p}_j) \cdot \left.\frac{\nabla f}{|\nabla f|}\right|_{\boldsymbol{p}_j}\right) \tag{5.20}$$

where $\nabla\chi_{\boldsymbol{p}_i}(\boldsymbol{p}_j)$ is the result of gradient of the occupancy function that is shifted from zero by the threshold value (of occupancy) at $\boldsymbol{p}_i$, that is being summed over at point $\boldsymbol{p}_j$. This way, when $\boldsymbol{p}_i = \boldsymbol{p}_j$, while the fundamental solution function's gradient blows up at these locations, the gradient of the shifted occupancy function that is zero, and this cancels out the effect of these singularities.

In the grid based integration method, all the non-trivial contributions come from grid points that are close enough to the boundary so that their derivative stencil crosses the boundary. In this context, these grid points adjacent to the boundary are referred to as irregular points. In order to solve for $\beta$, we consider each irregular point as a source and we enumerate over the source points as explained above. The discrete elements of the coefficient matrix, $A$, are then

$$A_{ij} = \frac{1}{2}\delta_{ij} + h^n \left(\nabla_{\boldsymbol{p}_j}\Phi(\boldsymbol{p}_i, \boldsymbol{p}_j) \cdot \left.\frac{\nabla f}{|\nabla f|}\right|_{\boldsymbol{p}_j}\right) \left(\nabla\chi_{\boldsymbol{p}_i}(\boldsymbol{p}_j) \cdot \left.\frac{\nabla f}{|\nabla f|}\right|_{\boldsymbol{p}_j}\right) \tag{5.21}$$

where $\delta_{ij}$ is the Kronecker delta, and $h^n$ is the grid spacing in n-dimensions.

Here is the summary of the steps we follow in the numerical solution of the problem:

1. Detect irregular points. The number of detected irregular points is referred to as $N$.
2. Construct the linear algebra problem $A\beta = b$ where, $A$ is the $N \times N$ matrix and $b$ is created by sampling the boundary function $b(\boldsymbol{p})$, for the irregular points.
3. Solve for $\beta$ with a matrix solver such as LU solver.
4. Computing $A_{\text{full}}$,the full domain version of $A$ (not just the irregular points).
5. Reconstruct $u$ with $u = A_{\text{full}}\beta$. If there are memory constraints, $u$ can instead be solved with computing one dot product at a time with a loop in the serial approach or in parallel in the GPU programming approach.

### 5.3 Results and Discussion

Listing A.9 provided in Appendix A shows the whole code used in the numerical computation of boundary integral on the given boundary and shape. To highlight the algorithm from the previous section, Listing 5.1 shows how irregular points are detected where

Listing 5.1: Detection of irregular points.

```
bool isIrregular(const int3& pt, const double *grid, const double *stencil,
  int rad, double delta, const dim3& dim, double eps) {
  double3 gradChi = computeGradChi(grid, stencil, rad, pt, delta, dim, 0.0);
  return computeMagnitude(gradChi) > eps;
}
```

`computeGradChi` function is shown in Listing 5.2.

Listing 5.2: Computation of $\nabla\chi$.

```
double3 computeGradChi(const double *grid, const double *stencil, int rad,
  const int3& pt, double delta, const dim3& dim, double thresh) {
  return make_double3(
    -xDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0,
    -yDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0,
    -zDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0);
}
```

Once all the irregular points ($N$ points) are detected, the $N$-by-$N$ $A$ matrix is formed. Listing 5.3 shows the loop used in forming $F$, which is the matrix $A$ before the diagonal elements are added

Listing 5.3: Double loop populating $F$ matrix.

```
  for (int r = 0; r < N; r++) {
    for (int c = 0; c < N; c++) {
```

```
      F(r, c) = computeElementOfA(irregular_pts[c], irregular_pts[r], grid,
        stencil, rad, lim, delta, dim);
    }
  }
```

where the function `computeElementOfA` is shown in Listing 5.5. In order the obtain the final form of matrix $A$, matrix $F$ is then summed with $1/2$ identity matrix as

Listing 5.4: Obtaining the final $A$ matrix.

```
Eigen::MatrixXf A = F + Eigen::MatrixXf::Identity(N, N) / 2.0;
```

Listing 5.5: Computation of elements of $A$ matrix.

```
double computeElementOfA(const int3& ptc_crs, const int3& ptr_crs,
  const double *grid, const double *stencil, int rad, const double3& lim,
  double delta, const dim3& dim, bool withThreshold=true) {
  bool problemIs3D = dim.z > 1;
  double thresh = withThreshold ? grid[flatten(ptc_crs, dim)] : 0.0;

  double3 gradPhi, normalizedGradF, gradChi;
  try {
    gradPhi = computeGradPhi(ptr_crs, ptc_crs, delta, lim, problemIs3D);
  } catch(std::runtime_error e) { return 0.0; }

  try {
    normalizedGradF = computeNormalizedGradF(grid, stencil, rad, ptr_crs,
      delta, dim);
  } catch(std::runtime_error e) { return 0.0; }

  gradChi = computeGradChi(grid, stencil, rad, ptr_crs, delta, dim, thresh);

  double spacingCoeff = delta * delta * (problemIs3D ? delta : 1);
  return spacingCoeff * dotProduct(gradPhi, normalizedGradF) *
```

```
21     dotProduct(gradChi, normalizedGradF);
22 }
```

Breaking down Listing 5.5, `computeGradChi` was provided previously, and `computeGradPhi` and `computeNormalizedGradF` are provided in Listings 5.6 and 5.7.

Listing 5.6: Computation of $\nabla\Phi$.

```
1 double3 computeGradPhi(const int3& ptc_crs, const int3& ptr_crs, double delta,
2   const double3& lim, bool isProblem3D) {
3   double3 ptc = crs2xyz(ptc_crs, delta, lim);
4   double3 ptr = crs2xyz(ptr_crs, delta, lim);
5   double3 diff = {ptc.x - ptr.x, ptc.y - ptr.y, ptc.z - ptr.z};
6   double magDiff = computeMagnitude(diff);
7   if (magDiff < EPS) {
8     throw std::runtime_error("Zero magnitude exception.");
9   }
10  double coeff = isProblem3D ? -1 / ((4 * M_PI) * std::pow(magDiff, 3))
11                             : -1 / ((2 * M_PI) * std::pow(magDiff, 2));
12  return make_double3(coeff * diff.x, coeff * diff.y, coeff * diff.z);
13 }
```

Listing 5.7: Computation of $\frac{\nabla f}{|\nabla f|}$.

```
1 double3 computeNormalizedGradF(const double *grid, const double *stencil,
2   int rad, const int3& pt_crs, double delta, const dim3& dim) {
3   double3 gradF = {xDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0),
4                    yDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0),
5                    zDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0)};
6   double magGradF = computeMagnitude(gradF);
7   if (magGradF == 0.0) {
8     throw std::runtime_error("Zero magnitude exception.");
9   }
10  return make_double3(gradF.x / magGradF, gradF.y / magGradF,
```

```
11                          gradF.z / magGradF);
12 }
```

Boundary array is then constructed by sampling the the given boundary function, $b(\boldsymbol{p})$, solely on the irregular points. The resulting $A\beta = b$ system that has been created by taking only the irregular points into account is then solved for $\beta$ using an LU decomposition/solver provided by the Eigen C++ package [43]. Finally, $u$ is reconstructed and the results are written to a comma-separated value (CSV) file. The resulting file is then plotted using a plotting software such as Mathematica [44]. The rest of the code is provided in Listing A.9 in the Appendix A.

For the purposes of showing a sample application of boundary integral with direct integration, here we look at a simple heat transfer problem on a disk. Both the interior and the exterior region (also known as interior Dirichlet and exterior Dirichlet problems respectively) are solved. While the Equation 5.15 is used in solution of the interior Dirichlet problem, for the exterior Dirichlet problem, integral equation for the exterior problem differs only by a sign as [35]

$$\int_{\partial\Omega} \beta(\boldsymbol{q}) \frac{\partial \Phi}{\partial n_{\boldsymbol{q}}^{+}} ds(\boldsymbol{q}) - \frac{1}{2}\beta(\boldsymbol{p}) = b(\boldsymbol{p}). \tag{5.22}$$

The slight tweaking that is needed to be done in code is a single liner, where Listing 5.4 needs to be changed to

Listing 5.8: Subtracting $\frac{1}{2}\delta_{ij}$ instead of adding for the exterior problem.

```
1  Eigen::MatrixXf A = F - Eigen::MatrixXf::Identity(N, N) / 2.0;
```

For the purposes of our example, our 2D domain is defined by $x, y \in [-2, 2]$, and the disk on xy-plane is centered at the origin with radius 1. The boundary function is given as $f(x) = x$.

Since this is a steady state problem, it is a special case, and its solution can be expressed as

$$u(r,\theta) = a_0 + \sum_{n=1}^{\infty} \left[ a_n \Big(\frac{r}{R}\Big)^n \cos n\theta + b_n \Big(\frac{r}{R}\Big)^n \sin n\theta \right] \tag{5.23}$$

where R is the radius of the disk, and which, in the case of the boundary condition $f(x) = x$, can be simplified as

$$u(r,\theta) = \frac{r}{R} \cos\theta \tag{5.24}$$

which, in turn, can be converted to rectangular coordinates as

$$u(x,y) = \frac{x}{R} \tag{5.25}$$

Mathematica plot of the results over the computational grid obtained from the direct boundary integration code discussed above is shown in Figure 5.1. The solution for the interior region looks reasonable where for the exterior values, they are both not relevant and fail to satisfy the BCs. This is expected, since the solver is supposed to solve only the interior problem.

Isolating the 2D disk and plotting in Mathematica again, Figure 5.2 shows only the interior domain solution, which is the meaningful of the part of the Figure 5.1.

The difference between the sampled values of the exact solution given in Equation 5.25 and the numerical results from the direct integration solution on the interior grid points is plotted in Figure 5.3. It can be observed that there is a small systematic error away from the boundaries. As we approach the boundary, there occurs a more noisy region with higher errors. Overall, the maximum error is still less than 5%.

Following the same approach for the exterior region, with the updated line as shown in Listing 5.8 with the same geometry and boundary conditions, Figure 5.4 shows the results of the same heat distribution problem for the exterior region. This time, it is the exterior region that looks plausible and the interior region that looks off. Again, this is expected, since the solver is supposed to solve only the exterior problem.

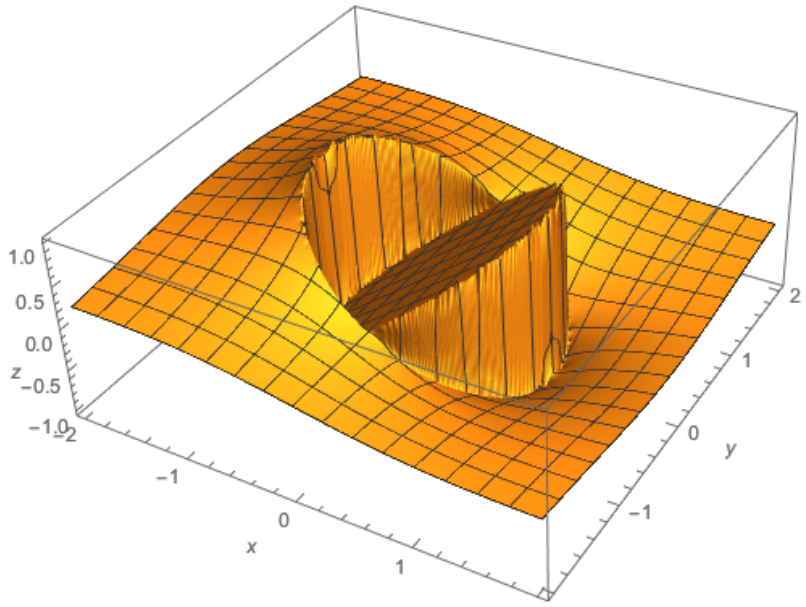



Figure 5.1: Boundary integral solution with direct integration method of the interior problem for heat distribution in a 2D disk with radius 1 centered at origin plotted in full computational grid with Mathematica.

This time, isolating the exterior region of the disk, we obtain Figure 5.5 which shows only the points in the exterior domain solution, which is the meaningful of the part of the Figure 5.4.

Finally, along the same lines with Figure 5.3, we plot the difference between the analytical solution and the boundary integral solution with direct integration method of the problem for the exterior domain. Figure 5.6 shows the differences. The maximum difference is again less than 5% even near the boundaries, and the occurrence of a low systematic error away from the boundaries is also observed.

While Figures 5.1 to 5.6 show the applicability of direct integration to boundary problems qualitatively, we can also do a quantitative analysis as well. Here we analyze the root mean square error for the difference between the formula values and numerical results for different grid spacings. For the interior problem, with the radius of the circle, $R = 1$, we log-log plot the RMSE values vs. $R/\Delta$ values, where $\Delta$ is the grid spacing. Figure 5.7 shows this plot

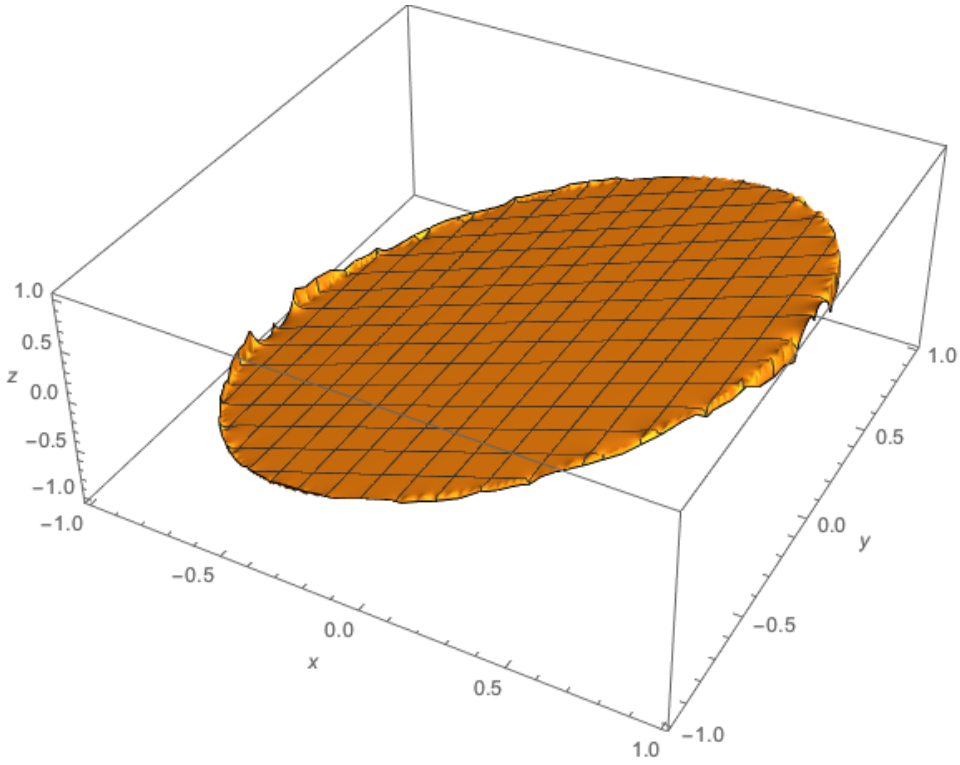



Figure 5.2: Boundary integral solution with direct integration method of the interior problem for heat distribution in a 2D disk with radius 1 centered at origin plotted with Mathematica. Difference from previous figure is that this plot only shows the interior domain, which is the relevant part from the previous figure.

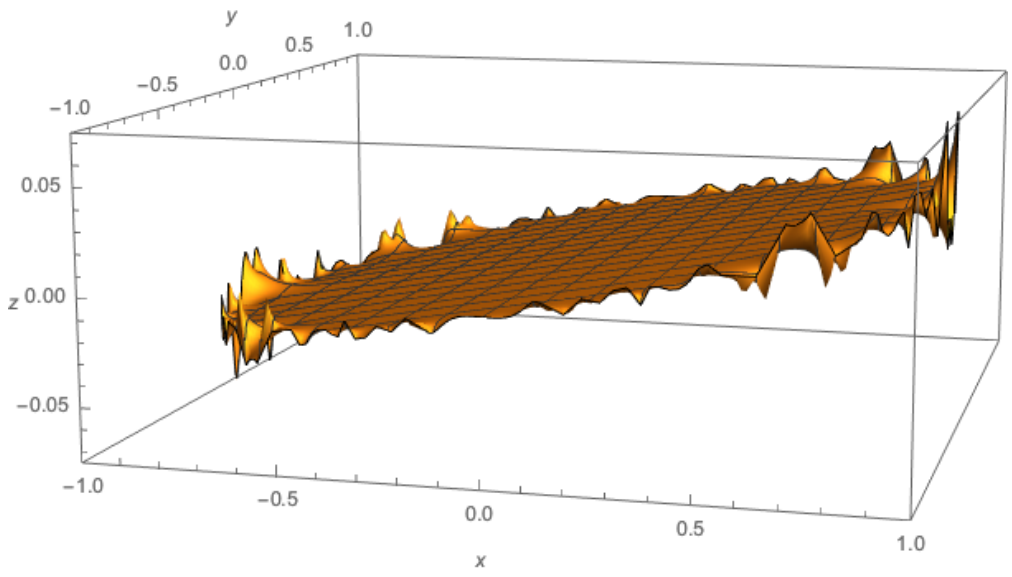



Figure 5.3: Difference between the analytical solution and the boundary integral solution with direct integration method of the interior problem for heat distribution in a 2D disk with radius 1 centered at origin plotted with Mathematica. Notice the z axis' scale difference compared to previous figures. This shows that the error is very low in general except the boundaries where there is some noise which can be mitigated with increase in grid resolution.

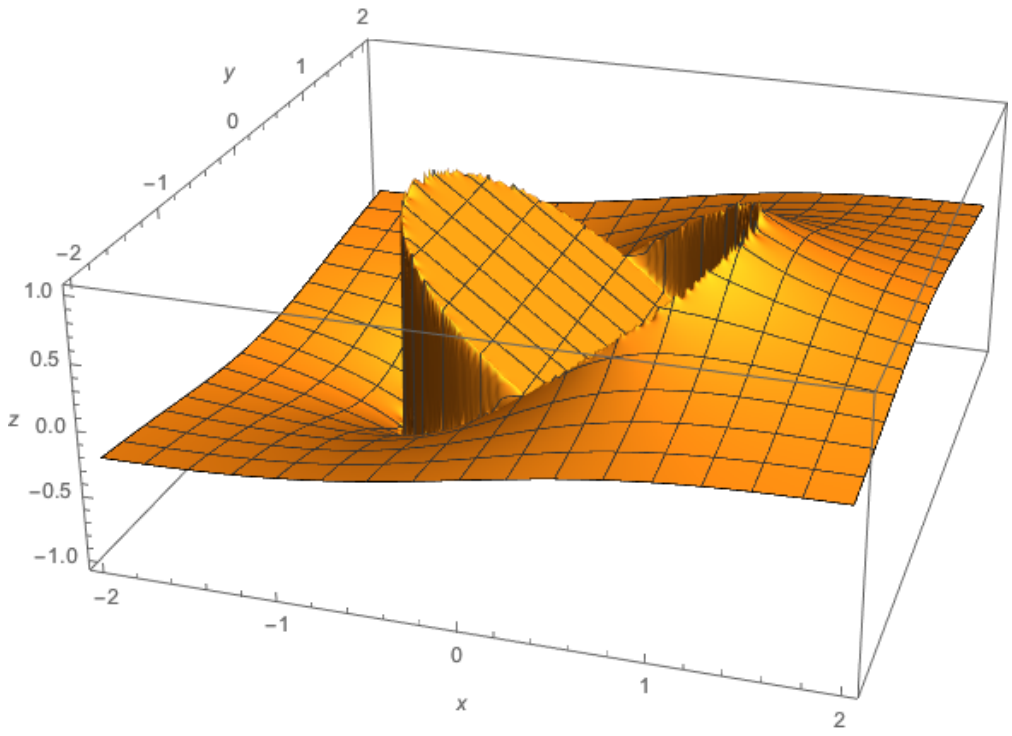



Figure 5.4: Boundary integral solution with direct integration method of the exterior problem for heat distribution in a 2D disk with radius 1 centered at origin plotted in full domain with Mathematica.

and the clear convergence trend. As the $\Delta$ gets smaller, so do the RMSE values.

Applying same type of analysis to the exterior problem, we obtain Figure 5.8 which shows a similar behavior as Figure 5.7.

### *5.4 GPU Implementation and Performance Analysis*

Here, the parallelization of the steps that were presented in Section 5.2 is discussed. Due to the nature of the algorithms implemented used in the solution of the problem, some CUDA libraries designed for high performance computing and linear algebra are needed. When it comes to solving systems of linear algebraic equations with CUDA, there is no need for custom kernel implementations because a large amount of effort by other researchers has been directed at creating efficient parallel linear solvers. As it is the case with the widely used

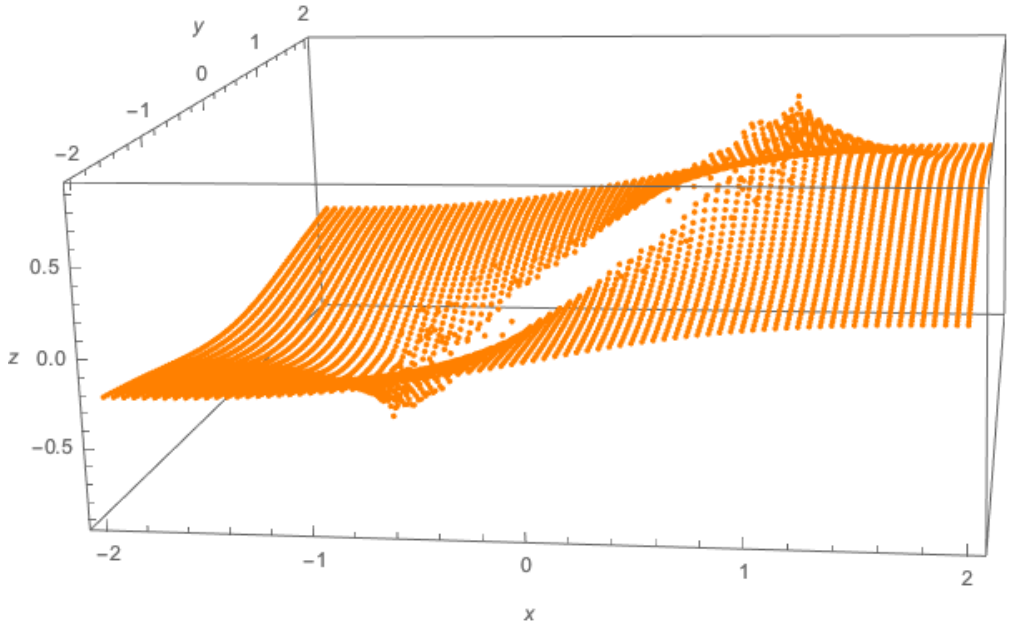



Figure 5.5: Points on the exterior domain of the exterior problem for heat distribution in a 2D disk with radius 1 centered at origin.

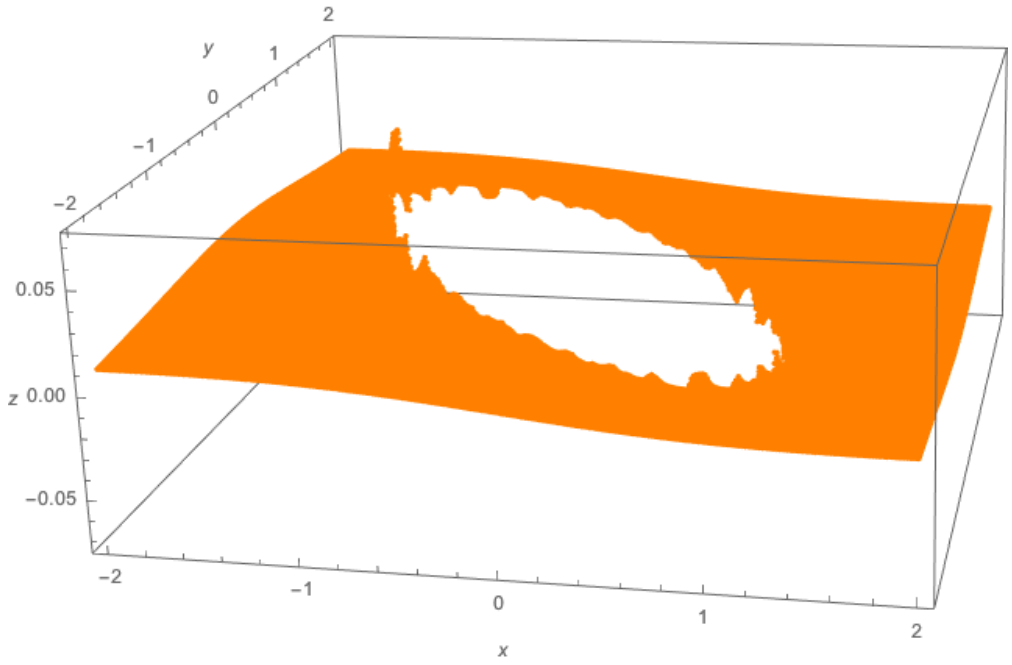



Figure 5.6: Difference between the analytical solution and the boundary integral solution with direct integration method of the exterior problem for heat distribution in a 2D disk with radius 1 centered at origin plotted with Mathematica. Similar to Figure 5.3, the z-axis scale is much lower to capture the difference in the exterior region.

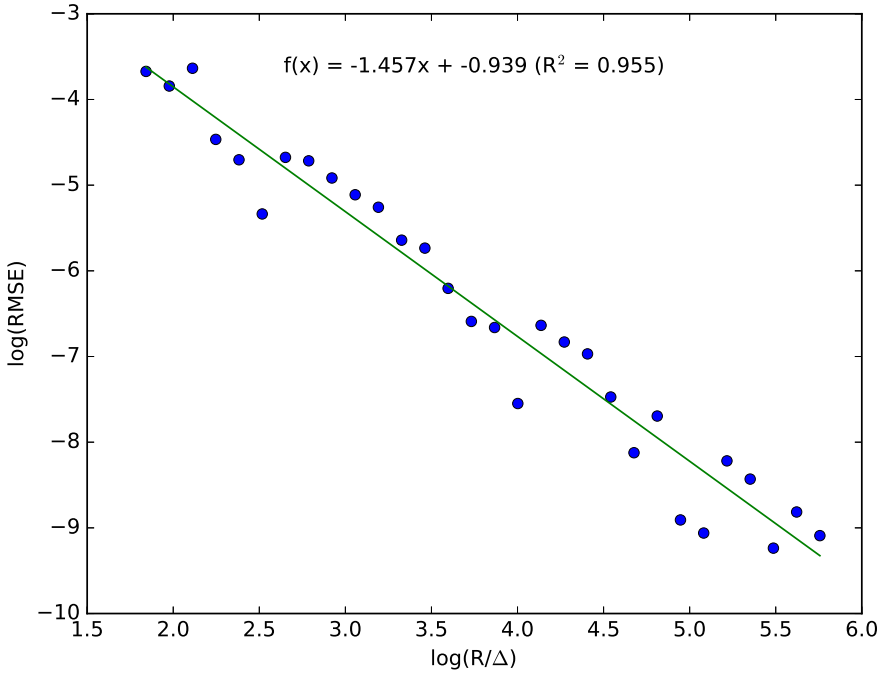



Figure 5.7: Loglog plot with curve fitting of RMSE vs. $R/\Delta$ for the interior heat distribution problem.

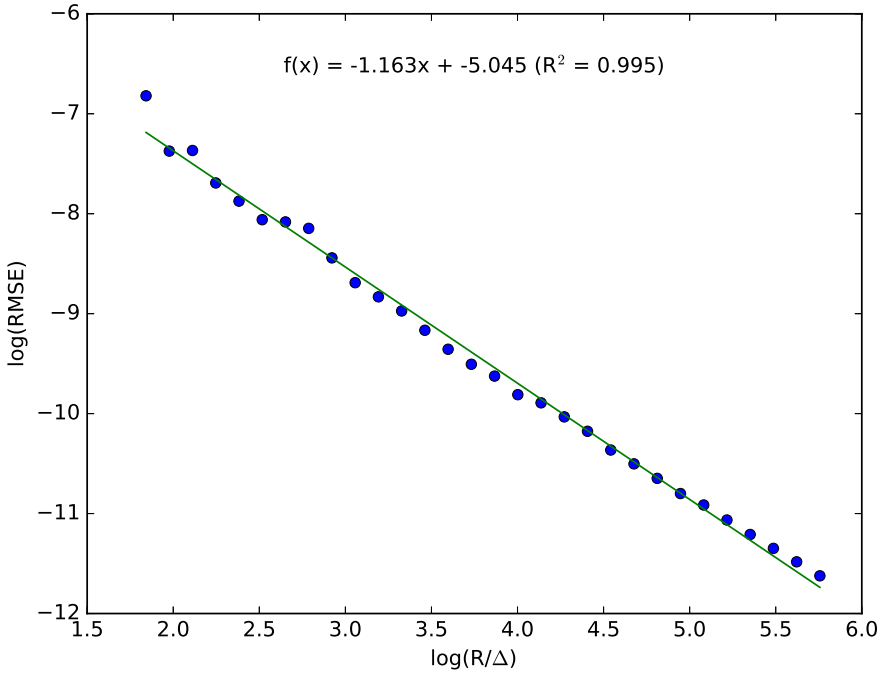



Figure 5.8: Loglog plot with curve fitting of RMSE vs. $R/\Delta$ for the exterior heat distribution problem.

CPU-based linear algebra library Basic Linear Algebra Subprograms (BLAS), NVIDIA has a GPU implementation called cuBLAS. More recently introduced, cuSOLVER is also an analog of a well-known CPU-based linear algebra library called Linear Algebra Package (LAPACK) [45, 46, 47, 12]. These two libraries provide us the required linear algebra functionality with the familiarity of these well-known libraries and with high performance promised by CUDA. Listing 5.9 shows the kernel wrapper function that is used for this problem. As it can be seen on lines 33, 45, 54, and 109, in addition to the library functions, we use 4 different custom kernel functions.

Listing 5.9: Wrapper function that calls the surface area kernel function

```
1 void parallelSolveSystem(double *u, const double *grid, const double *stencil,
2   int rad, const double3& lim, double delta, const dim3& dim) {
3   const int problemSize = dim.x * dim.y * dim.z;
4   double *d_grid = 0;
5   double *d_stencil = 0;
6   int3 *d_irreg_pts = 0;
7   int *d_irreg_count = 0;
8   dim3 *d_dim = 0;
9   double3 *d_lim = 0;
10
11  cudaMalloc(&d_grid, problemSize * sizeof(double));
12  cudaMemcpy(d_grid, grid, problemSize * sizeof(double),
13    cudaMemcpyHostToDevice);
14
15  cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
16  cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
17    cudaMemcpyHostToDevice);
18
19  cudaMalloc(&d_irreg_pts, problemSize * sizeof(int3));
20
21  cudaMalloc(&d_irreg_count, sizeof(int));
```

```
22  cudaMemset(d_irreg_count, 0, sizeof(int));
23
24  cudaMalloc(&d_dim, sizeof(dim3));
25  cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
26
27  cudaMalloc(&d_lim, sizeof(double3));
28  cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
29
30  const dim3 filterKernelBlockSize(TX, TY, TZ);
31  const dim3 filterKernelGridSize(divUp(dim.x, TX), divUp(dim.y, TY),
32                                  divUp(dim.z, TZ));
33  filterKernel<<<filterKernelGridSize, filterKernelBlockSize>>>(d_irreg_pts,
34    d_irreg_count, d_grid, d_stencil, rad, delta, d_dim);
35
36  int irreg_count = 0;
37  cudaMemcpy(&irreg_count, d_irreg_count, sizeof(int),
38    cudaMemcpyDeviceToHost);
39  double *d_F = 0;
40  cudaMalloc(&d_F, irreg_count * irreg_count * sizeof(int3));
41
42  const dim3 matrixKernelBlockSize(TX, TY);
43  const dim3 matrixKernelGridSize(divUp(irreg_count, TX),
44                                  divUp(irreg_count, TY));
45  computeFMatrixKernel<<<matrixKernelGridSize, matrixKernelBlockSize>>>(d_F,
46    irreg_count, irreg_count, d_irreg_pts, d_grid, d_stencil, rad, d_lim,
47    delta, d_dim);
48
49  double *d_g = 0;
50  cudaMalloc(&d_g, irreg_count * sizeof(double));
51
52  const dim3 boundaryKernelBlockSize(TX1D);
53  const dim3 boundaryKernelGridSize(divUp(irreg_count, TX1D));
54  boundaryKernel<<<boundaryKernelGridSize, boundaryKernelBlockSize>>>(d_g,
```

```
55    irreg_count, d_irreg_pts, delta, d_lim);
56
57  const int lda = irreg_count, ldb = irreg_count;
58
59  // Initialize the CUSOLVER and CUBLAS context.
60  cusolverDnHandle_t cusolverDnH = 0;
61  cublasHandle_t cublasH = 0;
62  cusolverDnCreate(&cusolverDnH);
63  cublasCreate(&cublasH);
64
65  // Initialize solver parameters.
66  double *tau = 0, *work = 0;
67  int *devInfo = 0, Lwork = 0;
68  cudaMalloc(&tau, irreg_count * sizeof(double));
69  cudaMalloc(&devInfo, sizeof(int));
70  const double alphaCoeff = 1;
71
72  // Calculate the size of work buffer needed.
73  cusolverDnDgeqrf_bufferSize(cusolverDnH, irreg_count, irreg_count, d_F,
74    lda, &Lwork);
75  cudaMalloc(&work, Lwork*sizeof(double));
76
77  // A = QR with CUSOLVER
78  cusolverDnDgeqrf(cusolverDnH, irreg_count, irreg_count, d_F, lda, tau, work,
79    Lwork, devInfo);
80
81  cudaDeviceSynchronize();
82
83  // z = (Q^T)b with CUSOLVER, z is m x 1
84  cusolverDnDormqr(cusolverDnH, CUBLAS_SIDE_LEFT, CUBLAS_OP_T, irreg_count, 1,
85    irreg_count, d_F, lda, tau, d_g, ldb, work, Lwork, devInfo);
86  cudaDeviceSynchronize();
87
```

```
88  // Solve Rx = z for x with CUBLAS, x is n x 1.
89  cublasDtrsm(cublasH, CUBLAS_SIDE_LEFT, CUBLAS_FILL_MODE_UPPER, CUBLAS_OP_N,
90    CUBLAS_DIAG_NON_UNIT, irreg_count, 1, &alphaCoeff, d_F, lda, d_g, ldb);
91
92  double *d_beta = d_g;
93  cublasDestroy(cublasH);
94
95  cusolverDnDestroy(cusolverDnH);
96
97  cudaFree(d_F);
98  cudaFree(tau);
99  cudaFree(devInfo);
100 cudaFree(work);
101
102 double *d_u = 0;
103 cudaMalloc(&d_u, problemSize * sizeof(double));
104 cudaMemset(d_u, 0, problemSize * sizeof(double));
105
106 const dim3 fullAKernelblockSize(TX1D);
107 const dim3 fullAKernelgridSize(divUp(dim.x * dim.y * dim.z, TX1D));
108
109 solveForTheFullAKernel<<<fullAKernelgridSize, fullAKernelblockSize>>>(d_u,
110   d_beta, d_irreg_pts, d_grid, d_stencil, rad, d_lim, delta,
111   d_dim, irreg_count);
112
113 cudaMemcpy(u, d_u, problemSize * sizeof(double), cudaMemcpyDeviceToHost);
114 cudaFree(d_u);
115 cudaFree(d_stencil);
116 cudaFree(d_grid);
117 cudaFree(d_irreg_pts);
118 cudaFree(d_irreg_count);
119 cudaFree(d_dim);
120 cudaFree(d_lim);
```

```
121   cudaFree(d_g);
122 }
```

The first kernel, `filterKernel`, of which the code is provided in Listing 5.10, detects and filters the irregular points, and populates the array of irregular points.

Listing 5.10: Wrapper function that calls the surface area kernel function

```
1  __global__
2  void filterKernel(int3 *irreg_arr, int *irreg_count, const double *grid,
3    const double *stencil, int rad, double delta, const dim3 *dim) {
4    const int c = threadIdx.x + blockDim.x * blockIdx.x;
5    const int r = threadIdx.y + blockDim.y * blockIdx.y;
6    const int s = threadIdx.z + blockDim.z * blockIdx.z;
7    if (c >= dim->x || r >= dim->y || s >= dim->z) return;
8    int3 pt = {c, r, s};
9    if (isIrregular(pt, grid, stencil, rad, delta, *dim, EPS)) {
10     irreg_arr[atomicAdd(irreg_count, 1)] = make_int3(c, r, s);
11   }
12 }
```

The second, `computeFMatrixKernel`, creates and computes the elements of the matrix $A_{ij} = \frac{1}{2}\delta_{ij} + h^n\left(\nabla\Phi \cdot \frac{\nabla f}{|\nabla f|}\right)\left(\nabla\chi \cdot \frac{\nabla f}{|\nabla f|}\right)$.
Listing 5.11 shows the code of `computeFMatrixKernel`.

Listing 5.11: Wrapper function that calls the surface area kernel function

```
1  __global__
2  void computeFMatrixKernel(double *F_arr, int width, int height,
3    const int3 *irregular_pts, const double *grid, const double *stencil,
4    int rad, const double3 *lim, double delta, const dim3 *dim) {
5    // Column-major ordering
6    const int c = threadIdx.x + blockDim.x * blockIdx.x;
7    const int r = threadIdx.y + blockDim.y * blockIdx.y;
```

```
8   if (c >= width || r >= height) return;
9
10  if (r == c) {
11    F_arr[c * height + r] = 1.0 / 2.0;
12    return;
13  }
14
15  F_arr[c * height + r] = computeElementOfA(irregular_pts[c],
16    irregular_pts[r], grid, stencil, rad, *lim, delta, *dim);
17 }
```

The third kernel is `boundaryKernel`, which creates and computes the elements of the boundary array for the irregular points that were found with `filterKernel`. The code for `boundaryKernel` is provided in Listing 5.12.

Listing 5.12: Wrapper function that calls the surface area kernel function

```
1 __global__
2 void boundaryKernel(double *g_arr, int N, const int3* irregular_pts,
3   double delta, const double3 *lim) {
4   const int i = threadIdx.x + blockDim.x * blockIdx.x;
5   if (i >= N) return;
6   g_arr[i] = boundaryFunc(crs2xyz(irregular_pts[i], delta, *lim));
7 }
```

The fourth and final kernel is `solveForTheFullAKernel`, which as its name implies, solves the system for all the points (hence the full A). Listing 5.13 shows the kernel code.

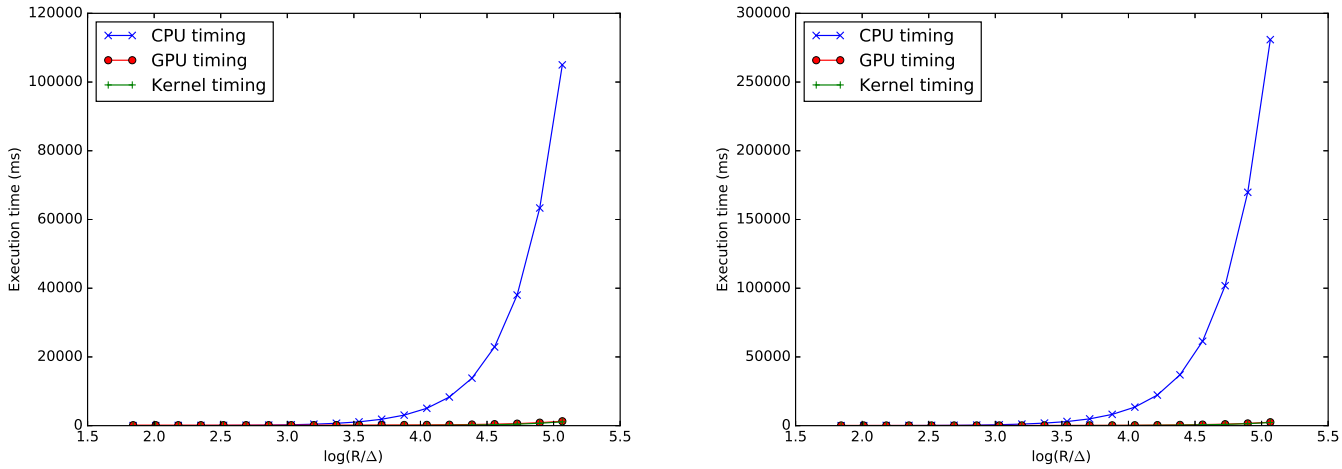
Listing 5.13: Wrapper function that calls the surface area kernel function

```
1 __global__
2 void solveForTheFullAKernel(double *u, const double *beta,
3   const int3 *irregular_pts, const double *grid, const double *stencil,
4   int rad, const double3 *lim, double delta, const dim3 *dim,
```

```
5   int irregular_pt_count) {
6
7   const int r = threadIdx.x + blockDim.x * blockIdx.x;
8   if (r >= dim->x * dim->y * dim->z) return;
9
10  for (int c = 0; c < irregular_pt_count; c++) {
11    u[r] += computeElementOfAWithoutThreshold(flatIdx2crs(r, *dim),
12      irregular_pts[c], grid, stencil, rad, *lim, delta, *dim) * beta[c];
13  }
14 }
```

In addition to these custom kernel functions, here, since we are solving a linear algebra system such as $Ax = b$, we follow a couple of steps. First of all we apply QR factorization, $A = QR$ where $Q$ is orthonormal ($Q^TQ = I$) and R is upper triangular [48]. For this, we use the cuSOLVER function named `cusolverDnDgeqrf`, which is the function that can apply the QR factorization on any real double precision $m$ by $n$ matrix. Premultiplying $QRx = b$ by both sides with $Q^T$, we obtain $Rx = Q^Tb$. The vector $z = Q^Tb$ is computed with the cuSOLVER function `cusolverDnDormqr`. Finally, The system $Rx = z$ is solved using a backsolver from the cuBLAS library, named `cublasDtrsm`, which is a solver for real double precision triangular systems. Since $R$ is upper triangular, `cublasDtrsm` is the ideal function for this.

Combining these custom kernels and library functions with the helper functions (full Listing is provided in Appendix A, Listing A.10), we now have the parallel implementation that gives the exact same results that were analyzed in the previous section. Since the parallel implementation returns identical results as the serial implementation, we focus on performance and execution times of the CUDA implementation including results using connection coefficients for the first three genera. Figures 5.9 and 5.10 show the performance comparison between the serial implementation and the CUDA implementation for the first three genera. Consistent with our previous experiences, we again see a speedup of about 100x with the usage of GPU.




(a) (b)

Figure 5.9: Timing comparison for boundary line integral problem in wavelet genus 1 (a) and 2 (b).

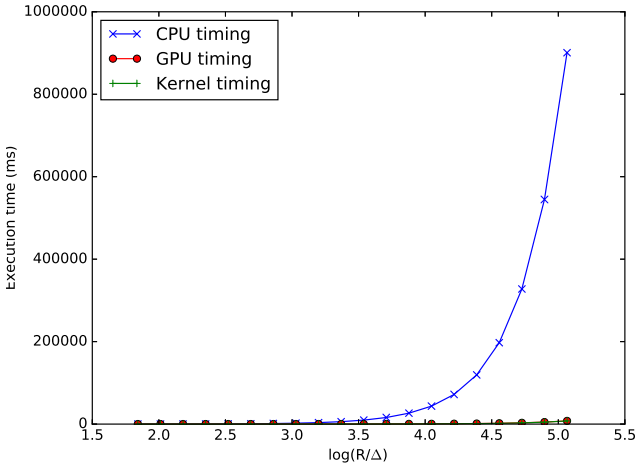



Figure 5.10: Timing comparison for boundary line integral problem in wavelet genus 3.

# Chapter 6

# CONCLUSIONS

In this final chapter, we discuss the meaning of the results of this study and the conclusions that can be drawn from them.

In each of the Chapters 2, 3, and 4, we studied the theory behind the direct integration for the integral operation that was the topic of the chapter and performed related numerical experiments. The results we obtained from numerical experiments and convergence observed as the grid spacing is reduced were consistent with Resnikoff and Wells' convergence theory which can be invoked on regular grid data derived from twice differentiable functions [6]. In addition to presenting direct integration methods under a unified framework and testing their convergence, general purpose graphics processing unit (GPGPU) programming techniques for acceleration of direct integration methods were also investigated. Results from GPGPU tests suggest that huge performance gains without any compromise on accuracy can be obtained with the help of CUDA. The main inspiration for this work came from previous work on grid-based surface integration [5]. Other implicit modeling and GPGPU research previously done in our lab such as M. Ensz's "Implicit solid modelling through manifold modification" [49], G. Marchelli's "GPU-Accelerated Tools for Medical Image Registration and Biomechanical Modeling" [50], and D. Zhang's "GPU Accelerated Signed Distance Voxel Modeling Systems" [51] were also motivational for this study. Kublik et al.'s paper on Implicit Boundary Integral Method (IBIM) was also critical in development of direct integration solution for boundary integral problems [41].

In Chapter 2, the topic of surface integrals was studied. Using the 19th century Schwarz Lantern paradox as a motivating example for the need of a more direct method than polygonization, we developed the theory using signum function and the Divergence Theorem.

After obtaining the integral formula, we discretized it and used Daubechies wavelet connection coefficients for the derivative stencils. Implementing the discrete formula in C++, we solved some example cases such as torus surface area, toroidal shell moment of inertia, and flame surface computation using direct numerical simulation (DNS) data [28]. Results show converging behavior, and another advantage was the bypassing of the polygonization step which may introduce noise and diverging behavior. Finally, we concluded the chapter with the GPGPU versions of the same samples. Significant performance gains in the order of 100x were observed with the CUDA implementations.

Chapter 3 treated the problem of computing volume integrals. This time, we implemented the solution of torus volume integration and compared it to to a straightforward Monte-Carlo implementation. It was observed that the direct integration performed at least as well as our Monte-Carlo implementation. In addition to just the volume computation, solid torus moment of inertia computation was also done and convergent behavior was also observed with it. Performance comparisons again showed that the CUDA implementation helped speeding up computations up to about 50x compared to direct integration on the CPU and about 10x compared to Monte-Carlo computations on the CPU. Comparing direct integral method to Monte-Carlo, another benefit was the bypassing of the intermediate step of pseudo-random number generation. Thanks to the direct integral method, the non-deterministic nature of Monte-Carlo were also eliminated.

Line integrals were the topic of Chapter 4. Using Stokes' and Divergence theorems, we developed the formulation needed for computation of line integrals. This time, the resulting formula and its implementation was a little bit more complicated but it was in the same nature of the previous two integrals. The case study made for this integral was about the length of the intersection curve between a pair of implicitly defined surfaces. Since it is possible to analytically calculate the intersection length of two intersecting spheres in a given configuration, we chose intersecting spheres as the sample of the chapter. In addition to just two spheres intersecting, we also applied a cutting algorithm on it, where the spheres were cut using a given plane. Results showed converging behavior in computing the intersection

line length and the CUDA implementation showed a speedup of almost 100x.

Finally, we devoted Chapter 5 to an application of direct integration of boundary integrals. Research of alternative methods to Finite Elements and Boundary Elements methods is an important topic and there is a big body of research done on this topic. Taking inspiration from some of the work done on this field, we applied our direct integration approach to the integral equation that is being used to solve a Dirichlet boundary value problem. This time, both the serial implementation and the CUDA implementation were a little more complicated as linear algebra operations and classification of points were operations that needed application of linear algebra libraries and some advanced C++ functionality. On the CUDA side, some CUDA libraries designed for high performance computing and linear algebra such as cuBLAS and cuSolver were used. In the same theme with the other Chapters, CUDA implementation showed performance gains in the order of 100x.

In summary, this study aimed at two main topics. Computation of integral properties of digital grid representations and performance optimization of these computations using GPU computing techniques. Results of the serial implementations were analyzed to observe that the proposed method of direct integration for integral properties performed accurately and showed converging behavior. This is promising, since it shows hope in methods that directly compute the desired quantities instead of using intermediate methods such as parametrization. The areas of application extend from 3D printing to medical applications and visualization software.

Considering the application areas of these methods, one of the most crucial aspects is performance. Modern hardware/software developments enable highly performant CAD and visualization tools and if a method that claims to be accurately measuring integral properties is going to be used in visualization, it needs to be highly performant. While in most chapters the development and conversation of discretized formulas were done using CPU implementations, it was observed that they were not very performant. For example, a high resolution computation of surface area of a digitized object required about 30 seconds. This is unacceptable for the goal of achieving live and interactive modeling. This was the point

were the second part of this study, GPU computing came into play. Every C++ implementation provided in this study was also converted to CUDA and performance analysis was done. For the aforementioned high resolution surface area computation that took about 30 seconds, the computation time was reduced to about 0.3 seconds including the overhead of memory transfers. Since in the live visualization case there are opportunities for interoperation between CUDA and graphics libraries, there is further possibility for performance gains potentially reducing runtime of each computation to about 0.1 second. While this may still be a little slower for showing the result at 24 fps, an optimization can be made for showing the result for every 3 frames or a slightly less accurate computation with coarser digitization can be chosen in order to have the results for each frame faster.

Overall, combination of direct methods with GPU computing is promising in offering a great alternative to mainstream CAD and visualization tools and this study shows that it is possible to obtain reliable results quickly in analyzing integral properties of objects with the proposed methods, providing availability for live applications such as medical analysis and live engineering design tools.

# BIBLIOGRAPHY

[1] C. Lin and J. Miller, "3d characterization and analysis of particle shape using x-ray microtomography (xmt)," *Powder Technology*, vol. 154, no. 1, pp. 61–69, 2005.

[2] M. A. Taylor, E. J. Garboczi, S. Erdogan, and D. Fowler, "Some properties of irregular 3-d particles," *Powder Technology*, vol. 162, no. 1, pp. 1–15, 2006.

[3] J. Lindblad and I. Nyström, "Surface area estimation of digitized 3d objects using local computations," in *Discrete Geometry for Computer Imagery*, pp. 267–278, Springer, 2002.

[4] Y. T. Lee and A. A. Requicha, "Algorithms for computing the volume and other integral properties of solids. i. known methods and open issues," *Communications of the ACM*, vol. 25, no. 9, pp. 635–641, 1982.

[5] D. Storti, "Using lattice data to compute surface integral properties of digitized objects," *Proceedings of IDMME—Virtual Concept, Bordeaux, France*, 2010.

[6] H. L. Resnikoff, O. Raymond Jr, *et al.*, *Wavelet analysis: the scalable structure of information*. Springer Science & Business Media, 2012.

[7] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[8] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[9] M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[10] N. Wilt, *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[11] NVIDIA, "Cuda C programming guide, v7.5," 2015.

[12] D. Storti and M. Yurtoglu, *CUDA for Engineers: An Introduction to High-performance Parallel Computing*. Addison-Wesley Professional, 2015.

[13] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[14] M. Harris, "Inside pascal: Nvidia's newest computing platform," *Retrieved from Parallel Forall: https://devblogs.nvidia.com/parallelforall/inside-pascal/*, 2016.

[15] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *ACM siggraph computer graphics*, vol. 21, pp. 163–169, ACM, 1987.

[16] A. Lopes and K. Brodlie, "Interactive approaches to contouring and isosurfacing for geovisualization," 2004.

[17] D. Akio and A. Koide, "An efficient method of triangulating equi-valued surfaces by using tetrahedral cells," *IEICE TRANSACTIONS on Information and Systems*, vol. 74, no. 1, pp. 214–224, 1991.

[18] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of hermite data," in *ACM transactions on graphics (TOG)*, vol. 21, pp. 339–346, ACM, 2002.

[19] E. L. Allgower and S. Gnutzmann, "Simplicial pivoting for mesh generation of implicity defined surfaces," *Computer Aided Geometric Design*, vol. 8, no. 4, pp. 305–325, 1991.

[20] F. Zames, "Surface area and the cylinder area paradox," *The Two-Year College Mathematics Journal*, vol. 8, no. 4, pp. 207–211, 1977.

[21] J. J. Koenderink and J. M. Rabins, "Solid shape," *Applied Optics*, vol. 30, p. 714, 1991.

[22] M. Berger, *Geometry I*. Springer Science & Business Media, 2009.

[23] G. Beck and I. Hafner, "Cylinder area paradox." http://demonstrations.wolfram.com/CylinderAreaParadox/, 2008.

[24] H. M. Schey, *Div, grad, curl, and all that: an informal text on vector calculus*. WW Norton & Company, 1996.

[25] I. Daubechies *et al.*, *Ten lectures on wavelets*, vol. 61. SIAM, 1992.

[26] P. Oonincx, "Daubechies wavelets," *Encyclopedia of Mathematics: https://www.encyclopediaofmath.org/index.php/Daubechies_wavelets*, 2011.

[27] Torus, "Torus — Wikipedia, the free encyclopedia," 2002. [Online; accessed 16-October-2016].

[28] W. Wang, *Kinematic study of the evolution and properties of flame surfaces in turbulent nonpremixed combustion with local extinction and reignition*. PhD thesis, 2013.

[29] Y.-S. Liu, J. Yi, H. Zhang, G.-Q. Zheng, and J.-C. Paul, "Surface area estimation of digitized 3d objects using quasi-monte carlo methods," *Pattern Recognition*, vol. 43, no. 11, pp. 3900–3909, 2010.

[30] T. Smith, G. Lange, and W. Marks, "Fractal methods and results in cellular morphology—dimensions, lacunarity and multifractals," *Journal of neuroscience methods*, vol. 69, no. 2, pp. 123–136, 1996.

[31] D. Storti, C. Finley, and M. Ganter, "Interval extensions of signed distance functions: isdf-reps and reliable membership classification," *Journal of Computing and Information Science in Engineering*, vol. 10, no. 2, p. 021012, 2010.

[32] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, and P. B. Kramer, "Numerical recipes: the art of scientific computing," 1987.

[33] A. Efimov, "Lipschitz condition," *Encyclopedia of Mathematics: https://www.encyclopediaofmath.org*, 2014. Accessed on 2016-09-01.

[34] J. Tatum, "Classical mechanics," *Texts*, vol. 2008, pp. 2002–2008, 2000.

[35] P. K. Kythe, *An introduction to boundary element methods*, vol. 4. CRC press, 1995.

[36] M. Costabel, "Principles of boundary element methods," *Computer Physics Reports*, vol. 6, no. 1, pp. 243–274, 1987.

[37] L. C. Wrobel, *The boundary element method, applications in thermo-fluids and acoustics*, vol. 1. John Wiley & Sons, 2002.

[38] J. Trevelyan, *Boundary elements for engineers: theory and applications*, vol. 1. Computational Mechanics, 1994.

[39] M. Aliabadi and P. Wen, *Boundary element methods in engineering and sciences*, vol. 4. World Scientific, 2011.

[40] F. Hartmann, *Introduction to boundary elements: theory and applications*. Springer Science & Business Media, 2012.

[41] C. Kublik, N. M. Tanushev, and R. Tsai, "An implicit interface boundary integral method for poisson's equation on arbitrary domains," *Journal of Computational Physics*, vol. 247, pp. 279–311, 2013.

[42] A.-M. Wazwaz, *A first course in integral equations*. World Scientific, second edition ed., 2015.

[43] G. Guennebaud, B. Jacob, *et al.*, "Eigen: A c++ linear algebra library," 2014.

[44] S. Wolfram, *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.

[45] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[46] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, *et al.*, *LAPACK Users' guide*. SIAM, 1999.

[47] Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, p. 27, 2008.

[48] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU Press, 2012.

[49] M. T. Ensz, "Implicit solid modelling through manifold modification," 1997.

[50] G. L. S. Marchelli, *GPU-Accelerated Tools for Medical Image Registration and Biomechanical Modeling*. PhD thesis, University of Washington, 2015.

[51] D. Zhang, *GPU Accelerated Signed Distance Voxel Modeling Systems*. PhD thesis, University of Washington, 2016.

# Appendix A

# CODE LISTINGS

Listing A.1: Surface integration code.

```
#include <cuda_runtime.h>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <algorithm>
#include <iostream>
#include <random>
#include <cstring>
#define EPS 1e-6
#define MAX_STENCIL_SIZE 25

double torusAtOrigin(double x, double y, double z, double majorRadius,
  double minorRadius) {
    double xyResult = majorRadius - std::sqrt(std::pow(x, 2) + std::pow(y, 2))
        ;
    return std::pow(xyResult, 2) + std::pow(z, 2) - std::pow(minorRadius, 2);
}

double shapeFunc(const double3& pt, double majorRadius, double minorRadius) {
  return torusAtOrigin(pt.x, pt.y, pt.z, majorRadius, minorRadius);
}

double id(double val, double thresh) { return val; }
```

```
24
25 double sgn(double val, double thresh) {
26   return (thresh < val) - (val < thresh);
27 }
28
29 int clip(int i, int iMax) {
30   return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
31 }
32
33 int flatten(const int3& crs, const dim3& dim) {
34   return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
35     clip(crs.z, dim.z) * dim.x * dim.y;
36 }
37
38 double deriv(const double *grid, const double *stencil, int radius,
39   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
40   double (*f)(double val, double thresh), double thresh) {
41   double result = 0.0;
42   for (int i = -radius; i <= radius; i++) {
43     int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
44                                 pt.z + (i * zd)), dim);
45     result += f(grid[loc], thresh) * stencil[radius + i];
46   }
47   return result / delta;
48 }
49
50 double xDeriv(const double *grid, const double *stencil, int radius,
51   const int3& pt, double delta, const dim3& dim,
52   double (*f)(double val, double thresh), double thresh) {
53   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
54 }
55
56 double yDeriv(const double *grid, const double *stencil, int radius,
```

```
57   const int3& pt, double delta, const dim3& dim,
58   double (*f)(double val, double thresh), double thresh) {
59   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
60 }
61
62 double zDeriv(const double *grid, const double *stencil, int radius,
63   const int3& pt, double delta, const dim3& dim,
64   double (*f)(double val, double thresh), double thresh) {
65   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
66 }
67
68 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
69   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
70                       -lim.z + crs.z * delta);
71 }
72
73 double dotProduct(const double3& v, const double3& w) {
74   return v.x * w.x + v.y * w.y + v.z * w.z;
75 }
76
77 double computeMagnitude(const double3& v) { return std::sqrt(dotProduct(v, v))
       ; }
78
79 double3 normalizedVec(const double3& v) {
80   double mag = computeMagnitude(v);
81   return make_double3(v.x / mag, v.y / mag, v.z / mag);
82 }
83
84 double surfArea(const double *grid, const double *stencil, int rad,
85   const double3 &lim, double delta, const dim3 &dim) {
86   double areaSum = 0.0;
87   for (int s = 0; s < dim.z; s++) {
88     for (int r = 0; r < dim.y; r++) {
```

```
      for (int c = 0; c < dim.x; c++) {
        int3 pt = make_int3(c, r, s);
        double3 df = make_double3(
          xDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
          yDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
          zDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0));
        double3 dchi = make_double3(
          xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
          zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
        double denom = computeMagnitude(df);
        if (denom < EPS) continue;
        areaSum += dotProduct(df, dchi) / denom;
      }
    }
  }
  return areaSum / 2.0 * std::pow(delta, 3);
}

double3 rotatedPointAboutAxis(const double3& pt, const double3& axis,
  double angleInRadians) {
  angleInRadians *= -1.0; // Reverse the angle
  double3 normalizedAxis = normalizedVec(axis);
  double qi = normalizedAxis.x * std::sin(angleInRadians / 2.0);
  double qj = normalizedAxis.y * std::sin(angleInRadians / 2.0);
  double qk = normalizedAxis.z * std::sin(angleInRadians / 2.0);
  double qr = std::cos(angleInRadians / 2.0);
  return make_double3(
    (1 - 2 * qj * qj - 2 * qk * qk) * pt.x + 2 * (qi * qj - qk * qr) * pt.y +
      2 * (qi * qk + qj * qr) * pt.z,
    2 * (qi * qj + qk * qr) * pt.x + (1 - 2 * qi * qi - 2 * qk * qk) * pt.y +
      2 * (qj * qk - qi * qr) * pt.z,
    2 * (qi * qk - qj * qr) * pt.x + 2 * (qi * qr + qj * qk) * pt.y +
```

```
122       (1 - 2 * qi * qi - 2 * qj * qj) * pt.z);
123 }
124
125 double3 translatedPoint(const double3& pt, const double3& delta) {
126   return make_double3(pt.x - delta.x, pt.y - delta.y, pt.z - delta.z);
127 }
128
129 double* stencilPicker(int genus) {
130   double *stencil = new double[MAX_STENCIL_SIZE]();
131   const double stencils[8][MAX_STENCIL_SIZE] = {
132     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0},
133     {-0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0}, // M = 1
134     {1.0/12.0, -2.0/3.0, 0.0, 2.0/3.0, -1.0/12.0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
135     {-1.0/2920.0, -16.0/1095.0, 53.0/365.0, -272.0/365.0, 0.0, 272.0/365.0,
        -53.0/365.0, 16.0/1095.0, 1.0/2920.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0},
136     {-1.0/1189272.0, 128.0/743295.0, 2645.0/1189272.0, -1664.0/49553.0,
        76113.0/396424.0, -39296.0/49553.0, 0.0, 39296.0/49553.0,
        -76113.0/396424.0, 1664.0/49553.0, -2645.0/1189272.0, -128.0/743295.0,
        1.0/1189272.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
137     {-5.0/18545664272, -2048.0/8113728119, -563818.0/10431936153,
        -1386496.0/5795520085, 17297069.0/2318208034, -735232.0/13780629,
        265226398.0/1159104017, -957310976.0/1159104017, 0.0,
        957310976.0/1159104017, -265226398.0/1159104017, 735232.0/13780629,
        -17297069.0/2318208034, 1386496.0/5795520085, 563818.0/10431936153,
        2048.0/8113728119, 5.0/18545664272, 0, 0, 0, 0, 0, 0, 0, 0},
138     {6.9686511520083E-13, -2.8996668057051E-09, 4.2069120451167E-07,
        1.2026575195723E-05, 4.2968915709948E-06, -1.5885615434757E-03,
        1.4545511041994E-02, -7.2440589997659E-02, 0.25855294414146,
        -0.85013666155592, 0.0, 0.85013666155592, -0.25855294414146,
```

```
        7.2440589997659E-02, -1.4545511041994E-02, 1.5885615434757E-03,
        -4.2968915709948E-06, -1.2026575195723E-05, -4.2069120451167E-07,
        2.8996668057051E-09, -6.9686511520083E-13, 0, 0, 0, 0},
139     {-6.6283900594600E-16, -1.2035273999989E-11, 4.1830548203747E-10,
        -2.1871130331900E-07, -1.6501679210868E-06, 4.2363946800701E-06,
        3.3734404776409E-04, -3.8814546576295E-03, 2.2687411014648E-02,
        -9.0189066217795E-02, 0.28296509452594, -0.86874391452377, 0.0,
        0.86874391452377, -0.28296509452594, 9.0189066217795E-02,
        -2.2687411014648E-02, 3.8814546576295E-03, -3.3734404776409E-04,
        -4.2363946800701E-06, 1.6501679210868E-06, 2.1871130331900E-07,
        -4.1830548203747E-10, 1.2035273999989E-11, 6.6283900594600E-16}
140   };
141   memcpy(stencil, stencils[genus], MAX_STENCIL_SIZE * sizeof(double));
142   return stencil;
143 }
144
145 int main(int argc, char** argv) {
146   if (argc < 3) {
147     std::cerr << "Usage: ./main.exe [delta] [genus]" << std::endl;
148     return 1;
149   }
150   // Initialize the grid
151   std::clock_t startTime, endTime;
152   std::random_device rd;
153   std::default_random_engine gen(rd());
154   const double delta = std::atof(argv[1]);
155   const double3 lim = {15.f, 15.f, 15.f}; // USER INPUT
156   const dim3 dim = dim3(static_cast<uint>(std::ceil(2 * lim.x / delta) + 1),
157                         static_cast<uint>(std::ceil(2 * lim.y / delta) + 1),
158                         static_cast<uint>(std::ceil(2 * lim.z / delta) + 1));
159   const double minorRadius = 2.f;
160   const double majorRadius = 10.f;
161   const int genus = std::atoi(argv[2]);
```

```
162  const int stencilRadius = genus == 1 ? 1 : (genus - 1) * 2;
163  double *stencil = stencilPicker(genus);
164  double *const grid = new double[dim.x * dim.y * dim.z]();
165
166  std::uniform_real_distribution<> delta_dis(-delta, delta);
167  std::uniform_real_distribution<> angle_dis(-M_PI, M_PI);
168  std::uniform_real_distribution<> axis_dis(0.00001, 1);
169  double3 rotationAxis = make_double3(axis_dis(gen), axis_dis(gen), axis_dis(
         gen));
170  double rotationAngle = angle_dis(gen);
171  double3 translationDelta = make_double3(delta_dis(gen), delta_dis(gen),
         delta_dis(gen));
172
173  // Creating the grid
174  for (int s = 0; s < dim.z; s++) {
175    for (int r = 0; r < dim.y; r++) {
176      for (int c = 0; c < dim.x; c++) {
177        int3 pt = make_int3(c, r, s);
178        double3 xyz = crs2xyz(pt, delta, lim);
179        xyz = rotatedPointAboutAxis(xyz, rotationAxis, rotationAngle);
180        xyz = translatedPoint(xyz, translationDelta);
181        grid[flatten(pt, dim)] = shapeFunc(xyz, majorRadius, minorRadius);
182      }
183    }
184  }
185
186  startTime = std::clock();
187  double area = surfArea(grid, stencil, stencilRadius, lim, delta, dim);
188  endTime = std::clock();
189  printf("%f,", area);
190  printf("%f\n",
191    (endTime - startTime) / (double)(CLOCKS_PER_SEC / 1000));
192  delete[] grid;
```

```
193   delete[] stencil;
194 }
```

Listing A.2: Moment of inertia computation using surface integration technique.

```
1  #include <cuda_runtime.h>
2  #include <cstdio>
3  #include <cstdlib>
4  #include <cmath>
5  #include <ctime>
6  #include <algorithm>
7  #include <iostream>
8
9  #define EPS 1e-6 // Replacement for std::numeric_limits<double>::epsilon()
10 #define MAX_STENCIL_SIZE 25
11
12 double torusAtOrigin(double x, double y, double z, double majorRadius,
13   double minorRadius) {
14     double xyResult = majorRadius - std::sqrt(std::pow(x, 2) + std::pow(y, 2))
           ;
15     return std::pow(xyResult, 2) + std::pow(z, 2) - std::pow(minorRadius, 2);
16 }
17
18 double shapeFunc(const double3& pt, double majorRadius, double minorRadius) {
19   return torusAtOrigin(pt.x, pt.y, pt.z, majorRadius, minorRadius);
20 }
21
22 double id(double val, double thresh) { return val; }
23
24 double sgn(double val, double thresh) {
25   return (thresh < val) - (val < thresh);
26 }
27
```

```
28 int clip(int i, int iMax) {
29   return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
30 }
31
32 int flatten(const int3& crs, const dim3& dim) {
33   return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
34     clip(crs.z, dim.z) * dim.x * dim.y;
35 }
36
37 double deriv(const double *grid, const double *stencil, int radius,
38   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
39   double (*f)(double val, double thresh), double thresh) {
40   double result = 0.0;
41   for (int i = -radius; i <= radius; i++) {
42     int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
43                                 pt.z + (i * zd)), dim);
44     result += f(grid[loc], thresh) * stencil[radius + i];
45   }
46   return result / delta;
47 }
48
49 double xDeriv(const double *grid, const double *stencil, int radius,
50   const int3& pt, double delta, const dim3& dim,
51   double (*f)(double val, double thresh), double thresh) {
52   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
53 }
54
55 double yDeriv(const double *grid, const double *stencil, int radius,
56   const int3& pt, double delta, const dim3& dim,
57   double (*f)(double val, double thresh), double thresh) {
58   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
59 }
60
```

```
61 double zDeriv(const double *grid, const double *stencil, int radius,
62   const int3& pt, double delta, const dim3& dim,
63   double (*f)(double val, double thresh), double thresh) {
64   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
65 }
66
67 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
68   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
69                       -lim.z + crs.z * delta);
70 }
71
72 double dotProduct(const double3& v, const double3& w) {
73   return v.x * w.x + v.y * w.y + v.z * w.z;
74 }
75
76 double computeMagnitude(const double3& v) { return std::sqrt(dotProduct(v, v))
     ; }
77
78 double3 normalizedVec(const double3& v) {
79   double mag = computeMagnitude(v);
80   return make_double3(v.x / mag, v.y / mag, v.z / mag);
81 }
82
83 double momentOfInertia(const double *grid, const double *stencil, int rad,
84   const double3 &lim, double delta, const dim3 &dim) {
85   double moiSum = 0.0;
86   for (int s = 0; s < dim.z; s++) {
87     for (int r = 0; r < dim.y; r++) {
88       for (int c = 0; c < dim.x; c++) {
89         int3 pt = make_int3(c, r, s);
90         double3 df = make_double3(
91           xDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
92           yDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
```

```
93            zDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0));
94          double3 dchi = make_double3(
95            xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
96            yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
97            zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
98          double denom = computeMagnitude(df);
99          if (denom < EPS) continue;
100         double3 xyz = crs2xyz(pt, delta, lim);
101         moiSum += ((xyz.x * xyz.x) + (xyz.y * xyz.y)) * dotProduct(df, dchi) /
                denom;
102       }
103     }
104   }
105   return moiSum / 2.0 * std::pow(delta, 3);
106 }
107
108 double3 rotatedPointAboutAxis(const double3& pt, const double3& axis,
109   double angleInRadians) {
110   angleInRadians *= -1.0; // Reverse the angle
111   double3 normalizedAxis = normalizedVec(axis);
112   double qi = normalizedAxis.x * std::sin(angleInRadians / 2.0);
113   double qj = normalizedAxis.y * std::sin(angleInRadians / 2.0);
114   double qk = normalizedAxis.z * std::sin(angleInRadians / 2.0);
115   double qr = std::cos(angleInRadians / 2.0);
116   return make_double3(
117     (1 - 2 * qj * qj - 2 * qk * qk) * pt.x + 2 * (qi * qj - qk * qr) * pt.y +
118       2 * (qi * qk + qj * qr) * pt.z,
119     2 * (qi * qj + qk * qr) * pt.x + (1 - 2 * qi * qi - 2 * qk * qk) * pt.y +
120       2 * (qj * qk - qi * qr) * pt.z,
121     2 * (qi * qk - qj * qr) * pt.x + 2 * (qi * qr + qj * qk) * pt.y +
122      (1 - 2 * qi * qi - 2 * qj * qj) * pt.z);
123 }
124
```

```
125 double3 translatedPoint(const double3& pt, const double3& delta) {
126   return make_double3(pt.x - delta.x, pt.y - delta.y, pt.z - delta.z);
127 }
128
129 double* stencilPicker(int genus) {
130   double *stencil = new double[MAX_STENCIL_SIZE]();
131   const double stencils[8][MAX_STENCIL_SIZE] = {
132     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0},
133     {-0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0}, // M = 1
134     {1.0/12.0, -2.0/3.0, 0.0, 2.0/3.0, -1.0/12.0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
135     {-1.0/2920.0, -16.0/1095.0, 53.0/365.0, -272.0/365.0, 0.0, 272.0/365.0,
        -53.0/365.0, 16.0/1095.0, 1.0/2920.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0},
136     {-1.0/1189272.0, 128.0/743295.0, 2645.0/1189272.0, -1664.0/49553.0,
        76113.0/396424.0, -39296.0/49553.0, 0.0, 39296.0/49553.0,
        -76113.0/396424.0, 1664.0/49553.0, -2645.0/1189272.0, -128.0/743295.0,
        1.0/1189272.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
137     {-5.0/18545664272, -2048.0/8113728119, -563818.0/10431936153,
        -1386496.0/5795520085, 17297069.0/2318208034, -735232.0/13780629,
        265226398.0/1159104017, -957310976.0/1159104017, 0.0,
        957310976.0/1159104017, -265226398.0/1159104017, 735232.0/13780629,
        -17297069.0/2318208034, 1386496.0/5795520085, 563818.0/10431936153,
        2048.0/8113728119, 5.0/18545664272, 0, 0, 0, 0, 0, 0, 0, 0},
138     {6.9686511520083E-13, -2.8996668057051E-09, 4.2069120451167E-07,
        1.2026575195723E-05, 4.2968915709948E-06, -1.5885615434757E-03,
        1.4545511041994E-02, -7.2440589997659E-02, 0.25855294414146,
        -0.85013666155592, 0.0, 0.85013666155592, -0.25855294414146,
        7.2440589997659E-02, -1.4545511041994E-02, 1.5885615434757E-03,
        -4.2968915709948E-06, -1.2026575195723E-05, -4.2069120451167E-07,
        2.8996668057051E-09, -6.9686511520083E-13, 0, 0, 0, 0},
```

```
139     {-6.6283900594600E-16, -1.2035273999989E-11, 4.1830548203747E-10,
           -2.1871130331900E-07, -1.6501679210868E-06, 4.2363946800701E-06,
           3.3734404776409E-04, -3.8814546576295E-03, 2.2687411014648E-02,
           -9.0189066217795E-02, 0.28296509452594, -0.86874391452377, 0.0,
           0.86874391452377, -0.28296509452594, 9.0189066217795E-02,
           -2.2687411014648E-02, 3.8814546576295E-03, -3.3734404776409E-04,
           -4.2363946800701E-06, 1.6501679210868E-06, 2.1871130331900E-07,
           -4.1830548203747E-10, 1.2035273999989E-11, 6.6283900594600E-16}
140   };
141   memcpy(stencil, stencils[genus], MAX_STENCIL_SIZE * sizeof(double));
142   return stencil;
143 }
144
145 int main(int argc, char** argv) {
146   if (argc < 3) {
147     std::cerr << "Usage: ./main.exe [delta] [genus]" << std::endl;
148     return 1;
149   }
150   // Initialize the grid
151   std::clock_t startTime, endTime;
152   const double delta = std::atof(argv[1]);
153   const double3 lim = {15.f, 15.f, 6.f}; // USER INPUT
154   const dim3 dim = dim3(static_cast<uint>(std::ceil(2 * lim.x / delta) + 1),
155                         static_cast<uint>(std::ceil(2 * lim.y / delta) + 1),
156                         static_cast<uint>(std::ceil(2 * lim.z / delta) + 1));
157   const double minorRadius = 2.f;
158   const double majorRadius = 10.f;
159   const int genus = std::atoi(argv[2]);
160   const int stencilRadius = genus == 1 ? 1 : (genus - 1) * 2;
161   double *stencil = stencilPicker(genus);
162   double *const grid = new double[dim.x * dim.y * dim.z]();
163
164   // Creating the grid
```

```
  for (int s = 0; s < dim.z; s++) {
    for (int r = 0; r < dim.y; r++) {
      for (int c = 0; c < dim.x; c++) {
        int3 pt = make_int3(c, r, s);
        double3 xyz = crs2xyz(pt, delta, lim);
        grid[flatten(pt, dim)] = shapeFunc(xyz, majorRadius, minorRadius);
      }
    }
  }

  startTime = std::clock();
  double moi = momentOfInertia(grid, stencil, stencilRadius, lim, delta, dim);
  endTime = std::clock();
  printf("%f,", moi);
  printf("%f\n",
    (endTime - startTime) / (double)(CLOCKS_PER_SEC / 1000));
  delete[] grid;
  delete[] stencil;
}
```

Listing A.3: Matlab code used in importing Weirong Wang's data.

```
% Brandon Blakeley
% Flamelet Data Import
% Original Simulation data from Weirong Wang
% 11, March 2016

clear all, close all, clc

%% Import Initial Z Data

filepath = './Case_R2_data/Z_FIELD/INITIAL_FIELD/mod_BB/';
F = dir(filepath);
```

```
display(F)

c = 0;
for i = 1 : length([F.isdir])
    if (F(i).isdir == 0)
        filename = F(i).name;

        fullname = [filepath filename];
        disp('Opening file:')
        disp(fullname)

        fileID = fopen(fullname, 'r', 'ieee-be.l64');    % Access binary data
            in big endian format

        tmp = fread(fileID, 1, 'integer*4');    % Reads first byte of data
            from document - this is the size (in bits) of data?
        sz = round(tmp/8);       % Number of double precision values in the
            file

        % Read data into Z
        Z(c*sz + 1 : (c+1)*sz) = fread(fileID, sz, 'double');

        fclose(fileID);
        clear tmp;
        c = c + 1;
    end
end

dim = nthroot(size(Z, 2), 3);
L = 2 * pi;
outfile = fopen('dataset.bin','a');
fwrite(outfile, dim, 'double');
fwrite(outfile, L, 'double');
```

```
fwrite(outfile, Z, 'double');
fclose(outfile);
```

Listing A.4: Component used in reading exported data into expected format.

```
#include "read.h"
#include <cstdio>

void read_file(const std::string& fileName, int *n, double *len, double **grid
    ) {
  FILE *file = std::fopen(fileName.c_str(), "rb");
  double n_val = 0.0;
  std::fread(&n_val, sizeof(double), 1, file);
  *n = n_val;
  std::fread(len, sizeof(double), 1, file);
  *grid = new double[*n * *n * *n];
  std::fread(*grid, sizeof(double), *n * *n * *n, file);
}
```

Listing A.5: Surface integration code used in computing flame surface area.

```
#include "read.h"

#include <cuda_runtime.h>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <algorithm>
#include <iostream>
#include <cstring>

#define EPS 1e-6 // Replacement for std::numeric_limits<double>::epsilon()
```

```
#define MAX_STENCIL_SIZE 25

double id(double val, double thresh) { return val; }

double sgn(double val, double thresh) {
  return (thresh < val) - (val < thresh);
}

int clipClamped(int i, int iMax) {
  return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
}

int clipPeriodically(int i, int iMax) {
  return (i + iMax) % iMax;
}

int flatten(const int3& crs, const dim3& dim) {
  return clipClamped(crs.x, dim.x) +
         clipClamped(crs.y, dim.y) * dim.x +
         clipClamped(crs.z, dim.z) * dim.x * dim.y;
}

double deriv(const double *grid, const double *stencil, int radius,
  const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
  double (*f)(double val, double thresh), double thresh) {
  double result = 0.0;
  for (int i = -radius; i <= radius; i++) {
    int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
                                pt.z + (i * zd)), dim);
    result += f(grid[loc], thresh) * stencil[radius + i];
  }
  return result / delta;
}
```

```
46
47 double xDeriv(const double *grid, const double *stencil, int radius,
48   const int3& pt, double delta, const dim3& dim,
49   double (*f)(double val, double thresh), double thresh) {
50   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
51 }
52
53 double yDeriv(const double *grid, const double *stencil, int radius,
54   const int3& pt, double delta, const dim3& dim,
55   double (*f)(double val, double thresh), double thresh) {
56   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
57 }
58
59 double zDeriv(const double *grid, const double *stencil, int radius,
60   const int3& pt, double delta, const dim3& dim,
61   double (*f)(double val, double thresh), double thresh) {
62   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
63 }
64
65 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
66   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
67                       -lim.z + crs.z * delta);
68 }
69
70 double dotProduct(const double3& v, const double3& w) {
71   return v.x * w.x + v.y * w.y + v.z * w.z;
72 }
73
74 double computeMagnitude(const double3& v) { return std::sqrt(dotProduct(v, v))
     ; }
75
76 double surfArea(const double *grid, const double *stencil, int rad,
77   const double3 &lim, double delta, const dim3 &dim) {
```

```
78   double areaSum = 0.0;
79   for (int s = 0; s < dim.z; s++) {
80     for (int r = 0; r < dim.y; r++) {
81       for (int c = 0; c < dim.x; c++) {
82         int3 pt = make_int3(c, r, s);
83         double3 df = make_double3(
84           xDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
85           yDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0),
86           zDeriv(grid, stencil, rad, pt, delta, dim, id, 0.0));
87         double3 dchi = make_double3(
88           xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
89           yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
90           zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
91         double denom = computeMagnitude(df);
92         if (denom < EPS) continue;
93         areaSum += dotProduct(df, dchi) / denom;
94       }
95     }
96   }
97   return areaSum / 2.0 * std::pow(delta, 3);
98 }
99
100 double* stencilPicker(int genus) {
101   double *stencil = new double[MAX_STENCIL_SIZE]();
102   const double stencils[8][MAX_STENCIL_SIZE] = {
103     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0},
104     {-0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0}, // M = 1
105     {1.0/12.0, -2.0/3.0, 0.0, 2.0/3.0, -1.0/12.0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
106     {-1.0/2920.0, -16.0/1095.0, 53.0/365.0, -272.0/365.0, 0.0, 272.0/365.0,
        -53.0/365.0, 16.0/1095.0, 1.0/2920.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
        0, 0, 0, 0, 0},
107     {-1.0/1189272.0, 128.0/743295.0, 2645.0/1189272.0, -1664.0/49553.0,
        76113.0/396424.0, -39296.0/49553.0, 0.0, 39296.0/49553.0,
        -76113.0/396424.0, 1664.0/49553.0, -2645.0/1189272.0, -128.0/743295.0,
        1.0/1189272.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
108     {-5.0/18545664272, -2048.0/8113728119, -563818.0/10431936153,
        -1386496.0/5795520085, 17297069.0/2318208034, -735232.0/13780629,
        265226398.0/1159104017, -957310976.0/1159104017, 0.0,
        957310976.0/1159104017, -265226398.0/1159104017, 735232.0/13780629,
        -17297069.0/2318208034, 1386496.0/5795520085, 563818.0/10431936153,
        2048.0/8113728119, 5.0/18545664272, 0, 0, 0, 0, 0, 0, 0, 0},
109     {6.9686511520083E-13, -2.8996668057051E-09, 4.2069120451167E-07,
        1.2026575195723E-05, 4.2968915709948E-06, -1.5885615434757E-03,
        1.4545511041994E-02, -7.2440589997659E-02, 0.25855294414146,
        -0.85013666155592, 0.0, 0.85013666155592, -0.25855294414146,
        7.2440589997659E-02, -1.4545511041994E-02, 1.5885615434757E-03,
        -4.2968915709948E-06, -1.2026575195723E-05, -4.2069120451167E-07,
        2.8996668057051E-09, -6.9686511520083E-13, 0, 0, 0, 0},
110     {-6.6283900594600E-16, -1.2035273999989E-11, 4.1830548203747E-10,
        -2.1871130331900E-07, -1.6501679210868E-06, 4.2363946800701E-06,
        3.3734404776409E-04, -3.8814546576295E-03, 2.2687411014648E-02,
        -9.0189066217795E-02, 0.28296509452594, -0.86874391452377, 0.0,
        0.86874391452377, -0.28296509452594, 9.0189066217795E-02,
        -2.2687411014648E-02, 3.8814546576295E-03, -3.3734404776409E-04,
        -4.2363946800701E-06, 1.6501679210868E-06, 2.1871130331900E-07,
        -4.1830548203747E-10, 1.2035273999989E-11, 6.6283900594600E-16}
111   };
112   memcpy(stencil, stencils[genus], MAX_STENCIL_SIZE * sizeof(double));
113   return stencil;
114 }
115
116 int main(int argc, char** argv) {
117   if (argc < 4) {
```

```
118     std::cerr << "not enough arguments call as: area.exe"
119               << " [datafileName] [isovalue] [genus]"
120               << std::endl;
121     return 1;
122   }
123   // Initialize the grid
124   std::clock_t startTime, endTime;
125   const double isovalue = std::atof(argv[2]);
126   const int genus = std::atoi(argv[3]);
127   int n = 0;
128   double len = 0.0;
129   double *grid = 0;
130   read_file(argv[1], &n, &len, &grid);
131   const dim3 dim = dim3(n, n, n);
132   const double delta = len / (n - 1);
133   const double3 lim = {len, len, len};
134   const int stencilRadius = genus == 1 ? 1 : (genus - 1) * 2;
135   double *stencil = stencilPicker(genus);
136   for (int i = 0; i < n * n * n; i++) grid[i] -= isovalue;
137
138   startTime = std::clock();
139   double area = surfArea(grid, stencil, stencilRadius, lim, delta, dim);
140   endTime = std::clock();
141   printf("%f,", area);
142   printf("%f\n",
143     (endTime - startTime) / (double)(CLOCKS_PER_SEC / 1000));
144   delete[] grid;
145   delete[] stencil;
146 }
```

Listing A.6: Surface integration code implemented on the GPU.

```
1 #include "kernel.h"
```

```
#define TX 4
#define TY 4
#define TZ 4

#define EPS 1e-6

int divUp(int a, int b) { return (a + b - 1)/b; }

__host__ __device__
int clip(int i, int iMax) {
  return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
}

__host__ __device__
int flatten(const int3& crs, const dim3& dim) {
  return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
    clip(crs.z, dim.z) * dim.x * dim.y;
}

__host__ __device__
double dotProduct(const double3& v, const double3& w) {
  return v.x * w.x + v.y * w.y + v.z * w.z;
}

__host__ __device__
double computeMagnitude(const double3& v) { return std::sqrt(dotProduct(v, v))
    ; }

__device__
double sgn(double val, double thresh) { return (thresh < val) - (val < thresh)
    ; }

__device__
```

```
33 double deriv(const double *grid, const double *stencil, int radius,
34   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
35   double (*f)(double val, double thresh), double thresh) {
36   double result = 0.0;
37   for (int i = -radius; i <= radius; i++) {
38     int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
39                                 pt.z + (i * zd)), dim);
40     result += f(grid[loc], thresh) * stencil[radius + i];
41   }
42   return result / delta;
43 }
44
45 __device__
46 double xDeriv(const double *grid, const double *stencil, int radius,
47   const int3& pt, double delta, const dim3& dim,
48   double (*f)(double val, double thresh), double thresh) {
49   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
50 }
51
52 __device__
53 double yDeriv(const double *grid, const double *stencil, int radius,
54   const int3& pt, double delta, const dim3& dim,
55   double (*f)(double val, double thresh), double thresh) {
56   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
57 }
58
59 __device__
60 double zDeriv(const double *grid, const double *stencil, int radius,
61   const int3& pt, double delta, const dim3& dim,
62   double (*f)(double val, double thresh), double thresh) {
63   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
64 }
65
```

```
66 __host__ __device__
67 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
68   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
69                       -lim.z + crs.z * delta);
70 }
71
72 __device__
73 double id(double val, double thresh) { return val; }
74
75 __device__
76 int3 flatIdx2crs(int flatIdx, const dim3& dim) {
77   int s = flatIdx / (dim.y * dim.x);
78   int rem3d = flatIdx - s * dim.y * dim.x;
79   int r = rem3d / dim.x;
80   int c = rem3d - r * dim.x;
81   return make_int3(c, r, s);
82 }
83
84 __global__
85 void surfAreaKernel(double *areaSum, const double *grid, const double *stencil
     ,
86   int rad, const double3 *lim, double delta, const dim3 *dim) {
87   const int c = threadIdx.x + blockDim.x * blockIdx.x;
88   const int r = threadIdx.y + blockDim.y * blockIdx.y;
89   const int s = threadIdx.z + blockDim.z * blockIdx.z;
90   if (c >= dim->x || r >= dim->y || s >= dim->z) return;
91   const int3 pt_crs = make_int3(c, r, s);
92   const int i = flatten(pt_crs, *dim);
93   const int s_c = threadIdx.x + rad;
94   const int s_r = threadIdx.y + rad;
95   const int s_s = threadIdx.z + rad;
96   const dim3 sBlockSz = dim3(blockDim.x + 2 * rad, blockDim.y + 2 * rad,
97     blockDim.z + 2 * rad);
```

```
98  const int s_i = flatten(make_int3(s_c, s_r, s_s), sBlockSz);
99  extern __shared__ double s_block[];
100 double *s_stencil = s_block + sBlockSz.x * sBlockSz.y * sBlockSz.z;
101 double *s_blockSum = s_stencil + 2 * rad + 1;
102
103 // set to 0 once per shared block
104 if (s_c == rad && s_r == rad && s_s == rad) {
105   for (int k = 0; k < 2 * rad + 1; k++) {
106     s_stencil[k] = stencil[k];
107   }
108   *s_blockSum = 0.0;
109 }
110
111 // Regular cells
112 s_block[s_i] = grid[i];
113
114 // Halo cells
115 if (threadIdx.x < rad) {
116   s_block[flatten(make_int3(s_c - rad, s_r, s_s), sBlockSz)] =
117     grid[flatten(make_int3(c - rad, r, s), *dim)];
118   s_block[flatten(make_int3(s_c + blockDim.x, s_r, s_s), sBlockSz)] =
119     grid[flatten(make_int3(c + blockDim.x, r, s), *dim)];
120 }
121
122 if (threadIdx.y < rad) {
123   s_block[flatten(make_int3(s_c , s_r - rad, s_s), sBlockSz)] =
124     grid[flatten(make_int3(c, r - rad, s), *dim)];
125   s_block[flatten(make_int3(s_c, s_r + blockDim.y, s_s), sBlockSz)] =
126     grid[flatten(make_int3(c, r + blockDim.y, s), *dim)];
127 }
128
129 if (threadIdx.z < rad) {
130   s_block[flatten(make_int3(s_c , s_r, s_s - rad), sBlockSz)] =
```

```
131       grid[flatten(make_int3(c, r, s - rad), *dim)];
132     s_block[flatten(make_int3(s_c, s_r, s_s + blockDim.z), sBlockSz)] =
133       grid[flatten(make_int3(c, r, s + blockDim.z), *dim)];
134   }
135
136   __syncthreads();
137
138   if ((c >= dim->x - rad) || (r >= dim->y - rad) || (s >= dim->z - rad) ||
139       c <= (rad - 1) || r <= (rad - 1) || s <= (rad - 1)) return;
140
141   const int3 s_pt = make_int3(s_c, s_r, s_s);
142
143   double3 df = make_double3(
144     xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
145     yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0),
146     zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, id, 0.0));
147
148   double3 dchi = make_double3(
149     xDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
150     yDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0),
151     zDeriv(s_block, s_stencil, rad, s_pt, delta, sBlockSz, sgn, 0.0));
152
153   double denom = computeMagnitude(df);
154
155   if (denom > EPS) atomicAdd(s_blockSum, dotProduct(df, dchi) / denom);
156
157   __syncthreads();
158
159   // add only once per shared block
160   if (s_c == rad && s_r == rad && s_s == rad) {
161     atomicAdd(areaSum, *s_blockSum);
162   }
163 }
```

```
164
165 double surfArea(const double *grid, const double *stencil, int rad,
166   const double3 &lim, double delta, const dim3 &dim) {
167   const int problemSize = dim.x * dim.y * dim.z;
168   double areaSum = 0.0;
169   double *d_areaSum = 0, *d_grid = 0, *d_stencil;
170   dim3 *d_dim = 0;
171   double3 *d_lim = 0;
172
173   cudaMalloc(&d_areaSum, sizeof(double));
174   cudaMemset(d_areaSum, 0, sizeof(double));
175
176   cudaMalloc(&d_grid, problemSize * sizeof(double));
177   cudaMemcpy(d_grid, grid, problemSize * sizeof(double),
        cudaMemcpyHostToDevice);
178
179   cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
180   cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
        cudaMemcpyHostToDevice);
181
182   cudaMalloc(&d_dim, sizeof(dim3));
183   cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
184
185   cudaMalloc(&d_lim, sizeof(double3));
186   cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
187
188   const dim3 blockSize(TX, TY, TZ);
189   const dim3 gridSize(divUp(dim.x, TX), divUp(dim.y, TY), divUp(dim.z, TZ));
190   const size_t smSz = ((TX + 2 * rad) * (TY + 2 * rad) * (TZ + 2 * rad) +
191                        2 * rad + 1 + 1) * sizeof(double);
192
193   surfAreaKernel<<<gridSize, blockSize, smSz>>>(d_areaSum, d_grid, d_stencil,
194     rad, d_lim, delta, d_dim);
```

```
195
196   cudaMemcpy(&areaSum, d_areaSum, sizeof(double), cudaMemcpyDeviceToHost);
197   cudaFree(d_areaSum);
198   cudaFree(d_grid);
199   cudaFree(d_stencil);
200   cudaFree(d_dim);
201   cudaFree(d_lim);
202   return std::pow(delta, 3) / 2.0 * areaSum;
203 }
```

Listing A.7: Volume integration applied on a torus together with Monte-Carlo method for comparison purposes

```
1  #include <cuda_runtime.h>
2  #include <cstdio>
3  #include <cstdlib>
4  #include <cmath>
5  #include <algorithm>
6  #include <ctime>
7  #include <random>
8  #include <iostream>
9
10 #define EPS 1e-6 // Replacement for std::numeric_limits<double>::epsilon()
11 #define MAX_STENCIL_SIZE 25
12
13 double torusAtOrigin(double x, double y, double z, double majorRadius,
14   double minorRadius) {
15     double xyResult = majorRadius - std::sqrt(std::pow(x, 2) + std::pow(y, 2))
         ;
16     return std::pow(xyResult, 2) + std::pow(z, 2) - std::pow(minorRadius, 2);
17 }
18
19 double shapeFunc(const double3& pt, double majorRadius, double minorRadius) {
```

```
20   return torusAtOrigin(pt.x, pt.y, pt.z, majorRadius, minorRadius);
21 }
22
23 double sgn(double val, double thresh) {
24   return (thresh < val) - (val < thresh);
25 }
26
27 int clip(int i, int iMax) {
28   return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
29 }
30
31 int flatten(const int3& crs, const dim3& dim) {
32   return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
33     clip(crs.z, dim.z) * dim.x * dim.y;
34 }
35
36 double deriv(const double *grid, const double *stencil, int radius,
37   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
38   double (*f)(double val, double thresh), double thresh) {
39   double result = 0.0;
40   for (int i = -radius; i <= radius; i++) {
41     int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
42                                 pt.z + (i * zd)), dim);
43     result += f(grid[loc], thresh) * stencil[radius + i];
44   }
45   return result / delta;
46 }
47
48 double xDeriv(const double *grid, const double *stencil, int radius,
49   const int3& pt, double delta, const dim3& dim,
50   double (*f)(double val, double thresh), double thresh) {
51   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
52 }
```

```
53
54 double yDeriv(const double *grid, const double *stencil, int radius,
55   const int3& pt, double delta, const dim3& dim,
56   double (*f)(double val, double thresh), double thresh) {
57   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
58 }
59
60 double zDeriv(const double *grid, const double *stencil, int radius,
61   const int3& pt, double delta, const dim3& dim,
62   double (*f)(double val, double thresh), double thresh) {
63   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
64 }
65
66 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
67   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
68                       -lim.z + crs.z * delta);
69 }
70
71 double dotProduct(const double3& v, const double3& w) {
72   return v.x * w.x + v.y * w.y + v.z * w.z;
73 }
74
75 double computeMagnitude(const double3& v) { return std::sqrt(dotProduct(v, v))
     ; }
76
77 double3 normalizedVec(const double3& v) {
78   double mag = computeMagnitude(v);
79   return make_double3(v.x / mag, v.y / mag, v.z / mag);
80 }
81
82 double volumeInt(const double *grid, const double *stencil, int rad,
83   const double3 &lim, double delta, const dim3 &dim) {
84   double volSum = 0.0;
```

```
85    for (int s = 0; s < dim.z; s++) {
86      for (int r = 0; r < dim.y; r++) {
87        for (int c = 0; c < dim.x; c++) {
88          int3 pt = make_int3(c, r, s);
89          double3 dchi = make_double3(
90            xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
91            yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
92            zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
93          double3 xyz = crs2xyz(pt, delta, lim);
94          volSum += dotProduct(dchi, xyz);
95        }
96      }
97    }
98    return volSum / 6.0 * std::pow(delta, 3);
99  }
100
101 double mcVolume(int n, const double3 &lim, double minorRadius,
102   double majorRadius) {
103   double volSum = 0.0;
104   std::random_device rd;
105   std::default_random_engine gen(rd());
106   std::uniform_real_distribution<> disx(-lim.x, lim.x);
107   std::uniform_real_distribution<> disy(-lim.y, lim.y);
108   std::uniform_real_distribution<> disz(-lim.z, lim.z);
109   for (int i = 0; i < n; ++i) {
110     double3 pt = make_double3(disx(gen), disy(gen), disz(gen));
111     if (shapeFunc(pt, majorRadius, minorRadius) < 0.0) {
112       volSum += 1.0;
113     }
114   }
115   return volSum / n * (2*lim.x) * (2*lim.y) * (2*lim.z);
116 }
117
```

```
118 double3 rotatedPointAboutAxis(const double3& pt, const double3& axis,
119   double angleInRadians) {
120   angleInRadians *= -1.0; // Reverse the angle
121   double3 normalizedAxis = normalizedVec(axis);
122   double qi = normalizedAxis.x * std::sin(angleInRadians / 2.0);
123   double qj = normalizedAxis.y * std::sin(angleInRadians / 2.0);
124   double qk = normalizedAxis.z * std::sin(angleInRadians / 2.0);
125   double qr = std::cos(angleInRadians / 2.0);
126   return make_double3(
127     (1 - 2 * qj * qj - 2 * qk * qk) * pt.x + 2 * (qi * qj - qk * qr) * pt.y +
128       2 * (qi * qk + qj * qr) * pt.z,
129     2 * (qi * qj + qk * qr) * pt.x + (1 - 2 * qi * qi - 2 * qk * qk) * pt.y +
130       2 * (qj * qk - qi * qr) * pt.z,
131     2 * (qi * qk - qj * qr) * pt.x + 2 * (qi * qr + qj * qk) * pt.y +
132      (1 - 2 * qi * qi - 2 * qj * qj) * pt.z);
133 }
134
135 double3 translatedPoint(const double3& pt, const double3& delta) {
136   return make_double3(pt.x - delta.x, pt.y - delta.y, pt.z - delta.z);
137 }
138
139 double* stencilPicker(int genus) {
140   double *stencil = new double[MAX_STENCIL_SIZE]();
141   const double stencils[8][MAX_STENCIL_SIZE] = {
142     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0},
143     {-0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0}, // M = 1
144     {1.0/12.0, -2.0/3.0, 0.0, 2.0/3.0, -1.0/12.0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
145     {-1.0/2920.0, -16.0/1095.0, 53.0/365.0, -272.0/365.0, 0.0, 272.0/365.0,
        -53.0/365.0, 16.0/1095.0, 1.0/2920.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0},
```

```
146     {-1.0/1189272.0, 128.0/743295.0, 2645.0/1189272.0, -1664.0/49553.0,
           76113.0/396424.0, -39296.0/49553.0, 0.0, 39296.0/49553.0,
           -76113.0/396424.0, 1664.0/49553.0, -2645.0/1189272.0, -128.0/743295.0,
           1.0/1189272.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
147     {-5.0/18545664272, -2048.0/8113728119, -563818.0/10431936153,
           -1386496.0/5795520085, 17297069.0/2318208034, -735232.0/13780629,
           265226398.0/1159104017, -957310976.0/1159104017, 0.0,
           957310976.0/1159104017, -265226398.0/1159104017, 735232.0/13780629,
           -17297069.0/2318208034, 1386496.0/5795520085, 563818.0/10431936153,
           2048.0/8113728119, 5.0/18545664272, 0, 0, 0, 0, 0, 0, 0, 0},
148     {6.9686511520083E-13, -2.8996668057051E-09, 4.2069120451167E-07,
           1.2026575195723E-05, 4.2968915709948E-06, -1.5885615434757E-03,
           1.4545511041994E-02, -7.2440589997659E-02, 0.25855294414146,
           -0.85013666155592, 0.0, 0.85013666155592, -0.25855294414146,
           7.2440589997659E-02, -1.4545511041994E-02, 1.5885615434757E-03,
           -4.2968915709948E-06, -1.2026575195723E-05, -4.2069120451167E-07,
           2.8996668057051E-09, -6.9686511520083E-13, 0, 0, 0, 0},
149     {-6.6283900594600E-16, -1.2035273999989E-11, 4.1830548203747E-10,
           -2.1871130331900E-07, -1.6501679210868E-06, 4.2363946800701E-06,
           3.3734404776409E-04, -3.8814546576295E-03, 2.2687411014648E-02,
           -9.0189066217795E-02, 0.28296509452594, -0.86874391452377, 0.0,
           0.86874391452377, -0.28296509452594, 9.0189066217795E-02,
           -2.2687411014648E-02, 3.8814546576295E-03, -3.3734404776409E-04,
           -4.2363946800701E-06, 1.6501679210868E-06, 2.1871130331900E-07,
           -4.1830548203747E-10, 1.2035273999989E-11, 6.6283900594600E-16}
150   };
151   memcpy(stencil, stencils[genus], MAX_STENCIL_SIZE * sizeof(double));
152   return stencil;
153 }
154
155 int main(int argc, char** argv) {
156   if (argc < 3) {
157     std::cerr << "Usage: ./main.exe [delta] [genus]" << std::endl;
```

```
158     return 1;
159   }
160   // Initialize the grid
161   std::clock_t startTime, endTime;
162   std::random_device rd;
163   std::default_random_engine gen(rd());
164   const double delta = std::atof(argv[1]);
165   const double3 lim = {15.f, 15.f, 15.f}; // USER INPUT
166   // Monte Carlo number of points
167   const int nx = std::ceil(2 * lim.x / delta);
168   const int ny = std::ceil(2 * lim.y / delta);
169   const int nz = std::ceil(2 * lim.z / delta);
170   const int n = nx * ny * nz;
171   const dim3 dim = dim3(static_cast<uint>(std::ceil(2 * lim.x / delta) + 1),
172                         static_cast<uint>(std::ceil(2 * lim.y / delta) + 1),
173                         static_cast<uint>(std::ceil(2 * lim.z / delta) + 1));
174   const double minorRadius = 2.f;
175   const double majorRadius = 10.f;
176   const int genus = std::atoi(argv[2]);
177   const int stencilRadius = genus == 1 ? 1 : (genus - 1) * 2;
178   double *stencil = stencilPicker(genus);
179   double *const grid = new double[dim.x * dim.y * dim.z]();
180
181   std::uniform_real_distribution<> delta_dis(-delta, delta);
182   std::uniform_real_distribution<> angle_dis(-M_PI, M_PI);
183   std::uniform_real_distribution<> axis_dis(0.00001, 1);
184   double3 rotationAxis = make_double3(axis_dis(gen), axis_dis(gen), axis_dis(
        gen));
185   double rotationAngle = angle_dis(gen);
186   double3 translationDelta = make_double3(delta_dis(gen), delta_dis(gen),
        delta_dis(gen));
187
188   // Creating the grid
```

```
189   for (int s = 0; s < dim.z; s++) {
190     for (int r = 0; r < dim.y; r++) {
191       for (int c = 0; c < dim.x; c++) {
192         int3 pt = make_int3(c, r, s);
193         double3 xyz = crs2xyz(pt, delta, lim);
194         xyz = rotatedPointAboutAxis(xyz, rotationAxis, rotationAngle);
195         xyz = translatedPoint(xyz, translationDelta);
196         grid[flatten(pt, dim)] = shapeFunc(xyz, majorRadius, minorRadius);
197       }
198     }
199   }
200
201   startTime = std::clock();
202   double volume = volumeInt(grid, stencil, stencilRadius, lim, delta, dim);
203   endTime = std::clock();
204   printf("%f,", volume);
205   printf("%f\n", mcVolume(n, lim, minorRadius, majorRadius));
206   delete[] grid;
207   delete[] stencil;
208 }
```

Listing A.8: Moment of inertia computation on a solid torus using volume integration technique.

```
1 #include <cuda_runtime.h>
2 #include <cstdio>
3 #include <cstdlib>
4 #include <cmath>
5 #include <ctime>
6 #include <algorithm>
7 #include <iostream>
8
9 #define EPS 1e-6 // Replacement for std::numeric_limits<double>::epsilon()
```

```
10 #define MAX_STENCIL_SIZE 25
11
12 double torusAtOrigin(double x, double y, double z, double majorRadius,
13   double minorRadius) {
14     double xyResult = majorRadius - std::sqrt(std::pow(x, 2) + std::pow(y, 2))
         ;
15     return std::pow(xyResult, 2) + std::pow(z, 2) - std::pow(minorRadius, 2);
16 }
17
18 double shapeFunc(const double3& pt, double majorRadius, double minorRadius) {
19   return torusAtOrigin(pt.x, pt.y, pt.z, majorRadius, minorRadius);
20 }
21
22 double id(double val, double thresh) { return val; }
23
24 double sgn(double val, double thresh) {
25   return (thresh < val) - (val < thresh);
26 }
27
28 int clip(int i, int iMax) {
29   return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
30 }
31
32 int flatten(const int3& crs, const dim3& dim) {
33   return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
34     clip(crs.z, dim.z) * dim.x * dim.y;
35 }
36
37 double deriv(const double *grid, const double *stencil, int radius,
38   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
39   double (*f)(double val, double thresh), double thresh) {
40   double result = 0.0;
41   for (int i = -radius; i <= radius; i++) {
```

```
42      int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
43                                  pt.z + (i * zd)), dim);
44      result += f(grid[loc], thresh) * stencil[radius + i];
45    }
46    return result / delta;
47  }
48
49  double xDeriv(const double *grid, const double *stencil, int radius,
50    const int3& pt, double delta, const dim3& dim,
51    double (*f)(double val, double thresh), double thresh) {
52    return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
53  }
54
55  double yDeriv(const double *grid, const double *stencil, int radius,
56    const int3& pt, double delta, const dim3& dim,
57    double (*f)(double val, double thresh), double thresh) {
58    return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
59  }
60
61  double zDeriv(const double *grid, const double *stencil, int radius,
62    const int3& pt, double delta, const dim3& dim,
63    double (*f)(double val, double thresh), double thresh) {
64    return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
65  }
66
67  double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
68    return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
69                        -lim.z + crs.z * delta);
70  }
71
72  double dotProduct(const double3& v, const double3& w) {
73    return v.x * w.x + v.y * w.y + v.z * w.z;
74  }
```

```
75
76 double momentOfInertia(const double *grid, const double *stencil, int rad,
77   const double3 &lim, double delta, const dim3 &dim) {
78   double inertiaSum = 0.0;
79   for (int s = 0; s < dim.z; s++) {
80     for (int r = 0; r < dim.y; r++) {
81       for (int c = 0; c < dim.x; c++) {
82         int3 pt = make_int3(c, r, s);
83         double3 dchi = make_double3(
84           xDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
85           yDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0),
86           zDeriv(grid, stencil, rad, pt, delta, dim, sgn, 0.0));
87         double3 xyz = crs2xyz(pt, delta, lim);
88         // g = x^2 + y^2 => u = (x^3 i + y^3 j) / 3
89         xyz.x = xyz.x * xyz.x * xyz.x;
90         xyz.y = xyz.y * xyz.y * xyz.y;
91         xyz.z = 0;
92         inertiaSum += dotProduct(dchi, xyz);
93       }
94     }
95   }
96   return inertiaSum / 6.0 * std::pow(delta, 3);
97 }
98
99 double* stencilPicker(int genus) {
100   double *stencil = new double[MAX_STENCIL_SIZE]();
101   const double stencils[8][MAX_STENCIL_SIZE] = {
102     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0},
103     {-0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0}, // M = 1
104     {1.0/12.0, -2.0/3.0, 0.0, 2.0/3.0, -1.0/12.0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
105     {-1.0/2920.0, -16.0/1095.0, 53.0/365.0, -272.0/365.0, 0.0, 272.0/365.0,
            -53.0/365.0, 16.0/1095.0, 1.0/2920.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0},
106     {-1.0/1189272.0, 128.0/743295.0, 2645.0/1189272.0, -1664.0/49553.0,
            76113.0/396424.0, -39296.0/49553.0, 0.0, 39296.0/49553.0,
            -76113.0/396424.0, 1664.0/49553.0, -2645.0/1189272.0, -128.0/743295.0,
            1.0/1189272.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
107     {-5.0/18545664272, -2048.0/8113728119, -563818.0/10431936153,
            -1386496.0/5795520085, 17297069.0/2318208034, -735232.0/13780629,
            265226398.0/1159104017, -957310976.0/1159104017, 0.0,
            957310976.0/1159104017, -265226398.0/1159104017, 735232.0/13780629,
            -17297069.0/2318208034, 1386496.0/5795520085, 563818.0/10431936153,
            2048.0/8113728119, 5.0/18545664272, 0, 0, 0, 0, 0, 0, 0, 0},
108     {6.9686511520083E-13, -2.8996668057051E-09, 4.2069120451167E-07,
            1.2026575195723E-05, 4.2968915709948E-06, -1.5885615434757E-03,
            1.4545511041994E-02, -7.2440589997659E-02, 0.25855294414146,
            -0.85013666155592, 0.0, 0.85013666155592, -0.25855294414146,
            7.2440589997659E-02, -1.4545511041994E-02, 1.5885615434757E-03,
            -4.2968915709948E-06, -1.2026575195723E-05, -4.2069120451167E-07,
            2.8996668057051E-09, -6.9686511520083E-13, 0, 0, 0, 0},
109     {-6.6283900594600E-16, -1.2035273999989E-11, 4.1830548203747E-10,
            -2.1871130331900E-07, -1.6501679210868E-06, 4.2363946800701E-06,
            3.3734404776409E-04, -3.8814546576295E-03, 2.2687411014648E-02,
            -9.0189066217795E-02, 0.28296509452594, -0.86874391452377, 0.0,
            0.86874391452377, -0.28296509452594, 9.0189066217795E-02,
            -2.2687411014648E-02, 3.8814546576295E-03, -3.3734404776409E-04,
            -4.2363946800701E-06, 1.6501679210868E-06, 2.1871130331900E-07,
            -4.1830548203747E-10, 1.2035273999989E-11, 6.6283900594600E-16}
110   };
111   memcpy(stencil, stencils[genus], MAX_STENCIL_SIZE * sizeof(double));
112   return stencil;
113 }
114
```

```
115 int main(int argc, char** argv) {
116   if (argc < 3) {
117     std::cerr << "Usage: ./main.exe [delta] [genus]" << std::endl;
118     return 1;
119   }
120   // Initialize the grid
121   std::clock_t startTime, endTime;
122   const double delta = std::atof(argv[1]);
123   const double3 lim = {15.0, 15.0, 15.0}; // USER INPUT
124   const dim3 dim = dim3(static_cast<uint>(std::ceil(2 * lim.x / delta) + 1),
125                         static_cast<uint>(std::ceil(2 * lim.y / delta) + 1),
126                         static_cast<uint>(std::ceil(2 * lim.z / delta) + 1));
127   const double minorRadius = 2.0;
128   const double majorRadius = 10.0;
129   const int genus = std::atoi(argv[2]);
130   const int stencilRadius = genus == 1 ? 1 : (genus - 1) * 2;
131   double *stencil = stencilPicker(genus);
132   double *const grid = new double[dim.x * dim.y * dim.z]();
133
134   // Creating the grid
135   for (int s = 0; s < dim.z; s++) {
136     for (int r = 0; r < dim.y; r++) {
137       for (int c = 0; c < dim.x; c++) {
138         int3 pt = make_int3(c, r, s);
139         double3 xyz = crs2xyz(pt, delta, lim);
140         grid[flatten(pt, dim)] = shapeFunc(xyz, majorRadius, minorRadius);
141       }
142     }
143   }
144
145   startTime = std::clock();
146   double inertia = momentOfInertia(grid, stencil, stencilRadius, lim, delta,
       dim);
```

```
147   endTime = std::clock();
148   printf("%f,", inertia);
149   printf("%f\n",
150     (endTime - startTime) / (double)(CLOCKS_PER_SEC / 1000));
151   delete[] grid;
152   delete[] stencil;
153 }
```

Listing A.9: Implementation of Boundary Integral

```
1  #include <cuda_runtime.h>
2  #include <Eigen/Dense>
3  #include <cmath>
4  #include <fstream>
5  #include <iostream>
6  #include <vector>
7  #include <ctime>
8  #include <stdexcept>
9
10 #define EPS 1e-6
11
12 double id(double val, double thresh) { return val; }
13
14 double sgn(double val, double thresh) {
15   return (thresh < val) - (val < thresh);
16 }
17
18 int clip(int i, int iMax) {
19   return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
20 }
21
22 int flatten(const int3& crs, const dim3& dim) {
23   return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
```

```
24      clip(crs.z, dim.z) * dim.x * dim.y;
25 }
26
27 double dotProduct(const double3& v, const double3& w) {
28   return v.x * w.x + v.y * w.y + v.z * w.z;
29 }
30
31 double computeMagnitude(const double3& v) {
32   return std::sqrt(dotProduct(v, v));
33 }
34
35 double deriv(const double *grid, const double *stencil, int radius,
36   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
37   double (*f)(double val, double thresh), double thresh) {
38   double result = 0.0;
39   for (int i = -radius; i <= radius; i++) {
40     int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
41                                 pt.z + (i * zd)), dim);
42     result += f(grid[loc], thresh) * stencil[radius + i];
43   }
44   return result / delta;
45 }
46
47 double xDeriv(const double *grid, const double *stencil, int radius,
48   const int3& pt, double delta, const dim3& dim,
49   double (*f)(double val, double thresh), double thresh) {
50   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
51 }
52
53 double yDeriv(const double *grid, const double *stencil, int radius,
54   const int3& pt, double delta, const dim3& dim,
55   double (*f)(double val, double thresh), double thresh) {
56   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
```

```
57 }
58
59 double zDeriv(const double *grid, const double *stencil, int radius,
60   const int3& pt, double delta, const dim3& dim,
61   double (*f)(double val, double thresh), double thresh) {
62   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
63 }
64
65 int3 flatIdx2crs(int flatIdx, const dim3& dim) {
66   int s = flatIdx / (dim.y * dim.x);
67   int rem3d = flatIdx - s * dim.y * dim.x;
68   int r = rem3d / dim.x;
69   int c = rem3d - r * dim.x;
70   return make_int3(c, r, s);
71 }
72
73 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
74   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
75                       -lim.z + crs.z * delta);
76 }
77
78 double3 computeGradPhi(const int3& ptc_crs, const int3& ptr_crs, double delta,
79   const double3& lim, bool isProblem3D) {
80   double3 ptc = crs2xyz(ptc_crs, delta, lim);
81   double3 ptr = crs2xyz(ptr_crs, delta, lim);
82   double3 diff = {ptc.x - ptr.x, ptc.y - ptr.y, ptc.z - ptr.z};
83   double magDiff = computeMagnitude(diff);
84   if (magDiff < EPS) {
85     throw std::runtime_error("Zero magnitude exception.");
86   }
87   double coeff = isProblem3D ? -1 / ((4 * M_PI) * std::pow(magDiff, 3))
88                              : -1 / ((2 * M_PI) * std::pow(magDiff, 2));
89   return make_double3(coeff * diff.x, coeff * diff.y, coeff * diff.z);
```

```
90 }
91
92 double3 computeNormalizedGradF(const double *grid, const double *stencil,
93   int rad, const int3& pt_crs, double delta, const dim3& dim) {
94   double3 gradF = {xDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0),
95                    yDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0),
96                    zDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0)};
97   double magGradF = computeMagnitude(gradF);
98   if (magGradF == 0.0) {
99     throw std::runtime_error("Zero magnitude exception.");
100  }
101  return make_double3(gradF.x / magGradF, gradF.y / magGradF,
102                      gradF.z / magGradF);
103 }
104
105 double3 computeGradChi(const double *grid, const double *stencil, int rad,
106   const int3& pt, double delta, const dim3& dim, double thresh) {
107   return make_double3(
108     -xDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0,
109     -yDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0,
110     -zDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0);
111 }
112
113 double computeElementOfA(const int3& ptc_crs, const int3& ptr_crs,
114   const double *grid, const double *stencil, int rad, const double3& lim,
115   double delta, const dim3& dim, bool withThreshold=true) {
116   bool problemIs3D = dim.z > 1;
117   double thresh = withThreshold ? grid[flatten(ptc_crs, dim)] : 0.0;
118
119   double3 gradPhi, normalizedGradF, gradChi;
120   try {
121     gradPhi = computeGradPhi(ptr_crs, ptc_crs, delta, lim, problemIs3D);
122   } catch(std::runtime_error e) { return 0.0; }
```

```
123
124   try {
125     normalizedGradF = computeNormalizedGradF(grid, stencil, rad, ptr_crs,
126       delta, dim);
127   } catch(std::runtime_error e) { return 0.0; }
128
129   gradChi = computeGradChi(grid, stencil, rad, ptr_crs, delta, dim, thresh);
130
131   double spacingCoeff = delta * delta * (problemIs3D ? delta : 1);
132   return spacingCoeff * dotProduct(gradPhi, normalizedGradF) *
133     dotProduct(gradChi, normalizedGradF);
134 }
135
136 double computeElementOfAWithoutThreshold(const int3& ptc_crs,
137   const int3& ptr_crs, const double *grid, const double *stencil, int rad,
138   const double3& lim, double delta, const dim3& dim) {
139   return computeElementOfA(ptc_crs, ptr_crs, grid, stencil, rad, lim, delta,
140     dim, false);
141 }
142
143 bool isIrregular(const int3& pt, const double *grid, const double *stencil,
144   int rad, double delta, const dim3& dim, double eps) {
145   double3 gradChi = computeGradChi(grid, stencil, rad, pt, delta, dim, 0.0);
146   return computeMagnitude(gradChi) > eps;
147 }
148
149 double shapeFunc(const double3& pt) {
150   double radius = 1.0;
151   // Circle with radius 1.
152   return pt.x * pt.x + pt.y * pt.y - radius;
153 }
154
155 double boundaryFunc(const double3& pt) {
```

```
156   return pt.x;
157 }
158
159 void serialSolveSystem(double *u_arr, const double *grid, const double *
      stencil,
160   int rad, const double3& lim, double delta, const dim3& dim) {
161   // Finding irregular points
162   std::vector<int3> irregular_pts;
163
164   for (int s = 0; s < dim.z; s++) {
165     for (int r = 0; r < dim.y; r++) {
166       for (int c = 0; c < dim.x; c++) {
167         int3 pt = {c, r, s};
168         if (isIrregular(pt, grid, stencil, rad, delta, dim, EPS)) {
169           irregular_pts.push_back(pt);
170         }
171       }
172     }
173   }
174
175   const int N = irregular_pts.size();
176
177   // Computing the A Matrix
178   Eigen::MatrixXf F(N, N);
179
180   for (int r = 0; r < N; r++) {
181     for (int c = 0; c < N; c++) {
182       F(r, c) = computeElementOfA(irregular_pts[c], irregular_pts[r], grid,
183         stencil, rad, lim, delta, dim);
184     }
185   }
186
187   // Adding the diagonal elements to obtain the final form of A
```

```
188   Eigen::MatrixXf A = F + Eigen::MatrixXf::Identity(N, N) / 2.0;
189
190   // Creating the RHS
191   Eigen::VectorXf g(N);
192   for (int r = 0; r < N; r++) {
193     g(r) = boundaryFunc(crs2xyz(irregular_pts[r], delta, lim));
194   }
195
196   Eigen::VectorXf beta = A.fullPivLu().solve(g);
197
198   // Finding u for all points
199   for (int r = 0; r < dim.x * dim.y * dim.z; r++) {
200     for (int c = 0; c < N; c++) {
201       u_arr[r] += computeElementOfAWithoutThreshold(flatIdx2crs(r, dim),
202        irregular_pts[c], grid, stencil, rad, lim, delta, dim) * beta[c];
203     }
204   }
205 }
206
207 int main(int argc, char** argv) {
208   if (argc < 2) {
209     std::cerr << "Usage: ./main.exe delta [outputFileName]" << std::endl;
210     return 1;
211   }
212
213   std::clock_t startTime, currTime;
214
215   const std::string outputFileName = (argc >= 3) ? argv[2] : "result.csv";
216   const double delta = std::atof(argv[1]);
217   const double3 lim = {2.f, 2.f, 0.f}; // USER INPUT
218   const dim3 dim = dim3(static_cast<uint>(std::ceil(2 * lim.x / delta) + 1),
219                         static_cast<uint>(std::ceil(2 * lim.y / delta) + 1),
220                         static_cast<uint>(std::ceil(2 * lim.z / delta) + 1));
```

```
221  double *const grid = new double[dim.x * dim.y * dim.z]();
222  const double stencil[] = {-0.5, 0.0, 0.5};
223  const int rad = 1;
224
225  // Creating the grid
226  for (int s = 0; s < dim.z; s++) {
227    for (int r = 0; r < dim.y; r++) {
228      for (int c = 0; c < dim.x; c++) {
229        grid[flatten(make_int3(c, r, s), dim)] =
230          shapeFunc(crs2xyz(make_int3(c, r, s), delta, lim));
231      }
232    }
233  }
234
235  // Finding u for all points
236  double *const u = new double[dim.x * dim.y * dim.z]();
237
238  startTime = std::clock();
239  serialSolveSystem(u, grid, stencil, rad, lim, delta, dim);
240  currTime = std::clock();
241
242  double computationTime = (currTime - startTime) / (CLOCKS_PER_SEC / 1000);
243
244  std::cout << "Time elapsed: " << computationTime << " ms" << std::endl;
245
246  // Write the results into a csv file
247  std::ofstream outFile(outputFileName.c_str());
248
249  for (int i = 0; i < dim.x * dim.y * dim.z; i++) {
250    double3 xyzLoc = crs2xyz(flatIdx2crs(i, dim), delta, lim);
251    outFile << xyzLoc.x << "," << xyzLoc.y << "," << xyzLoc.z << "," << u[i]
         << std::endl;
252  }
```

```
253
254   outFile.close();
255   delete[] grid;
256   delete[] u;
257 }
```

Listing A.10: Implementation of the boundary integral kernel file

```
1  #include "kernel.h"
2  #include <stdexcept>
3  #include <limits>
4  #include <cusolverDn.h>
5  #include <cstdio>
6
7  #include "helper_cuda.h"
8
9  #define TX 8
10 #define TY 8
11 #define TZ 8
12 #define TX1D 128
13
14 #define EPS 1e-6
15
16 __host__ __device__
17 int divUp(int a, int b) { return (a + b - 1)/b; }
18
19 __host__ __device__
20 int clip(int i, int iMax) {
21   return i > (iMax - 1) ? (iMax - 1) : (i < 0 ? 0 : i);
22 }
23
24 __host__ __device__
25 int flatten(const int3& crs, const dim3& dim) {
```

```
26   return clip(crs.x, dim.x) + clip(crs.y, dim.y) * dim.x +
27     clip(crs.z, dim.z) * dim.x * dim.y;
28 }
29
30 __host__ __device__
31 double dotProduct(const double3& v, const double3& w) {
32   return v.x * w.x + v.y * w.y + v.z * w.z;
33 }
34
35 __host__ __device__
36 double computeMagnitude(const double3& v) { return std::sqrt(dotProduct(v, v))
     ; }
37
38 __host__ __device__
39 double sgn(double val, double thresh) { return (thresh < val) - (val < thresh)
     ; }
40
41 __host__ __device__
42 double deriv(const double *grid, const double *stencil, int radius,
43   const int3& pt, double delta, const dim3& dim, int xd, int yd, int zd,
44   double (*f)(double val, double thresh), double thresh) {
45   double result = 0.0;
46   for (int i = -radius; i <= radius; i++) {
47     int loc = flatten(make_int3(pt.x + (i * xd), pt.y + (i * yd),
48                                 pt.z + (i * zd)), dim);
49     result += f(grid[loc], thresh) * stencil[radius + i];
50   }
51   return result / delta;
52 }
53
54 __host__ __device__
55 double xDeriv(const double *grid, const double *stencil, int radius,
56   const int3& pt, double delta, const dim3& dim,
```

```
57   double (*f)(double val, double thresh), double thresh) {
58   return deriv(grid, stencil, radius, pt, delta, dim, 1, 0, 0, f, thresh);
59 }
60
61 __host__ __device__
62 double yDeriv(const double *grid, const double *stencil, int radius,
63   const int3& pt, double delta, const dim3& dim,
64   double (*f)(double val, double thresh), double thresh) {
65   return deriv(grid, stencil, radius, pt, delta, dim, 0, 1, 0, f, thresh);
66 }
67
68 __host__ __device__
69 double zDeriv(const double *grid, const double *stencil, int radius,
70   const int3& pt, double delta, const dim3& dim,
71   double (*f)(double val, double thresh), double thresh) {
72   return deriv(grid, stencil, radius, pt, delta, dim, 0, 0, 1, f, thresh);
73 }
74
75 __host__ __device__
76 double3 computeGradChi(const double *grid, const double *stencil, int rad,
77   const int3& pt, double delta, const dim3& dim, double thresh) {
78   return make_double3(
79     -xDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0,
80     -yDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0,
81     -zDeriv(grid, stencil, rad, pt, delta, dim, sgn, thresh) / 2.0);
82 }
83
84 __host__ __device__
85 bool isIrregular(const int3& pt, const double *grid, const double *stencil,
86   int rad, double delta, const dim3& dim, double eps) {
87   double3 gradChi = computeGradChi(grid, stencil, rad, pt, delta, dim, 0.0);
88   return computeMagnitude(gradChi) > eps;
89 }
```

```
90
91 __host__ __device__
92 double3 crs2xyz(const int3& crs, double delta, const double3& lim) {
93   return make_double3(-lim.x + crs.x * delta, -lim.y + crs.y * delta,
94                       -lim.z + crs.z * delta);
95 }
96
97 __host__ __device__
98 double boundaryFunc(const double3& pt) {
99   return pt.x;
100 }
101
102 __host__
103 double shapeFunc(const double3& pt) {
104   double radius = 1.0;
105   // Circle with radius 1.
106   return pt.x * pt.x + pt.y * pt.y - radius;
107 }
108
109 __host__ __device__
110 double computeElementOfA(const int3& ptc_crs, const int3& ptr_crs,
111   const double *grid, const double *stencil, int rad, const double3& lim,
112   double delta, const dim3& dim, bool withThreshold) {
113   bool problemIs3D = dim.z > 1;
114   double thresh = withThreshold ? grid[flatten(ptc_crs, dim)] : 0.0;
115
116   double3 gradPhi, normalizedGradF, gradChi;
117   int err_status = computeGradPhi(&gradPhi, ptr_crs, ptc_crs, delta, lim,
118     problemIs3D);
119   if (err_status != 0) return 0.0;
120
121   err_status = computeNormalizedGradF(&normalizedGradF, grid, stencil, rad,
122     ptr_crs, delta, dim);
```

```
123   if (err_status != 0) return 0.0;
124
125   gradChi = computeGradChi(grid, stencil, rad, ptr_crs, delta, dim, thresh);
126
127   double spacingCoeff = delta * delta * (problemIs3D ? delta : 1);
128   return spacingCoeff * dotProduct(gradPhi, normalizedGradF) *
129     dotProduct(gradChi, normalizedGradF);
130 }
131
132 __host__ __device__
133 double id(double val, double thresh) { return val; }
134
135 __host__ __device__
136 int3 flatIdx2crs(int flatIdx, const dim3& dim) {
137   int s = flatIdx / (dim.y * dim.x);
138   int rem3d = flatIdx - s * dim.y * dim.x;
139   int r = rem3d / dim.x;
140   int c = rem3d - r * dim.x;
141   return make_int3(c, r, s);
142 }
143
144 __host__ __device__
145 int computeGradPhi(double3 *gradPhi, const int3 &ptc_crs, const int3 &ptr_crs,
146   double delta, const double3 &lim, bool isProblem3D) {
147   double3 ptc = crs2xyz(ptc_crs, delta, lim);
148   double3 ptr = crs2xyz(ptr_crs, delta, lim);
149   double3 diff = {ptc.x - ptr.x, ptc.y - ptr.y, ptc.z - ptr.z};
150   double magDiff = computeMagnitude(diff);
151   if (magDiff < EPS) return 1;
152   double coeff = isProblem3D ? -1 / ((4 * M_PI) * std::pow(magDiff, 3))
153                              : -1 / ((2 * M_PI) * std::pow(magDiff, 2));
154   gradPhi->x = coeff * diff.x;
155   gradPhi->y = coeff * diff.y;
```

```
156   gradPhi->z = coeff * diff.z;
157   return 0;
158 }
159
160 __host__ __device__
161 int computeNormalizedGradF(double3 *normalizedGradF, const double *grid,
162   const double *stencil, int rad, const int3 &pt_crs, double delta,
163   const dim3 &dim) {
164   double3 gradF = {xDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0),
165                    yDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0),
166                    zDeriv(grid, stencil, rad, pt_crs, delta, dim, id, 0.0)};
167   double magGradF = computeMagnitude(gradF);
168   if (magGradF == 0.0) return 1;
169   normalizedGradF->x = gradF.x / magGradF;
170   normalizedGradF->y = gradF.y / magGradF;
171   normalizedGradF->z = gradF.z / magGradF;
172   return 0;
173 }
174
175 __host__ __device__
176 double computeElementOfAWithoutThreshold(const int3& ptc_crs,
177   const int3& ptr_crs, const double *grid, const double *stencil, int rad,
178   const double3& lim, double delta, const dim3& dim) {
179   return computeElementOfA(ptc_crs, ptr_crs, grid, stencil, rad, lim, delta,
180     dim, false);
181 }
182
183 __global__
184 void filterKernel(int3 *irreg_arr, int *irreg_count, const double *grid,
185   const double *stencil, int rad, double delta, const dim3 *dim) {
186   const int c = threadIdx.x + blockDim.x * blockIdx.x;
187   const int r = threadIdx.y + blockDim.y * blockIdx.y;
188   const int s = threadIdx.z + blockDim.z * blockIdx.z;
```

```
189   if (c >= dim->x || r >= dim->y || s >= dim->z) return;
190   int3 pt = {c, r, s};
191   if (isIrregular(pt, grid, stencil, rad, delta, *dim, EPS)) {
192     irreg_arr[atomicAdd(irreg_count, 1)] = make_int3(c, r, s);
193   }
194 }
195
196 __global__
197 void boundaryKernel(double *g_arr, int N, const int3* irregular_pts,
198   double delta, const double3 *lim) {
199   const int i = threadIdx.x + blockDim.x * blockIdx.x;
200   if (i >= N) return;
201   g_arr[i] = boundaryFunc(crs2xyz(irregular_pts[i], delta, *lim));
202 }
203
204 __global__
205 void computeFMatrixKernel(double *F_arr, int width, int height,
206   const int3 *irregular_pts, const double *grid, const double *stencil,
207   int rad, const double3 *lim, double delta, const dim3 *dim) {
208   // Column-major ordering
209   const int c = threadIdx.x + blockDim.x * blockIdx.x;
210   const int r = threadIdx.y + blockDim.y * blockIdx.y;
211   if (c >= width || r >= height) return;
212
213   if (r == c) {
214     F_arr[c * height + r] = 1.0 / 2.0;
215     return;
216   }
217
218   F_arr[c * height + r] = computeElementOfA(irregular_pts[c],
219     irregular_pts[r], grid, stencil, rad, *lim, delta, *dim);
220 }
221
```

```
__global__
void solveForTheFullAKernel(double *u, const double *beta,
  const int3 *irregular_pts, const double *grid, const double *stencil,
  int rad, const double3 *lim, double delta, const dim3 *dim,
  int irregular_pt_count) {

  const int r = threadIdx.x + blockDim.x * blockIdx.x;
  if (r >= dim->x * dim->y * dim->z) return;

  for (int c = 0; c < irregular_pt_count; c++) {
    u[r] += computeElementOfAWithoutThreshold(flatIdx2crs(r, *dim),
      irregular_pts[c], grid, stencil, rad, *lim, delta, *dim) * beta[c];
  }
}

void parallelSolveSystem(double *u, const double *grid, const double *stencil,
  int rad, const double3& lim, double delta, const dim3& dim) {
  const int problemSize = dim.x * dim.y * dim.z;
  double *d_grid = 0;
  double *d_stencil = 0;
  int3 *d_irreg_pts = 0;
  int *d_irreg_count = 0;
  dim3 *d_dim = 0;
  double3 *d_lim = 0;

  cudaMalloc(&d_grid, problemSize * sizeof(double));
  cudaMemcpy(d_grid, grid, problemSize * sizeof(double),
    cudaMemcpyHostToDevice);

  cudaMalloc(&d_stencil, (2 * rad + 1) * sizeof(double));
  cudaMemcpy(d_stencil, stencil, (2 * rad + 1) * sizeof(double),
    cudaMemcpyHostToDevice);

```

```
255  cudaMalloc(&d_irreg_pts, problemSize * sizeof(int3));
256
257  cudaMalloc(&d_irreg_count, sizeof(int));
258  cudaMemset(d_irreg_count, 0, sizeof(int));
259
260  cudaMalloc(&d_dim, sizeof(dim3));
261  cudaMemcpy(d_dim, &dim, sizeof(dim3), cudaMemcpyHostToDevice);
262
263  cudaMalloc(&d_lim, sizeof(double3));
264  cudaMemcpy(d_lim, &lim, sizeof(double3), cudaMemcpyHostToDevice);
265
266  const dim3 filterKernelBlockSize(TX, TY, TZ);
267  const dim3 filterKernelGridSize(divUp(dim.x, TX), divUp(dim.y, TY),
268                                  divUp(dim.z, TZ));
269  filterKernel<<<filterKernelGridSize, filterKernelBlockSize>>>(d_irreg_pts,
270    d_irreg_count, d_grid, d_stencil, rad, delta, d_dim);
271
272  int irreg_count = 0;
273  cudaMemcpy(&irreg_count, d_irreg_count, sizeof(int),
274    cudaMemcpyDeviceToHost);
275  double *d_F = 0;
276  cudaMalloc(&d_F, irreg_count * irreg_count * sizeof(int3));
277
278  const dim3 matrixKernelBlockSize(TX, TY);
279  const dim3 matrixKernelGridSize(divUp(irreg_count, TX),
280                                  divUp(irreg_count, TY));
281  computeFMatrixKernel<<<matrixKernelGridSize, matrixKernelBlockSize>>>(d_F,
282    irreg_count, irreg_count, d_irreg_pts, d_grid, d_stencil, rad, d_lim,
283    delta, d_dim);
284
285  double *d_g = 0;
286  cudaMalloc(&d_g, irreg_count * sizeof(double));
287
```

```
288  const dim3 boundaryKernelBlockSize(TX1D);
289  const dim3 boundaryKernelGridSize(divUp(irreg_count, TX1D));
290  boundaryKernel<<<boundaryKernelGridSize, boundaryKernelBlockSize>>>(d_g,
291    irreg_count, d_irreg_pts, delta, d_lim);
292
293  const int lda = irreg_count, ldb = irreg_count;
294
295  // Initialize the CUSOLVER and CUBLAS context.
296  cusolverDnHandle_t cusolverDnH = 0;
297  cublasHandle_t cublasH = 0;
298  cusolverDnCreate(&cusolverDnH);
299  cublasCreate(&cublasH);
300
301  // Initialize solver parameters.
302  double *tau = 0, *work = 0;
303  int *devInfo = 0, Lwork = 0;
304  cudaMalloc(&tau, irreg_count * sizeof(double));
305  cudaMalloc(&devInfo, sizeof(int));
306  const double alphaCoeff = 1;
307
308  // Calculate the size of work buffer needed.
309  cusolverDnDgeqrf_bufferSize(cusolverDnH, irreg_count, irreg_count, d_F,
310    lda, &Lwork);
311  cudaMalloc(&work, Lwork*sizeof(double));
312
313  // A = QR with CUSOLVER
314  cusolverDnDgeqrf(cusolverDnH, irreg_count, irreg_count, d_F, lda, tau, work,
315    Lwork, devInfo);
316
317  cudaDeviceSynchronize();
318
319  // z = (Q^T)b with CUSOLVER, z is m x 1
320  cusolverDnDormqr(cusolverDnH, CUBLAS_SIDE_LEFT, CUBLAS_OP_T, irreg_count, 1,
```

```
321     irreg_count, d_F, lda, tau, d_g, ldb, work, Lwork, devInfo);
322   cudaDeviceSynchronize();
323
324   // Solve Rx = z for x with CUBLAS, x is n x 1.
325   cublasDtrsm(cublasH, CUBLAS_SIDE_LEFT, CUBLAS_FILL_MODE_UPPER, CUBLAS_OP_N,
326     CUBLAS_DIAG_NON_UNIT, irreg_count, 1, &alphaCoeff, d_F, lda, d_g, ldb);
327
328   double *d_beta = d_g;
329   cublasDestroy(cublasH);
330
331   cusolverDnDestroy(cusolverDnH);
332
333   cudaFree(d_F);
334   cudaFree(tau);
335   cudaFree(devInfo);
336   cudaFree(work);
337
338   double *d_u = 0;
339   cudaMalloc(&d_u, problemSize * sizeof(double));
340   cudaMemset(d_u, 0, problemSize * sizeof(double));
341
342   const dim3 fullAKernelblockSize(TX1D);
343   const dim3 fullAKernelgridSize(divUp(dim.x * dim.y * dim.z, TX1D));
344
345   solveForTheFullAKernel<<<fullAKernelgridSize, fullAKernelblockSize>>>(d_u,
346     d_beta, d_irreg_pts, d_grid, d_stencil, rad, d_lim, delta,
347     d_dim, irreg_count);
348
349   cudaMemcpy(u, d_u, problemSize * sizeof(double), cudaMemcpyDeviceToHost);
350   cudaFree(d_u);
351   cudaFree(d_stencil);
352   cudaFree(d_grid);
353   cudaFree(d_irreg_pts);
```

```
354   cudaFree(d_irreg_count);
355   cudaFree(d_dim);
356   cudaFree(d_lim);
357   cudaFree(d_g);
358 }
```