ORNL/TM-2008/00

Technical Report: Toward a Scalable Algorithm to Compute High-Dimensional Integrals of Arbitrary Functions

8/13/2010

Prepared by
Abigail Snyder
Summer Intern 2010

Yu (Cathy) Jiao, Ph.D.
R&D Staff

ORNL/TM-2008/00

Toward a Scalable Algorithm to Compute High-Dimensional Integrals of Arbitrary Functions

Abigail Snyder
Yu (Cathy) Jiao, Ph.D.

Date Published: August, 2010

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

ABSTRACT

Neutron experiments at the Spallation Neutron Source (SNS) at Oak Ridge National Laboratory (ORNL) frequently generate large amounts of data (on the order of 10^6-10^{12} data points). Hence, traditional data analysis tools run on a single CPU take too long to be practical and scientists are unable to efficiently analyze all data generated by experiments. Our goal is to develop a scalable algorithm to efficiently compute high-dimensional integrals of arbitrary functions. This algorithm can then be used to integrate the four-dimensional integrals that arise as part of modeling intensity from the experiments at the SNS. Here, three different one-dimensional numerical integration solvers from the GNU Scientific Library were modified and implemented to solve four-dimensional integrals. The results of these solvers on a final integrand provided by scientists at the SNS can be compared to the results of other methods, such as quasi-Monte Carlo methods, computing the same integral. A parallelized version of the most efficient method can allow scientists the opportunity to more effectively analyze all experimental data.

1. INTRODUCTION AND MOTIVATION

High-dimensional integrals arise in many areas of science, particularly physics and experimental mathematics. Specifically, experiments performed at the Spallation Neutron Source produce huge quantities of data. Traditional approaches to analyzing the data involve a more or less brute force application of numerical integration schemes such as the Monte Carlo methods. However, when applied to 10^{12} data points, this approach becomes intractably time consuming. Hence, it is necessary to develop an algorithm that modifies, parallelizes and implements a more traditional numerical integration scheme efficiently and scalably. By developing an efficient, scalable algorithm, the data from SNS experiments can be successfully analyzed in full. The goal of scalability is an important one so that the analysis may be performed on both large computing clusters and individual computers with multi-core processors (given that a desktop with 4-8 processing cores is now common). A search of the literature shows that there is no such highly parallel adaptive quadrature solver for high-dimensional integrals currently available.

2. LITERATURE REVIEW

As computers have become more powerful, research into numerical techniques has attempted to make use of each improvement. Thus, there exists a huge amount of literature on the topic of numerical integration in general and specifically on algorithms to compute high-dimensional integrals. The difficulty is finding literature that deals with arbitrary (rather than smooth) integrands. The most common methods are Monte Carlo, quasi-Monte Carlo and quadrature methods. Figure 1 shows the numerical integration algorithms considered.

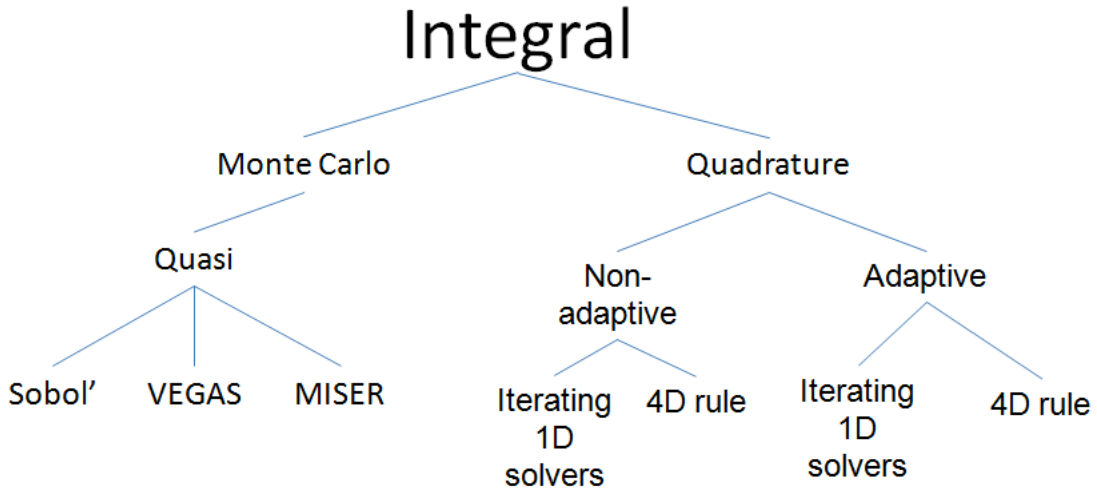

Figure 1. Numerical integration algorithms

2.1 MONTE CARLO METHOD

The textbook by Lemieux provides an introduction to Monte Carlo methods. Simple Monte Carlo integration approximates an integral as the average of the value of the integrand function evaluated at N pseudorandom points in the domain of integration, i.e.

$$\int_a^b f(x)\,dx \cong V\frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

for N pseudorandom points x_i in the volume of the domain of integration V [8]. Unfortunately, this method has some very serious weakness lying in the nature of pseudorandom numbers. Pseudorandom numbers can cluster in the space of integration, causing one region of the space to weight the sample more heavily than others [8]. This leads to lower accuracy.

Two popular methods used to deal with this weakness are recursive stratified sampling and importance sampling. Recursive stratified sampling (similar to the popular algorithm MISER) estimates the error following a Monte Carlo integration. If the error is too large, the region is subdivided and each subregion is integrated, repeating until the error estimate meets a desired tolerance. To keep the number of subdivisions at a minimum, the region is subdivided only in the dimension that will be most beneficial [5]. Unfortunately, this choice of dimension depends on the integrand function and so recursive stratified sampling does not lend itself to evaluating an arbitrary integral. Importance sampling (similar to the popular algorithm VEGAS) takes its points x_i for evaluation from the probability distribution described by |f|. This allows for the x_i's to come from the regions of integration contributing most to the value of the integral [5]. Again, this does not lend itself to implementation in a general solver for the evaluation of an arbitrary integral.

2.2 QUASI-MONTE CARLO METHOD

The quasi-Monte Carlo method uses the same approximation as the Monte Carlo method except that it evaluates f at points x_i of a low-discrepancy sequence (rather than pseudorandom numbers). The idea of a low-discrepancy sequence is that the space of integration is covered by points for evaluation in more ordered and better-covering way [5]. One of the more popular low-discrepancy sequences used for quasi-Monte Carlo integration is the Sobol sequence, in which each x_i is determined by using a primitive polynomial and performing bit-by-bit exclusive-or operations on combinations of the coefficients of the primitive polynomial and previous terms. It is detailed in [1].

2.3 ONE-DIMENSIONAL GAUSSIAN QUADRATURE METHOD

Generally, Gaussian quadrature approximates an integral by

$$\int_a^b f(x)\,dx \cong \sum_{i=1}^{N} w_i f(x_i)$$

for weights w_i and abscissas x_i coming from a chosen orthogonal polynomial [5]. Non-adaptive quadrature methods will simply stop following this evaluation. Adaptive quadrature methods perform an error estimate on the evaluation and, if the error exceeds a given tolerance, the region of integration is subdivided and the quadrature rule is applied separately to each subregion. That is, the region of integration is initially covered with N points to approximate the integral by an N-point quadrature rule. Following subdivision, 2N points are used to approximate the integral by applying an N-point quadrature rule to two subregions. This is repeated for any subregion with large error until the error tolerance is reached [5]. By focusing on regions with the largest error, adaptive methods are more capable of handling difficult (quickly changing for example) integrands than non-adaptive methods because it is possible to break difficult areas down into areas small enough to eliminate the difficulties locally. Unfortunately, this can be quite time-consuming. However, in one-dimension, it is still quite efficient [5].

Gauss-Kronrod rules are an extension of a given N-point Gauss rule. The Kronrod extension adds N+1 points to the N-point rule, yielding a higher-order rule without having to recalculate entirely new points (the original N points get reused) [7]. The difference between an evaluation using the Gauss rule and an evaluation using the corresponding Gauss-Kronrod rule provides a convenient error estimate for use in adaptive schemes.

2.4 MULTI-DIMENSIONAL GAUSSIAN QUADRATURE METHODS

There are two approaches to using multi-dimensional Gaussian quadrature. The first is to recursively call one-dimensional quadrature rules [5].That is, integrate with respect to x_1. Then integrate the result with respect to x_2. This result is then integrated with respect to x_3, and so on. This advantage to this is that it mimics integration by hand and so is very intuitive. And it does work fairly well for smooth integrands in low dimensions [5]. However, it can become very inefficient for complicated integrands or in higher dimensions. The second option is to develop a multi-dimensional quadrature rule, such as by using Smolnyak's construct and taking the tensor product of one-dimensional rules. For integrands that are functions with bounded mixed derivatives, this proves to be very efficient when Gauss-Patterson quadrature rules (an extension that goes beyond the Kronrod extension) are used (even in comparison to quasi-Monte Carlo methods)[7]. The main disadvantage is that this is a much more complicated method to develop rigorously and implement.

2.5 PARALLEL NUMERICAL INTEGRATION

In general, there is significantly more work involved in efficiently parallelizing an adaptive quadrature method than in parallelizing a Monte Carlo method. This is due to the more challenging requirements for proper load distribution. It has been concluded [2] that the most efficient method is using a message passing programming model with each processor maintaining an independent list of

subregions and where the load is balanced by comparing error with a fixed neighbor after a fixed number of integrations and transferring the subregions with the largest error to the neighbor. The number of integrations can be reduced to further improve efficiency. The issue with the reduced method is that it is dependent on a parameter that may take some time to tune. It has further been shown [9] that adaptive quadrature methods parallelized in this way can perform better than a parallelized quasi-Monte Carlo method for certain types of integrands (mostly smooth) and in very low dimensions (<5) both in terms of accuracy versus processing time and accuracy per integrand evaluation. Unfortunately, scalability has only been explored up to 16 [9] and 30 [2] processors and there does not appear to exist a highly-parallel adaptive quadrature method.

3. STATE OF THE ART

3.1 EXISTING SOFTWARE PACKAGES

There are a variety of algorithms and source code available for numerical integration. The two freely available, open source software packages examined were the GNU Scientific Library (GSL) [6] and the QUADRULE package [3].

The GSL offers a variety of solvers for one-dimensional numerical integration. Specifically, it includes solvers capable of solving functions with singularities, functions with known singular points, functions on an infinite interval, singular functions, oscillatory functions and Fourier integrals. The majority of the implementations are adaptive.

The QUADRULE package contains a huge amount of source code for generating different quadrature rules and implementing the generated rules in a non-adaptive solver. In general, it is an excellent resource as a basis to start writing code for a solver, but it would not be possible to implement a truly accurate, efficient multidimensional solver with only the available source code. See Appendix C for more details regarding the necessary steps to use QUADRULE as a basis for a multidimensional solver. Some of the programs examined in detail were (with Burkardt's description of what each does):

- quad_mpi – a parallelized example of a one-dimensional numerical integration
- quadrature_test – "a program which reads the definition of a multidimensional quadrature rule from three files, applies the rule to a number of test integrals, and prints the results."
- quadrule –"a library which defines quadrature rules for approximating integrals;" i.e., have to use both quadrule and quadrature_test to evaluate an integral, also have to pick a rule
- product_rule – "a C++ program which creates a multidimensional quadrature rule by using a product of one-dimensional quadrature rules."
- quadrature_rules – "a dataset directory which contains examples of quadrature rules."
- sparse_grid_gp – "a dataset directory which contains examples of sparse grids, using the idea of a level to control the number of points, and assigning point locations using the Gauss Patterson rule."

It is clear that each of these programs provides very useful elements. However, directly combining several of them will not result in a comprehensive, efficient multidimensional solver.

3.2 ALGORITHM SELECTION

In general, Monte Carlo methods perform faster than Gaussian quadrature. However, this speed comes at the cost of lower precision. Therefore, it is necessary to compare the performance of both Monte Carlo and quadrature methods on several test integrals in order to determine which

performs with the desired combination of precision and speed. This research focuses on recursively calling one-dimensional adaptive quadrature solvers four times to solve a four-dimensional integral. This method was chosen as a first approach due to the ready availability of open-source code that could be easily adapted. Future approaches may involve developing a four-dimensional quadrature rule and an adaptive solver using it.

Functions from [5] serve as the basis of the framework for recursive integration, calling each of three different solvers from the GNU Scientific Library (GSL) [6]. The solvers chosen from the GSL as the final choices for implementation are QNG, QAG, and QAGS.

QNG is a non-adaptive method that successively applies the 10-point, 21-point, 43-point and 87-point Gauss-Kronrod integration rules until the estimate of integral is within desired error limits. This makes the QNG method closer to an adaptive method than a truly non-adaptive method. Adaptive methods intelligently subdivide only the regions where the error exceeds a given tolerance; QNG subdivides the entire region when the error exceeds a given tolerance.

QAG adaptively applies a 15, 21, 31, 41, 51, or 61 point Gauss-Kronrod rule according to the user's choice until the estimate of the integral is within desired error limits. This can require a user to do extra testing to determine the most efficient choice for solving an integral.

QAGS adaptively applies a 21 point Gauss-Kronrod rule until the estimate of the integral is within desired error limits. While it is capable of handling some functions with singularities (unlike QNG or QAG), it cannot solve a function when the singularity occurs at an abscissa.

4. IMPLEMENTATION

4.1 GOAL

The goal is to solve a final integral from experiments at the Spallation Neutron Source calculating the intensity at a given point $(\vec{Q},E) = (Qx,Qy,Qz,E)$:

$$I_{calculated}(\vec{Q},E) = \iint S(\overrightarrow{Q'}+\vec{Q},E'+E)R(\overrightarrow{Q'},E')d\overrightarrow{Q'}dE' \quad (1)$$

for machine resolution function $R(\vec{Q},E)$ and neutron scattering function $S(\vec{Q},E)$. This is the integral that must be computed at the 10^{12} data points.

The resolution function, for R_0 constant and M a symmetric matrix, is given by:

$$R(\vec{Q},E) = R_0 \mathrm{x} e^{-([Qx,Qy,Qz,E]\mathrm{T}M[Qx,Qy,Qz,E])} \quad (2)$$

The scattering function is given by:

$$S(Q,E) = S_{background} + S_{incoh} + S_{ladder} \quad (3)$$

for

$$S_{background} = constant \quad (4)$$

$$S_{incoh} = \frac{maxincoh}{\sqrt{2\pi}sincoh} \mathrm{x} e^{-\frac{E^2}{2sincoh^2}} \quad (5)$$

where *maxincoh* is a constant parameter and *sincoh* is the standard deviation.

$$S_{ladder} = \frac{maxint}{\sqrt{2\pi}sigmaE} \mathrm{x} e^{-\frac{(E-w)^2}{2sigmaE^2}} \mathrm{x} \frac{fdperp \,\mathrm{x}\, ff^2}{w} \quad (6)$$

where *maxint* is a constant parameter, *sigmaE* is the standard deviation, *w* is the dispersion given by:

$$w = gap + \frac{width}{2}(1+\cos(2\pi Qx)) \quad (7)$$

fdperp is a dimer form factor given by:

$$fdperp = sin^2(\pi \mathrm{x}(0.3904 \mathrm{x} Qx + 0.3832 \mathrm{x} Qy)) \quad (8)$$

and *ff* is a magnetic form factor given by:

$$ff = j_0[0]\mathrm{x}e^{-j_0[1]\mathrm{x}s_2} + j_0[2]\mathrm{x}e^{-j_0[3]\mathrm{x}s_2} + j_0[4]\mathrm{x}e^{-j_0[5]\mathrm{x}s_2} + j_0[6] + (1-\frac{2}{g})\mathrm{x}s_2\mathrm{x}(j_2[0]\mathrm{x}e^{-j_2[1]\mathrm{x}s_2} + j_2[2]\mathrm{x}e^{-j_2[3]\mathrm{x}s_2} + j_2[4]\mathrm{x}e^{-j_2[5]\mathrm{x}s_2} + j_2[6]) \quad (9)$$

for eight element arrays j_0 and j_2 and s_2 given by:

$$s2=0.25x(\frac{Qx^2}{a^2}+\frac{Qy^2}{b^2}+\frac{Qz^2}{c^2}) \quad (10)$$

for constants a, b and c

While the space of data is infinite, the limits of integration must be rescaled in each dimension to make the problem finite and more easily solvable. After rescaling, the new limits of integration become $(-\pi/2, \pi/2)$ for each dimension. This rescaling is performed by sending each variable to the tangent of that variable, i.e. Qx becomes $\tan(Qx)$.

4.2 IMPLEMENTATION DETAILS

The function quad3d.h from [5] was used as a basis to develop a program that recursively calls GSL one-dimensional integration solvers four times to solve a four dimensional integral. The main programming challenge was the strict function definition requirements of the GSL solvers. This necessitated the global definition of limits of integration and error tolerances (parameters that GSL integrators call) since the function definitions of each round of integration cannot be changed to accommodate passing the limits of integration as parameters. The general algorithm for numerical integration is depicted in Figure 2.

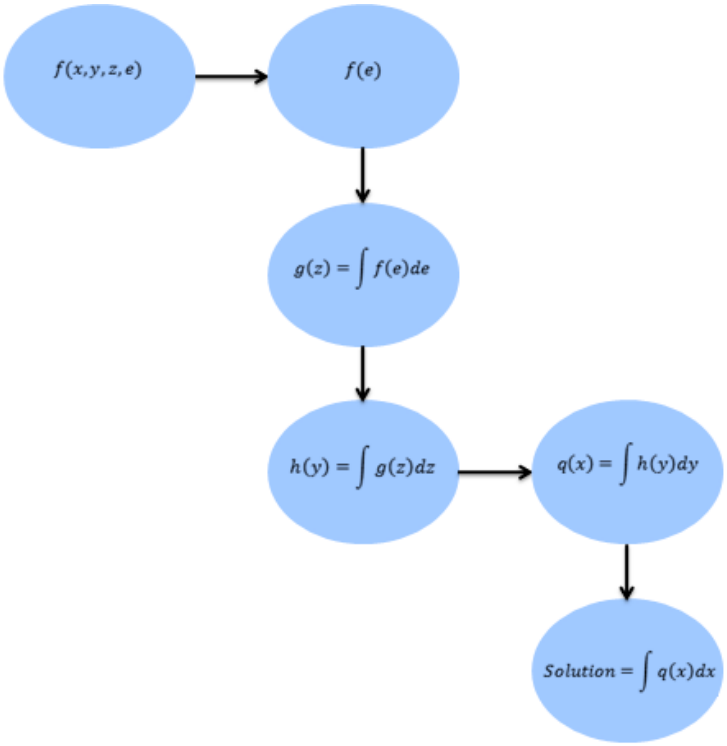

Figure 2. Flowchart depicting numerical integration

Two methods of implementing the integrand function f were explored. Both use the same algorithm for numerical integration. The difference is that Implementation 1 defines the integrand explicitly within the function f and Implementation 2 defines the integrand as external functions called by f. When evaluating the same integral, both implementations should return the same result at about the same speed. However, our experiments contradict this assumption. The first implementation

uses the following pseudo code:

Implementation 1 – Integrand Explicitly Defined in f

Step 1: Define the function to be integrated (for fixed $(\overrightarrow{Q_\iota}, E_i)$)
f(Qx', Qy', Qz', E')=$S\left(\overrightarrow{Q_\iota} + \overrightarrow{Q'}, E_i + E'\right)R(\overrightarrow{Q'}, E')$, the limits of integration and error tolerances.
Step 2: Pass f(Qx′, Qy′, Qz′, E′) to a framework function that holds Qx′, Qy′ and Qz′ fixed, i.e. that sets f(Qx′, Qy′, Qz′, E′) = f(E′).
Step 3: Pass f(E′) to a function calling the GSL integration method chosen to integrate f(e) with respect to E′.
Step 4: Set g(Qz′)=result of Step 3.
Step 5: Pass g(Qz′) to a function calling the GSL integration method chosen to integrate g(Qz′) with respect to Qz′.
Step 6: Set h(Qy′)=result of Step 5.
Step 7: Pass h(Qy′) to a function calling the GSL integration method chosen to integrate h(Qy′) with respect to Qy′.
Step 8: Set q(Qx′)=result of Step 7.
Step 9: Pass q(Qx′) to a function calling the GSL integration method chosen to integrate q(Qx′) with respect to Qx′.
Step 10: Return the result of Step 9.

The second implementation for the integrand function f used in equation (1) at a fixed data point $(\overrightarrow{Q_\iota}, E_i) = (Qx_i, Qy_i, Qz_i, E_i)$ is to have the integrand function f call the separately defined resolution and scattering functions. By comparison, Implementation 1 would have the resolution and scattering functions explicitly defined in the integrand function f. The pseudo code is very similar to Implementation 1, with only the implementation of the integrand function f changing:

Implementation 2 – Integrand Defined Separately and Called by f

Step 1: Define the resolution function R.
Step 2: Define the scattering function S.
Step 3: Define the function to be integrated (for fixed $(\overrightarrow{Q_\iota}, E_i)$)
f(Qx', Qy', Qz', E')=$S\left(\overrightarrow{Q_\iota} + \overrightarrow{Q'}, E_i + E'\right)R(\overrightarrow{Q'}, E')$, the limits of integration and error tolerances.
Step 4: Pass f(Qx′, Qy′, Qz′, E′) to a framework function that holds Qx′, Qy′ and Qz′ fixed, i.e. that sets f(Qx′, Qy′, Qz′, E′) = f(E′).
Step 5: Pass f(E′) to a function calling the GSL integration method chosen to integrate f(e) with respect to E′.
Step 6: Set g(Qz′)=result of Step 5.
Step 7: Pass g(Qz′) to a function calling the GSL integration method chosen to integrate g(Qz′) with respect to Qz′.
Step 8: Set h(Qy′)=result of Step 7.
Step 9: Pass h(Qy′) to a function calling the GSL integration method chosen to integrate h(Qy′) with respect to Qy′.
Step 10: Set q(Qx′)=result of Step 9.
Step 11: Pass q(Qx′) to a function calling the GSL integration method chosen to integrate q(Qx′) with respect to Qx′.
Step 12: Return the result of Step 11.

To perform the integration at all data points $(\overrightarrow{Q_\iota}, E_i) = (Qx_i, Qy_i, Qz_i, E_i)$, it is necessary to include a loop to generate the data points and a loop to evaluate equation (1) at each data point.

5. PERFORMANCE EVALUATION

5.1 PERFORMANCE METRICS: ACCURACY AND SPEED

Accuracy and speed were used as the performance metrics to evaluate Implementation 1 and Implementation 2. Both methods of integrand implementation (Implementation 1and Implementation 2) of QNG, QAG and QAGS were tested on several functions that could be solved analytically. Accuracy tests were performed using three different Genz functions [4] as integrands (see Table 1). To implement a Genz function as an integrand, the function R was taken to be of constant value 1 and the function S was taken to be the Genz function at the fixed point (Qx_i, Qy_i, Qz_i, E_i) = (0,0,0,0). In other words, function f only depends on the S function (the Genz function).

5.2 SENSITIVITY TO INTEGRAND

Three of the Genz test functions in four dimensions were used to determine the sensitivity of quadrature methods to integrand: oscillatory, Gaussian and product peak. Again, it is assumed that S takes the form of these functions. The functions are defined in Table 1. The actual value comes from integrating from (-π/2, π/2) for each dimension.

Table 1. Three Genz Functions

Function	Actual Value
$f_1(x,y,z,t) = \cos(2\pi + x + y + z + t)$	16
$f_2(x,y,z,t) = e^{-(x^2+y^2+z^2+t^2)}$	8.8708
$f_3(x,y,z,t) = \frac{1}{x^2y^2z^2t^2}$	$\frac{4^4}{\pi^4}$

For ease of visualization, each function is plotted in two dimensions in Figure 3 (oscillatory), Figure 4 (Gaussian) and Figure 5 (product peak).

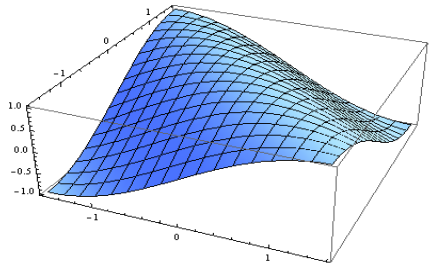

Figure 3. Oscillatory $f(x,y) = \cos(2\pi + x + y)$

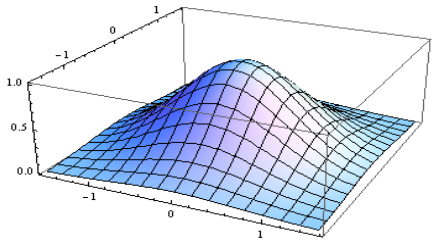

Figure 4. Gaussian $f(x,y) = e^{-(x^2+y^2)}$

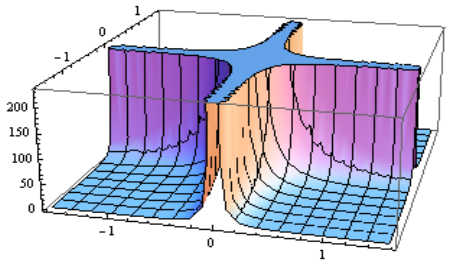

Figure 5. Product Peak $f(x, y) = \frac{1}{x^2 y^2}$

Each Genz function was tested using the integrand definition of both Implementation 1 and Implementation 2 for each of QNG, QAG and QAGS and many of Mathematica's [10] numerical integration methods.

The comparison with Mathematica's methods provides valuable information about which methods might be promising for future implementation: the comparison of Mathematica's various quadrature rules and Monte Carlo methods with Mathematica's Gauss-Kronrod rule allows some standard for comparison to the implementation of the GSL solvers (which also use various Gauss-Kronrod rules).

In general, this provides a more comprehensive comparison of numerical methods. To gain an estimate of evaluation time for each numerical integration method a for loop was used to call integration 100 times when the integration took less than one second. The execution time was then divided by 100 to determine an average evaluation time. When integration is greater than one second, only one evaluation is performed. If a single integrand evaluation takes greater than 15 minutes, the evaluation is stopped.

It can be concluded that all numerical integration methods are quite sensitive to the choice of integrand (not just quadrature methods). In general, it is not possible to numerically integrate functions with infinite discontinuities in the region of integration. Specifically, it was found that for the Gaussian and oscillatory functions, the GSL solvers perform with better accuracy than the Monte Carlo methods and the same accuracy as Mathematica's quadrature methods. It was found that the product peak function is not integrable by any method. The quadrature methods return errors due to the integrand being infinity at 0 (an abscissa). The Monte Carlo methods return an incorrect value due to the steepness of the integrand's slopes. Finally, it was found that the GSL solvers are among the fastest methods for solving the integrals. Results of the tests can be found in Table 2 (oscillatory), Table 3 (Gaussian) and Table 4 (product peak). Figure 6 (oscillatory) and Figure 7 (Gaussian) compare the different methods versus the execution time for evaluating the integral.

It was found that for sufficiently smooth integrands, all solvers used in Implementation 1 perform with perfect accuracy. When Implementation 2 is used, the solvers tend to perform with an average accuracy of 10^{-2}. For, again, sufficiently smooth integrands, it was found that QNG was the fastest method on average, with Implementation 2 being faster than Implementation 1 at evaluating the same integrand. This is contrary to the expectation that both integrand implementations would perform similarly. Indeed, since function calls usually incur additional computational overhead and therefore, it would be expected that Implementation 2 performed more slowly. However, Implementation 2 is significantly faster at the cost of a lower precision. Therefore, users need to carefully weigh the tradeoff between speed and accuracy when choosing a specific implementation.

Table 2. Results of testing Genz Oscillatory function for GSL solvers and Mathematica's methods

Method	Description	Average Time/Integral (s)	Accuracy (Returned-Actual)
N[Integrate[]]	Evaluates analytically as far as possible, then numerically what remains	0.46297	0
NIntegrate[]	Mathematica's default, adaptive numerical integrator	0.28016	0
NIntegrate[], Method: GlobalAdaptive	Adaptive subdivisions based on global error estimates	0.2764	0
NIntegrate[], Method: GlobalAdaptive, Rule: CartesianRule	Uses Cartesian product of rules for the quadrature rule	0.09344	0
NIntegrate[], Method: GlobalAdaptive, Rule: ClenshawCurtisRule	Uses Clenshaw-Curtis rule for the quadrature rule	20.1099	0
NIntegrate[], Method: GlobalAdaptive, Rule: GaussKronrodRule	Uses the Kronrod extension of Gaussian quadrature	0.09421	0
NIntegrate[], Method: GlobalAdaptive, Rule: LobattoKronrodRule	Uses the Kronrod extension of Gauss-Lobatto quadrature	1.25172	0
NIntegrate[], Method: GlobalAdaptive, Rule: NewtonCotesRule	Uses the Newton-Cotes rule to approximate	689.813	0
NIntegrate[], Method: GlobalAdaptive, Rule: TrapezoidalRule	Uniform points in one dimension	>15 min	n/a
NIntegrate[], Method: Local Adaptive	Adaptive subdivisions based on local error estimates	0.67	0
NIntegrate[], Method: LocalAdaptive, Rule: CartesianRule	Uses Cartesian product of rules for the quadrature rule	0.69687	0
NIntegrate[], Method: LocalAdaptive, Rule: ClenshawCurtisRule	Uses Clenshaw-Curtis rule for the quadrature rule	11.0053	0
NIntegrate[], Method: LocalAdaptive, Rule: GaussKronrodRule	Uses the Kronrod extension of Gaussian quadrature	0.67219	0

NIntegrate[], Method: LocalAdaptive, Rule: LobattoKronrodRule	Uses the Kronrod extension of Gauss-Lobatto quadrature	0.66141	0
NIntegrate[], Method: LocalAdaptive, Rule: NewtonCotesRule	Uses the Newton-Cotes rule to approximate	17.218	0
NIntegrate[], Method: LocalAdaptive, Rule: TrapezoidalRule	Uniform points in one dimension	>15 min	n/a
NIntegrate[], Method: DoubleExponential	Uses the Double Exponential (Tanh-Sinh) quadrature rule	21.8628	0
NIntegrate[], Method: MonteCarlo	Uses Monte Carlo integration	0.12469	0.2282
NIntegrate[], Method: AdaptiveMonteCarlo	Uses adaptive Monte Carlo integration	0.26437	0.0789
NIntegrate[], Method: QuasiMonteCarlo	Uses quasi-Monte Carlo integration	0.29078	0.0009
NIntegrate[], Method: AdaptiveQuasiMonteCarlo	Uses adaptive quasi-Monte Carlo integration	2.36906	0.0119
QNG – Implementation 1	GSL non-adaptive solver	0.03203	0
QAG – Implementation 1	GSL adaptive solver	0.02453	0
QAGS – Implementation 1	GSL adaptive solver for singularities	0.06718	0
QNG – Implementation 2	GSL non-adaptive solver	0.218	0.0001
QAG – Implementation 2	GSL adaptive solver	0.109	0.0001
QAGS – Implementation 2	GSL adaptive solver for singularities	0.218	0.0001

NOTE: n/a in Total Time means the integral was only evaluated once. n/a in Result means no result was returned

Table 3. Results of testing Genz Gaussian function for GSL solvers and Mathematica's methods

Method	**Description**	**Time/Integral (s)**	**Accuracy (Returned-Actual)**
NIntegrate[]	Mathematica's default, adaptive numerical integrator	0.40688	~0
N[Integrate[]]	Evaluates analytically as far as possible, then numerically what remains	0.23516	~0

NIntegrate[], Method: GlobalAdaptive	Adaptive subdivisions based on global error estimates	0.40343	~0
NIntegrate[], Method: GlobalAdaptive, Rule: CartesianRule	Uses Cartesian product of rules for the quadrature rule	1.26578	~0
NIntegrate[], Method: GlobalAdaptive, Rule: ClenshawCurtisRule	Uses Clenshaw-Curtis rule for the quadrature rule	9.875	~0
NIntegrate[], Method: GlobalAdaptive, Rule: GaussKronrodRule	Uses the Kronrod extension of Gaussian quadrature	1.406	~0
NIntegrate[], Method: GlobalAdaptive, Rule: LobattoKronrodRule	Uses the Kronrod extension of Gauss-Lobatto quadrature	1.359	~0
NIntegrate[], Method: GlobalAdaptive, Rule: NewtonCotesRule	Uses the Newton-Cotes rule to approximate	940.297	~0
NIntegrate[], Method: GlobalAdaptive, Rule: TrapezoidalRule	Uniform points in one dimension	>15 min	n/a
NIntegrate[], Method: Local Adaptive	Adaptive subdivisions based on local error estimates	0.09297	0.00002
NIntegrate[], Method: LocalAdaptive, Rule: CartesianRule	Uses Cartesian product of rules for the quadrature rule	1.703	~0
NIntegrate[], Method: LocalAdaptive, Rule: ClenshawCurtisRule	Uses Clenshaw-Curtis rule for the quadrature rule	8.812	~0
NIntegrate[], Method: LocalAdaptive, Rule: GaussKronrodRule	Uses the Kronrod extension of Gaussian quadrature	1.703	~0
NIntegrate[], Method: LocalAdaptive, Rule: LobattoKronrodRule	Uses the Kronrod extension of Gauss-Lobatto quadrature	0.76907	~0
NIntegrate[], Method: LocalAdaptive, Rule: NewtonCotesRule	Uses the Newton-Cotes rule to approximate	14.188	~0
NIntegrate[], Method: LocalAdaptive, Rule: TrapezoidalRule	Uniform points in one dimension	234.406	~0

NIntegrate[], Method: DoubleExponential	Uses the Double Exponential (Tanh-Sinh) quadrature rule	14.782	~0
NIntegrate[], Method: MonteCarlo	Uses Monte Carlo integration	0.06641	0.12253
NIntegrate[], Method: AdaptiveMonteCarlo	Uses adaptive Monte Carlo integration	0.06313	0.03146
NIntegrate[], Method: QuasiMonteCarlo	Uses quasi-Monte Carlo integration	0.18906	0.00785
NIntegrate[], Method: AdaptiveQuasiMonteCarlo	Uses adaptive quasi-Monte Carlo integration	0.22172	0.01324
QNG – Implementation 1	GSL non-adaptive solver	0.03516	~0
QAG – Implementation 1	GSL adaptive solver	1.05031	~0
QAGS – Implementation 1	GSL adaptive solver for singularities	0.06969	~0
QNG – Implementation 2	GSL non-adaptive solver	0.187	~0.022
QAG – Implementation 2	GSL adaptive solver	2.937	~0.022
QAGS – Implementation 2	GSL adaptive solver for singularities	0.187	~0.022

NOTE: n/a in Total Time means the integral was only evaluated once. n/a in Result means no result was returned. Error is approximate due to the fact that the solution must be truncated at some value.

Table 4. Results of testing Genz Product Peak function for GSL solvers and Mathematica's methods

Method	Description	Time/Integral (s)	Accuracy (Returned-Actual)
NIntegrate[]	Mathematica's default, adaptive numerical integrator	1.89625	None returned
N[Integrate[]]	Evaluates analytically as far as possible, then numerically what remains	0.32813	None returned
NIntegrate[], Method: GlobalAdaptive	Adaptive subdivisions based on global error estimates	1.88782	None returned
NIntegrate[], Method: GlobalAdaptive, Rule: CartesianRule	Uses Cartesian product of rules for the quadrature rule	70.9467	None returned
NIntegrate[], Method: GlobalAdaptive, Rule: ClenshawCurtisRule	Uses Clenshaw-Curtis rule for the quadrature rule	38.953	None returned

NIntegrate[], Method: GlobalAdaptive, Rule: GaussKronrodRule	Uses the Kronrod extension of Gaussian quadrature	58.719	None returned
NIntegrate[], Method: GlobalAdaptive, Rule: LobattoKronrodRule	Uses the Kronrod extension of Gauss-Lobatto quadrature	38.078	None returned
NIntegrate[], Method: GlobalAdaptive, Rule: NewtonCotesRule	Uses the Newton-Cotes rule to approximate	4.547	None returned
NIntegrate[], Method: GlobalAdaptive, Rule: TrapezoidalRule	Uniform points in one dimension	39.39	None returned
NIntegrate[], Method: Local Adaptive	Adaptive subdivisions based on local error estimates	>15 min	n/a
NIntegrate[], Method: LocalAdaptive, Rule: CartesianRule	Uses Cartesian product of rules for the quadrature rule	>15 min	n/a
NIntegrate[], Method: LocalAdaptive, Rule: ClenshawCurtisRule	Uses Clenshaw-Curtis rule for the quadrature rule	>15 min	n/a
NIntegrate[], Method: LocalAdaptive, Rule: GaussKronrodRule	Uses the Kronrod extension of Gaussian quadrature	>15 min	n/a
NIntegrate[], Method: LocalAdaptive, Rule: LobattoKronrodRule	Uses the Kronrod extension of Gauss-Lobatto quadrature	>15 min	n/a
NIntegrate[], Method: LocalAdaptive, Rule: NewtonCotesRule	Uses the Newton-Cotes rule to approximate	>15 min	n/a
NIntegrate[], Method: LocalAdaptive, Rule: TrapezoidalRule	Uniform points in one dimension	>15 min	n/a
NIntegrate[], Method: DoubleExponential	Uses the Double Exponential (Tanh-Sinh) quadrature rule	0.02687	None returned
NIntegrate[], Method: MonteCarlo	Uses Monte Carlo integration	0.18453	$O(10^{11})$
NIntegrate[], Method: AdaptiveMonteCarlo	Uses adaptive Monte Carlo integration	0.16359	$O(10^{25})$
NIntegrate[], Method: QuasiMonteCarlo	Uses quasi-Monte Carlo integration	0.35797	$O(10^{14})$

NIntegrate[], Method: AdaptiveQuasiMonteCarlo	Uses adaptive quasi-Monte Carlo integration	0.09266	$O(10^{18})$
QNG – Implementation 1	GSL non-adaptive solver	n/a	n/a
QAG – Implementation 1	GSL adaptive solver	n/a	n/a
QAGS – Implementation 1	GSL adaptive solver for singularities	n/a	n/a
QNG – Implementation 2	GSL non-adaptive solver	n/a	n/a
QAG – Implementation 2	GSL adaptive solver	n/a	n/a
QAGS – Implementation 2	GSL adaptive solver for singularities	n/a	n/a

NOTE: N/A in Total Time means the integral was only evaluated once. N/A in Result and Time/Integral means no result was returned.

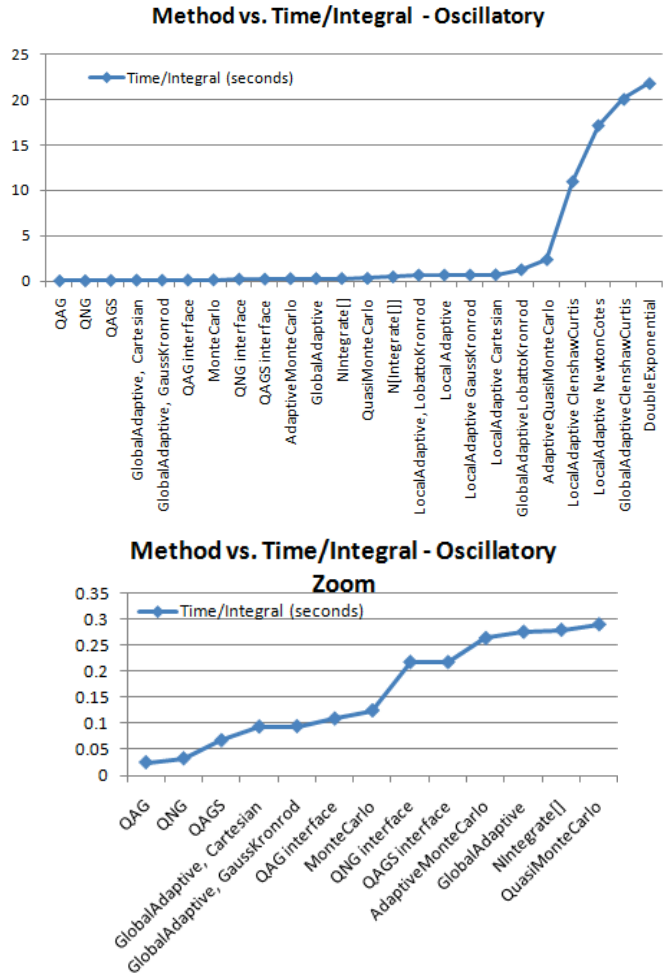

Figure 6. Method vs. Time/Integral plots of the Genz Oscillatory function

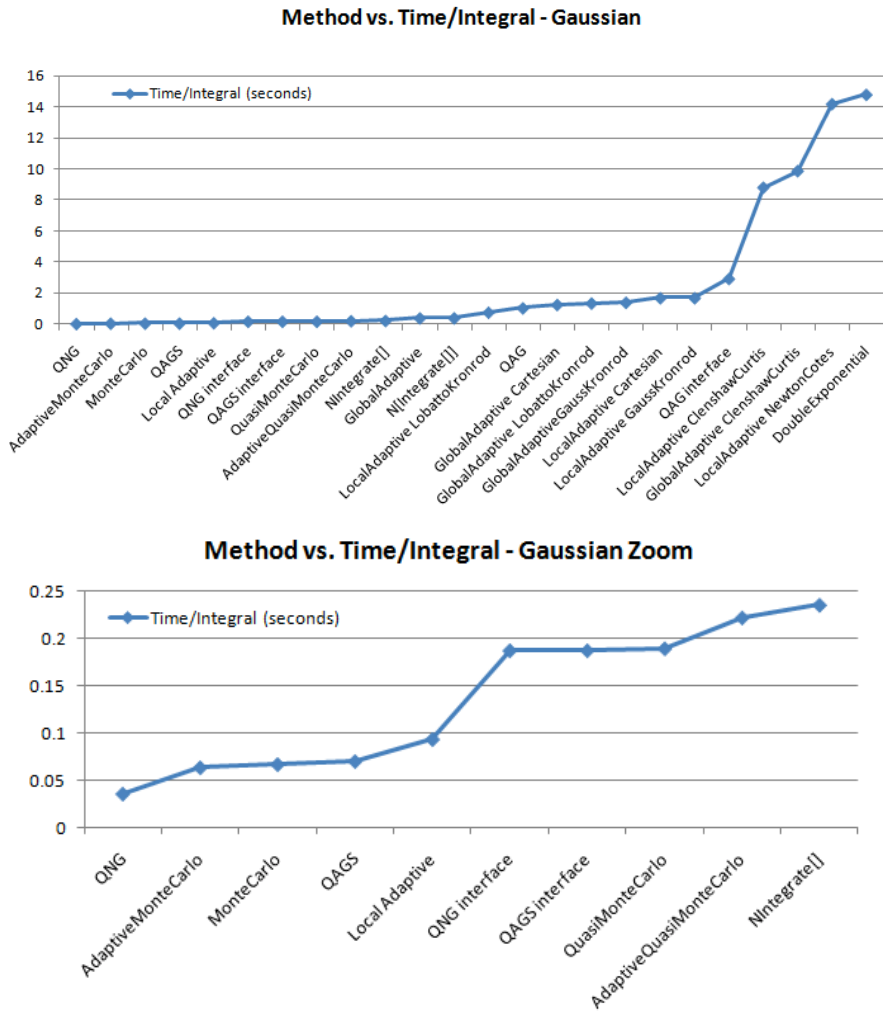

Figure 7. Method vs. Time/Integral for Genz Gaussian function

5.3 SERIAL TEST ON EQUATION (1)

The example integrand (R defined in (2), S defined in (3)-(10)) at the point (Qx_i, Qy_i, Qz_i, E_i) = (-1,-1,-1,0) was evaluated using Implementation 1 and Implementation 2 for each of QNG, QAG, and QAGS. The results can be found in Table 5. Accuracy is defined as the difference between the returned value and the true value.

Table 5. Results of testing equation (1)

Implementation 1	Time (s)	Returned	Accuracy
QNG	Error: failed to reach tolerance with highest-order rule		
QAG	13290.843	29.9458	~0.00116
QAGS	46920.328	29.9458	~0.00116
Implementation 2	**Time (s)**	**Returned**	**Accuracy**
QNG	Error: failed to reach tolerance with highest-order rule		
QAG	7902.8	29.9458	~0.00116
QAGS	31401.5	29.9458	~0.00116

It can be seen that neither version of QNG was capable of solving the final integrand. The fastest evaluation, by Implementation 2 of QAG, took more than two hours. All four tests returning a result returned the same result: 29.9458. The extreme time required for solving this integrand can be attributed to two things. First, quadrature methods perform best on sufficiently smooth functions in low dimensions (three dimensions or less). This integrand is not sufficiently smooth for these methods and the dimension of the problem is larger than ideal. Second, the method of recursively calling one-dimensional integration four times can be computationally expensive. In particular, the weights and abscissas must be generated each time integration is performed.

6. CONCLUSIONS AND FUTURE WORK

To evaluate all 10^{12} integrals for the full problem would take more than 225 million years using the fastest method, Implementation 2 of QAG (over two hours to solve the integral at one point). This is too slow to be useful. Especially by comparison to the Monte Carlo methods implemented by others: evaluating the final integrand at the point (Qx_i, Qy_i, Qz_i, E_i) = (-1,-1,-1,0) took less than a second using a similarly implemented quasi-Monte Carlo method. Since parallelization of the code was to be done by parallelizing over data points, it is not a worthwhile exercise to parallelize any of the quadrature methods currently implemented.

Future work on quadrature methods should explore alternative implementations. Recall, the method implemented here was to call one-dimensional solvers recursively four times. While this is an intuitive first method (it mimics the order of analytical integration by hand), it is clearly not sufficiently efficient. An alternative method is to explore generating four-dimensional quadrature rules. A simple four-dimensional rule would be to take either the Cartesian or tensor product of one-dimensional rules (see [8]). The four-dimensional rule would then be summed once to approximate the integral, as compared to the previous method of summing a one-dimensional rule four times. There is open-source code available that can serve as a starting point in such a construction (see Appendix C). However, it will require significant modification to solve arbitrary integrals efficiently.

7. REFERENCES

[1] P. Bratley and B. Fox, "Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator," *ACM Transactions on Mathematical Software,* vol. 14, no. 1, Mar. 1988.

[2] J. Bull and T. Freeman, Parallel algorithms for multi-dimensional integration. *Parallel and Distributed Computing Practices*, vol. 1, no. 1, pp. 89-102, 1998.

[3] J. Burkardt, *QUADRULE.* [Online source code] Available: http://people.sc.fsu.edu/~jburkardt/cpp_src/cpp_src.html

[4] J. Burkardt, *TESTPACK.* [Online source code] Available: http://people.sc.fsu.edu/~jburkardt/cpp_src/cpp_src.html

[5] B. Flannery, W. Press, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C*. New York: Cambridge University Press, 1992. [E-book] Available: http://www.nrbook.com/a/bookcpdf.php.

[6] M. Galassi et al, *GNU Scientific Library Reference Manual.* 3rd Ed. [E-book] Available: http://www.gnu.org/software/gsl/

[7] T. Gerstner and M. Griebel, "Numerical Integration Using Sparse Grids," *Numerical Algorithms,* vol. 18, no. 3-4, Jan. 1998.

[8] C. Lemieux, *Monte Carlo and Quasi-Monte Carlo Sampling*, 1st ed. New York: Addison Springer New York, 2009. [E-book] Available: Google Books e-book.

[9] R. Schürer, "Parallel High-Dimensional Integration: Quasi-Monte Carlo vs. Adaptive Cubature Rules" in *Lecture Notes in Computer Science*. Heidelberg: Springer Berlin, 2001. [E-book] Available: SpringerLink e-book.

[10] Wolfram Research, Inc., *Mathematica,* Version 7.0. Champaign, IL: 2008.

APPENDIX A. ENVIRONMENT SETUP

Eclipse

The Eclipse Galileo C/C++ IDE version 3.5 for a 32-bit Windows machine was downloaded from http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/helios/R/eclipse-cpp-helios-win32.zip. The automatic download process went smoothly and the program was installed at C:\Users\8y7\Documents\eclipse-cpp-galileo-SR2-win32. Eclipse requires the download of a complier (either Cygwin or MinGW compilers are recommended) in order to work.

Following the recommendations of the "Before you begin" section of the Eclipse C/C++ Development User Guide (under the help section of Eclipse), the MinGW compiler and gdb debugger were installed. Specifically, MinGW version 5.1.6 was downloaded as MinGW-5.1.6.exe from http://sourceforge.net/projects/mingw/files/, following the links from the "Before you begin" guide. It was downloaded to C:\MinGW.

The only possible dependencies clear from the instructions in the "Before you begin" guide were the gdb debugger and MSYS. Hence gdb-6.6.tar.bz2 was downloaded from http://sourceforge.net/mingw/gdb-6.6.tar.bz2, also following the links from the "Before you begin" guide. Following the recommendation of the guide to extract the contents of the debugger to the same location where MinGW was installed, the contents were extracted to C:\MinGW under the folder "debugger" (C:\MinGW\debugger). To install MSYS, msys-1.0.10.exe was downloaded from http://sourceforge.net/projects/mingw/files/MSYS/BaseSystem/msys-1.0.10/MSYS-1.0.10.exe/download. The msys-1.0.10.exe was run and MSYS was installed in C:\msys\1.0. The installation auto-suggested MinGW as location of program's shortcuts and this option was selected.

To use the solvers from the GSL, the GSL must be properly linked to Eclipse. Unfortunately, there does not appear to be any comprehensive or formal instructions for doing this available. Instructions from http://www.eclipse.org/forums/index.php?t=msg&goto=231303&S=e12b33cc44ec8098c3abf3d5a3bc9056#msg_231303, http://www.eclipse.org/forums/index.php?t=msg&goto=497917& and http://whatwouldnickdo.com/wordpress/328/eclipse-cdt-and-linux-libraries/ are being used to link the library.

Specifically, gsl-1.11.tar.gz was downloaded from ftp://mirrors.usc.edu/pub/gnu/gsl/ (a mirror from the GSL website) and its files were extracted to C:\Users\8y7\workspace. In Eclipse, "C:\Users\8y7\workspace\gsl-1.11\include"(quotes included) was added to Project Properties -> C/C++ General -> Paths and Symbols -> Include for both Gnu C and Gnu C++ languages. GSL was added to Project Properties -> C/C++ Build -> Settings -> Tool Settings -> MinGW C++ Linker -> Libraries -> Libraries (-l). And "C:\Users\8y7\workspace\gsl-1.11" (quotes included) was added to Project Properties -> C/C++ Build -> Settings -> Tool Settings -> MinGW C++ Linker -> Libraries -> Library Search Path (-L). This, however, presents errors when trying to run a sample program that calls GSL functions (such as the one from the GSL manual available at http://www.gnu.org/software/gsl/manual/html_node/Numerical-integration-examples.html).

Therefore, Eclipse was used primarily as a text editor. Cygwin was used for compiling and running programs since it includes the GSL.

Cygwin

Cygwin was downloaded as an alternative to linking the GSL to Eclipse. Following the link from the Cygwin website http://www.cygwin.com/, setup.exe was downloaded and run. The setup is fairly automated and files are installed at C:/cygwin. The mirror ftp://ftp.gtlib.gatech.edu was chosen. The "Install" option was chosen for all files (rather than the "Default"). Any dependencies the installer recognized were fixed automatically.

Testing the same GSL example program used in Eclipse (available at http://www.gnu.org/software/gsl/manual/html_node/Numerical-integration-examples.html) resulted in a successful run with results that match those given by the GSL.

The GSL was installed and compiled on the Oak Ridge Institutional Cluster (OIC) by logging on to the OIC and using the commands:
scp gsl-1.11.tar.gs userid@bes-inter.ornl.gov:~/
gunzip gsl-1.11.tar.gz
tar –xvf gsl-1.11.tar
mkdir libgsl
./configure --prefix=/home/PATH_WHERE_GSL_INSTALLED
make
make check
make install
make clean
No errors were returned during this process.
Using the commands:

export LD_LIBRARY_PATH=/home/PATH_WHERE_GSL_INSTALLED/lib:
$LD_LIBRARY_PATH

g++ main.cpp -o main -I/home/ PATH_WHERE_GSL_INSTALLED/include -L/home/
PATH_WHERE_GSL_INSTALLED/lib –lgsl -lgslcblas –lm

successfully runs a program main.cpp on the OIC using the GSL.

APPENDIX B. LIST OF LINKS TO RESOURCES USED

1. Website to download Eclipse: http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/helios/R/eclipse-cpp-helios-win32.zip
2. Website to download MinGW compiler: http://sourceforge.net/projects/mingw/files/
3. Website to download gdb debugger: http://sourceforge.net/mingw/gdb-6.6.tar.bz2
4. Website to download MSYS: http://sourceforge.net/projects/mingw/files/MSYS/BaseSystem/msys-1.0.10/MSYS-1.0.10.exe/download
5. Website 1 for instructions linking the GSL to Eclipse (unsuccessful): http://www.eclipse.org/forums/index.php?t=msg&goto=231303&S=e12b33cc44ec8098c3abf3d5a3bc9056#msg_231303
6. Website 2 for instructions linking the GSL to Eclipse (unsuccessful): http://www.eclipse.org/forums/index.php?t=msg&goto=497917&
7. Website 3 for instructions linking the GSL to Eclipse (unsuccessful): http://whatwouldnickdo.com/wordpress/328/eclipse-cdt-and-linux-libraries/
8. Mirror GSL was downloaded from: ftp://mirrors.usc.edu/pub/gnu/gsl/
9. Example used to test GSL installation: http://www.gnu.org/software/gsl/manual/html_node/Numerical-integration-examples.html
10. Cygwin website setup.exe was downloaded from: http://www.cygwin.com/
11. Mirror Cygwin was downloaded from: ftp://ftp.gtlib.gatech.edu
12. Location of quadrule collection of code: http://people.sc.fsu.edu/~jburkardt/cpp_src/quadrule/quadrule.html.
13. Tutorial on passing pointers to functions: http://www.cplusplus.com/doc/tutorial/pointers/

APPENDIX C. REVIEW OF QUADRULE – A COLLECTION OF CODE USEFUL FOR FUTURE RESEARCH

Collection of code available at http://people.sc.fsu.edu/~jburkardt/cpp_src/quadrule/quadrule.html. The variety of code provides excellent examples. The general organization of the available programs also raises an interesting idea that may cut down on computing time: to have the weights and abscissas pre-computed and read in from a text file.
Some of the programs examined in detail were:

- quad_mpi – a parallelized example of a one-dimensional numerical integration
- quadrature_test – "a program which reads the definition of a multidimensional quadrature rule from three files, applies the rule to a number of test integrals, and prints the results."
- quadrule –"a library which defines quadrature rules for approximating integrals;" i.e., have to use both quadrule and quadrature_test to evaluate an integral, also have to pick a rule
- product_rule – "a C++ program which creates a multidimensional quadrature rule by using a product of one-dimensional quadrature rules."
- quadrature_rules – "a dataset directory which contains examples of quadrature rules."
- sparse_grid_gp – "a dataset directory which contains examples of sparse grids, using the idea of a level to control the number of points, and assigning point locations using the Gauss Patterson rule."

A variety of other quadrature rules are implemented in the same source. Unfortunately, all of the implementations are non-adaptive. So to develop an adaptive solver using one four-dimensional quadrature rule (as in [8]), rather than recursively calling four one-dimensional rules, it would be necessary to combine elements from a variety of the code available. For example, sparse_grid_gp could be used to generate the one-dimensional rule used by product_rule to generate the four-dimensional rule. Then quadrature_test could be used as a basis to develop a solver using the four-dimensional rule. Note that it would have to be made adaptive. To parallelize, quad_mpi would need to be expanded to handle four-dimensional integrands and the four-dimensional adaptive solver.

INTERNAL DISTRIBUTION

1. Abigail Snyder
2–4. Dr. Yu (Cathy) Jiao
5.
6.
7.
8.
9–10.
11. ORNL Office of Technical Information and Classification

EXTERNAL DISTRIBUTION

None permitted until completion and review.