

Evaluating Radial Matrix Elements with Cubic Spline Interpolation

Abraham Flores
Advisor: Dr. Mark Caprio
University of Notre Dame

June 20 2016
Edited: July 20 2016

1 Introduction

This pdf will serve as a guide to the C++ routines and the process we used to evaluate these integrals. The starting point of this project was to use two computational methods to evaluate integrals from zero to infinity. First we would analytically change the bounds of the integral to be zero to one. From there if we could generate the integrand we could then use the GNU Scientific Library (GSL) to generate cubic splines of the integrand. Making the integral into trivial power rule integration.

1.1 Requirements

In order to make use of the routines I have created you must have the *GSL* built and ready to use. If you have windows, use MingGW and built GSL with msys correctly (putting the library into the MinGW Include and LIB folder) than the makefiles should* work. If not see Additional comments for compile instructions.

In order to use the WaveFunction Class files you will need *Eigen*.

Link to DropBox containing files

2 Process

This section will describe the mathematical methods and coding details necessary to have a good understanding of where the core pieces of code comes from.

2.1 Change of Variable

If we did not translate the integral to finite bounds than we would have to truncate our integral and risk losing information. Using the process outlined in *Computational Physics* our change of variable to z becomes

$$z = \frac{r}{r+1} \quad \text{or equivalently} \quad r = \frac{z}{1-z} \quad (1)$$

$$\text{Then } dr = \frac{dz}{(1-z)^2} \quad (2)$$

$$\text{Yielding } \int_0^\infty f(r)dr = \int_0^1 \frac{1}{(1-z)^2} f\left(\frac{z}{1-z}\right)dz \quad (3)$$

Now we have our integrand on numerical boundaries. In our function evaluator the new variable z is passed in, so we must solve for r using (1) from there we easily the integrand, by declaring dr as (2) then coding our function and returning the product of the two. Once we can do this we are ready to generate the splines.

2.2 Cubic Spline Interpolation

The heavy mathematics of Cubic Spline Interpolation is taken care of by the GSL however here is the outline. We have an array of data points x and y . We can break up the data points into groups, say groups of four because we need exactly four to match a cubic function to four data points. That may sound easy but we will need to link the endpoints of each group. Meaning each generated cubic will depend on the last generating a large system of equations. Which can be readily solved as a matrix problem. Here are the Restrictions put in place for Cubic Spline Interpolation by the GSL:

- Take vector or array of data points separate into groups of four
- Approximate each group with a cubic function
- Model given function with piecewise cubic
- Match the first and second derivative at each data point
- first and last endpoint have their second derivative set to zero

2.3 GSL Routines and Types

GSL Interpolation types:

gsl_interp_accel: This function returns a pointer to an accelerator object, which is a kind of iterator for interpolation lookups. It tracks the state of lookups, thus allowing for application of various acceleration strategies.

gsl_interp_cspline: Cubic spline with natural boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The second derivative is chosen to be zero at the first point and last point.

gsl_interp_akima: Non-rounded Akima spline with natural boundary conditions. This method uses the non-rounded corner algorithm of Wodicka

gsl_interp_steffen: Steffen's method guarantees the monotonicity of the interpolating function between the given data points. Therefore, minima and maxima can only occur exactly at the data points, and there can never be spurious oscillations between data points. The interpolated function is piecewise cubic in each interval. The resulting curve and its first derivative are guaranteed to be continuous, but the second derivative may be discontinuous.

gsl_spline: Returns a pointer to a newly allocated interpolation object of type t for n size data-points

GSL Interpolation Functions:

Using the higher interface functions we can just store our splines within the spline data structure.

gsl_spline_init(gsl_spline * spline, const double xa[], const double ya[], size_t size): This function initializes the interpolation object `interp` for the data (x_a, y_a) where x_a and y_a are arrays of size `size`. The interpolation object `The xa data array is always assumed to be strictly ordered, with increasing x values; the behavior for other arrangements is not defined.`

gsl_spline_eval_integ(const gsl_spline * spline, double a, double b, gsl_interp_accel * acc): Evaluates the integral based off of the object `spline`.

These next routines just free the data structures and delete their memory.

`gsl_spline_free (gsl_spline);`

`gsl_interp_accel_free (gsl_interp_accel * acc);`

3 Routines

This section will define the necessary arguments, returns and the use of each routine available.

3.1 spline.h

Namespace: spline

CubicIntegrate(double *x, double *y, int n)

Arguments:

- x: Array of Doubles taken by reference, x-axis of data points
- y: Array of Doubles taken by reference, y-axis of data points
- n: Integer number of data points. This is the number of points in each array

Return: (double) Using the GSL routines it evaluates and returns the given integral

Usage: Makes use of GSL's Cubic Spline Interpolation. Takes two arrays of matching x-y data points and approximates the integral over the x-range

AkimaIntegrate(double *x, double *y, int n)

Arguments:

- x: Array of Doubles taken by reference, x-axis of data points
- y: Array of Doubles taken by reference, y-axis of data points
- n: Integer number of data points. This is the number of points in each array

Return: (double) Using the GSL routines it evaluates and returns the given integral

Usage: Makes use of GSL's akima spline interpolation. Takes two arrays of matching x-y data points and approximates the integral over the x-range

SteffenIntegrate(double *x, double *y, int n)

Arguments:

- x: Array of Doubles taken by reference, x-axis of data points
- y: Array of Doubles taken by reference, y-axis of data points
- n: Integer number of data points. This is the number of points in each array

Return: (double) Using the GSL routines it evaluates and returns the given integral

Usage: Makes use of GSL's Steffen spline interpolation. Takes two arrays of matching x-y data points and approximates the integral over the x-range

3.2 wavefunction_class.h

Class that Generates a Radial WaveFunction Data Structure. It stores the relevant data members and has some useful member functions. It uses an enum class to create relevant types for the each basis.

enum class Basis:

- Harmonic Oscillator in Coordinate Space = Basis::HC
- Harmonic Oscillator in Momentum Space = Basis::HM
- Laguerre in Coordinate Space = Basis::LC
- Laguerre in Momentum = Space Basis::LM

class WaveFunction:

Private Data Members

- int n_ : n quantum number, default = 1
Accessed with WF.n()
- int l_ : l quantum number, default = 0
Accessed with WF.l()
- double b_ : Length parameter, default = 1

Accesed with WF.b()
 Basis basis_ : Basis of the wavefunction, default = Basis::HC
 Accesed with WF.basis()

Constructors:

WaveFunction name = WaveFunction()
 WaveFunction name = WaveFunction(int n, int l, double b, Basis basis)
 WaveFunction name = WaveFunction(int n, int l, Basis basis)
 $*b = \sqrt{\frac{2}{2l+3}}$

Member Functions:

int n(): returns the value of data member n_
 int l(): returns the value of data member l_
 double b(): returns the value of data member b_
 Basis basis(): returns the value of data member basis_

FindRoots()

Uses Gauss Quadrature to determine the roots of orthogonal polynomials. See references for process.

Returns (vector<double>) the roots of the function sorted from min to max in a vector

MakeBounds(WaveFunction wf)

Calls FindRoots to determine the roots of their overlap. Then makes pairs:

[(0- > first), (first- > middle)..., (middle- > last), (last- > ∞)]

Returns (vector<pair<double, double>>) a vector of pairs of doubles that can be used as bounds of integration. This Method does not help CSI, So I do not use it. However other Integration methods could probably make use of this function. double

MatrixElement(int n, WaveFunction wf, int order)

Arguments:

n: Number of data points to be evaluated, (Linear time relationship, 10x the number of points means 10x the time the integral will take)

wf: Other WaveFunction in the matrix element

order: The power of the operator.

Return: (double) $\langle \psi_{nl}(r) | r^{order} | \psi_{n'l'}(r) \rangle$

Usage: Needed for r^2, r, k^2, k and overlap matrix elements

3.3 wavefunction_basis.h

Factorial(int n)

Arguments:

Integer n

Return: (double) n!

Usage: Calculates the factorial of n

LaguerrePolynomial(int n, int k, double x)

DO NOT USE THIS ROUTINE: Not accurate enough

Arguments:

n: Order of Laguerre polynomial

k: Order of the generalized Laguerre polynomial

x: point to evaluate the Laguerre polynomial at

Return: (double) Evaluates $L_n^k(x)$

Usage: Needed for the Coordinate space Radial wavefunctions in the Laguerre basis and Harmonic Oscillator basis

JacobiPolynomial(int n, double a, double b, double x)

Arguments:

- n: Order of Jacobi polynomial
- a: parameter of Jacobi polynomials
- b: parameter of Jacobi polynomials
- x: point to evaluate the Jacobi polynomial at

Return: (double) Evaluates $P_n^{(a,b)}(x)$

Usage: Needed for the Momentum space Radial wavefunctions in the Laguerre basis

The Basis Generators are *Overloaded functions*.**HarmonicCoordinate(double r, WaveFunction wf)****HarmonicCoordinate(double r, int n, int l, double b)**

Arguments:

- r: Point to evaluate the Harmonic Oscillator wavefunction at
- b: Length parameter
- n: n quantum number of the wavefunction
- l: l quantum number of the wavefunction

Return: (double) Evaluates $R_{nl}(r)$

Usage: The original basis for the many body problem, needed for matrix elements in coordinate eigenspace

HarmonicMomentum(double k, WaveFunction wf)**HarmonicMomentum(double k, int n, int l, double b)**

Arguments:

- k: Momentum to evaluate the Harmonic Oscillator wavefunction at
- b: Length parameter
- n: n quantum number of the wavefunction
- l: l quantum number of the wavefunction

Return: (double) Evaluates $\tilde{R}_{nl}(k)$

Usage: The original basis for the many body problem, needed for matrix elements in momentum eigenspace

LaguerreCoordinate(double r, WaveFunction wf)**LaguerreCoordinate(double r, int n, int l, double b)**

Arguments:

- r: Point to evaluate the Radial wavefunction at
- b: Length parameter
- n: n quantum number of the Radial wavefunction
- l: l quantum number of the Radial wavefunction

Return: (double) Evaluates $S_{nl}(r)$

Usage: Needed to evaluate matrix elements

LaguerreMomentum(double k, WaveFunction wf)**LaguerreMomentum(double k, int n, int l, double b)**

Arguments:

- k: Momentum to evaluate the Radial wavefunction at
- b: Length parameter
- n: n quantum number of the radial wavefunction
- l: l quantum number of the Radial wavefunction

Return: (double) Evaluates $\tilde{S}_{nl}(k)$

Usage: Needed to evaluate Matrix elements

4 Testing

The general test is to see the convergence of the integration as a function of the number of data points. In Figures 1. we can see the integral fluctuate roughly between ± 1 of the correct value until the Integral converges to the correct value at $N = 3000$. This gives us a pretty good idea of how many data points we will need to be accurate.

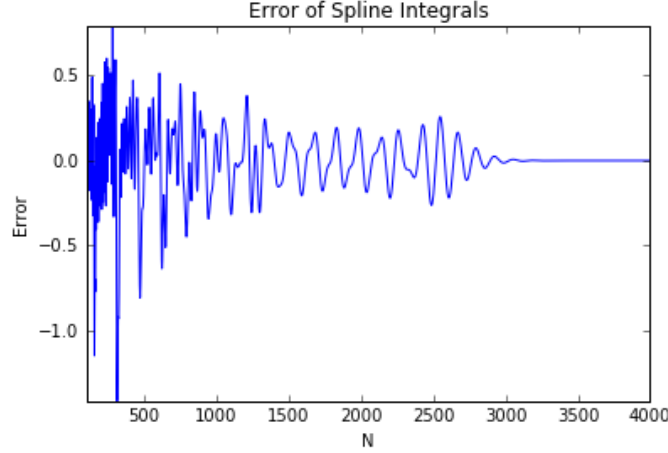


Figure 1: Convergence of $\int_0^\infty S_{50\,32}(r)S_{47\,32}(r) dr$ as a Function of Data Points

4.1 Mathematica Vs. CSI

Mathematica was the standard I held random tests too, tests that I could not analytically solve myself. One of these tests being the evaluation of

$$\int_0^\infty \sin^4(e^{-r}) dr \quad (4)$$

both Mathematica and CSI agreed that the integral was approximately 0.16063.

4.2 Testing each Routine

All polynomials were rigorously tested against Mathematica. It was found that Laguerre_Polynomials broke down for large values of x. This required me to use the GSL routines for Associated Laguerre functions.

The Laguerre and Harmonic Oscillator basis functions were tested against

$$\int_0^\infty R_{n'l}(r)R_{nl}(r) dr = \delta_{n'n} \quad (5)$$

$$\int_0^\infty \tilde{R}_{n'l}(k)\tilde{R}_{nl}(k) dk = \delta_{n'n} \quad (6)$$

$$\int_0^\infty S_{n'l}(r)S_{nl}(r) dr = \delta_{n'n} \quad (7)$$

$$\int_0^\infty \tilde{S}_{n'l}(k)\tilde{S}_{nl}(k) dk = \delta_{n'n} \quad (8)$$

with $n, l < 20$, 1000 data points were sufficient enough to get an accurate result. However when $n, l > 40$, The data points needed to be scaled up to a maximum of 10000 until the change of variable broke down from discontinuities.

4.3 Debugging

If GSL takes an input it does not like it will throw its generic error back, sometimes leaving you baffled. The best process I found was explicitly checking the inputs of each GSL routine. For example making sure the number of points lined up with the size of the array. The only problem I am still not sure why it throws an error is in `gsl_spline_eval_integ()` it will not accept the range 0 to 1 but rather 0 to $x[n-1]$ which means 0 to $1 - \frac{1}{n}$. Other common errors we thrown because of coding the function to evaluate was wrong or had a discontinuity I did not see. Simply use if statements to change the value to zero or evaluate the limit yourself.

5 Limits of CSI and Change of Variable

The GSL's cubic spline integration is great but it will break down eventually. The problems that always popped when CSI or change of variable failed were almost exclusively linked to discontinuities.

5.1 Dividing by Zero

When coding your function within the domain zero to one, you should make sure you are not evaluating $1/0$. This may throw an error in GSL or cause your integral to be slightly off and in most cases will simply return nan as the value of your integral.

5.2 Slow Functions

When a function does not get to zero fast enough it will push all of the evaluated points into a very small range near 1, generating a vertical asymptote.

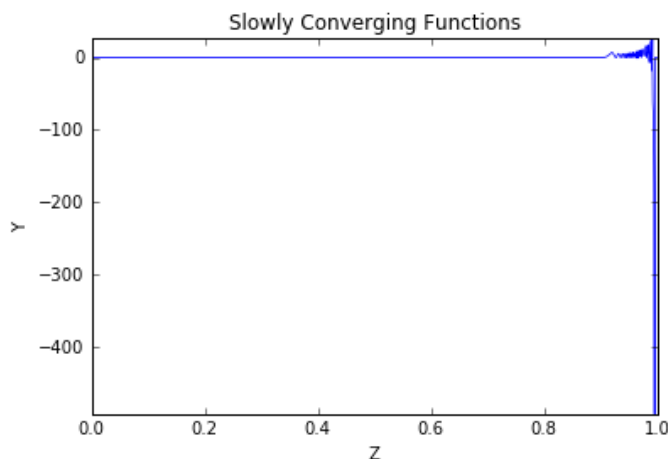


Figure 2: Evaluating $S_{60\,45}(z)S_{59\,45}(z) \frac{1}{(1-z)^2} dz = y$

Change of Variable breaks down as the function forms a vertical asymptote. Eventually the plot will just be zero all the way to one because all the information of the function was lost by changing the domain of the function.

6 Resources

6.1 Change of Variable

Computational Physics

Chaper 5.8: Integrals over infinite ranges

6.2 Spline Interpolation

Spline Interpolation

-General overview of splines

Cubic Splines

-Specific details on solving cubic splines

Numerical Recipes

-Coding Splines

Chapter 3: Splines

Section 3.3: Cubic Splines

GSL MANUAL

Chapter 28: Interpolation

Chapter 40: Splines

Chapter 7.22: Laguerre Functions

6.3 Polynomials

Laguerre

Jacobi

6.4 Laguerre Basis

Coulomb-Sturmian basis for the nuclear many-body problem

-Dr. Caprios paper: Defines R_{nl} , \tilde{R}_{nl} , S_{nl} and \tilde{S}_{nl}

6.5 Gauss Quadrature

Calculating Gauss Quadrature Rules

Section 2: How to find the roots of orthogonal polynomials

6.6 Additional C++ Information

Overloaded functions

7 Additional Comments

7.1 Future Updates

-I am working on recoding the polynomials using backwards recurrence for greatly improved accuracy.

7.2 Compiling

Process is the same as always. Build the object files, then create your executable by linking them.

```
g++ -std=c++11 -c example.cpp -I"GSL Header File Location"-Wall
```

```
g++ -std=c++11 -static example.o linkedfiles.o -L"GSL LIBs Location" -lgsl -lgslcblas -lm -o Name
```

You can always modify the make files yourself, simply modify the directories of Eigen, GSL libs and GSL headers.

7.3 Contact Information

If you have any questions, comments or find any errors please email me at floresab@msu.edu