COMPUTATIONAL METHODS IN MOLECULAR QUANTUM MECHANICS

A document submitted toward the completion of Chemistry 451

presented by

Joshua Cook

California State University, Northridge

Dr. Jussi Eloranta

2015

Summary

Documenting work in the laboratory of Dr. Jussi Eloranta, a professor in the Department of Biochemistry and Chemistry at California State University, Northridge. Dr. Eloranta specializes in the empirical and computational analysis of the quantum behavior of helium that has been cooled very near to absolute zero. This work deals with computational mathematics applied to problems in modern quantum chemistry, including the diagonalization of large matrices.

Table of Contents

1

Introduction

Schrödinger's Equation

A dynamic quantum mechanical system is governed by a linear partial differential equation [@olver2014introduction;@strauss1992partial]. Developed by Erwin Schrödinger and named for him, the equation is written

$$i\hbar\frac{\partial\psi}{\partial t} = H\psi$$

{#eq:time_dependent_schrodinger}

where $i = \sqrt{-1}$, $\hbar$ is Planck's constant, H is the self-adjoint[1], linear operator known as the Hamiltonian, and ψ is a wave function corresponding to a quantum mechanical state of the system.

An eigenequation is a relationship describing a vector or function that is invariant

[1]The adjoint of a linear operator L is the unique linear operator L^* that satisfies $\langle\langle L[u], v\rangle\rangle = \langle u, L^*v\rangle$ An operator is called self-adjoint if $L = L^*$. For a self-adjoint operator, the eigenvalues are real and the eigenvectors for distinct eigenvalues are orthogonal. Even where there is degeneracy in the eigenvalues, it is always possible for form a complete basis set of orthogonal eigenvectors.

with respect to a given linear operator. Given a linear operator $\Gamma : \mathbb{F}^n \to \mathbb{F}^n$, λ and ψ are said to be an eigenvalue and eigenvector respectively of Γ, if

$$\Gamma\psi = \lambda\psi \text{ for } \psi \neq 0$$

{#eq:eigenequation}

Schrödinger's equation can be written independent of time as

$$H\psi = E\psi$$

{#eq:time_independent_schrodinger}

and takes the form of an eigenequation. Then, solutions are sets of eigenfunctions representing quantum states, their corresponding eigenvalues representing energy levels.

In fact, these eigenfunctions are *wave functions*, the most complete description that can be given of a physical system. Wave functions equate to the probability that a given measurement will result from a single measurement of an observable[2].

Most treatments of quantum mechanics delineate a set of postulates [@atkins2011molecular;@Eloran necessary in the formulation of quantum mechanics. For our purposes suffice it to assume that wave functions:

1. are functions
2. are continous and differentiable
3. are finite valued
4. are normal[3]

[2]It is part and parcel of the inherent strangeness of quantum mechanics that we can not describe a quantum system more accurately than by a measure of probability. More precisely, for a wavefunction $\phi(\mathbf{r})$, the probability that $\mathbf{r}$ is returned by a measurement is $p := |\phi(\mathbf{r})|^2$.

[3]Note that this fact combined with the self-adjointness of the operators we are working with combine to give us *orthonormal* wave functions.

Particle in a Box

Central to Schrödinger's formulation is the requirement of boundary conditions. A quantum mechanical problem with no boundary conditions is ill defined[4]. Furthermore, the quantization of the system falls out of the required bound conditions.

The simplest characterization of a quantum system is the so called "Particle in a Box" problem in one dimension. Here, the potential energy within a confined range (say from 0 to a) is set to zero and the potential energy outside of the range, infinite. The Hamiltonian then becomes

$$H = T + V = \begin{cases} -\frac{\hbar^2}{2m}\nabla^2 & : x \in (0, a) \\ \infty & : x \notin (0, a) \end{cases}$$

{#eq:hamiltonian_particle_in_a_box}

Note that this characterization provides the boundary conditions required for quantization, namely that $\psi(0) = \psi(a) = 0$. Intuitively, we can easily reason that if the potential energy at 0 and at a is infinite, then the probability that a particle will be at 0 or at a is nil. We can use these bounds to solve the time-independent Schrödinger equation in one-dimension.

$$-\frac{\hbar^2}{2m}\nabla^2\psi(x) = E\psi(x)$$

{#eq:particle_in_a_box_eigenequation}

$$-\psi''(x) = \frac{2Em}{\hbar^2}\psi(x)$$

{#eq:particle_in_a_box_intermediate}

$$-\psi''(x) = \lambda^2\psi(x) \quad \text{where } \lambda = \frac{\sqrt{2mE}}{\hbar}$$

{#eq:solution_particle_in_a_box}

This equation has solutions of the form $\psi(x) = A\sin(\lambda x)$ where $\lambda = \frac{n\pi}{a}$, $n \in \mathbb{N}$. Recalling that we require normality in our wavefunctions, we can find A.

[4] see the Ultraviolet Catastrophe

$$\left\langle A\sin\left(\frac{n\pi}{a}x\right)\middle| A\sin\left(\frac{n\pi}{a}x\right)\right\rangle = 1 \implies A = \sqrt{\frac{2}{a}}$$

{#eq:adjointness_finds_lambda}

Then, we have the eigenfunctions, $\psi(x) = \sqrt{\frac{2}{a}}\sin\left(\frac{n\pi}{a}x\right)$ with corresponding eigenvalue energy levels $E = \frac{n^2\pi^2\hbar^2}{2ma^2}$. Note that the energy can only take values defined by the eigenproblem, $E_1 = \frac{\pi^2\hbar^2}{2ma^2}$, $E_2 = \frac{4\pi^2\hbar^2}{2ma^2}$, … In other words, the energy is discrete and not continuous. It is quantized. Furthermore, it is of note that there is no "zero". The lowest possible value for the energy, the "ground" state, is $E_1 = \frac{\pi^2\hbar^2}{2ma^2}$.

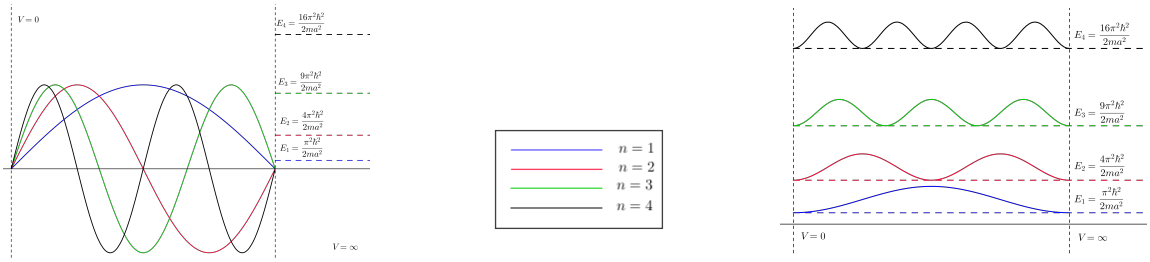

Figure 1.1: Wave Functions & Energies in a 1-D Box

Figure 1.2: Probability Distributions in a 1-D Box

The wave function should not be interpreted as some sort of function of space and time. This is a solution to the time-independent Schrödinger equation and describes a *steady-state*. Recall that the solutions should be used to generate probability distributions. Then when the energy corresponds to E_i, $p := |\psi_i|^2$ describes the probability that the particle will be found at a given location.

Discrete Representations

At the core of this work is the analogy between discrete and continuous mathematics — matrix equations and differential equations. The time-independent Schrödinger equation ([@eq:time_independent_schrodinger]) is essentially the second-order differential equation

$$H\psi = E\psi \implies -\frac{d^2}{dx^2}\psi = \lambda\psi$$

{#eq:second_order_diff_eqn}

generally solved by $y = \cos \omega x$ and $y = \sin \omega x$ with $\lambda = \omega^2$.

Analogously, we consider the matrix equation (and eigenproblem)

$$-D\mathbf{u} = \lambda\mathbf{u}$$

{#eq:second_difference_eigenproblem}

where D represents a difference matrix with a specific set of boundary conditions, $\mathbf{u}$ is an eigenvector, and λ is an eigenvalue.

Representing a Vector in `numpy`

It helps to begin to think of this in terms of how we might represent a function in `numpy`. Here, we represent the linear function, $f(x) = x$ as $\mathbf{u}$ as an evenly spaced vector from 0 to 1 with steps of $h = 0.1$.

```
In [1]: import numpy

In [2]: u = numpy.linspace(0,1,11)

In [3]: u
Out[3]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

Finite Differences

Consider the derivative:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

{#eq:first_derivative}

It is not possible to represent a function $f : \mathbb{R} \to \mathbb{R}$ computationally because computers are discrete in nature and require discrete representation. Nor is it possible to symbolically represent an operation like taking the derivative. Toward a discrete representation, we consider our function as a vector and then look at the finite difference method, representing the derivative as a difference operator.

Toward the difference operator we take

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

{#eq:first_difference}

Recalling that $\mathbf{u}[0] \approx f(0.0)$, $\mathbf{u}[1] \approx f(0.1)$, $\mathbf{u}[2] \approx f(0.2)$…,

$$
\begin{aligned}
\frac{d}{dx}\mathbf{u}[0] &= \frac{1}{0.1}\left(\mathbf{u}[1] - \mathbf{u}[0]\right) = 1 \\
\frac{d}{dx}\mathbf{u}[1] &= \frac{1}{0.1}\left(\mathbf{u}[2] - \mathbf{u}[1]\right) = 1 \\
\frac{d}{dx}\mathbf{u}[2] &= \frac{1}{0.1}\left(\mathbf{u}[3] - \mathbf{u}[2]\right) = 1 \\
&\dots \\
\frac{d}{dx}\mathbf{u}[9] &= \frac{1}{0.01}\left(\mathbf{u}[10] - \mathbf{u}[9]\right) = 1 \\
\frac{d}{dx}\mathbf{u}[10] &= \frac{1}{0.01}\left(\mathbf{u}[11] - \mathbf{u}[10]\right) = 1
\end{aligned}
$$

More generally,

$$
\frac{d}{dx}\mathbf{u}[x] \approx \frac{1}{h}\left(\mathbf{u}[x+h] - \mathbf{u}[x]\right) = \frac{1}{h}\begin{pmatrix}
1 & -1 & 0 & \dots & 0 & 0 & 0 \\
0 & -1 & 1 & \dots & 0 & 0 & 0 \\
\vdots & & & \ddots & & & \vdots \\
0 & 0 & 0 & \dots & 1 & -1 & 0 \\
0 & 0 & 0 & \dots & 0 & 1 & -1
\end{pmatrix}\mathbf{u} = A\mathbf{u}
$$

{#eq:first_difference_equation}

Then the first derivative can be approximated by this matrix

$$
\frac{d}{dx} \approx A = \frac{1}{h}\begin{pmatrix}
1 & -1 & 0 & \dots & 0 & 0 & 0 \\
0 & -1 & 1 & \dots & 0 & 0 & 0 \\
\vdots & & & \ddots & & & \vdots \\
0 & 0 & 0 & \dots & 1 & -1 & 0 \\
0 & 0 & 0 & \dots & 0 & 1 & -1
\end{pmatrix}
$$

{#eq:first_difference_operator}

Toward Second Difference Operator

Consider

$$
\begin{aligned}
f''(x) &\approx \frac{f'(x+h) - f'(x-h)}{2h} \\
&\approx \frac{\frac{f(x+h)-f(x)}{2h} - \frac{f(x)-f(x-h)}{2h}}{2h} \\
&\approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \qquad \text{(with } 2h \text{ recast as } h\text{)}
\end{aligned}
$$

$$
f''(x) \approx \frac{1}{h^2}\left(f(x+h) - 2f(x) + f(x-h)\right)
$$

{#eq:second_difference}

Now consider the following discrete representation

```
                |  2 -1  0  0  0  0  0  0  0  0  0 | |0.00| = |-|
                | -1  2 -1  0  0  0  0  0  0  0  0 | |0.01| = |2|
                |  0 -1  2 -1  0  0  0  0  0  0  0 | |0.04| = |2|
                |  0  0 -1  2 -1  0  0  0  0  0  0 | |0.09| = |2|
                |  0  0  0 -1  2 -1  0  0  0  0  0 | |0.16| = |2|
-1/(.01)        |  0  0  0  0 -1  2 -1  0  0  0  0 | |0.25| = |2|
                |  0  0  0  0  0 -1  2 -1  0  0  0 | |0.36| = |2|
                |  0  0  0  0  0  0 -1  2 -1  0  0 | |0.49| = |2|
                |  0  0  0  0  0  0  0 -1  2 -1  0 | |0.64| = |2|
                |  0  0  0  0  0  0  0  0 -1  2 -1 | |0.81| = |2|
                |  0  0  0  0  0  0  0  0  0 -1  2 | |1.00| = |-|
```

Note that the vector being multiplied by the matrix corresponds to the values of $f(x) = x^2$ at $x = 0.0, 0.1, 0.2, \ldots, 0.8, 0.9, 1.0$. Note that the matrix is being multiplied by $\frac{1}{h^2} = \frac{1}{.01}$ where $h = 0.1$ or the step of our vector representing $f(x)$. Note that the returned value is the constant 2 which corresponds to $f''(x) = 2$.

Computational Science

Strang's *Computational Science and Engineering* [@strang2007computational] devotes the bulk of its first chapter to a discussion of this very matrix

$$D = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 & 0 & 0 \\ -1 & 2 & -1 & \dots & 0 & 0 & 0 \\ 0 & -1 & 2 & \dots & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \dots & 2 & -1 & 0 \\ 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & \dots & 0 & -1 & 2 \end{pmatrix} \approx \frac{d^2}{dx^2}$$

This matrix has several significant properties:

1. It is symmetric.
2. It is sparse.
3. It is tridiagonal.
4. The matrix has constant diagonals.
5. It is invertible.
6. It is positive definite[5].

Numpy and Scipy as Wrapper to BLAS and Lapack

If performance is a priority than computational mathematics must be done using C and its libraries BLAS, Lapack, and Arpack. Prior to understanding the underlying mathematics, we posit that rapid prototyping and an interactive development environment should be prioritized over performance. Toward this end, we offer as an alternative to C and its computational libraries, Python and its computational libraries Numpy and Scipy. Numpy and Scipy are open-source (free in all senses). Numpy and Scipy offer a robust interactive development environment in IPython. Furthermore, we believe that Python syntax is descended from C syntax and note that Numpy and Scipy are high-level wrappers to the same Fortran functions being

[5]This means that the matrix will have n positive (and therefore real) eigenvalues. From this we know that the matrix D is self-adjoint.

used in BLAS and Lapack, and therefore porting a robust and vetted algorithm from Python to C should be straight-forward where non-trivial.

Later in this document we explore the Gnu Scientific Library.

Worth further invesigation are going directly to netlib which is largely maintained in Fortran. Netlib maintains BLAS, LAPACK, PLASMA, and MAGMA.

Implementing a Second Difference Matrix in numpy

We have written the following function based upon *CSE* [@strang2007computational]. It creates four matrices, each corresponding to a different set of boundary conditions: `D`, for Dirichlet[6], `R` for Robin[7], `N` for Neuman[8], or `C` for circular[9].

```
def secondDiff(type,n=10,sparse=False):
    '''
    secondDiff Create finite difference model matrix.
    TYPE is one of the characters 'D', 'R', 'N', or 'C'.
    3rd argument is boolean for sparseness
    '''
    import numpy, scipy.sparse
    e = numpy.ones(n)
    e_off = numpy.ones(n-1)
    D = scipy.sparse.csr_matrix(
      scipy.sparse.diags([e_off,-2*e,e_off],[-1,0,1]))

    if (str(type) == 'R' or str(type) == 'T' or
        str(type) == 'N' or str(type) == 'B'):
        D[0,0] = -1
    if (str(type) == 'N' or str(type) == 'B'):
        D[n-1,n-1] = -1
    if str(type) == 'C':
        D[0,n-1] = 1
        D[n-1,0] = 1

    if sparse == False: return D.todense()
    else: return D
```

[6]Also known as *fixed-fixed conditions* and defined by zeroes at the boundaries of the function.

[7]Also known as *free-fixed conditions* and defined by zero in the first derivative of the function at one end of the function and zero in the value of the function at the other.

[8]Also known as *free-free conditions* and defined by zeroes in the first derivative of the function at the boundaries of the function.

[9]Defined by equality in the value of the function and in the value of its first derivative at the boundaries

Discrete Representation of the Particle in a Box

To conclude this introduction, we note that we can discretely represent the Hamiltonian operator ([@eq:hamiltonian_particle_in_a_box]) describing the "particle in a box" using the matrix `D` defined by this function.

We use the IPython interactive terminal to execute the commands and find its `%paste` magic function very useful.

We first load the function `secondDiff`.

We then load the following python modules:

```
In [2]: import numpy, numpy.linalg, scipy.linalg, matplotlib.pyplot

## -- End pasted test --
```

We define the matrix equation describing the particle in a box as

$$-D\mathbf{u} = \frac{2mE}{\hbar^2}\mathbf{u}$$

{#eq:matrix_eqn_particle_in_a_box}

For D, which describes Dirichlet boundary conditions, we have

- $\theta_k = k\pi/(n+1)$
- eigenvalues, $E_k = 2 - 2\cos\theta_k$
- eigenfunctions, $u_k = (\sin k\pi h, \sin 2k\pi h, \ldots, \sin n\pi h)$

for $k \in [1, n]$.

We define each of these in our IPython session.

```
In [3]: %paste
def K_theta(k,n):
  return k*numpy.pi/(n+1)

def K_eigenvalues(n):
  return 2*numpy.ones(n) - 2*numpy.cos(K_theta(numpy.linspace(n,1,n),n))

def K_eigenfunction(k,n):
  vec = numpy.sin(K_theta(numpy.linspace(1,n,n),n)*k)
  return vec/numpy.linalg.norm(vec)

## -- End pasted text --
```

We next create our second difference matrices for Dirichlet boundary conditions with $n = 2, 3, 4, 5$. We display `K4` for visual inspection.

```
In [4]: %paste
K2 = -secondDiff('D',2)
K3 = -secondDiff('D',3)
K4 = -secondDiff('D',4)
K5 = -secondDiff('D',5)
print K4

## -- End pasted text --
[[ 2. -1. -0. -0.]
 [-1.  2. -1. -0.]
 [-0. -1.  2. -1.]
 [-0. -0. -1.  2.]]
```

Finding Eigenvalues

We then use the built-in eigensolver in `numpy.linalg` to find the eigenvalues and compare it to the values generated by our function. Note that the first element in the array returned by `numpy.linalg.eig` is an array of the eigenvalues.

```
In [5]: %paste
e2 = numpy.linalg.eig(K2)
e3 = numpy.linalg.eig(K3)
e4 = numpy.linalg.eig(K4)
e5 = numpy.linalg.eig(K5)
print e2[0]
print K_eigenvalues(2)
print e3[0]
print K_eigenvalues(3)
print e4[0]
print K_eigenvalues(4)
print e5[0]
print K_eigenvalues(5)

## -- End pasted text --
[ 3.  1.]
[ 3.  1.]
[ 3.41421356  2.          0.58578644]
[ 3.41421356  2.          0.58578644]
[ 3.61803399  2.61803399  0.38196601  1.38196601]
[ 3.61803399  2.61803399  1.38196601  0.38196601]
[ 3.73205081  3.          2.          0.26794919  1.        ]
[ 3.73205081  3.          2.          1.          0.26794919]
```

Finding Eigenvectors

The second element in the array by `numpy.linalg.eig` is a matrix of the eigenfunctions. Please understand that we are liberal with our implicit understanding that eigenfunctions and eigenvectors are, for our purposes, synonymous.

```
In [6]: %paste
print e2[1]
print K_eigenfunction(1,2)
print K_eigenfunction(2,2)

## -- End pasted text --
[[ 0.70710678  0.70710678]
 [-0.70710678  0.70710678]]
[ 0.70710678  0.70710678]
[ 0.70710678 -0.70710678]

In [7]: %paste
print e4[1]
print K_eigenfunction(1,4)
print K_eigenfunction(2,4)
print K_eigenfunction(3,4)
print K_eigenfunction(4,4)

## -- End pasted text --
[[-0.37174803 -0.60150096 -0.37174803 -0.60150096]
 [ 0.60150096  0.37174803 -0.60150096 -0.37174803]
 [-0.60150096  0.37174803 -0.60150096  0.37174803]
 [ 0.37174803 -0.60150096 -0.37174803  0.60150096]]
[ 0.37174803  0.60150096  0.60150096  0.37174803]
[ 0.60150096  0.37174803 -0.37174803 -0.60150096]
[ 0.60150096 -0.37174803 -0.37174803  0.60150096]
[ 0.37174803 -0.60150096  0.60150096 -0.37174803]
```

Comparison of timing

For comparison of timing, we introduce a function that will return the exact same values as the built-in eigensolver.

```
In [8]: %paste
def my_eig(n):
    vals = []
    vals.append(K_eigenvalues(n))
    eigenvectors = numpy.matrix(K_eigenfunction(1,n))
    for i in range(n-1):
        eigenvectors = numpy.r_[eigenvectors,
                                numpy.matrix(K_eigenfunction(i+2,n))]
    vals.append(eigenvectors)
    return vals

## -- End pasted text --
```

We visually inspect the three by three output.

```
In [9]: %paste
print my_eig(3)
print numpy.linalg.eig(K3)

## -- End pasted text --
[array([ 3.41421356,  2.        ,  0.58578644]),
matrix([[  5.00000000e-01,   7.07106781e-01,   5.00000000e-01],
        [  7.07106781e-01,   8.65956056e-17,  -7.07106781e-01],
        [  5.00000000e-01,  -7.07106781e-01,   5.00000000e-01]])]
(array([ 3.41421356,  2.        ,  0.58578644]),
matrix([[ -5.00000000e-01,  -7.07106781e-01,   5.00000000e-01],
        [  7.07106781e-01,   4.05925293e-16,   7.07106781e-01],
        [ -5.00000000e-01,   7.07106781e-01,   5.00000000e-01]]))
```

We use the IPython magic function `%timeit` to run our timing comparisons.

```
In [10]: %paste
%timeit numpy.linalg.eig(K5)
%timeit my_eig(5)

## -- End pasted text --
The slowest run took 6.73 times longer than the fastest.
This could mean that an intermediate result is being cached
10000 loops, best of 3: 31.8 µs per loop
1000 loops, best of 3: 188 µs per loop
```

We note that the built-in eigensolver is nearly six times faster than our algorithm.

```
In [11]: %paste
K10 = secondDiff('D',10)
%timeit numpy.linalg.eig(K10)
%timeit my_eig(10)

## -- End pasted text --
10000 loops, best of 3: 49.8 µs per loop
1000 loops, best of 3: 427 µs per loop
```

We note that the built-in eigensolver is nearly ten times faster than our algorithm.

```
In [12]: %paste
K100 = secondDiff('D',100)
%timeit numpy.linalg.eig(K100)
%timeit my_eig(100)

## -- End pasted text --
100 loops, best of 3: 9.06 ms per loop
100 loops, best of 3: 4.97 ms per loop
```

This last result is astounding. Suddenly our algorithm is almost twice as fast as the built-in solver.

Exploring This Result

We wish to collect data over times to find eigenvalues and eigenvectors for increasing values of n. Note that our method does not actually use the matrix, but rather uses analytical results based on our knowledge of the second difference matrix for Dirichlet boundary conditions. We will need, however, to pass a matrix to the built-in eigensolver. We wish to create this matrix outside of the timer so as not to penalize the eigensolver.

We wrote the following simple IPython script:

```
for i in range(3,300):
    matrix = secondDiff('D',n)
    result_my  = %timeit -o my_eig(n)
    result_sys = %timeit -o numpy.linalg.eig(matrix)
    %store result_my.best, result_sys.best >> output.txt
```

```
import numpy, numpy.linalg, matplotlib.pyplot
data = numpy.genfromtxt('output_1442991327.csv', delimiter=',')
indep = data[:,0]
my_eig_data = data[:,1]
sys_eig_data = data[:,2]
my_eigensolver = matplotlib.pyplot.plot(indep,my_eig_data)
system_eigensolver = matplotlib.pyplot.plot(indep,sys_eig_data)
matplotlib.pyplot.legend(["My Eigensolver", "Numpy's Eigensolver"])
matplotlib.pyplot.show()
```

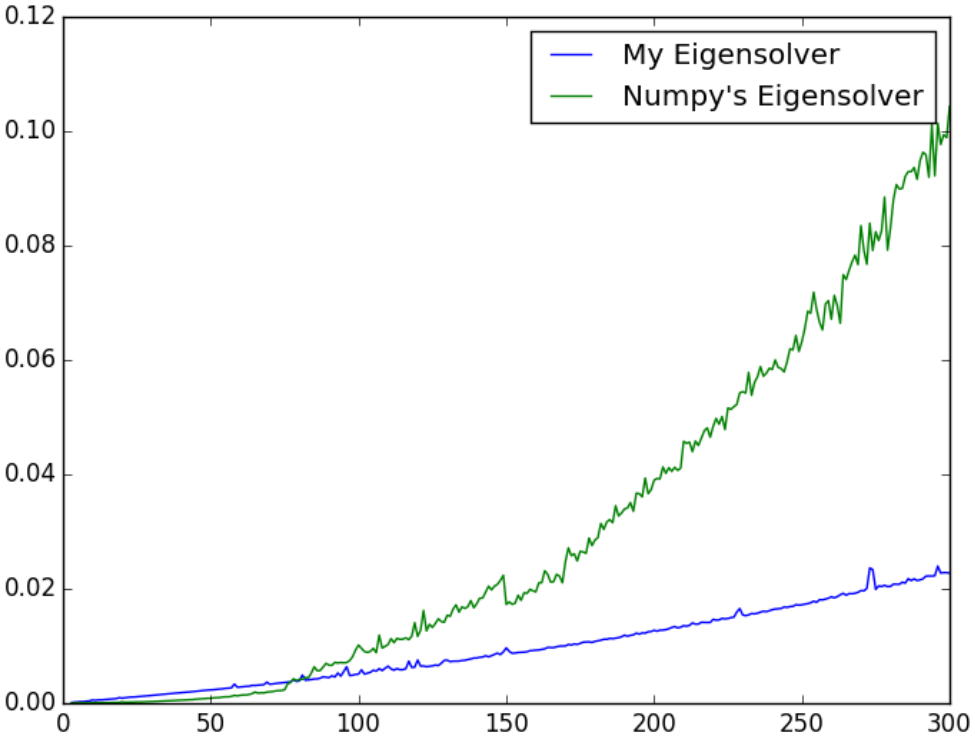

Figure 1.3: Scaling to $n = 300$

Plotting the first few eigenfunctions

```
def for_plot_K_eigenfunction(k,n):
  vec = K_eigenfunction(k,n)
  soln = np.insert(vec,0,0)
  soln = np.insert(np.zeros(1),0,soln)
  return soln

def plot_m_K_eigenfunctions(m,n):
  for i in range(m+1):
  plt.plot(np.linspace(0,1,n+2),for_plot_K_eigenfunction(i,n))

plot_m_K_eigenfunctions(4,10)
plot_m_K_eigenfunctions(4,20)
plot_m_K_eigenfunctions(4,1000)
```

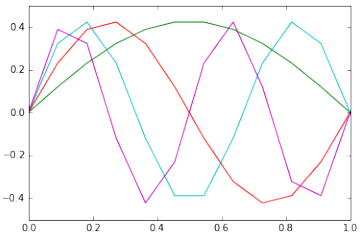

Figure 1.4: First four eigenfunctions, n=10

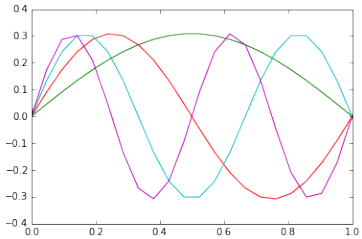

Figure 1.5: First four eigenfunctions, n=20

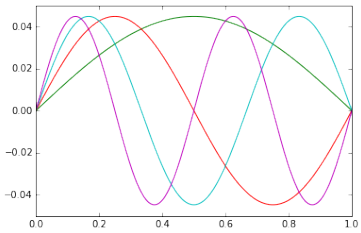

Figure 1.6: First four eigenfunctions, n=10000

2

Power Method

Solving Molecular Quantum Mechanics Problems Computationally

At its essence the work in Dr. Eloranta's lab is an extension of the simple power method for finding an eigenvector. Also known as the Von Mises Iteration, the simple power method operates on a few well conditionings, all of which are satisfied by our Hamiltonian matrices — namely we require self-adjoint matrices, and all that this implies.

The power method will only return the dominant eigenvector of an operator. Essentially, the next vector in the iteration is calculated by multiplying the current vector by the matrix being examined and then normalizing.

Simple Power Method

1. Choose a starting vector $\mathbf{x}^{(0)} \in \mathcal{R}^n$ with $||\mathbf{x}^{(0)}|| = 1$.
2. $k = 0$
3. `while` some convergence criteria is not satisfied
   i. $k := k + 1$
   ii. $\mathbf{y}^{(k)} := A\mathbf{x}^{(k-1)}$
   iii. $\mu_k :=||\mathbf{y}^{(k)}||$
   iv. $\mathbf{x}^{(k)} := \mathbf{y}^{(k)}/\mu_k$

The eigenvalue can be found by calculating

$$\begin{aligned} \lambda u &= Au \\ u^T \lambda u &= u^T Au \\ \lambda &= u^T Au \qquad \text{(because } u \text{ is normalized)} \end{aligned}$$

Here we define an iteration and that iterate 100 times to find our the eigenvector associated with the largest eigenvalue.

```
def power_iteration(A,u,n):
  for i in range(n):
    u      = A.dot(u)
    mu     = numpy.sqrt(u.dot(u))
    u      = u/mu
  eigvec = u
  eigval = eigval = u.dot(A.dot(u))
  return eigval, eigvec

In [1]: import numpy, scipy.linalg, numpy.random

In [2]: B = numpy.random.rand(4,4)

In [3]: C = B.dot(B.T) # returns a symmetric matrix

In [4]: y = numpy.random.rand(4)

In [5]: power_iteration(C,u,100)
Out[5]: (5.4509787314661642,
 array([ 0.57243721,  0.54380344,  0.38428034,  0.47845802]))

In [6]: scipy.linalg.eigh(C,eigvals=(3,3))
Out[6]:
(array([ 5.45097873]),
 array([[-0.57243721],
        [-0.54380344],
        [-0.38428034],
        [-0.47845802]]))
```

Comparison of Timing

```
In [7]: %timeit power_iteration(C,u,100)
```

```
1000 loops, best of 3: 351 µs per loop

In [8]: %timeit scipy.linalg.eigh(C,eigvals=(3,3))
The slowest run took 6.32 times longer than the fastest.
This could mean that an intermediate result is being cached
10000 loops, best of 3: 26 µs per loop
```

It is shown that our power iteration is considerably slower than the built-in eigensolver. We are, however, hard coding the number of iterations required.

Stopping Criteria

It would behoove us to explore a better stopping criteria than simply iterate 100 times. Toward this we propose the use of the norm of the residual vector

$$\mathbf{r} = A\mathbf{u}^* - \lambda^*\mathbf{u}^*$$

We can then stop our calculation when $||\mathbf{r}|| < \epsilon$ for any desired ϵ.

```
def power_iteration(A,u,n,eps=0.00001):
  r_mag = 1
  while(r_mag > eps):
    u      = A.dot(u)
    mu     = numpy.sqrt(u.dot(u))
    u      = u/mu
    eigval = eigval = u.dot(A.dot(u))
    r      = A.dot(u)-eigval*u
    r_mag  = numpy.sqrt(r.dot(r))
  eigvec = u
  return eigval, eigvec

In [9]: %timeit power_iteration(C,u,100)
The slowest run took 6.79 times longer than the fastest.
This could mean that an intermediate result is being cached
10000 loops, best of 3: 42.9 µs per loop
```

While we are not beating the built-in solver, we are certainly within an order of magnitude and are satisfied with these results.

Iterative Techniques

vector norm a function from $\mathbb{R}^n$ to $\mathbb{R}$

l_2 norm is also called the Euclidean norm

$||\mathbf{x}||_2 = (\sum_{i=1}^{n} x_i^2)^{1/2}$

convergence A sequence $\left\{\mathbf{x}^{(k)}\right\}_{k=1}^{\infty}$ of vectors in $\mathbb{R}^n$ is said to converge to $\mathbf{x}$ with respect to the norm $||\cdot|| < \epsilon$, if given any $\epsilon > 0$, there exists an integer $N(\epsilon)$ such that

$$||\mathbf{x}^{(k)} - \mathbf{x}|| < \epsilon, \text{ for all } k \geq N(\epsilon)$$

spectral radius the spectral radius of a matrix A is defined by

$$\rho(A) = \max|\lambda||$$

where λ is an eigenvalue of A.

convergent matrices an n by n matrix is convergent if for all $1 \leq i, j \leq n$

$$\lim_{k \to \infty} (A^k)_{ij} = 0$$

3

Imaginary Time Propagation Method

Applications of Imaginary Time Propagation Method in Material Research

We are seeking approximate solutions to the time-independent Schrödinger equation

$$H\Psi_i = E_i\Psi_i \qquad \text{(eqn. 1)}$$
$$\text{with } \hat{H} = \hat{T} + \hat{V} = -\frac{\hbar^2}{2m}\Delta + V$$

If we can find the eigenfunctions that satisfy these equations, the corresponding eigenvalues can be found by calculating the expectation value

$$\langle\psi_i|H|\psi_i\rangle$$

Our approach is the solve the time-dependent Schrödinger equation

$$i\hbar\frac{\partial}{\partial t}\Psi = \hat{H}\Psi$$

by converting it via a Wick rotation (let $t = -i\tau$) to a simple heat equation:

$$\frac{\partial}{\partial t}\Psi = -\frac{\hat{H}}{\hbar}\Psi \qquad \text{(eqn. 2)}$$

Our solutions are then given by

$$\Psi(r,t) = e^{-\hat{H}\tau/\hbar}\psi(r,0) = \left(e^{-\hat{H}\Delta\tau/\hbar}\right)^n \psi(r,0) \qquad \text{(eqn. 3)}$$

Then $\left(e^{-\hat{H}\Delta\tau/\hbar}\right)^n$ has the same eigenfunctions as $H\Psi_i = E_i\Psi_i$. If we iterate the decay equation it will converge on the ground state.

We can solve eqn. 3 by treating it as an analog of the power method or its generalization the subspace iteration.

As presented at NSF PREM Colloquium.

Helium droplets not only provide a unique matrix environment for high resolution spec- troscopy and studying molecular solvation but also allow to use guest molecules as probes of the surrounding quantum medium.1–3 After the initial discovery of the helium droplet tech- nique for spectroscopic applications, attention quickly turned into characterizing the physical properties of the helium droplets themselves. The groundbreaking experiments by the Toen-nies group employed the glyoxal molecule as a probe to study the helium droplet response through optical absorption spectrum.
the Imaginary Time Propagation

The imaginary time propagation method (ITP) relies on solving the time-dependent Schrödinger equation in imaginary time. We perform a Wick Rotation (setting $t = -i\tau$) to transform the time-dependent Schrödinger into a simple diffusion equation

$$\frac{\partial\psi(r,\tau)}{\partial\tau} = -\frac{\hat{H}}{\hbar}\psi(r,\tau) \implies \psi(r,\tau) = \exp(-\hat{H}\tau/\hbar)\psi(r,0) \qquad (3.1)$$

Iterative Solutions to Eigenproblems

This can be thought of as the analog to a power solution or subspace iteration. As $\tau \to \infty$, $\psi(r, \tau)$ will converge on the eigenfunction for the ground state.

In practice, a random vector is chosen as the initial state. A time-propagation will yield the ground state eigenvector. If a vector other than the ground state is desired, N separate wave functions are propagated. Each higher state eigenvector is required to be orthogonal to the lower eigenvectors and are thus discovered through the iterative process. Approximate orthogonality is enforced in the following way.

$$\frac{\partial \psi_i(r,\tau)}{\partial \tau} = -\frac{\hat{H}}{\hbar}\psi(r,\tau) - \lambda \sum_{j<i}^{N} |\langle \psi_j(r,\tau)|\psi_i(r,\tau)\rangle|^2 \tag{3.2}$$

In order to implement the solution computationally, the exponential operator is approximated using the Cayley unitary form, transforming the eigenvalue problem into a linear problem:

$$\exp(-H\Delta\tau) \approx \left(1 + \frac{1}{2}H\Delta\tau\right)^{-1}\left(1 - \frac{1}{2}H\Delta\tau\right)$$
$$\implies \left(1 + \frac{1}{2}H\Delta\tau\right)\psi(r,\tau+\Delta\tau) = \left(1 - \frac{1}{2}H\Delta\tau\right)\psi(r,\tau)$$

Stopping Criteria

A formula for the absolute error, ΔE_i present in $E_i(\tau)$ can be written in terms of the quantum mechanical standard deviation of H and is used as a stopping criteria.

$$\Delta E_i = |E_i - \langle \psi_i(r,\tau)|H|\psi_i(r,\tau)| \leq \sqrt{2}\sqrt{\langle \psi_i(r,\tau)|H^2|\psi_i(r,\tau)\rangle - \langle \psi_i(r,\tau)|H|\psi_i(r,\tau)\rangle^2} \tag{3.3}$$

Results

The ITP method reduces the solving of an eigenproblem to an iterative power solution via the solution of a linear equation. % ITP is being implemented in practice parallelized, using C and openBLAS. In current practice has been shown to have better scalability than the implicitly restarted Lanczos method as implemented in ARPACK.

For the purposes of training, the algorithm is being reimplemented in Python and Numpy/Scipy. The solution of the linear equation being the most computationally expensive, theoretically involving a matrix inversion at each iteration, seven linear solution schemes were speed-tested versus Numpy's built-in eigensolver: the Numpy solver, the Scipy solver, Scipy's conjugate gradient squared (CGS) solver, a Cholevsky decomposition method, two of Scipy's sparse solvers, and a pre-inversion of the matrix iterated.

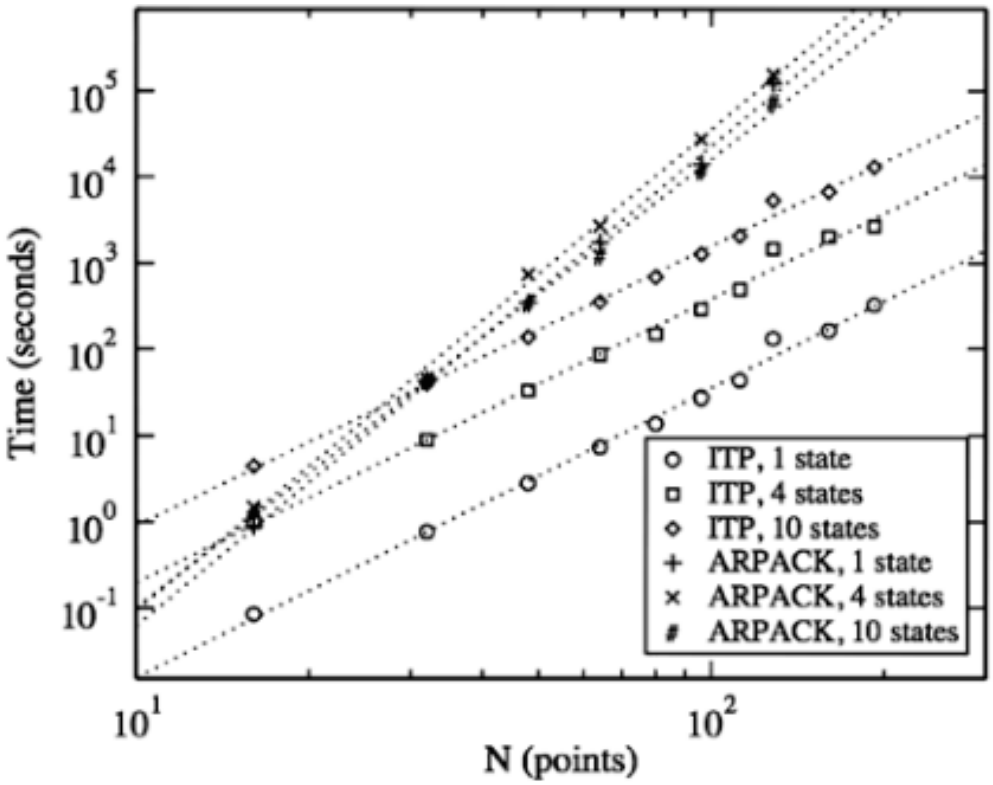

Figure 3.1: Computational scaling of the ITP and implicitly restarted Lanczos methods (ARPACK) for 1, 4 and 10 states.

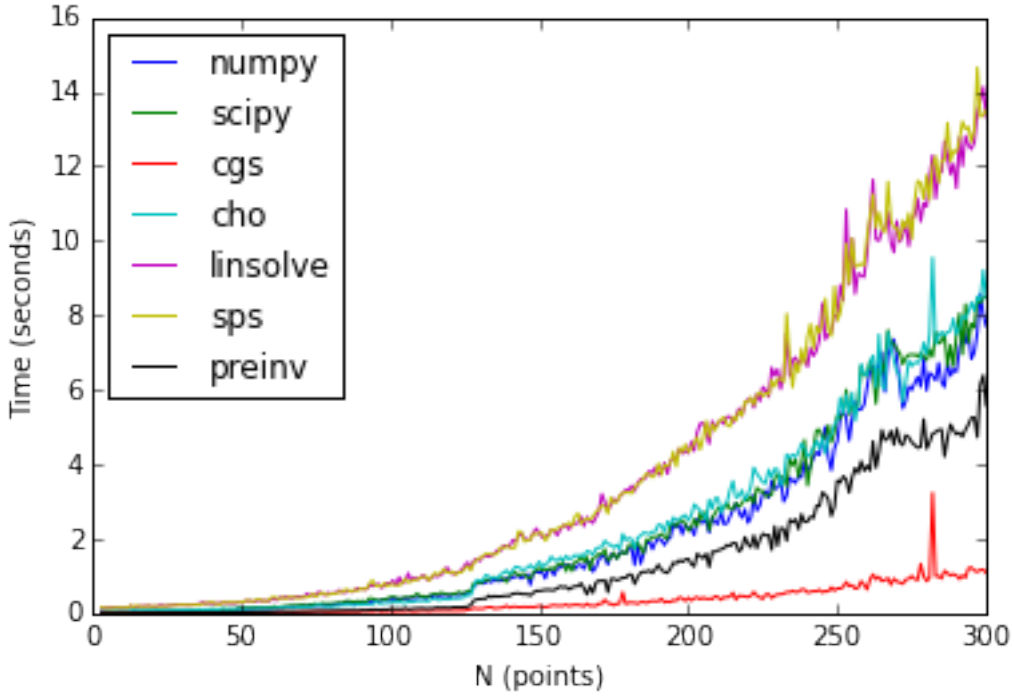

Figure 3.2: Implementation of seven linear solving schemes in Python.

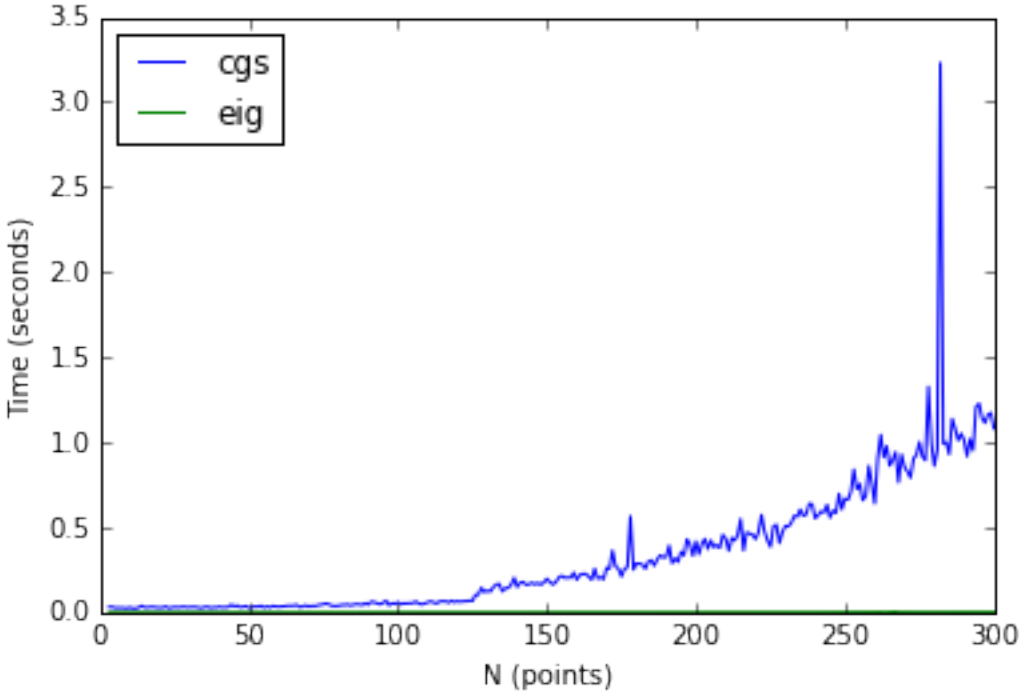

Figure 3.3: Conjugate Gradient Squared Solver ITP v Built-in Eigensolver

The CGS solver was found to be the fastest. In comparison to the built-in eigensolver, however, it falls well short of reasonable performance standards. Furthermore, the CGS method has been found to be the least stable of the methods (note the spike at $N \approx 280$).

Applications to Bosonic Density Functional Theory

Helium clusters were modeled by the Orsay-Trento DFT (OT-DFT) and the interaction with the guest molecule was included through an external potential. To compute the effective moment of inertia of the molecule–helium complex, we include an additional energy term of the form $-\omega L_z$ and compute the "rotating" groundstate energy by minimizing

$$E[\Psi,\omega] = \int \left\{ \frac{\hbar^2}{2m}|\nabla\Psi|^2 + \epsilon_{OT}[\Psi] + V_{X-\mathrm{He}}|\Psi|^2 - \omega\Psi * L_z\Psi \right\} \tag{3.4}$$

The non-linear Schrödinger-type equation arising from the minimization of eq. 4 is solved by means of imaginary time propagation.

Bosonic Density Functional Theory

In the experiment, bosonic density functional theory (DFT) is the method used to obtain calculated rotational constant values. Density functional theory is a technique that plays an important role in determining the key components that can explain why the moment of inertia decreases when rotational superfluidity takes place. The results obtained using DFT are compared with experimental data and Quantum Monte Carlo (QMC) values and there is a similar agreement which is shown by the appearance firs-turn over point.

The first minimum appearing in molecular rotational constants as a function of helium droplet size has been previously associated with the onset of superfluidity in these finite systems. We investigate this relationship by bosonic density functional theory calculations of classical molecular rotors (OCS, N2O, CO and HCN) interacting with the surrounding helium. The calculated rotational constants are in fair agreement with the existing experimental data, demonstrating the applicability of the theoretical model. By inspecting the spatial evolution of the global phase and density, the increase in the rotational constant after the first minimum is shown to correlate with continuous coverage of the molecule by helium and appearance of angular phase coherence rather than completion of the first solvent shell. We assign the observed phenomenon to quantum phase transition between a localized state and

one-dimensional superfluid, which represents the onset of rotational superfluidity in small helium droplets.

A

Appendix

B

Glossary

Adjoint rises in many fields of mathematics. It can be defined in infinite-dimensional complex scalar product spaces (finite-dimensional case).

Boundary Conditions

Classical Mechanics no limitations in the accuracy with which observables may be measured.

Complete Basis Set In a particular space a list of vectors or functions that is linearly independent and spans the space.

Continuous A set of data is said to be continuous if the values belonging to the set can take on ANY value within a finite or infinite interval. Some examples in quantum mechanics are position and momentum.

Difference Matrix

Differentiable

Differential Operator

Discrete A set of data is said to be discrete if the values belonging to the set are distinct and separate (unconnected values). An example in quantum mechanics is spin states.

Eigenequation a relationship describing a vector or function that is invariant with

respect to a given linear operator. Given an operator $\Gamma : \mathbb{F}^n \to \mathbb{F}^n$, where $\mathbb{F}^n$ represents either the Real or Complex field of dimenion n, Γ is said to be in the set of linear operators over the space $\mathbb{F}^n$. Then, λ and ψ are said to be an eigenvalue and eigenvector respectively of the linear operator Γ, if $\lambda \in \mathbb{F}$, $\psi \neq 0$ and $\Gamma\psi = \lambda\psi$ ([@eq:eigenequation])

Eigenfunction

Eigenvector it's a measurable quantity known as observables in quantum mechanics.

Eigenvalue if we are given the following equation: , then the eigenvalue of an operator is .

Energy the sum of potential and kinetic energies where is kinetic energy and is potential energy.

Hamiltonian Operator Classically, the Hamiltonian corresponds to the total energy of a system. The Hamiltonian operator is the corresponding quantum mechanical operator and is equal to the sum of the kinetic and potential energy operators.

$H = T + V = -\frac{\hbar^2}{2m}\nabla^2 + V$ ([@eq:definition_hamiltonian])

The expectation value of the Hamiltonian operator is an associated energy, E that is totally conserved.

Invertible

Kinetic Energy

Linear Combination Given a complete set of basis functions, it is possible to write a wavefunction describing the state of a system as a linear combination of this basis set. In other words, $\psi(\mathbf{r}) = \sum_{i=1}^{\infty} c_i\phi_i\$$ ([@eq:linear_combination]), where $\Gamma\phi_i = a_i\phi_i$.

Linear Operator

Observable any dynamical variable that can be measured.

Planck's Constant

Positive Definite

Potential Energy

Probability

Quantum Mechanics

Schrödinger's Equation Time-Dependent is

Time-Independent $H\psi = E\psi$ ([@eq:time_independent_schrodinger])

Self-Adjoint Linear Operator The adjoint of a linear operator L is the unique linear operator L^* that satisfies
$\langle\langle L[u], v\rangle\rangle = \langle u, L^* v\rangle$ ([@eq:adjoint])
An operator is called self-adjoint if $L = L^*$. For a self-adjoint operator, the eigenvalues are real and the eigenvectors are orthogonal.

Sparse

Spectral Theory

State

Steady-State

Symmetric Matrix

System

Tridiagonal

Vector

Wave Function

References

Software Packages

C

GSL

Central to this work is the GNU Scientific Library which offers implementations of `openblas` (Open Basic Linear Algebra System) and `lapack` (Linear Algebra Package) necessary for the completion of this work. We have used `homebrew` to install `gsl` and `pkg-config`

```
$ brew install pkg-config
$ brew install gsl
$ pkg-config --libs gsl
-L/usr/local/Cellar/gsl/1.16/lib -lgsl -lgslcblas -lm
```

and then added the linker statements to our makefile:

```
#Tool Definitions
CC=gcc
CFLAGS=-I. -I$(PATHU) -DTEST
CFLAGS+=-I/usr/local/opt/openblas/include
CFLAGS+=-lgsl -lgslcblas -lm
```

Known Eigenvalues

We have written a short python script in order to calculate a few known eigenvalues, in this case for the 4th order Hibert Matrix.

```
import numpy
```

```
import scipy
from scipy.linalg import *

h4 = hilbert(4)
eigs_h4 = eig(h4)
print eigs_h4
```

Test-Driven Development

We begin the project with a simple tests to verify the known eigenvalues we have previously generated.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>
#include "minunit.h"

#define DIMENSION_OF(a) (sizeof(a)/sizeof(a[0]))

gsl_vector * symmetric_eigenvalues(double * data, int return_values);

float round_to_5_places(float num);

int tests_run = 0;

double expected_eigenvalues[] = {1.500214,0.169141,0.006738,0.000097};

double fourth_order_hilbert[] =
{
  1.0  , 1/2.0, 1/3.0, 1/4.0,
  1/2.0, 1/3.0, 1/4.0, 1/5.0,
  1/3.0, 1/4.0, 1/5.0, 1/6.0,
  1/4.0, 1/5.0, 1/6.0, 1/7.0
};
```

```
static char * test_find_eigenvalues()
{
 gsl_vector * actual_eigenvalues = symmetric_eigenvalues(fourth_order_hilbert,0);

  for (int i = 0; i < DIMENSION_OF(expected_eigenvalues); i++)
  {
     float expected = expected_eigenvalues[i];
   float actual = round_to_5_places(gsl_vector_get(actual_eigenvalues, i));

     printf("\nexpected: %f actual: %f\n",
       expected,actual);
     mu_assert("error, eigenvalues do not match",
         expected == actual);
  }
  return 0;
}

static char * all_tests()
{
  mu_run_test(test_find_eigenvalues);
  return 0;
}

int main(int argc, char **argv)
{
  char * result = all_tests();
  if (result != 0)
  {
    printf("%s\n", result);
  }
  else
  {
    printf("All Tests Passed\n");
```

```
  }
  printf("Tests run: %d\n", tests_run);

  return result != 0;
}

gsl_vector * symmetric_eigenvalues(double * data, int return_values)
{
  gsl_matrix_view my_matrix = gsl_matrix_view_array (data, 4, 4);

  gsl_vector * my_evals = gsl_vector_alloc (4);
  gsl_matrix * my_evecs = gsl_matrix_alloc (4, 4);

 gsl_eigen_symmv_workspace * my_workspace = gsl_eigen_symmv_alloc (4);

 gsl_eigen_symmv (&my_matrix.matrix, my_evals, my_evecs, my_workspace);

  gsl_eigen_symmv_free (my_workspace);

  gsl_eigen_symmv_sort (my_evals, my_evecs, GSL_EIGEN_SORT_ABS_DESC);

  return my_evals;
}

float round_to_5_places(float num)
{
  float nearest = roundf(num * 1000000) / 1000000;
  return nearest;
}
```

Comments and Further Consideration

I am not pleased with this particular test of the Symmetic Eigensolver implemented in GSL. It is very "numerical". I think a better test would be more mathematical, such as a test of any matrix and the eigenequation itself, $Ax = \lambda x$. I did, however,

confirm that MinUnit is an excellent way to begin to implement a TDD mindset in C development. I think that the next step is to begin to write more tests around the openBlas implementation in the GSL.

MinUnit

As per MinUnit's description, "MinUnit is an extremely simple unit testing framework written in C. It uses no memory allocation, so it should work fine under almost any circumstance, including ROMable code."

```
/* file: minunit.h */
 #define mu_assert(message, test) do { if (!(test)) return message; } while (0)
 #define mu_run_test(test) do { char *message = test(); tests_run++; \
                                if (message) return message; } while (0)
 extern int tests_run;
```

As per MinUnit's description: "No, that's not a typo. It's just 3 lines of code."

MinUnit has also been added to `tools`.

I wanted to be able to add more information to tests being run using the `minunit.h` testing harness. In particular, I wanted to think of a test run as a "test" and an individual assert within that test run as a subtest. I wanted to be able to name each of these and have their status displayed in `STDOUT`.

My MinUnit Implementation

```
/* file: my_minunit.h */
#define mu_assert(subtest_desc, test,message) do { \
   if (!(test)) { \
   printf("subtest: \"%s\" FAILED\n", subtest_desc); \
   return message; \
   } \
   else { \
     printf("subtest: \"%s\" PASSED\n", subtest_desc); \
   } \
} while (0)
```

```
#define mu_run_test(test_desc,test) do { \
  printf("\nTest: \"%s\"\n",test_desc); \
  char *message = test(); tests_run++; \
  if (message) return message; } while (0)

extern int tests_run;
```

A little less minimal but still small. I also added this file to `/usr/include` so as to not have to tell `gcc` where to find it.

A typical output

```
Test: "Test of GSL"
y: -0.177597     expected_y: -0.177597
subtest: "value of zero order Bessel function of the first kind" PASSED

Test: "Test of Rectangular Complex Number Struct"
subtest: "real part of a rectangular complex number" PASSED
subtest: "imaginary part of a rectangular complex number" PASSED
subtest: "real part of a rectangular complex number after redefinition" PASSED
subtest: "imaginary part of a rectangular complex number after redefinition" PASSED

All Tests Passed
Tests run: 2
```

Spec file for this output

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
#include <gsl/gsl_complex_math.h>
#include <math.h>
#include <my_minunit.h>

int tests_run = 0;
float round_to_6_places(float num);
```

```
static
char * test_gsl_via_0_order_bessel_function_of_the_first_kind ()
{
  double x = 5.0;
  double y = round_to_6_places(gsl_sf_bessel_J0 (x));
  double expected_y = round_to_6_places(-1.775967713143382920e-01);
  printf("y: %f \t expected_y: %f\n",y,expected_y);
  mu_assert
  (
    "value of zero order Bessel function of the first kind",
    y == expected_y,
    "y: not equal to expected_y"
  );
  return 0;
}

static
char * test_gsl_rectangular_complex_number_struct()
{
  double x = 2.43728;
  double y = 3.23412;

  gsl_complex test_rect_complex_number = gsl_complex_rect ( x, y );

  mu_assert
  (
    "real part of a rectangular complex number",
    GSL_REAL(test_rect_complex_number) == x,
    "real part of rectangular complex number does not match expected"
  );

  mu_assert
  (
```

```
    "imaginary part of a rectangular complex number",
    GSL_IMAG(test_rect_complex_number) == y,
   "imaginary part of rectangular complex number does not match expected"
  );

  GSL_SET_REAL(&test_rect_complex_number,y);
  GSL_SET_IMAG(&test_rect_complex_number,x);

  mu_assert
  (
    "real part of a rectangular complex number after redefinition",
    GSL_REAL(test_rect_complex_number) == y,
   "redefined real part of rectangular complex number does not match expected"
  );

  mu_assert
  (
   "imaginary part of a rectangular complex number after redefinition",
    GSL_IMAG(test_rect_complex_number) == x,
   "redefined imaginary part of rectangular complex number does not match expected"
  );

  return 0;
}

static
char * all_tests ()
{
 mu_run_test("Test of GSL", test_gsl_via_0_order_bessel_function_of_the_first_kind
  mu_run_test("Test of Rectangular Complex Number Struct",
              test_gsl_rectangular_complex_number_struct);
  return 0;
}
```

```
int
main(int argc, char **argv)
{
  char * result = all_tests();
  if (result != 0)
  {
    printf("%s\n", result);
  }
  else
  {
    printf("\nAll Tests Passed\n");
  }
  printf("Tests run: %d\n", tests_run);

  return result != 0;
}

float
round_to_6_places(float num)
{
  float nearest = roundf(num * 10000000) / 10000000;
  return nearest;
}
```

Python

While Python comes preinstalled with Mac OS X, installing through homebrew ensures that you have the most updated version as well as moves the management of Python to homebrew.

```
brew install python
```

Python comes installed by default, but best to get the latest and greatest.

The above installs `pip`.

Install Basic Python Packages

Most of the pip installs require that they be run as `sudo`. `brew` is not a fan of `sudo` and should not have the same requirements.

```
pip install numpy
pip install scipy
pip install matplotlib
```

Python as a Tool for Developing C Algorithms

emulating minunit in python [[Needs Work]]

IPython

```
pip install ipython
```

This did not install all of the dependencies required to run IPython notebook. It was fairly straightforward to identify the missing dependencies, however, by attempting to run `IPython Notebook`.

```
ipython notebook
```

The following three dependencies were succesively required:

```
pip install pyzmqi
pip install jinjaz
pip install tornado
```

Numpy

Scipy

Unix-like

Pandoc

If you need to convert files from one markup format into another, pandoc is your swiss-army knife. Pandoc can convert documents in markdown, reStructuredText, textile,

HTML, DocBook, LaTeX, MediaWiki markup, TWiki markup, OPML, Emacs Org-Mode, Txt2Tags, Microsoft Word docx, EPUB, or Haddock markup to

- HTML formats: XHTML, HTML5, and HTML slide shows using Slidy, reveal.js, Slideous, S5, or DZSlides.
- Word processor formats: Microsoft Word docx, OpenOffice/LibreOffice ODT, OpenDocument XML
- Ebooks: EPUB version 2 or 3, FictionBook2
- Documentation formats: DocBook, GNU TexInfo, Groff man pages, Haddock markup
- Page layout formats: InDesign ICML
- Outline formats: OPML
- TeX formats: LaTeX, ConTeXt, LaTeX Beamer slides
- PDF via LaTeX
- Lightweight markup formats: Markdown, reStructuredText, AsciiDoc, MediaWiki markup, DokuWiki markup, Emacs Org-Mode, Textile
- Custom formats: custom writers can be written in lua.

Pandoc was (and is) used to render "final" pdfs of the work done. Pandoc files were written in a hybrid of markdown (for document formatting) and Latex (for math rendering) that mirror the presentation format of IPython Notebooks.

Install Pandoc

```
brew install pandoc
```

You will need a reasonably robust LaTeXinstallation.