

**COMPUTATIONAL METHODS IN  
MOLECULAR QUANTUM MECHANICS**

A document submitted toward the completion of CHEMISTRY 451

presented by  
**Joshua Cook**  
**Osma Gomez**

*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.



*Contents*

*Summary*      3

*Introduction*      7

*Power Method*      19

*Imaginary Time Propagation Method*      23

*Appendix*      29

*Glossary*      31

*Software Packages*      33



## Introduction

A DYNAMIC QUANTUM MECHANICAL SYSTEM is governed by a linear partial differential equation. Developed by Erwin Schrödinger and named for him, the equation is written

$$i\hbar \frac{\partial \psi}{\partial t} = H\psi \quad (1)$$

where  $i = \sqrt{-1}$ ,  $\hbar$  is Planck's constant,  $H$  is the self-adjoint<sup>1</sup>, linear operator known as the Hamiltonian, and  $\psi$  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 with respect to a given linear operator. Given a linear operator  $\Gamma : \mathbb{F}^n \rightarrow \mathbb{F}^n$ ,  $\lambda$  and  $\psi$  are said to be an eigenvalue and eigenvector respectively of  $\Gamma$ , if

$$\Gamma\psi = \lambda\psi \text{ for } \psi \neq 0 \quad (2)$$

Schrödinger's equation can be written independent of time as

$$H\psi = E\psi \quad (3)$$

and takes the form of an eigenequation. Then, solutions are sets of eigenfunctions representing quantum states ~~and~~ 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<sup>2</sup>.

~~Later sections will clearly~~ delineate a set of postulates necessary in the formulation of quantum mechanics. For the time being suffice it to assume that wave functions:

1. are functions
2. are continuous and differentiable
3. are finite valued
4. are normal<sup>3</sup>

@Strauss  
@Oliver

<sup>1</sup> The adjoint of a linear operator  $L$  is the unique linear operator  $L^*$  that satisfies  $\langle L[u], v \rangle = \langle u, L^*[v] \rangle$ . 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 to form a complete basis set of orthogonal eigenvectors.

@Axler

<sup>2</sup> 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  $p := |\phi(\mathbf{r})|^2$ .

<sup>3</sup> 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<sup>4</sup>. 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

$$H = T + V = \begin{cases} -\frac{\hbar^2}{2m} \nabla^2 & : x \in (0, a) \\ \infty & : x \notin (0, a) \end{cases} \quad (4)$$

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.

$$-\frac{\hbar^2}{2m} \nabla^2 \psi(x) = E \psi(x) \quad (5)$$

$$-\psi''(x) = \frac{2Em}{\hbar^2} \psi(x) \quad (6)$$

$$-\psi''(x) = \lambda^2 \psi(x) \quad \text{where } \lambda = \frac{\sqrt{2mE}}{\hbar} \quad (7)$$

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  $A$ .

$$\left\langle A \sin\left(\frac{n\pi}{a}x\right) \middle| A \sin\left(\frac{n\pi}{a}x\right) \right\rangle = 1 \implies A = \sqrt{\frac{2}{a}} \quad (8) \quad \text{Full Solution in Appendix?}$$

Then, we have the eigenfunctions,  $\psi(x) = \sqrt{\frac{2}{a}} \sin\left(\frac{n\pi}{a}x\right)$  with corresponding eigenvalue energy levels<sup>5</sup>  $E = \frac{n^2 \pi^2 \hbar^2}{2ma^2}$ .

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.

@Atkins, Molecular Quantum Mechanics on Traveling Particles

<sup>4</sup> see the **Ultraviolet Catastrophe**  
Link, Wikipedia, say something about this

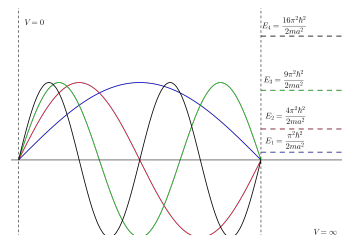


Figure 1: Wave Functions & Energies in a 1-D Box

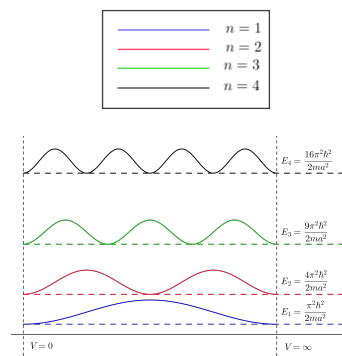


Figure 2: Probability Distributions in a 1-D Box

<sup>5</sup> Note that the energy can only take values defined by the eigenproblem,  $E_1 = \frac{\pi^2 \hbar^2}{2ma^2}$ ,  $E_2 = \frac{4\pi^2 \hbar^2}{2ma^2}$ , ... i.e. the energy is quantized.



## 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. 3) is essentially the second-order differential equation

$$H\psi = E\psi \implies -\frac{d^2}{dx^2}\psi = \lambda\psi \quad (9)$$

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

ANALOGOUSLY, WE CONSIDER THE MATRIX EQUATION (and eigenproblem)

$$-D\mathbf{u} = \lambda\mathbf{u} \quad (10)$$

where  $D$  represents a difference matrix with a specific set of boundary conditions,  $\mathbf{u}$  is an eigenvector, and  $\lambda$  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  $\mathbf{u}$  as an evenly spaced vector from 0 to 1.

what we are doing here is defining  $u$ , a linear function,  $f(x) = x$  as an evenly spaced vector

```
In [1]: import numpy
```

typically `linspace` is not used for this purpose but rather to describe an independent vector or step

```
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:

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

It is not possible to represent a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  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

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

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

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

More generally,

Missing a step

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

Then the first derivative can be approximated by this matrix

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



*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 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 investigation 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* (Strang 2007). It creates four matrices, each corresponding to a different set of boundary conditions: D, for Dirichlet<sup>7</sup>, R for Robin<sup>8</sup>, N for Neuman<sup>9</sup>, or C for circular<sup>10</sup>.

```
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
```

<sup>7</sup> Also known as *fixed-fixed conditions* and defined by zeroes at the boundaries of the function.

<sup>8</sup> 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.

<sup>9</sup> Also known as *free-free conditions* and defined by zeroes in the first derivative of the function at the boundaries of the function.

<sup>10</sup> 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. 4) describing the “particle in a box” using the matrix  $D$  defined by this function.

We load the following python modules:

```
from math import pi
import numpy as np
import numpy.linalg as la
import scipy.linalg as sla
import matplotlib.pyplot as plt
```

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

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

For  $D$ , which describes Dirichlet boundary conditions, we have<sup>11</sup>

- $\theta_k = k\pi/(n+1)$
  - eigenvalues,  $E_k = 2 - 2\cos\theta_k$
  - eigenfunctions,  $u_k = (\sin k\pi h, \sin 2k\pi h, \dots, \sin n\pi h)$
- for  $k \in [1, n]$ .

```
def K_theta(k,n):
    return k*pi/(n+1)

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

def K_eigenfunction(k,n):
    vec = np.sin(K_theta(np.linspace(1,n,n),n)*k)
    return vec/la.norm(vec)
```

Create the second difference matrix for Dirichlet boundary conditions with  $n = 2, 3, 4, 5$ .

```
K2 = secondDiff('D',2)
K3 = secondDiff('D',3)
K4 = secondDiff('D',4)
K5 = secondDiff('D',5)
print K4
```

```
[[ 2. -1.  0.  0.]
 [-1.  2. -1.  0.]
 [ 0. -1.  2. -1.]
 [ 0.  0. -1.  2.]]
```

<sup>11</sup> **THIS IS THE REAL DISCUSSION. HOW DID WE GET THIS!?**

Use the built-in eigensolver in `numpy.linalg` to find the eigenvalues.

```
e2 = la.eig(K2)
e3 = la.eig(K3)
e4 = la.eig(K4)
e5 = la.eig(K5)
```

The first element in the returned array is a vector of the eigenvalues.

```
e2[0], K_eigenvalues(2)
(array([ 3.,  1.]),
 array([ 3.,  1.]))

e3[0], K_eigenvalues(3)
(array([ 3.4142,  2.,  0.5858]),
 array([ 3.4142,  2.,  0.5858]))

e4[0], K_eigenvalues(4)
(array([ 3.618,  2.618,  0.382,  1.382]),
 array([ 3.618,  2.618,  1.382,  0.382]))

e5[0], K_eigenvalues(5)
(array([ 3.7321,  3.,  2.,  0.2679,  1. ]),
 array([ 3.7321,  3.,  2.,  1.,  0.2679]))
```

The second element in the returned array is a matrix of the eigenvectors.

```
e2[1],K_eigenfunction(1,2),K_eigenfunction(2,2)
(matrix([[ 0.7071,  0.7071],
         [-0.7071,  0.7071]]),
 array([ 0.7071,  0.7071]),
 array([ 0.7071, -0.7071]))

e3[1],K_eigenfunction(1,3),K_eigenfunction(2,3),K_eigenfunction(3,3)
(matrix([[ -5.0000e-01,  -7.0711e-01,  5.0000e-01],
         [ 7.0711e-01,  4.0593e-16,  7.0711e-01],
         [ -5.0000e-01,  7.0711e-01,  5.0000e-01]]),
 array([ 0.5,  0.7071,  0.5 ]),
 array([ 7.0711e-01,  8.6596e-17, -7.0711e-01]),
 array([ 0.5, -0.7071,  0.5 ]))

print e4[1]
print K_eigenfunction(1,4)
```

```

print K_eigenfunction(2,4)
print K_eigenfunction(3,4)
print K_eigenfunction(4,4)

[[-0.3717 -0.6015 -0.3717 -0.6015]
 [ 0.6015  0.3717 -0.6015 -0.3717]
 [-0.6015  0.3717 -0.6015  0.3717]
 [ 0.3717 -0.6015 -0.3717  0.6015]]
[ 0.3717  0.6015  0.6015  0.3717]
[ 0.6015  0.3717 -0.3717 -0.6015]
[ 0.6015 -0.3717 -0.3717  0.6015]
[ 0.3717 -0.6015  0.6015 -0.3717]

print e5[1]
K_eigenfunction(1,5), K_eigenfunction(2,5),
    K_eigenfunction(3,5), K_eigenfunction(4,5), K_eigenfunction(5,5)

[[ 2.8868e-01  5.0000e-01  5.7735e-01  2.8868e-01 -5.0000e-01]
 [-5.0000e-01 -5.0000e-01 -1.2201e-16  5.0000e-01 -5.0000e-01]
 [ 5.7735e-01  6.3900e-16 -5.7735e-01  5.7735e-01 -3.6852e-16]
 [-5.0000e-01  5.0000e-01  2.0903e-16  5.0000e-01  5.0000e-01]
 [ 2.8868e-01 -5.0000e-01  5.7735e-01  2.8868e-01  5.0000e-01]]

(array([ 0.2887,  0.5    ,  0.5774,  0.5    ,  0.2887]),
 array([ 5.0000e-01,  5.0000e-01,  7.0705e-17, -5.0000e-01, -5.0000e-01]),
 array([ 5.7735e-01,  7.0705e-17, -5.7735e-01, -1.4141e-16,  5.7735e-01]),
 array([ 5.0000e-01, -5.0000e-01, -1.4141e-16,  5.0000e-01, -5.0000e-01]),
 array([ 0.2887, -0.5    ,  0.5774, -0.5    ,  0.2887]))

```

### *Comparison of timing*

For comparison of timing, I wrote a function that returns the exact same values as the built-in eigensolver.

```

def my_eig(n):
    vals = [], []
    vals[0] = K_eigenvalues(n)
    for i in range(n):
        vals[1].append(K_eigenfunction(i+1,n))
    return vals

%timeit la.eig(secondDiff('D',5))
1000 loops, best of 3: 249 µs per loop

%timeit my_eig(5)

```



10000 loops, best of 3: 131  $\mu$ s per loop

```
%timeit la.eig(secondDiff('D',10))
```

1000 loops, best of 3: 268  $\mu$ s per loop

```
%timeit my_eig(10)
```

10000 loops, best of 3: 239  $\mu$ s per loop

```
%timeit la.eig(secondDiff('D',100))
```

100 loops, best of 3: 10.3 ms per loop

```
%timeit my_eig(100)
```

100 loops, best of 3: 2.08 ms per loop

```
%timeit la.eig(secondDiff('D',1000))
```

1 loops, best of 3: 1.4 s per loop

```
%timeit my_eig(1000)
```

10 loops, best of 3: 53.2 ms per loop

```
%timeit la.eig(secondDiff('D',10000))
```

Hung the system, I suspect in the creation of the matrix.

```
%timeit my_eig(10000)
```

1 loops, best of 3: 3.56 s per loop

*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)
```

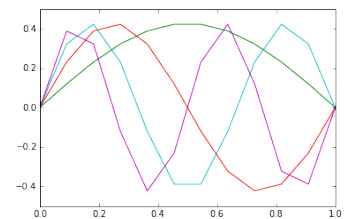


Figure 3: First four eigenfunctions,  $n=10$

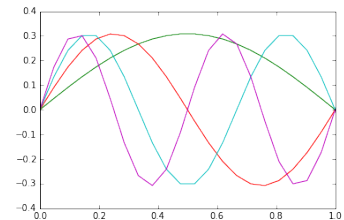


Figure 4: First four eigenfunctions,  $n=20$

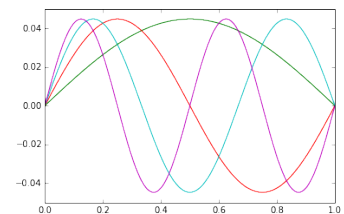


Figure 5: First four eigenfunctions,  $n=1000$



## *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

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

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  $\epsilon$ .

```
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  $\mu$ s 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.



## *Imaginary Time Propagation Method*

### *Applications of Imaginary Time Propagation Method in Material Research*

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

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

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 to 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:

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

Our solutions are then given by

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

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 spectroscopy and studying molecular solvation but

also allow to use guest molecules as probes of the surrounding quantum medium.<sup>1–3</sup> After the initial discovery of the helium droplet technique for spectroscopic applications, attention quickly turned into characterizing the physical properties of the helium droplets themselves. The groundbreaking experiments by the Toennies 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

$$\frac{\partial \psi(r, \tau)}{\partial \tau} = -\frac{\hat{H}}{\hbar} \psi(r, \tau) \implies \psi(r, \tau) = \exp(-\hat{H}\tau/\hbar) \psi(r, 0) \quad (11)$$

*Iterative Solutions to Eigenproblems* This can be thought of as the analog to a power solution or subspace iteration. As  $\tau \rightarrow \infty$ ,  $\psi(r, \tau)$  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.

$$\frac{\partial \psi_i(r, \tau)}{\partial \tau} = -\frac{\hat{H}}{\hbar} \psi_i(r, \tau) - \lambda \sum_{j < i}^N |\langle \psi_j(r, \tau) | \psi_i(r, \tau) \rangle|^2 \quad (12)$$

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:

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

*Stopping Criteria* A formula for the absolute error,  $\Delta E_i$  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.

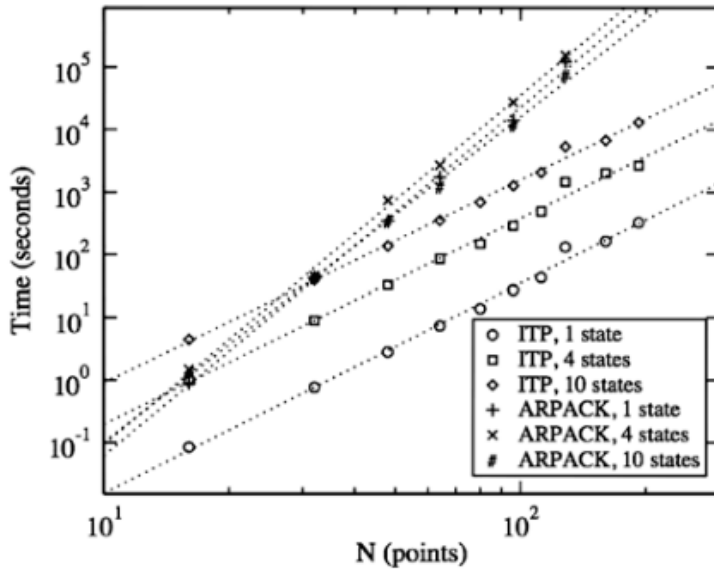
$$\Delta E_i = |E_i - \langle \psi_i(r, \tau) | H | \psi_i(r, \tau) \rangle| \leq \sqrt{2} \sqrt{\langle \psi_i(r, \tau) | H^2 | \psi_i(r, \tau) \rangle - \langle \psi_i(r, \tau) | H | \psi_i(r, \tau) \rangle^2} \quad (13)$$



## 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.



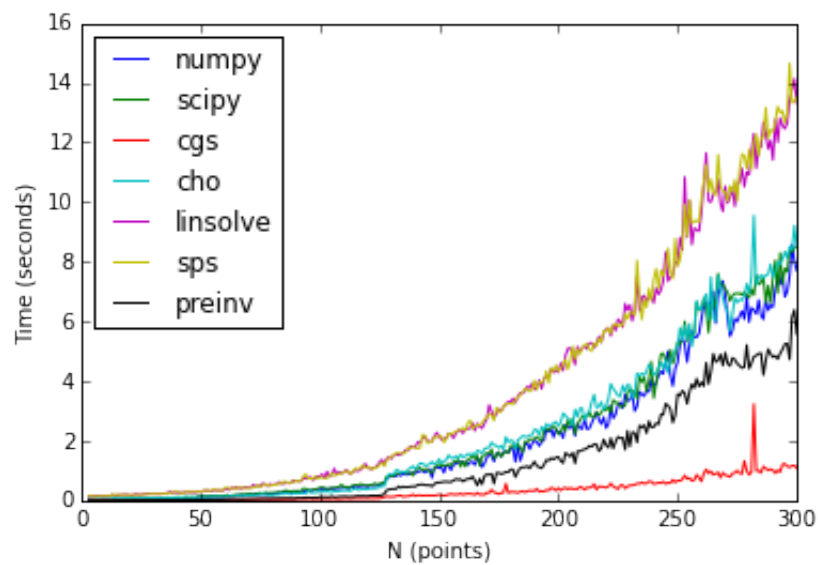


Figure 7: Implementation of seven linear solving schemes in Python.

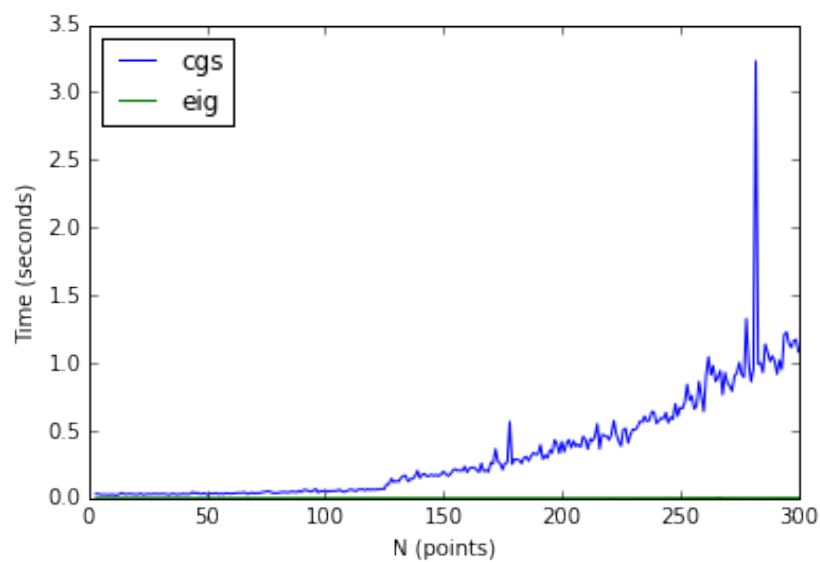


Figure 8: Conjugate Gradient Squared Solver ITP v Built-in Eigensolver

### *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

$$E[\Psi, \omega] = \int \left\{ \frac{\hbar^2}{2m} |\nabla \Psi|^2 + \epsilon_{OT}[\Psi] + V_{X-\text{He}} |\Psi|^2 - \omega \Psi * L_z \Psi \right\} \quad (14)$$

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 first-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, N<sub>2</sub>O, 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.



## Appendix

### Structuring a Scientific Project

The below are important highlights from *A Quick Guide to Organizing Computational Biology Projects* by William Noble.

It is generally a good idea to store all of the files relevant to one project under a common root directory.

use a top-level organization that is logical, with chronological organization at the next level, and logical organization below that

In parallel with this chronological directory structure, I find it useful to maintain a chronologically organized lab notebook. This is a document that resides in the root of the results directory and that records your progress in detail. Entries in the notebook should be dated, and they should be relatively verbose, with links or embedded images or tables displaying the results of the experiments that you performed.

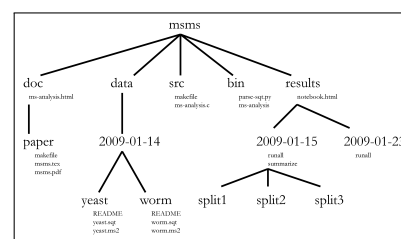


Figure 9: Structuring a Scientific Project

### Carrying Out a Single Experiment

record every operation that you perform

create either a README file, in which I store every command line that I used while performing the experiment, or a driver script (I usually call this runall) that carries out the entire experiment automatically

I work in a combination of shell scripts, Python, and C

Whatever you decide, you should end up with a file that is parallel to the lab notebook entry

Here are some rules of thumb that I try to follow when developing the driver script:

1. Record every operation that you perform.
2. Comment generously.
3. Avoid editing intermediate files by hand.

Many simple editing operations can be performed using standard Unix utilities such as sed, awk, grep, head, tail, sort, cut, and paste.

4. Store all file and directory names in this script.
5. Use relative pathnames to access other files within the same project.
6. Make the script restartable.

For experiments that take a long time to run, I find it useful to be able to obtain a summary of the experiment's progress thus far.

### *Command Lines versus Scripts versus Programs*

1. Driver Script
2. Single-use Script
3. Project-specific script
4. Multi-project script.

Regardless of how general a script is supposed to be, it should have a clearly documented interface.

### *The Value of Version Control*

provides a form of backup

version control provides a historical record that can be useful for tracking down bugs or understanding old results.

invaluable for collaborative projects

changes should be checked in at least once a day

it is possible to check in your changes on a “branch” of the project

should only be used for files that you edit by hand

## *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 Symmetric 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 pyzmq
pip install Jinja2
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 L<sup>A</sup>T<sub>E</sub>X installation.

Strang, Gilbert. 2007. *Computational Science and Engineering*. Vol. 1. Wellesley-Cambridge Press Wellesley.