Computational Physics HW4

Luke Bouma

August 10, 2015

1 Asymmetric quantum well

Quantum mechanics can be formulated as a matrix problem, and solved on a computer using linear algebra methods. For example. particle mass M, well of width L, some potential $V(x)$ in the well from $x = 0$ to $x = L$. We can't solve such a problem analytically in general, but we can on a computer!

In a pure state of energy E, the spatial part of the wavefunction obeys the time-independent Schrödinger equation $\hat{H}\psi(x) = E\psi(x)$, where $\hat{H}$ is

$$\hat{H} = -\frac{\hbar^2}{2M}\frac{d^2}{dx^2} + V(x).$$

For simplicity, assume the walls are infinitely high, so that $\psi(0) = \psi(L) = 0$. Thus the wavefunction can be expressed as a Fourier series:

$$\psi(x) = \sum_{n=1}^{\infty} \psi_n \sin\left(\frac{n\pi x}{L}\right)$$

where ψ_n are Fourier coefficients.

1.1 Manipulating Schrödinger's eqn

Noting that for m, n positive integers,

$$\int_0^L \sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L} = \begin{cases} L/2 & \text{if } m = n \\ 0 & \text{otherwise,} \end{cases}$$

show that the Schrödinger equation $\hat{H}\psi = E\psi$ implies that

$$\sum_{n=1}^{\infty}\psi_n \int_0^L \sin\frac{\pi m x}{L}\hat{H}\sin\frac{\pi n x}{L}\mathrm{d}x = \frac{1}{2}LE\psi_m.$$

And hence, defining a matrix $\mathbf{H}$ with elements

$$\begin{aligned} H_{mn} &= \frac{2}{L}\int_0^L \sin\frac{\pi m x}{L}\hat{H}\sin\frac{\pi n x}{L}\mathrm{d}x \\ &= \frac{2}{L}\int_0^L \sin\frac{\pi m x}{L}\left[-\frac{\hbar^2}{2M}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right]\sin\frac{\pi n x}{L}\mathrm{d}x \end{aligned}$$

show that Schrödinger's equation can be written in matrix form as $\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi}$, where $\boldsymbol{\psi}$ is the vector $(\psi_1, \psi_2, ...)$. Thus $\boldsymbol{\psi}$ is an eigenvector of the Hamiltonian matrix $\mathbf{H}$, with eigenvalue E. If we can calculate the eigenvalues of this matrix, then we know the allowed energies of the particle in the well.

Showing the claim is true: Since $\hat{H}\psi = E\psi$, inserting the Fourier-expanded version of $\psi(x)$ into Schrödinger's equation, we see:

$$\begin{aligned}
\hat{H}\psi &= E\psi \\
\hat{H}\sum_{n=1}^{\infty}\psi_n \sin\left(\frac{n\pi x}{L}\right) &= E\sum_{n=1}^{\infty}\psi_n \sin\left(\frac{n\pi x}{L}\right) \\
\sum_{m=1}^{\infty}\sin\left(\frac{m\pi x}{L}\right)\hat{H}\sum_{n=1}^{\infty}\psi_n \sin\left(\frac{n\pi x}{L}\right) &= \sum_{m=1}^{\infty}\sin\left(\frac{m\pi x}{L}\right)E\sum_{n=1}^{\infty}\psi_n \sin\left(\frac{n\pi x}{L}\right) \\
\sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\psi_n \sin\left(\frac{m\pi x}{L}\right)\hat{H}\sin\left(\frac{n\pi x}{L}\right) &= E\sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\psi_n \sin\left(\frac{m\pi x}{L}\right)\sin\left(\frac{n\pi x}{L}\right) \\
\sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\int_0^L \mathrm{d}x\ \psi_n \sin\left(\frac{m\pi x}{L}\right)\hat{H}\sin\left(\frac{n\pi x}{L}\right) &= E\sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\psi_n\int_0^L \mathrm{d}x\ \sin\left(\frac{m\pi x}{L}\right)\sin\left(\frac{n\pi x}{L}\right) \\
\sum_{n=1}^{\infty}\psi_n\int_0^L \mathrm{d}x\ \sin\left(\frac{m\pi x}{L}\right)\hat{H}\sin\left(\frac{n\pi x}{L}\right) &= EL\psi_m/2
\end{aligned}$$

where in the final line we used the orthogonality of the sine functions.

1.2 For $V(x) = ax/L$, evaluate the integral in H_{mn} analytically.

Also, show that the matrix is real and symmetric. We said that

$$\begin{aligned}
H_{mn} &= \frac{2}{L}\int_0^L \sin\frac{\pi m x}{L}\left[-\frac{\hbar^2}{2M}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right]\sin\frac{\pi n x}{L}\mathrm{d}x, \text{ so} \\
&= \frac{2}{L}\int_0^L \sin\frac{\pi m x}{L}\left[-\frac{\hbar^2}{2M}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + \frac{ax}{L}\right]\sin\frac{\pi n x}{L}\mathrm{d}x \\
&= \frac{2}{L}\int_0^L \left(\frac{ax}{L}\sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L} + \frac{\hbar^2}{2M}\frac{\pi^2 n^2}{L^2}\sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L}\right)\mathrm{d}x \\
&= \frac{2}{L}\left(\frac{a}{L}\begin{cases} 0 & \text{if } m \neq n \text{ and both even or both odd} \\ -\left(\frac{2L}{\pi}\right)^2\frac{mn}{(m^2-n^2)^2} & \text{if } m \neq n \text{ and one is even, one is odd} \\ L^2/4 & \text{if } m = n \end{cases} + \begin{cases} \frac{\hbar^2\pi^2 n^2}{2ML^2}\frac{L}{2} & \text{if } m = n \\ 0 & \text{otherwise} \end{cases}\right)
\end{aligned}$$

$$H_{mn} = \begin{cases} \frac{a}{2} + \frac{\hbar^2\pi^2 n^2}{2ML^2} & \text{if } m = n \\ 0 & \text{if } m \neq n \text{ and both even or both odd} \\ -\frac{8a}{\pi^2}\frac{mn}{(m^2-n^2)^2} & \text{if } m \neq n \text{ and one is even, one is odd} \end{cases}$$

Clearly, all of these terms are real. To see that $\mathbf{H}$ is symmetric, we only need to consider $-\frac{8a}{\pi^2}\frac{mn}{(m^2-n^2)^2}$. Do it in a dopey-picture:

1-based indexing:

$$\frac{mn}{(m^2-n^2)^2}$$

clearly equal for
1,2 ↔ 2,1
3,4 ↔ 4,3, ..., etc!

$$H = \begin{pmatrix} \cdot & 1,2 & 0 & 1,4 & 0 & & & \\ 2,1 & \cdot & 2,3 & 0 & & & & \\ 0 & 3,2 & \cdot & 3,4 & 0 & & & \\ 4,1 & 0 & 4,3 & \cdot & & & & \\ 0 & & 0 & & \cdot & & & \\ & & & & & \cdot & & \\ & & & & & & \cdot & \end{pmatrix}$$

1.3 Find the energy spectrum

Then writing the program to calculate H_{mn} for an electron in a well,

```
from numpy import zeros, pi
from numpy.linalg import eigvals

eV = 1.6022e-19
a = 10 * eV
M = 9.1094e-31
L = 5e-10
hbar = 1.0545717e-34

N = 10
H = zeros([N,N], float)

for m in range(N):
    for n in range(N):
        if n == m: #careful of indexing
            H[m, n] = a/2 + (hbar*pi*(n+1))**2/(2*M*L**2)
        elif n != m and ((m%2 == 0 and n%2 == 0) or (m%2 == 1 and n%2 ==1)):
            H[m,n] = 0.
        elif n != m and ((m%2 == 0 and n%2 == 1) or (m%2 == 1 and n%2 ==0)):
            H[m,n] = -8*a/pi**2 * (m+1)*(n+1) / ((m+1)*(m+1) - (n+1)*(n+1))**2

eigs = sort(eigvals(H)/eV)
print(eigs[0:10])
```

gives the desired minimum at around 5.84eV. Specifically, it prints

```
[   5.83634232    11.18099441    18.66266893    29.14379965    42.65445165
   59.18435979    78.72813749   101.28388661   126.8493641    155.55283499]
```

1.4 Repeat, with $N = 100$:

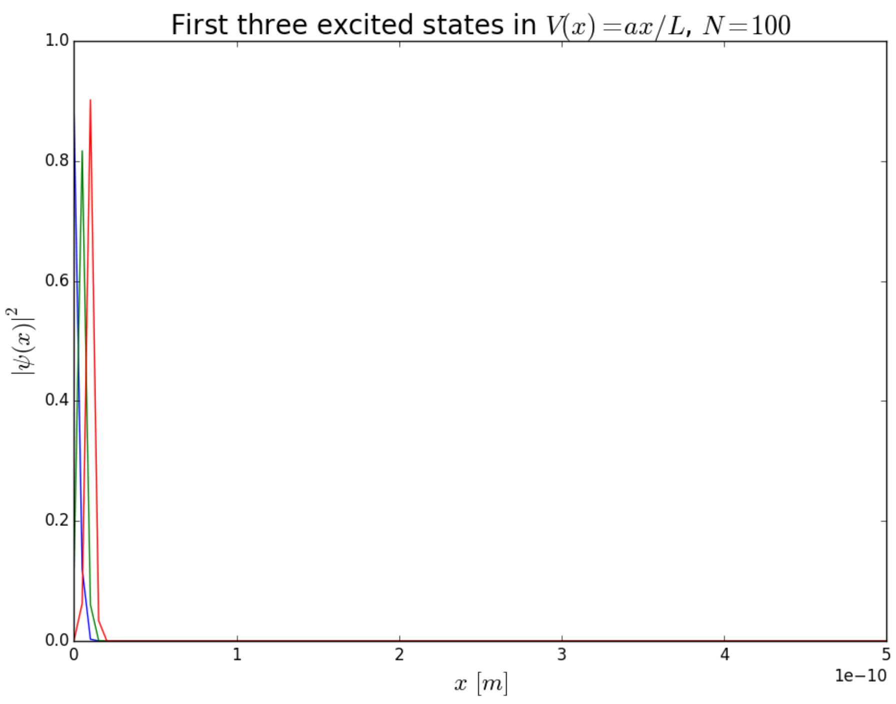
We get a spectrum of

```
[   5.83634191    11.18099309    18.66266706    29.14379085    42.65444253
   59.1843072     78.72808566   101.28325566   126.84853175   155.42321038]
```

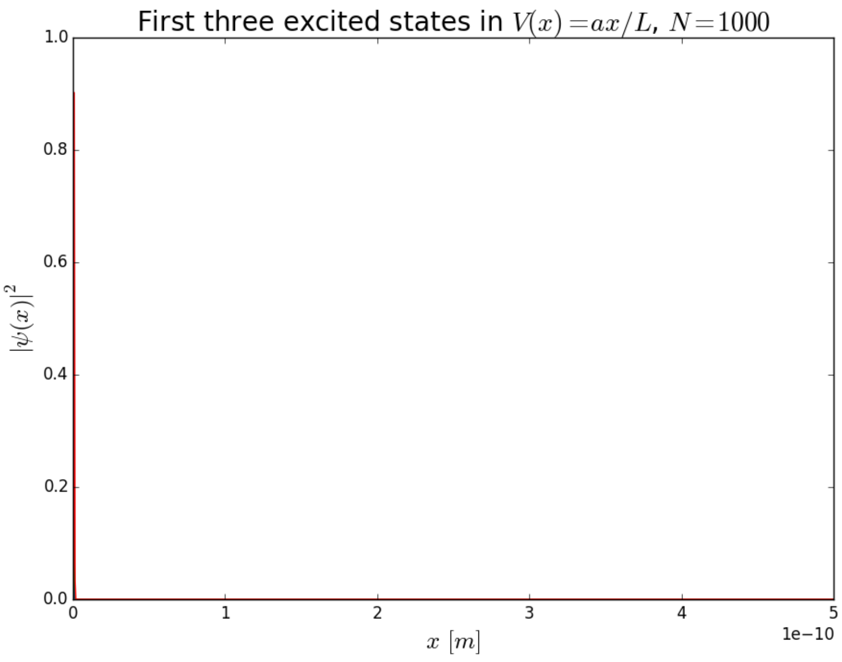
which has eigenvalues that are slightly lower than those of the $N = 10$ Hamiltonian, past like the fifth decimal place. We take this to mean that the larger N gives us slightly more accuracy (since if the eigenvalue finding algorithm works anything like the QR algorithm, then it's trying to minimize the eigenvalues).

1.5 Calcuate $\psi(x)$ for the ground state and first two excited states

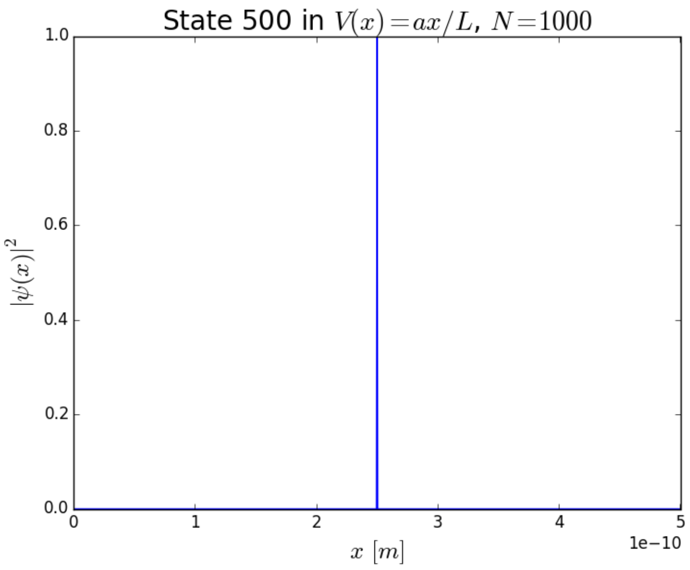
Then plot $|\psi(x)|^2$ as a function of x in each of these three states. Is the normalization of the wavefunction, $\int_0^L |\psi(x)|^2 \mathrm{d}x = 1$, well satisfied?



where the ground state is closest to the origin. Obviously, the normalization here isn't very well-obeyed. Look at what happens when $N = 1000$ though:



but also



To generate these plots, we used the code

```
vals, vecs = eig(H)
sortInd = argsort(vals) #return array of indices that sort eigenvalues
vals = sort(vals/eV)

print(vals[0:10])
x = array(linspace(0, L, N))

for i in range(1):
    plt.plot(x, vecs[:, sortInd[i]]**2)
plt.xlim(-1e-10, 5e-10)
plt.xlabel('$x$ $[m]$', fontsize=18)
plt.ylabel('$|\psi(x)|^2$', fontsize=18)
plt.title('State 1 in $V(x)=ax/L$, $N=1000$', fontsize=20)
plt.show()
```

and basically, in regard to the normalization, we notice that it seems pretty broken for the lowest energy eigenstates, but it gets better for higher-energy ones, especially at high N

2 Glycosis modeling

Glycosis is a biochemical process in the body breaks down glucose to release energy. It can be modelled by the equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -x + ay + x^2 y, \quad \frac{\mathrm{d}y}{\mathrm{d}t} = b - ay - x^2 y,$$

where x and y represent concentrations of two chemicals, ADP and F6P, and a and b are positive constants. An important feature of first order linear equations like these is their stationary points – the points at which both x and y stop changing simultaneously. Setting derivatives to zero, these points are the solutions of

$$0 = -x + ay + x^2 y, \quad 0 = b - ay - x^2 y.$$

2.1 Find the analytical solutions to these equations.

From the first, and then the second,

$$\begin{aligned} y &= \frac{x}{a+x^2} = \frac{b}{a+x^2} \\ \implies x &= b, \\ \implies y &= \frac{b}{a+b^2}. \end{aligned}$$

2.2 Rearrange, and write program to solve for stationary points with the relaxation method. The method should fail to converge.

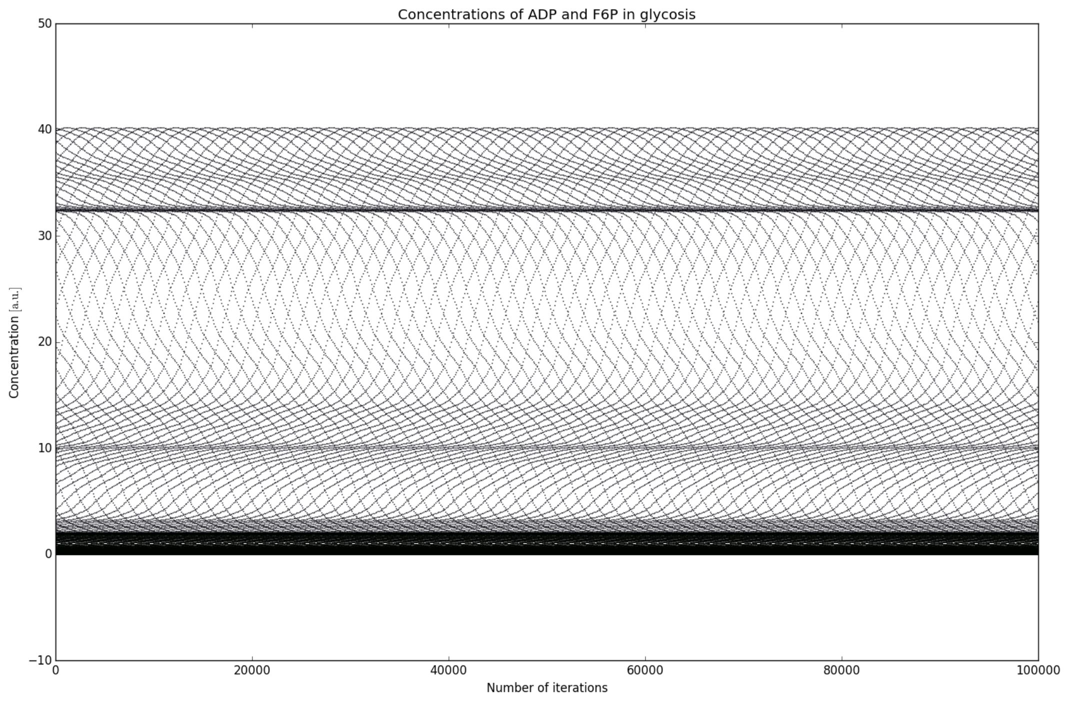
Obviously we can rearrange the equations to be

$$x = y(a+x^2), \quad y = \frac{b}{a+x^2}.$$

Recall what the relaxation method is: for nonlinear equations that aren't solvable analytically, the relaxation method says: just iterate the equation. I.e., to solve $x = f(x)$, just guess some $x = x_0$, and iterate. We showed in lecture 12 that the condition for convergence is $|f'(x_0)| > 1$. Here we're told to set $a = 1, b = 2$ so

$$x = y(1+x^2), \quad y = \frac{2}{1+x^2}.$$

When we're expecting to see the concentrations tending towards steady state, what we see instead is:



which, while it looks similar to those Feigenbaum plots, doesn't look like a steady-state. The code we used for this was

```
import matplotlib.pyplot as plt
from numpy import zeros, arange, copy

def f0(x, y):
    return y*(1 + x*x)
def f1(x):
    return 2/(1 + x*x)

x0 = 1.
y0 = 1.
numIter = 1e5
x = zeros([numIter], float)
y = copy(x)
x[0], y[0] = x0, y0

# want x=2, y=2/5
n = 1
while n < numIter:
    x[n], y[n] = f0(x[n-1],y[n-1]), f1(x[n-1])
    n += 1

plt.scatter(arange(numIter), x, s=0.1, c='blue')
plt.scatter(arange(numIter), y, s=0.1, c='green')
plt.xlim([0, numIter])
plt.xlabel('Number of iterations')
plt.ylabel('Concentration $[\mathrm{a.u.}]$')
plt.title('Concentrations of ADP and F6P in glycosis')
plt.show()
```

2.3 Rearrange the equations so that the relaxation method converges, giving a solution.

For the equations

$$x = f_0(x, y), \qquad f_0(x, y) = y(1 + x^2)$$
$$y = f_1(x), \qquad f_1(x) = \frac{2}{1 + x^2},$$

we know that at least one eigenvalue of the Jacobian,

$$\left.\begin{pmatrix} \frac{\partial f_0}{\partial x} & \frac{\partial f_0}{\partial y} \\ \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \end{pmatrix}\right|_{x_0, y_0} = \left.\begin{pmatrix} 2xy & 1 + x^2 \\ \frac{-4x}{(1+x^2)^2} & 0 \end{pmatrix}\right|_{x_0, y_0}$$

must be greater than or equal to 1. The eigenvalues of that one are crazy (and easy to compute with say, Mathematica).

Obviously the actual solution here is $x = 2, y = 2/5$. If we just input these numbers as the initial guesses, we stay converged, but that's obviously cheating.
Not sure what to do here.

3 Definite programming

Gonna skip, because I did this in the optimization class, and there's no coding.

4 Fast Fourier transform

Recall the definition of the discrete Fourier transform (following Jake VanderPlas):

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N},$$

and the inverse transform:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N}.$$

This is a transformation from a 'configuration' space (it could be any kind of data – spatial, sound, image, etc., provided it is periodic in time) to frequency space, and vice-versa. The FFT does it faster, in $\mathcal{O}(N \log N)$ steps, instead of the first-guess implementation of $\mathcal{O}\left(N^2\right)$. Let's try doing that first guess implementation first:

```
import numpy as np

def DFTslow(x):
    x = np.asarray(x, dtype=float)        # conv-> float array
    N = len(x)
    n = np.reshape(np.arange(N),(1,N))    # n row vector
    k = n.reshape((N,1))                  # k col vector
    M = np.exp(-2j * np.pi * k * n / N) # lets us matrix mult
    X = np.dot(M, x)
    return X

x = np.random.random(1024)
print(np.allclose(DFTslow(x), np.fft.fft(x)))
```

where the 'allclose?' query returns true, so we know we're doing the right thing.

Now, let's go for the fast Fourier transform. To see the symmetry we're going to exploit, write X_{N+k} from our definition of the DFT:

$$\begin{aligned} X_{N+k} &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi(N+k)n/N} \\ &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} e^{-i2\pi n} \\ &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} = X_k \end{aligned}$$

and in the second line, we used $e^{-i2\pi n} = 1$ for integer n. Similarly, for any integer i, $X_{k+iN} = X_k$. Now we start splitting the full DFT into even and odd terms:

$$\text{Let } E_k = \sum_{m=0}^{\frac{1}{2}N-1} x_{2m} e^{\left(-i\frac{2\pi k(2m)}{N}\right)} = \sum_{m=0}^{\frac{1}{2}N-1} x_{2m} e^{\left(-i\frac{2\pi km}{N/2}\right)}.$$

Similarly for the odd terms, $n = 2r + 1$ for $r = 0, 1, 2, ..., \frac{N}{2} - 1$:

$$\begin{aligned} \sum_{m=0}^{\frac{1}{2}N-1} x_{2m+1} e^{\left(-i\frac{2\pi k(2m+1)}{N}\right)} &= e^{-i2\pi k/N} \sum_{m=0}^{\frac{1}{2}N-1} x_{2m+1} e^{\left(-i\frac{2\pi km}{N/2}\right)} \\ &= e^{-i2\pi k/N} O_k, \end{aligned}$$

so that

$$X_k = E_k + e^{-i2\pi k/N} O_k \qquad (\star)$$

Now each term consists of $(N/2) \times N$ computations (total: N^2). However, they now have symmetries! The range of k is $0 \le k < N$, while the range of m is $0 \le m < M \equiv N/2$. From the symmetry property we wrote at the beginning, this means we only need to perform *half* the computations for each sub-problem, making our problem size $\mathcal{O}\left(M^2\right)$.

We can keep going: repeatedly halve the size of the smaller Fourier transforms, and halve the computational cost each time, until the arrays are small enough that the benefit is lost. This means we can implement the FFT recursively (this won't be fast, but it gets the idea):

```
def FFTrecursive(x):
    x = np.asarray(x, dtype=float)
    N = len(x)
    if N <= 32:
        return DFTslow(x)
    else:
        X_even = FFTrecursive(x[::2])    #x[i:j:k] slice i start, j stop, k step
        X_odd = FFTrecursive(x[1::2])
        twFactor = np.exp(-2j * np.pi * np.arange(N) / N)   #k from 0 to N-1
        return np.concatenate([X_even + twFactor[:N / 2] * X_odd,
                               X_even + twFactor[N / 2:] * X_odd])  #concat odd and even
                                   parts

x = np.random.random(1024)
print(np.allclose(DFTslow(x), np.fft.fft(x)))
```

Finally, a vectorized implementation looks like this:

```
def FFT_vectorized(x):
    x = np.asarray(x, dtype=float)
    N = x.shape[0]

    if np.log2(N) % 1 > 0:
        raise ValueError("size of x must be a power of 2")

    N_min = min(N, 32)

    # Perform an O[N^2] DFT on all length-N_min sub-problems at once
    n = np.arange(N_min)
    k = n.reshape(N_min,1)
    M = np.exp(-2j * np.pi * n * k / N_min)
    X = np.dot(M, x.reshape((N_min, -1)))

    # build-up each level of the recursive calculation all at once
    while X.shape[0] < N:
        X_even = X[:, :X.shape[1] / 2]
        X_odd = X[:, X.shape[1] / 2:]
        factor = np.exp(-1j * np.pi * np.arange(X.shape[0])
                        / X.shape[0])[:, None]
        X = np.vstack([X_even + factor * X_odd,
                       X_even - factor * X_odd])

    return X.ravel()
```

Honestly, this is a bit above where I am right now in Python-land, but it's good to get an idea for where I want to be.

5 Image deconvolution

5.1 Import image data and plot it as a density plot

```
import matplotlib.pyplot as plt

with open('blur.txt') as file:
    imgData2D = [[float(digit) for digit in line.split()] for line in file]

plt.imshow(imgData2D)
plt.gray()
plt.show()
```

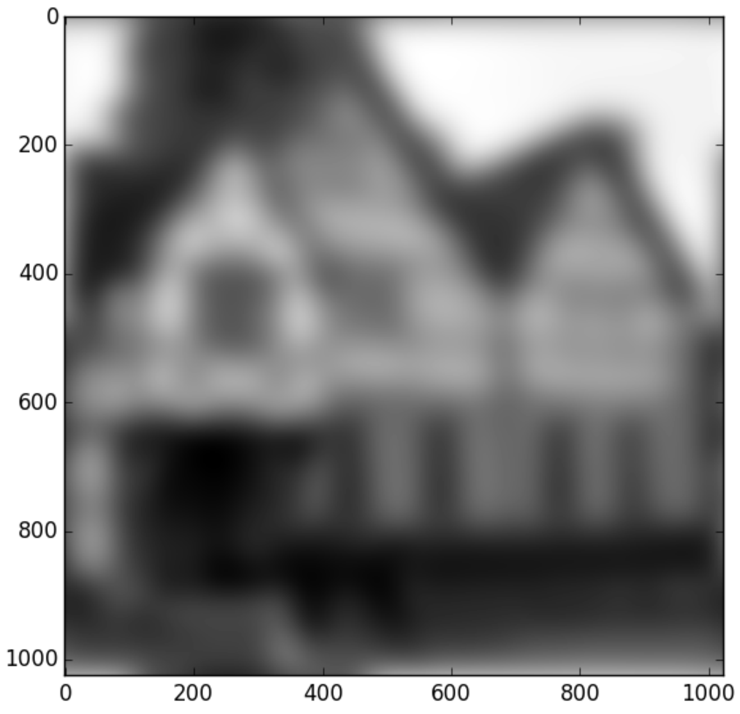


Figure 1: Blurred image from part 5a).

yields Fig. 1 below.

5.2 Make 2D Gaussian array, plot it.

This is plotting the function

$$f(x, y) = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right),$$

for $\sigma = 25$, with periodic boundary conditions:

```python
import numpy as np
def getGaussian(imgData):
    if np.shape(imgData)[0] != np.shape(imgData)[1]:
        raise ValueError("Only works on square images")
    N = np.shape(imgData)[0]
    x = np.arange(-N/2,N/2)
    y = np.arange(-N/2,N/2)
    f = np.zeros([N,N], float)
    sigma = 100
    prefac = -1/(2*sigma**2)
    for i in range(N):
        for j in range(N):
            f[i, j] = np.exp(prefac*(x[i]**2 + y[j]**2))
    f = np.roll(np.roll(f, N//2, axis=0), N//2, axis=1)
    return f
```

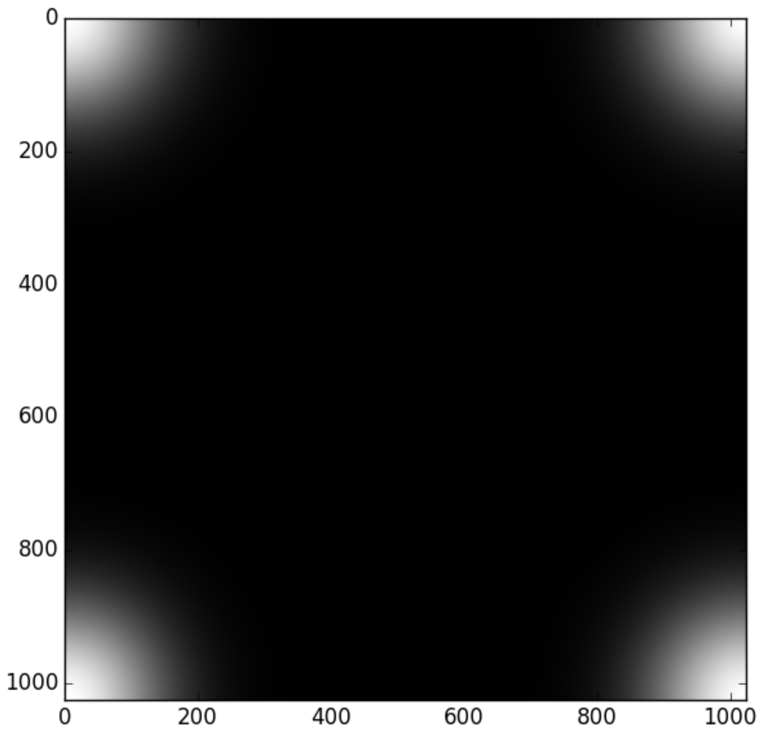


Figure 2: Gaussian blur with periodic BCs, using $\sigma = 100$. N.b. $\sigma = 25$ gives default blurring far less bright than Mark Wilde's....

5.3 Write full program

Want it to:

1. Read in blurred photo
2. Calculate point spread function
3. Fourier transform both of them
4. Divide one by the other
5. Perform an inverse FT to get the unblurred photo
6. Display the unblurred photo on the screen.

Ok, it's kind of crap, and I still don't understand a major part of it, but here's nothing:

```
import numpy as np
import matplotlib.pyplot as plt

def getGaussian(imgData):
    if np.shape(imgData)[0] != np.shape(imgData)[1]:
        raise ValueError("Only works on square images")
    N = np.shape(imgData)[0]
    x = np.arange(-N/2,N/2)
    y = np.arange(-N/2,N/2)
    f = np.zeros([N,N], float)
    sigma = 25
    prefac = -1/(2*sigma**2)
    for i in range(N):
        for j in range(N):
            f[i, j] = np.exp(prefac*(x[i]**2 + y[j]**2))
    f = np.roll(np.roll(f, N//2, axis=0), N//2, axis=1)
    return f

with open('blur.txt') as file:
    imgDat = [[float(digit) for digit in line.split()] for line in file]

ptSpread = getGaussian(imgDat)
imgDatFT = np.fft.fft2(imgDat)
ptSpreadFT = np.fft.fft2(ptSpread)

eps = 1e-3
resultFT = np.zeros(np.shape(imgDatFT))
for i in range(np.shape(imgDatFT)[0]):
    for j in range(np.shape(imgDatFT)[1]):
        if ptSpreadFT[i, j] > eps:
            resultFT[i, j] = imgDatFT[i, j] / ptSpreadFT[i, j]
        else:
            resultFT[i, j] = imgDatFT[i, j]
result = np.fft.ifft2(resultFT)

plt.imshow(np.real(result))
plt.gray()
plt.show()
```

which gives Figure ??. Things to notice include a bunch of weird artifacts, as well as the fact that

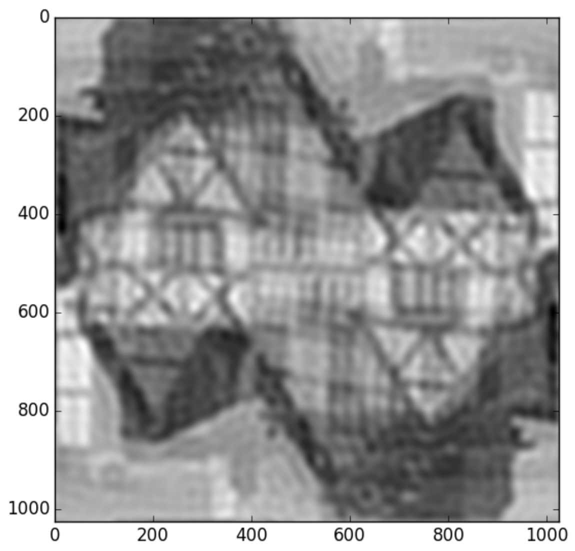


Figure 3: Crap deblurring. The top half of the image seems ‘alright’, the bottom half is a reflection.

the bottom half of the image is a reflection of the top. I don't know why that is. My first guess was that it had to do with `rfft2(x)` throwing away the 'redundant half' of the matrix during the manipulations, but that's wrong. The effect still crops up in that calculation too.

I think I'm going to call it here. It's maybe not the best way to wrap up, but I think I have a much better understanding of the concepts. We'll write a summary in the notes, and then turn them in.