Physics 465: Computing Project 5

September 19, 2012

Free Particles and the Fourier Transform

This week's project is about following the behavior of a free particle using the Fourier Transform as a tool to change to the k (or momentum) basis, evolve the time, then convert back to to the position basis to determine the new wavefunction. This requires the use of a new tool in python: The Fast Fourier Transform (FFT). The FFT is very closely related to the idea of a Fourier Transform (FT) described in your book. The FT is defined like so:

$$\phi(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-ikx} \psi(x) dx \tag{1}$$

The Inverse Fourier Transform (IFT) is defined symmetrically:

$$\psi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{+ikx} \phi(k) dk \tag{2}$$

The FT and IFT transform continuous wavefunctions from "real space" to "k space" (with the FT) and back again (using the IFT). For machine computing it's nice to be able to generalize these ideas to sampled wavefunctions such as those we've been using, where we're really just tracking discrete numbers at specific locations in space. In this context the wavefunction $\psi(x)$ becomes an array of sampled values ψ_n and the transform $\phi(k)$ becomes a set of discrete values ϕ_k:

$$\phi_k = \sum_{n=0}^{N} e^{-ink2\pi/N} \psi_n \tag{3}$$

Where $k \in \{0, 1, 2, \ldots, N-1\}$ Note the similarity with Eq. 1. The inverse transform is also similar:

$$\psi_n = \frac{1}{N}\sum_{k=0}^{N} e^{+ink2\pi/N}\phi_k \tag{4}$$

One point is that while the integrals (Eq. 1 and Eq. 2) are over all space from $-\infty$ to $+\infty$, the corresponding sums are finite ranging from 0 to N. One obvious question is "What happened to the negative frequencies?". The answer is subtle, and will become more clear when we get to Dirac notation in chapter 3. In the mean time notice that when k is greater than $N/2$ the angle $2\pi k/N$ is greater than π. If you think about it, as you move through successive values of n, multiplying by a phasor with an angle greater than π is exactly the same as multiplying by an negative phase! So.. the negative frequencies are just the terms where k is greater than $N/2$.

How do we compute the FFT in python?

Easy! Here is some sample code:

```
#-------------------------------------------------------------------------#

NA=150                          # how many arrows?
a=30.0                          # range of x is -a/2 to a/2 in units
                                # of $\sqrt{\hbar/m\omega}$
x = linspace(-a/2, a/2, NA)     # NA locations from -a/2 to a/2
kMin = 2*pi/a
k0 = 10*kMin
sigma = a/15.0
psi=exp(1j*k0*x - ((x+3*a/8)/sigma)**2)   # gaussian wave packet
psi = psi/sqrt((abs(psi)**2).sum())       # normalize

alist = []
for i in range(NA):
    alist.append(arrow(pos=(x[i],0,0), color=color.red))
    SetArrowFromCN(arrowScale*psi[i],alist[i])

phi0 = fft.fft(psi)  # Compute the FFT at t=0.

#-------------------------------------------------------------------------#
```

Note that the actual calculation of the FFT is the very last line only. Getting the FFT is easy! The inverse FFT is just as easy, except you'd say

```
psi = fft.ifft(phi)         # get the inverse FFT.
```

How do the ϕ_k evolve in time?

For a free particle at least, the answer is easy. The Hamiltonian depends only on momentum, so each momentum component just gets multiplied by $e^{-i\omega_k t}$ where ω_k is the frequency that corresponds to a particular value of k. We could write a loop, given and array ϕ that does this like so:

```
for k in range(len(phi0)):              # iterate through elements of phi
    if k>(N/2):                         # check for k>N/2
        p=hbar*(k-N)*pi/a               # momentum is negative
    else:
        p=hbar*k*pi/a                   # momentum is positive
    omega = (0.5*p**2/(2.0*m*hbar))     # get omega from (kinetic) energy
    phi[k] = phi0[k]*exp(-1j*omega*t)  # rotate phasor by time factor...
```

However, this is extremely slow! It would much better to use the fact that the ϕ_k are in an array structure. If we could get ω to be a parallel *array* all this code could be reduced to:

```
phi = phi0*exp(-1j*omega*t)      # rotate phasor by time factor...
```

But how to do it? It's not too hard, but it takes some thought. The main trouble is that when $k > N/2$ the momentum changes sign and the magnitude of the momentum is proportional to the difference between k and N. There is are two features *piecewise* and *lambda* that can help us with this.

The function *piecewise* is used for created arrays based on piecewise array functions. Say you wanted to create a an array that matched x in dimension but whose elements were:

$$y = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases} \tag{5}$$

This is exactly what *piecewise* is for. You can do it with one line of code like this:

```
y = piecewise(x, [x<0, x==0, x>0], [-1, 0, 1])
```

You can also pass in functions for the 'values' and the functions will be called to fill in:

```
y = piecewise(x, [x<0, x==0, x>0], [f1, f2, f3])
```

To do what we want we need to check to see if n is greater than $N/2$ and return $n - N$ if it is, or n if it isn't.

```
n=arange(N)
```

```
def f1(n):
    return n

def f2(n):
    return n - N

y = piecewise(n, [n<N/2, n>=N/2], [f1, f2])
```

However, we can even improve on this using the `lambda` syntax. This is used to define simple functions in place without requiring a full `def` clause. We could just say:

```
n=arange(N)

f1 = lambda n:n
f2 = lambda n:n - N

y = piecewise(n, [n<N/2, n>=N/2], [f1, f2])
```

Or even easier:

```
n=arange(N)
y = piecewise(n, [n<N/2, n>=N/2], [lambda n:n,lambda n:n-N])
```

Here is some sample code, and some more explanation:

```
hbar=1.0                        # use units where hbar = 1
m=1.0                           # and m=1.0
NA=500                          # how many arrows?
a=30.0                          # range of x is -a/2 to a/2

x = linspace(-a/2, a/2, NA)     # NA locations from -a/2 to a/2
n = arange(NA)                  # n = array([0,1,2,3,.... N-1])

#
# convert n to an array that changes sign above NA/2 to n-NA
n = piecewise(n, [n<NA/2, n>=NA/2], [lambda n:n, lambda n:n - NA])
k = 2*n*pi/a
Energy = (hbar*k)**2/(2.0*m)    # get the kinetic energy
omega = Energy/hbar             # get the frequency
```

This gets us set up with the right `omega` array so we can shift all the phases of ϕ_k, very quickly, with a single line of code.

So.. what are we supposed to do?

Your mission is to produce two deliverables:

1) A 3D representation of the complex wavefunction corresponding to a gaussian wave-packet which should start out looking like this:

Fig. 1.

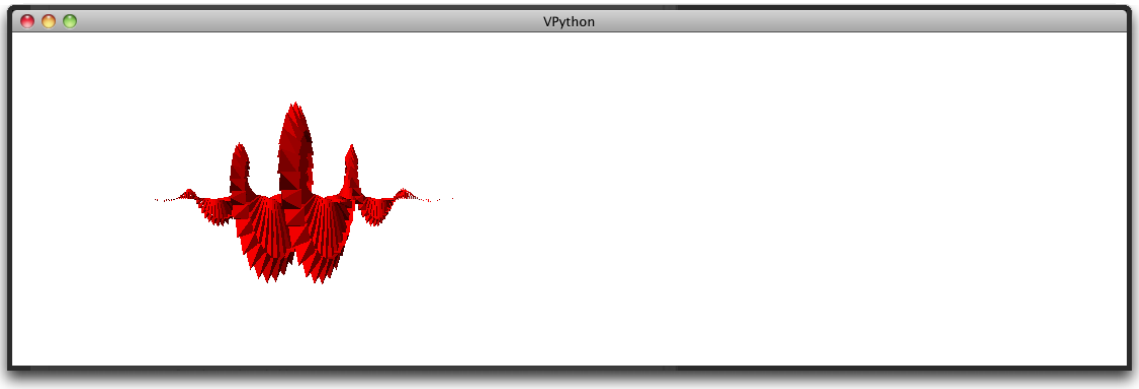

Figure 1: The initial wavefunction, 500 arrows, a=30.0

Later on your display should look something like this Fig. 2.

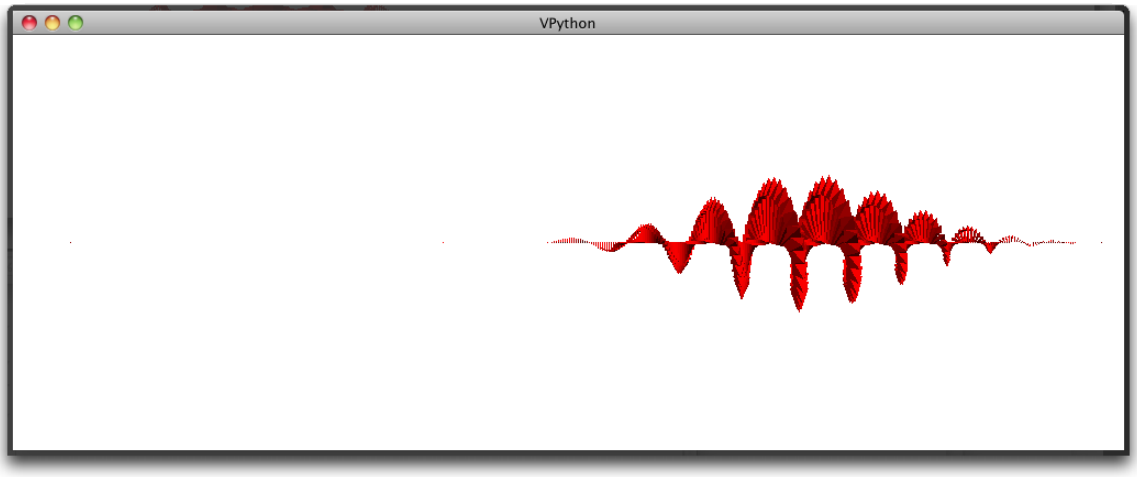

Figure 2: The wavefunction later, 40 arrows, a=6.0

Please include at least one screen capture of your wavefunction and include it in your report.

2) A graph of the expectation value of the position of the particle as a function of time, and a graph of the uncertainty of the position σ_x as a function of time. With my setup

I get graphs that look like Fig. 3 and Fig. 4.

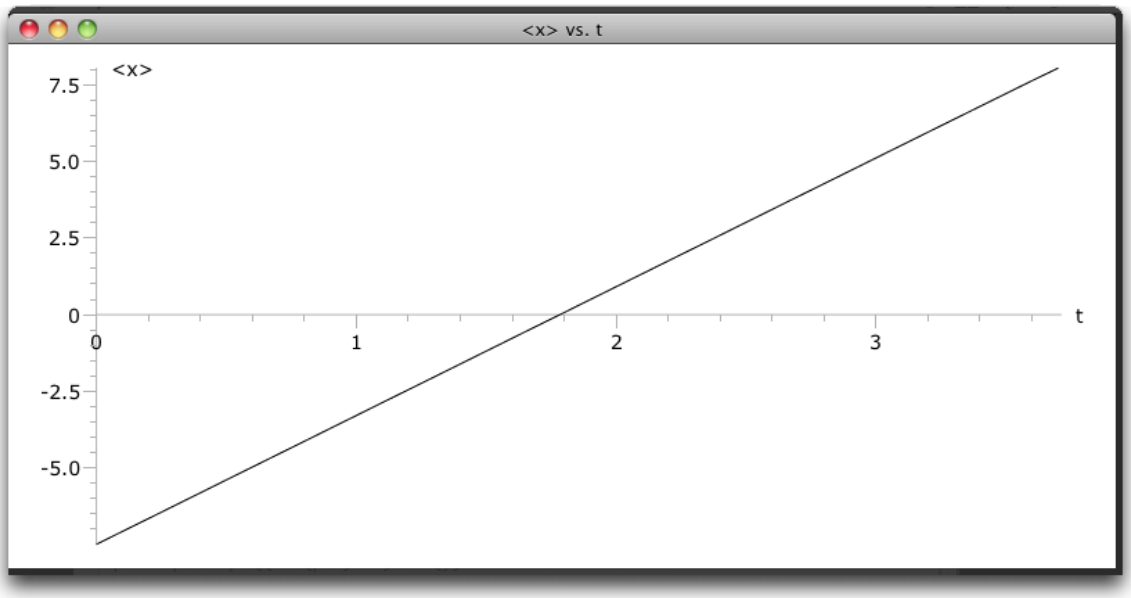

Figure 3: Expectation of position vs. time

I've loaded a 'starter' program on the "K" drive which follows:

```
#
# cp5: Solution to Computing Project 5: FFT evolution
#

from visual import *
from visual.graph import *
from safcn import SetArrowFromCN  # defined in a file safcn.py

gd = gdisplay(title="<x> vs. t", xtitle="t", ytitle="<x>",
              foreground=color.black, background=color.white)
gr = gcurve(color=color.black)
scene.background=color.white

hbar=1.0                        # use units where hbar = 1
m=1.0                           # and m=1.0
NA=500                          # how many arrows?
a=30.0                          # range of x is -a/2 to a/2

x = linspace(-a/2, a/2, NA)     # NA locations from -a/2 to a/2
n = arange(NA)                  # n = array([0,1,2,3,.... N-1])
n = piecewise(n, [n<NA/2, n>=NA/2], [lambda n:n, lambda n:n - NA])
```

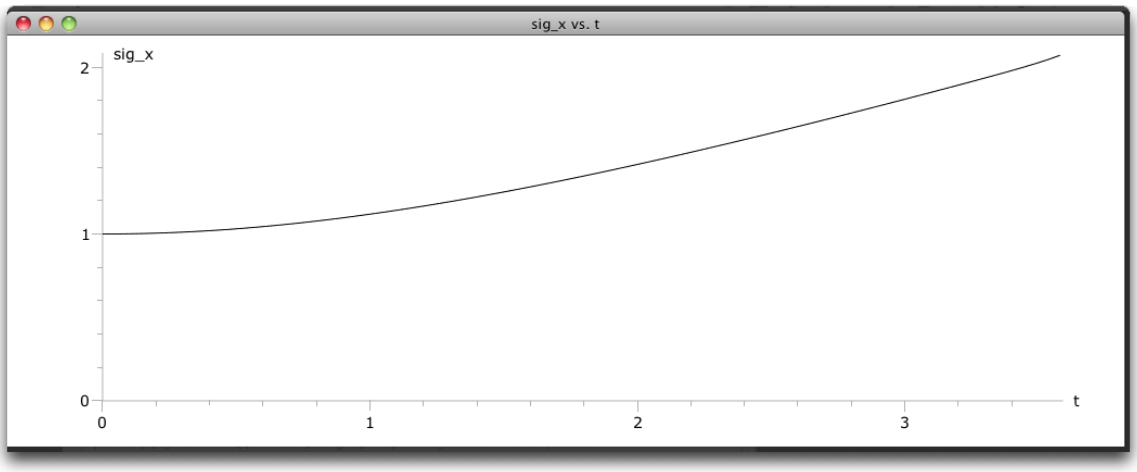

Figure 4: Uncertainty in position vs. time.

```
k = 2*n*pi/a
Energy = (hbar*k)**2/(2.0*m)   # get the kinetic energy
omega = Energy/hbar            # get the frequency

t = 0.0
dt = 0.01
kMin = 2*pi/a
k0 = 20*kMin
sigma = a/15.0
arrowScale = sqrt(NA*a*sigma)/10.0

psi=exp(1j*k0*x - ((x+1*a/4)/sigma)**2)   # gaussian wave packet
psi = psi/sqrt((abs(psi)**2).sum())     # normalize

alist = []
for i in range(NA):
    alist.append(arrow(pos=(x[i],0,0), color=color.red))
    SetArrowFromCN(arrowScale*psi[i],alist[i])

phi0 = fft.fft(psi)  # fft at t=0

updateScreen = False
while True:
    rate(1.0/dt)

    if scene.kb.keys: # event waiting to be processed?
        s = scene.kb.getkey() # get keyboard info
```

```
        if s == ' ':
            updateScreen ^= 1

    if updateScreen:
#
# here is where you'll want to compute the
# time evolution of phi (using phi0 as the t=0 value)
# and then compute the inverse FFT to get back psi
# at the current time. Then update the arrows
# and get expectation values to graph...
#
```

Questions

1) If you allow the program continue to run past the point shown in Fig. 3 what happens to the expectation value of position vs. time? Does this seem like a reasonable result for a free particle? What's happening?.

2) In problem 2.22 you will find the behavior of the gaussian envelope of a free particle wave packet over time. Does the graph of σ_x vs time appear to be reasonable given the result of problem 2.22. Justify your answer with a numerical comparison.