Quantum Python: Animating the Schrodinger Equation

Sep 05, 2012

Update: a reader contributed some improvements to the Python code presented below. Please see the pySchrodinger (https://github.com/jakevdp/pySchrodinger) github repository for updated code

In a previous post (/blog/2012/08/18/matplotlib-animation-tutorial/) I explored the new animation capabilities of the latest matplotlib (http://matplotlib.sourceforge.net) release. It got me wondering whether it would be possible to simulate more complicated physical systems in real time in python. Quantum Mechanics was the first thing that came to mind. It turns out that by mixing a bit of Physics knowledge with a bit of computing knowledge, it's quite straightforward to simulate and animate a simple quantum mechanical system with python.

The Schrodinger Equation

The dynamics of a one-dimensional quantum system are governed by the time-dependent Schrodinger equation:

$$i\hbar\frac{\partial\psi}{\partial t} = \frac{-\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V\psi$$

The *wave function* ψ is a function of both position x and time t, and is the fundamental description of the realm of the very small. Imagine we are following the motion of a single particle in one dimension. This wave function represents a probability of measuring the particle at a position x at a time t. Quantum mechanics tells us that (contrary to our familiar classical reasoning) this probability is not a limitation of our knowledge of the system, but a reflection of an unavoidable uncertainty about the position and time of events in the realm of the very small.

Still, this equation is a bit opaque, but to visualize the results we'll need to solve this numerically. We'll approach this using the split-step Fourier method.

The Split-step Fourier Method

A standard way to numerically solve certain differential equations is through the use of the Fourier transform. We'll use a Fourier convention of the following form:

$$\widetilde{\psi}(k,t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \psi(x,t) e^{-ikx} dx$$

Under this convention, the associated inverse Fourier Transform is given by:

$$\psi(x,t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \widetilde{\psi}(k,t) e^{ikx} dk$$

Substituting this into the Schrodinger equation and simplifying gives the Fourier-space form of the Schrodinger equation:

$$i\hbar \frac{\partial \widetilde{\psi}}{\partial t} = \frac{\hbar^2 k^2}{2m} \widetilde{\psi} + V(i\frac{\partial}{\partial k}) \widetilde{\psi}$$

The two versions of the Schrodinger equation contain an interesting symmetry: the time step in each case depends on a straightforward multiplication of the wave function ψ, as well as a more complicated term involving derivatives with respect to x or k. The key observation is that while the equation is difficult to evaluate fully within one of the forms, each basis offers a straightforward calculation of one of the two contributions. This suggests an efficient strategy to numerically solve the Schrodinger Equation.

First we solve the straightforward part of the x-space Schrodinger equation:

$$i\hbar \frac{\partial \psi}{\partial t} = V(x) \psi$$

For a small time step Δt, this has a solution of the form

$$\psi(x, t + \Delta t) = \psi(x,t) e^{-iV(x)\Delta t / \hbar}$$

Second, we solve the straightforward part of the k-space Schrodinger equation:

$$i\hbar \frac{\partial \widetilde{\psi}}{\partial t} = \frac{\hbar^2 k^2}{2m} \widetilde{\psi}$$

For a small time step Δt, this has a solution of the form

$$\widetilde{\psi}(k, t + \Delta t) = \widetilde{\psi}(k,t) e^{-i\hbar k^2 \Delta t / 2m}$$

Numerical Considerations

Solving this system numerically will require repeated computations of the Fourier transform of $\psi(x,t)$ and the inverse Fourier transform of $\widetilde{\psi}(k,t)$. The best-known algorithm for computation of numerical Fourier transforms is the Fast Fourier Transform (FFT), which is available in scipy (http://docs.scipy.org/doc/scipy/reference/fftpack.html) and efficiently computes the following form of the discrete Fourier transform:

$$\widetilde{F_m} = \sum_{n=0}^{N-1} F_n e^{-2\pi i n m / N}$$

and its inverse

$$F_n = \frac{1}{N} \sum_{m=0}^{N-1} \widetilde{F_m} e^{2\pi i n m / N}$$

We need to know how these relate to the continuous Fourier transforms defined and used above. Let's take the example of the forward transform. Assume that the infinite integral is well-approximated by the finite integral from a to b, so that we can write

$$\widetilde{\psi}(k,t) = \frac{1}{\sqrt{2\pi}} \int_{a}^{b} \psi(x,t) e^{-ikx} dx$$

This approximation ends up being equivalent to assuming that the potential $V(x) \to \infty$ at $x \leq a$ and $x \geq b$. We'll now approximate this integral as a Riemann sum of N terms, and define $\Delta x = (b - a)/N$, and $x_n = a + n\Delta x$:

$$\widetilde{\psi}(k,t) \simeq \frac{1}{\sqrt{2\pi}} \sum_{n=0}^{N-1} \psi(x_n,t) e^{-ikx_n} \Delta x$$

This is starting to look like the discrete Fourier transform! To bring it even closer, let's define $k_m = k_0 + m\Delta k$, with $\Delta k = 2\pi/(N\Delta x)$. Then our approximation becomes

$$\widetilde{\psi}(k_m,t) \simeq \frac{1}{\sqrt{2\pi}} \sum_{n=0}^{N-1} \psi(x_n,t) e^{-ik_m x_n} \Delta x$$

(Note that just as we have limited the range of x above, we have here limited the range of k as well. This means that high-frequency components of the signal will be lost in our approximation. The Nyquist sampling theorem tells us that this is an unavoidable consequence of choosing discrete steps in space, and it can be shown that the spacing we chose above exactly satisfies the Nyquist limit if we choose $k_0 = -\pi/\Delta x$).
Plugging our expressions for x_n and k_m into the Fourier approximation and rearranging, we find the following:

$$\left[\widetilde{\psi}(k_m,t)e^{imx_0\Delta k}\right] \simeq \sum_{n=0}^{N-1}\left[\frac{\Delta x}{\sqrt{2\pi}}\psi(x_n,t)e^{-ik_0x_n}\right]e^{-2\pi imn/N}$$

Similar arguments from the inverse Fourier transform yield:

$$\left[\frac{\Delta x}{\sqrt{2\pi}}\psi(x_n,t)e^{-ik_0x_n}\right] \simeq \frac{1}{N}\sum_{m=0}^{N-1}\left[\widetilde{\psi}(k_m,t)e^{-imx_0\Delta k}\right]e^{2\pi imn/N}$$

Comparing these to the discrete Fourier transforms above, we find that the *continuous* Fourier pair

$$\psi(x,t) \Longleftrightarrow \widetilde{\psi}(k,t)$$

corresponds to the *discrete* Fourier pair

$$\frac{\Delta x}{\sqrt{2\pi}}\psi(x_n,t)e^{-ik_0x_n} \Longleftrightarrow \widetilde{\psi}(k_m,t)e^{-imx_0\Delta k}$$

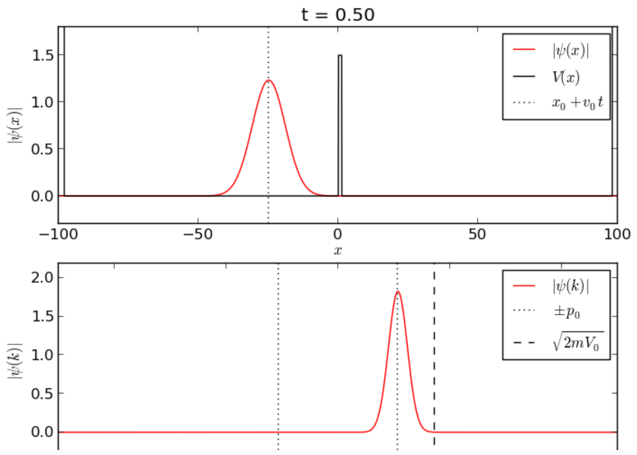
subject to the approximations mentioned above. This allows a fast numerical evaluation of the Schrodinger equation.

Putting It All Together: the Algorithm

We now put this all together using the following algorithm

1. Choose a, b, N, and k_0 as above, sufficient to represent the initial state of your wave function $\psi(x)$. (**Warning:** this is perhaps the hardest part of the entire solution. If limits in x or k are chosen which do not suit your problem, then the approximations used above can destroy the accuracy of the calculation!) Once these are chosen, then $\Delta x = (b-a)/N$ and $\Delta k = 2\pi/(b-a)$. Define $x_n = a + n\Delta x$ and $k_m = k_0 + m\Delta k$.
2. Discretize the wave-functions on this grid. Let $\psi_n(t) = \psi(x_n,t)$, $V_n = V(x_n)$, and $\widetilde{\psi}_m = \widetilde{\psi}(k_m,t)$.
3. To progress the system by a time-step Δt, perform the following:
4. Compute a half-step in x: $\psi_n \longleftarrow \psi_n \exp[-i(\Delta t/2)(V_n/\hbar)]$
5. Calculate $\widetilde{\psi}_m$ from ψ_n using the FFT.
6. Compute a full-step in k: $\widetilde{\psi}_m \longleftarrow \widetilde{\psi}_m \exp[-i\hbar(k\cdot k)\Delta t/(2m)]$
7. Calculate ψ_n from $\widetilde{\psi}_m$ using the inverse FFT.
8. Compute a second half-step in x: $\psi_n \longleftarrow \psi_n \exp[-i(\Delta t/2)(V_n/\hbar)]$
9. Repeat step 3 until the desired time is reached.

Note that we have split the x-space time-step into two half-steps: this turns out to lead to a more stable numerical solution than performing the step all at once. Those familiar with numerical integration algorithms may recognize this as an example of the well-known leap-frog integration technique.
To test this out, I've written a python code which sets up a particle in a box with a potential barrier. The barrier is high enough that a classical particle would be unable to penetrate it. A quantum particle, however, can "tunnel" through, leading to a non-zero probability of finding the particle on the other side of the partition. This quantum tunneling effect lies at the core of technologies as diverse as electron microsopy, semiconding diodes, and perhaps even the future of low-powered transistors.
The animation of the result is below (for a brief introduction to the animation capabilities of python, see this post (/blog/2012/08/18/matplotlib-animation-tutorial/)). The top panel shows the position-space wave function, while the bottom panel shows the momentum-space wave function.



0:00

Notice that the height of the potential barrier (denoted by the dashed line in the bottom panel) is far larger than the energy of the particle. Still, due to quantum effects, a small part of the wave function is able to tunnel through the barrier and reach the other side.

The python code used to generate this animation is included below. It's pretty long, but I've tried to comment extensively to make the algorithm more clear. If you're so inclined, you might try running the example and adjusting the potential or the input wave function to see the effect on the dynamics of the quantum system.

Schrodinger schrodinger.py download (/downloads/code/schrodinger.py)

```
"""
General Numerical Solver for the 1D Time-Dependent Schrodinger's equation.

author: Jake Vanderplas
email: vanderplas@astro.washington.edu
website: http://jakevdp.github.com
license: BSD
Please feel free to use and modify this, but keep the above information. Thanks!
"""

import numpy as np
from matplotlib import pyplot as pl
from matplotlib import animation
from scipy.fftpack import fft,ifft


class Schrodinger(object):
    """
    Class which implements a numerical solution of the time-dependent
    Schrodinger equation for an arbitrary potential
    """
    def __init__(self, x, psi_x0, V_x,
                 k0 = None, hbar=1, m=1, t0=0.0):
        """
        Parameters
        ----------
        x : array_like, float
            length-N array of evenly spaced spatial coordinates
        psi_x0 : array_like, complex
            length-N array of the initial wave function at time t0
        V_x : array_like, float
             length-N array giving the potential at each x
        k0 : float
            the minimum value of k.  Note that, because of the workings of the
            fast fourier transform, the momentum wave-number will be defined
            in the range
              k0 < k < 2*pi / dx
            where dx = x[1]-x[0].  If you expect nonzero momentum outside this
            range, you must modify the inputs accordingly.  If not specified,
            k0 will be calculated such that the range is [-k0,k0]
        hbar : float
            value of planck's constant (default = 1)
        m : float
            particle mass (default = 1)
        t0 : float
            initial tile (default = 0)
        """
        # Validation of array inputs
        self.x, psi_x0, self.V_x = map(np.asarray, (x, psi_x0, V_x))
        N = self.x.size
        assert self.x.shape == (N,)
        assert psi_x0.shape == (N,)
        assert self.V_x.shape == (N,)

        # Set internal parameters
        self.hbar = hbar
        self.m = m
        self.t = t0
        self.dt_ = None
        self.N = len(x)
        self.dx = self.x[1] - self.x[0]
        self.dk = 2 * np.pi / (self.N * self.dx)

        # set momentum scale
        if k0 == None:
            self.k0 = -0.5 * self.N * self.dk
        else:
            self.k0 = k0
        self.k = self.k0 + self.dk * np.arange(self.N)

        self.psi_x = psi_x0
        self.compute_k_from_x()

        # variables which hold steps in evolution of the
        self.x_evolve_half = None
        self.x_evolve = None
        self.k_evolve = None

        # attributes used for dynamic plotting
        self.psi_x_line = None
        self.psi_k_line = None
        self.V_x_line = None

    def _set_psi_x(self, psi_x):
        self.psi_mod_x = (psi_x * np.exp(-1j * self.k[0] * self.x)
                          * self.dx / np.sqrt(2 * np.pi))

    def _get_psi_x(self):
        return (self.psi_mod_x * np.exp(1j * self.k[0] * self.x)
                * np.sqrt(2 * np.pi) / self.dx)

    def _set_psi_k(self, psi_k):
        self.psi_mod_k = psi_k * np.exp(1j * self.x[0]
                                        * self.dk * np.arange(self.N))

    def _get_psi_k(self):
        return self.psi_mod_k * np.exp(-1j * self.x[0] *
                                        self.dk * np.arange(self.N))

    def _get_dt(self):
        return self.dt_

    def _set_dt(self, dt):
        if dt != self.dt_:
            self.dt_ = dt
            self.x_evolve_half = np.exp(-0.5 * 1j * self.V_x
                                         / self.hbar * dt )
            self.x_evolve = self.x_evolve_half * self.x_evolve_half
            self.k_evolve = np.exp(-0.5 * 1j * self.hbar /
                                   self.m * (self.k * self.k) * dt)

    psi_x = property(_get_psi_x, _set_psi_x)
    psi_k = property(_get_psi_k, _set_psi_k)
    dt = property(_get_dt, _set_dt)

    def compute_k_from_x(self):
        self.psi_mod_k = fft(self.psi_mod_x)

    def compute_x_from_k(self):
        self.psi_mod_x = ifft(self.psi_mod_k)

    def time_step(self, dt, Nsteps = 1):
        """
        Perform a series of time-steps via the time-dependent
        Schrodinger Equation.

        Parameters
        ----------
        dt : float
            the small time interval over which to integrate
        Nsteps : float, optional
            the number of intervals to compute.  The total change
            in time at the end of this method will be dt * Nsteps.
            default is N = 1
        """
        self.dt = dt

        if Nsteps > 0:
            self.psi_mod_x *= self.x_evolve_half

        for i in xrange(Nsteps - 1):
            self.compute_k_from_x()
            self.psi_mod_k *= self.k_evolve
            self.compute_x_from_k()
            self.psi_mod_x *= self.x_evolve

        self.compute_k_from_x()
        self.psi_mod_k *= self.k_evolve

        self.compute_x_from_k()
        self.psi_mod_x *= self.x_evolve_half

        self.compute_k_from_x()

        self.t += dt * Nsteps


######################################################################
# Helper functions for gaussian wave-packets

def gauss_x(x, a, x0, k0):
    """
    a gaussian wave packet of width a, centered at x0, with momentum k0
    """
    return ((a * np.sqrt(np.pi)) ** (-0.5)
            * np.exp(-0.5 * ((x - x0) * 1. / a) ** 2 + 1j * x * k0))

def gauss_k(k,a,x0,k0):
    """
    analytical fourier transform of gauss_x(x), above
    """
    return ((a / np.sqrt(np.pi))**0.5
            * np.exp(-0.5 * (a * (k - k0)) ** 2 - 1j * (k - k0) * x0))


######################################################################
# Utility functions for running the animation

def theta(x):
    """
    theta function :
      returns 0 if x<=0, and 1 if x>0
    """
    x = np.asarray(x)
    y = np.zeros(x.shape)
    y[x > 0] = 1.0
    return y

def square_barrier(x, width, height):
    return height * (theta(x) - theta(x - width))

######################################################################
# Create the animation

# specify time steps and duration
dt = 0.01
N_steps = 50
t_max = 120
frames = int(t_max / float(N_steps * dt))
```

```
# specify constants
hbar = 1.0   # planck's constant
m = 1.9      # particle mass

# specify range in x coordinate
N = 2 ** 11
dx = 0.1
x = dx * (np.arange(N) - 0.5 * N)

# specify potential
V0 = 1.5
L = hbar / np.sqrt(2 * m * V0)
a = 3 * L
x0 = -60 * L
V_x = square_barrier(x, a, V0)
V_x[x < -98] = 1E6
V_x[x > 98] = 1E6

# specify initial momentum and quantities derived from it
p0 = np.sqrt(2 * m * 0.2 * V0)
dp2 = p0 * p0 * 1./80
d = hbar / np.sqrt(2 * dp2)

k0 = p0 / hbar
v0 = p0 / m
psi_x0 = gauss_x(x, d, x0, k0)

# define the Schrodinger object which performs the calculations
S = Schrodinger(x=x,
                psi_x0=psi_x0,
                V_x=V_x,
                hbar=hbar,
                m=m,
                k0=-28)

######################################################################
# Set up plot
fig = pl.figure()

# plotting limits
xlim = (-100, 100)
klim = (-5, 5)

# top axes show the x-space data
ymin = 0
ymax = V0
ax1 = fig.add_subplot(211, xlim=xlim,
                      ylim=(ymin - 0.2 * (ymax - ymin),
                            ymax + 0.2 * (ymax - ymin)))
psi_x_line, = ax1.plot([], [], c='r', label=r'$|\psi(x)|$')
```

```
V_x_line, = ax1.plot([], [], c='k', label=r'$V(x)$')
center_line = ax1.axvline(0, c='k', ls=':',
                          label = r"$x_0 + v_0t$")

title = ax1.set_title("")
ax1.legend(prop=dict(size=12))
ax1.set_xlabel('$x$')
ax1.set_ylabel(r'$|\psi(x)|$')

# bottom axes show the k-space data
ymin = abs(S.psi_k).min()
ymax = abs(S.psi_k).max()
ax2 = fig.add_subplot(212, xlim=klim,
                      ylim=(ymin - 0.2 * (ymax - ymin),
                            ymax + 0.2 * (ymax - ymin)))
psi_k_line, = ax2.plot([], [], c='r', label=r'$|\psi(k)|$')

p0_line1 = ax2.axvline(-p0 / hbar, c='k', ls=':', label=r'$\pm p_0$')
p0_line2 = ax2.axvline(p0 / hbar, c='k', ls=':')
mV_line = ax2.axvline(np.sqrt(2 * V0) / hbar, c='k', ls='--',
                      label=r'$\sqrt{2mV_0}$')
ax2.legend(prop=dict(size=12))
ax2.set_xlabel('$k$')
ax2.set_ylabel(r'$|\psi(k)|$')

V_x_line.set_data(S.x, S.V_x)

######################################################################
# Animate plot
def init():
    psi_x_line.set_data([], [])
    V_x_line.set_data([], [])
    center_line.set_data([], [])

    psi_k_line.set_data([], [])
    title.set_text("")
    return (psi_x_line, V_x_line, center_line, psi_k_line, title)

def animate(i):
    S.time_step(dt, N_steps)
    psi_x_line.set_data(S.x, 4 * abs(S.psi_x))
    V_x_line.set_data(S.x, S.V_x)
    center_line.set_data(2 * [x0 + S.t * p0 / m], [0, 1])

    psi_k_line.set_data(S.k, abs(S.psi_k))
    title.set_text("t = %.2f" % S.t)
    return (psi_x_line, V_x_line, center_line, psi_k_line, title)

# call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                                frames=frames, interval=30, blit=True)


# uncomment the following line to save the video in mp4 format.  This
# requires either mencoder or ffmpeg to be installed on your system

#anim.save('schrodinger_barrier.mp4', fps=15, extra_args=['-vcodec', 'libx264'])

pl.show()
```

Edit: I changed the legend font properties to play nicely with older matplotlib versions. Thanks to Yann for pointing it out.

Posted by Jake Vanderplas Sep 05, 2012

Tweet G+1 15

Comments

24 Comments Pythonic Perambulations 1 Login

Recommend 10 Share Sort by Best

Join the discussion…

Some Physicist • 3 years ago

Very nice code and write-up. I think there's a pair of typos in the end of the Numerical Considerations section. I think the sign in the exponential next to the psi(k) should be positive. That's what you have in the code, and that's what I get from going through the derivation

3 • Reply • Share ›

Some Physicist Some Physicist • 3 years ago

Sorry if this is too detailed, but I think the method implemented in the code is not exactly what is described in the text. The text describes the Strang splitting method (1/2,1,1/2), while the code (for N_steps > 1) seems to start with part of a Strang step and then switch to the Lie-Trotter method (1,1) for (N_steps - 1) steps, then finish Strang. It works fine this way, but it was confusing at first. Any reason for it?

4 • Reply • Share ›

Jotaf • 4 years ago

Very inspiring! However, I have a small question about the underlying theory.

"...this probability is not a limitation of our knowledge of the system, but a reflection of an unavoidable uncertainty about the position and time of events in the realm of the very small."

I always had trouble with this sort of statement. My background is mostly statistics, so please pardon my ignorance.

I get that you cannot get an arbitrarily accurate measurement on both position and velocity simultaneously; their probability distributions are related in a way that prevents this. I know you didn't say this explicitly, but why do many physicists jump straight from there to the conclusion that particles are waves?

I can describe my knowledge of the state variables of a tennis ball using a joint probability distribution, but that doesn't mean it's a wave. The probability distribution is just a way to reason about uncertainty. In the quantum tunneling example you gave, it just means that 10% of the time a particle makes it through the barrier, not that 10% of every particle's wave makes it through. It seems contradictory to how probability works in all other realms of science, hence my confusion. Thank you!

1 • Reply • Share ›

Pedram ➦ Jotaf • 3 years ago

Your confusion is not uncommon. Einstein had the same problems believing what you're having trouble with.
Imagine a light source, a wall with two slits in it, and a surface that captures the light that bounces off it on the other end. Now, you ask yourself is light a wave or a particle? If it's a wave, then we should see an interference pattern on the surface, similar to the wave patterns in a lake surrounded by rocks. You turn on the light and you see an interference pattern. Brilliant, so light is a wave.

Now, try the same experiment but this time with a single photon. Surely, a single photon must go through only one or the other slit, and should exhibit no interference at all? But, when you send one photon at a time, you find the same interference pattern on the wall.

Now, how could a single photon interfere with itself? Because the wave function of the photon has not collapsed and so there's a fundamental and unbeatable uncertainty that surrounds its path. This isn't because our instruments are too clunky and imprecise but is due to the very nature of quantum particles.

Read more here: http://www.thekeyboard.org.uk/...

1 • Reply • Share ›

some dude • 9 months ago

great code - what im not seeing is the default value for k0 (k0=-28)
is there a reason why you are not using k0=None - which leads to k=(arange(N)-N/2)*dk..?
thx

• Reply • Share ›

杨通 • 9 months ago

I found there are some sign mistakes in the section of "Numerical Considerations". In this part, the exponent of the bracketed exponential term on the right-hand side of the last third equation (the inverse Fourier transform: K space--->X space) should be positive. And for the last equation, the discrete Fourier pair, the exponent of the exponential term on the right-hand side should also be positive.

• Reply • Share ›

杨通 • 9 months ago

I just follow your idea to solve the same problem using c instead of python. But I found that my gaussian package does not propagate with time. It just stays at the initial place. I have submitted my problem to http://stackoverflow.com/quest... .Can you help me to find what's wrong with my c codes?

Reply • Share ›

Farzan1World • a year ago

Fantastic job. I have become facinated about learning QM, math and all, and this is great simulation.

Reply • Share ›

Jean-Marc Sparenberg • 2 years ago

Great code! This is nearly exactly what I'm looking for for my quantum
physics classes. The only regret I have is that the phase of the
(complex) wave function is not represented, only its modulus.
Nevertheless, the phase contains very interesting physical content
(interferences, momentum, and so on).

The most elegant way I've found to plot the complex phase is to use rainbow colors, as done in Visual Quantum Mechanics by Bernd Thaller http://www.uni-graz.at/imawww/...
Unfortunately, this is written in Mathematica. Would you (or someone?) know of such a modulus-phase representation in Python? Many thanks in advance!

Reply • Share ›

Phoneli • 2 years ago

This a great introduction on the numerical method in solving schrodinger eqn. I just have a quick question: in your statement, you said we first solve the psi in x-space with V involved only (in this step the kinetic energy part ignored), then you solve the equation in k-space with kinetic energy term involved only but with V ignored. In what reason we can do that? For example, why we could ignore V while solving the k-space equation? why we could ignore p^2/2m while solving the x-space equation?

Reply • Share ›

Mark • 2 years ago

In the browser view of python (Ipython notebook) what line is missing in order to have animation in screen.
much like when you have charts in place with %matplotlib inline

Reply • Share ›

User • 3 years ago

Excellent application of Python.

Reply • Share ›

Hongze • 3 years ago

Great job Jake. In order to run the script on Mac OS, I added two lines before YOUR import command:

```
import matplotlib
matplotlib.use('TkAgg')
```

Reply • Share ›

Suman Acharya • 3 years ago

Hi jake,

it would be very helpful if you add more comment on the python code describing the mechanism. I found it little difficult to understand the way the wave function is being computed.

Reply • Share ›

Davide Olianas • 3 years ago

Thanks, this is great! I just had to change xrange(...) to range(...) in line 141 to make it work.

Reply • Share ›

Александр Емелин • 3 years ago

this is great, impressive, amazing! thank you very much!

Reply • Share ›

Fan Du • 3 years ago

Running the script on OS X Lion with Python 2.7.3 and matplotlib 1.2.0 keeps getting error: 'Line2D' object has no attribute '_animated'. Would you help me ?

Reply • Share ›

arymanus • 4 years ago

This code is awesome!!!

Reply • Share ›

Nguyễn Cảnh Hiếu • 4 years ago

Amazing, thanks !!!

Reply • Share ›

slam3085 • 4 years ago

Wonderful example and amazing blog. Thank you.

Reply • Share ›

Yann • 4 years ago

Isn't there an error in the above code ? Lines 256 and 272 doesn't execute because the 'fontsize' keyword doesn't exist for the legend method. Is it supposed to work fine ? Thanks for any answer.

Reply • Share ›

jakevdp Mod → Yann • 4 years ago

Hi Yann,
the fontsize keyword is in newer versions of matplotlib. In older versions, you can use legend(prop=dict(size=12)). I'm not entirely sure which version that was added, but note that you need v1.1+ for the animation tools.

Reply • Share ›

Giancarlo reyes fernandez • 4 years ago

chebre!!!

Reply • Share ›

Hauser Quaid • 4 years ago

Great job!

^ | ˅ • Reply • Share ›

ALSO ON **PYTHONIC PERAMBULATIONS**

Is Seattle Really Seeing an Uptick In Cycling?

18 comments • 2 years ago•

Avat **Jim Boyle** — I am really intrigued by your casual comment to use random forests for further analysis. I have used random forests in a model

Why Python is Slow: Looking Under the Hood

58 comments • 2 years ago•

Avat **Bennie The Bouncer** — This post illustrates another major problem with Python: significant whitespace. This forum doesn't preserve leading

Kernel Density Estimation in Python

28 comments • 3 years ago•

How Bad Is Your Colormap?

6 comments • 2 years ago•

Recent Posts

- Analyzing Pronto CycleShare Data with Python and Pandas (https://jakevdp.github.io/blog/2015/10/17/analyzing-pronto-cycleshare-data-with-python-and-pandas/)
- Out-of-Core Dataframes in Python: Dask and OpenStreetMap (https://jakevdp.github.io/blog/2015/08/14/out-of-core-dataframes-in-python/)
- Frequentism and Bayesianism V: Model Selection (https://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/)
- Learning Seattle's Work Habits from Bicycle Counts (Updated!) (https://jakevdp.github.io/blog/2015/07/23/learning-seattles-work-habits-from-bicycle-counts/)
- The Model Complexity Myth (https://jakevdp.github.io/blog/2015/07/06/model-complexity-myth/)

Follow @jakevdp 10.3K followers