Overview

Our focus today will be on solving the time-dependent Schrodinger equation. In other words, we want to see wavefunction evolve in time. To do this we will need to solve a linear algebra problem

The Crank-Nicolson algorithm

We'll start by writing down the very-familiar time-dependent Schrodinger equation:

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

and discretize it like we have done so many times before. This time, you might notice that things look a little different. The steps for discretizing this equation are given below:

$$i\hbar\frac{\psi_j^{n+1}-\psi_j^n}{\Delta t} = -\frac{\hbar^2}{2m}\frac{\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n}{dx^2} + V(x_j)\psi_j^n \tag{1}$$

$$i\hbar\frac{\psi_j^{n+1}-\psi_j^n}{\Delta t} = -\frac{\hbar^2}{2m}\frac{\frac{\psi_{j+1}^{n+1}+\psi_{j+1}^n}{2} - 2\frac{\psi_j^{n+1}+\psi_j^n}{2} + \frac{\psi_{j-1}^{n+1}+\psi_{j-1}^n}{2}}{dx^2} + V(x_j)\frac{\psi_j^{n+1}+\psi_j^n}{2} \tag{2}$$

$$i\hbar\frac{\psi_j^{n+1}-\psi_j^n}{\Delta t} = -\frac{\hbar^2\left(\psi_{j+1}^{n+1}+\psi_{j+1}^n - 2\psi_j^{n+1} - 2\psi_j^n + \psi_{j-1}^{n+1}+\psi_{j-1}^n\right)}{4mdx^2} + V(x_j)\frac{\psi_j^{n+1}+\psi_j^n}{2} \tag{3}$$

$$\psi_j^{n+1}\left(\frac{i\hbar}{\Delta t} - \frac{\hbar^2}{2mdx^2} - \frac{V(x_j)}{2}\right) + \psi_{j+1}^{n+1}\frac{\hbar^2}{4mdx^2} + \psi_{j-1}^{n+1}\frac{\hbar^2}{4mdx^2}$$
$$= \psi_j^n\left(\frac{i\hbar}{\Delta t} + \frac{\hbar^2}{2mdx^2} + \frac{V(x_j)}{2}\right) - \psi_{j+1}^n\frac{\hbar^2}{4mdx^2} - \psi_{j-1}^n\frac{\hbar^2}{4mdx^2} \tag{4}$$

1. Take the time to understand each step of the math above. Specifically, make sure you understand what happened from equation (1) and (2)
2. When we solved the wave equation, the next step was to rearrange the terms and solve for ψ_j^{n+1} hoping that the r.h.s didn't have any terms with $n+1$ in the subscript. Convince yourself that this is not possible in this case.

Since we can't rearrange the terms to solve for ψ_j^{n+1}, we must seek another way to a solution. Notice that equation (4) can be written in matrix form like this:

$$\mathbf{A}\psi^{n+1} = \mathbf{B}\psi^n$$

where the elements of $\mathbf{A}$ are given by

$$A_{j,k} = 0 \quad \text{except for :}$$

$$A_{j,j-1} = \frac{\hbar^2}{4mdx^2} \quad ; \quad A_{j,j} = \left(\frac{i\hbar}{\Delta t} - \frac{\hbar^2}{2mdx^2} - \frac{V(x_j)}{2}\right) \quad ; \quad A_{j,j+1} = \frac{\hbar^2}{4mdx^2}$$

and the elements of $\mathbf{B}$ are given by

$$B_{j,k} = 0 \quad \text{except for :}$$

$$B_{j,j-1} = -\frac{\hbar^2}{4mdx^2} \quad ; \quad B_{j,j} = \left(\frac{i\hbar}{\Delta t} + \frac{\hbar^2}{2mdx^2} + \frac{V(x_j)}{2}\right) \quad ; \quad B_{j,j+1} = -\frac{\hbar^2}{4mdx^2}$$

This means that each time we want to advance the wave into the future, we must form the appropriate matrices $\mathbf{A}$ and $\mathbf{B}$ and solve the linear algebra problem (read that last sentence again!)

Choose one of the problems below to complete:

1. Study the tunneling of a wave packet through a square step barrier. Be sure to vary the height of the barrier as part of your investigation.
2. Study the time evolution of a wave packet in a quadratic potential ($V = kx^2$).
3. Study the time evolution of a wave packet in an infiinte square well.
4. Study the reflection of a wave packet from a potential wall. A potential wall is essentially a step function: the potential is zero everywhere and then suddenly increases to a very large value.
5. Study the reflection of a wave packet from a potential cliff. A potential cliff is essentially a step function: the potential is zero everywhere and then suddenly decreases to a very large negative value.

Below you will find some tips/guidelines as well as a code template to help you get started. Read my comments carefully and ask any questions that come up.

1. Work in natural units where $\hbar = m = 1$.
2. The equations above tell you how to populate everything but the first and last row of the matrices. To enforce the wavefunction to be zero at the boundary, you should insert a 1 at the upper left and lower right entries of matrix A. (Just like we did on Tuesday when we solved the eigenvalue problem.)
3. Let your domain be $-L < x < L$ and start with a localized wave packet:

$$\psi(x,0) = \frac{1}{\sqrt{\sigma\sqrt{\pi}}}e^{\frac{ipx}{\hbar}}e^{\frac{-x^2}{2\sigma}}$$

   You'll have to play with values of σ, p, and L to ensure that your wavepacket is sufficiently localized.
4. The energy of the wave packet descibed above is given by:

$$\langle E\rangle = \int_{-\infty}^{\infty}\psi^*\hat{H}\psi dx = \frac{p^2}{2m} + \frac{\hbar^2}{4m\sigma^2}$$

   (I did the integration for you.) For those studying tunneling, you should investigate what happens when the barrier height V is greater than and less than this energy.
5. Remember that the wavefunction is complex and can't be plotted. Only $\psi^*(x)\psi(x)$ can be plotted. Use numpy's "conjugate" function to find the complex conjugate.
6. You'll notice that the wave packet broadens over time. You'll want to investigate the how the rate of broadening is affected by the initial width of the packet. Can you explain the results?
7. You should also notice that there is not instability in this algorithm. You can makde dt as big as you like and it won't blow up. Cool huh!

Problem 1 - Square Barrier Tunneling Effect on Ψ

I will be time evolving the wave packet as it pertains to representing an electron of mass m_e and momentum p in 1 dimension. To convert all to natural units, I will be making these simplifying assumptions

$$\hbar = m_e = 1$$

With these assumptions, I will have to be conservative in my approach to usage of the parameters p, σ, and L. In other words, they should be on a similar order of magnitude.

Additionally, I will be enforcing a boundary condition such that the particle's wavefunction is bound within an infinite potential at $-5 \leq x \leq 5$ to cause the wavefunction to reflect back and give several interaction results of the wavefunction with the potential barrier. What this means is that the matrix B has zeros on its top left and bottom right corners. In other words, the wavefunction vanishes to zero at those boundaries.

Method - Crank Nicolson

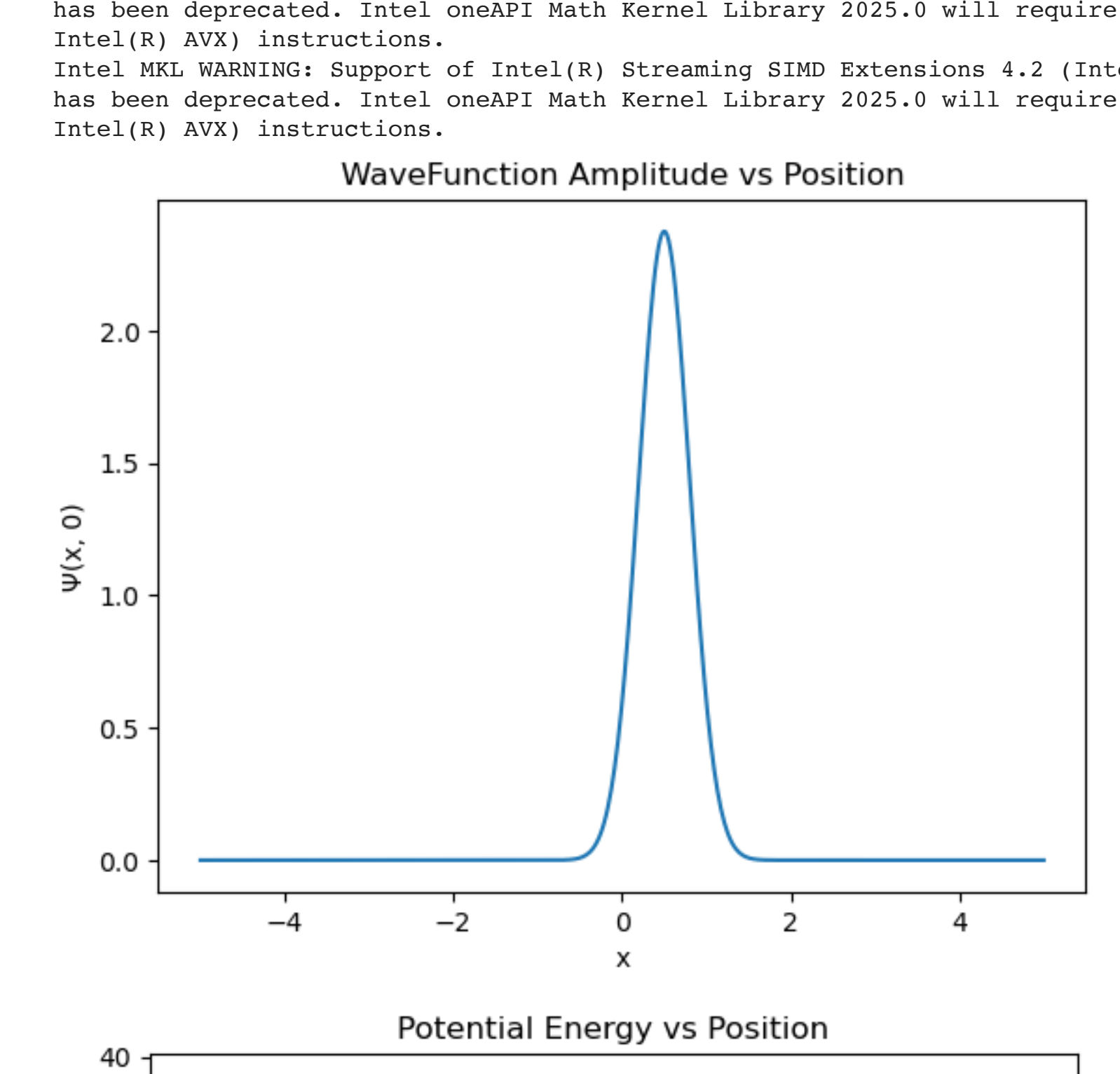
I will be making use of equation (4) to modify my wavefunction $\Psi(x,t)$.

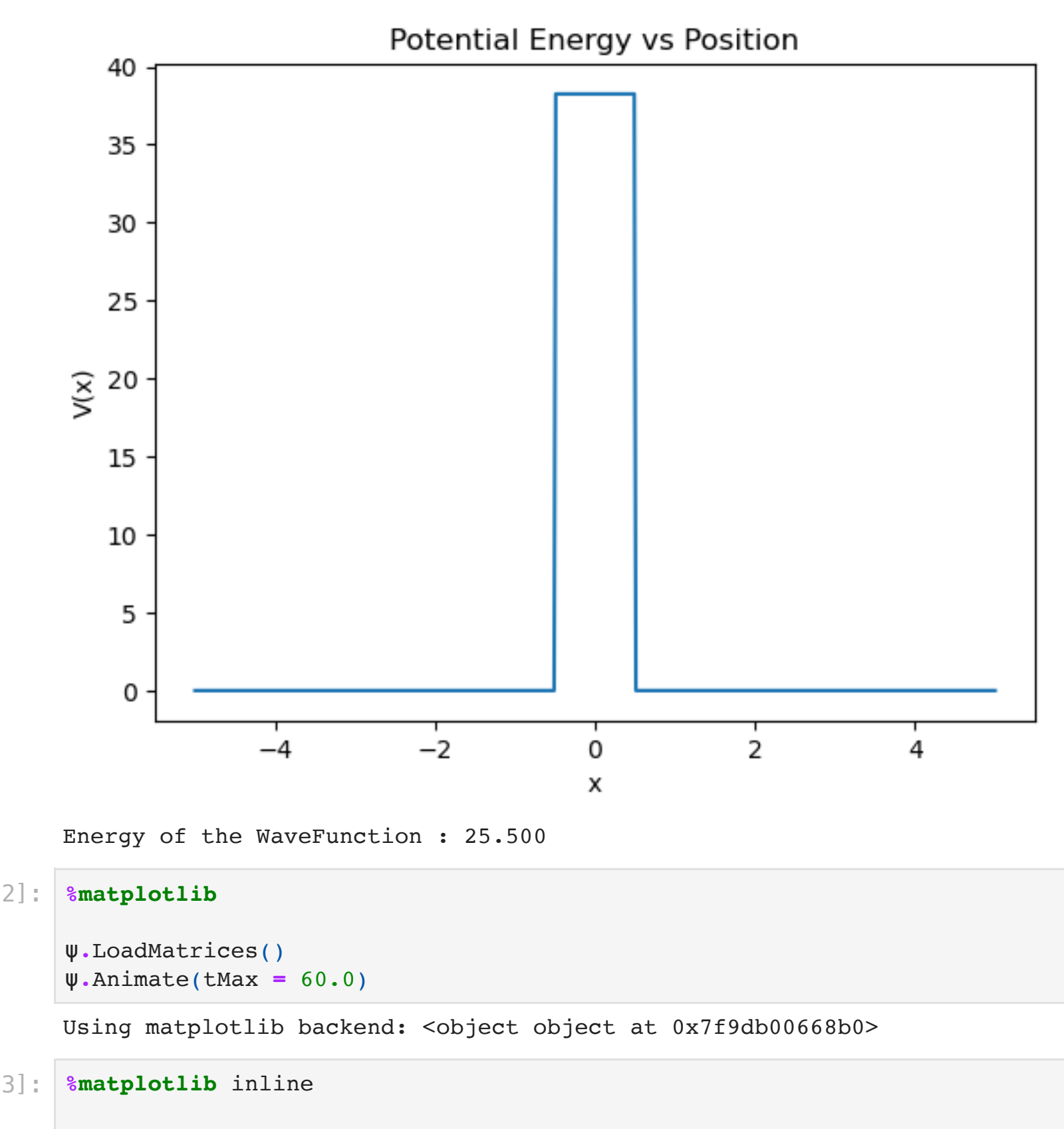
Case 1 - V > E

I expect the solution to the Schrodinger equation in the barrier region to have a dipping shape, demonstrating a type of exponential solution which connects to oscillatory or sinusoidal solutions on either opposing end of the barrier.

In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

class CrankNicolson:
    def __init__(self,
                 nSpacial : int,
                 x0 : float,
                 dt : float,
                 σ : float,
                 p : float,
                 L : float,
                 high : bool
            ):
        """
            nSpacial : int -> number of spacial points to plot (width of diagram)
            dt : float -> time step size
            σ : float -> a parameter to control width of initial wave packet (smaller σ means wider wave packet
            p : float -> the initial momentum of the particle-wavepacket
            L : float -> half the width of the diagram to plot
        """

        self.nSpacial = nSpacial
        self.dt = dt
        self.x, self.dx = np.linspace(- L, L, nSpacial, retstep = True)
        self.σ = σ
        self.p = p
        self.L = L
        self.Ψ = self.InitializeWaveFunction(x0)
        self.E = p * p / 2 + 1 / (4 * σ * σ)
        # This means make the potential more than E if true, else make it half of E
        self.VBoundary = 1.5 * self.E if high else 0.5 * self.E

    def V(self, x):
        # Simple step function -> @ x = 0, there is a boundary. On the positive side, V = 10
        # and on the negative side, V = 0
        return self.VBoundary if abs(x) <= 0.5 else 0


    def InitializeWaveFunction(self, x0 : float):
        # x0 represents an offset from center to move the wave to the right or left
        return 1 / (self.σ * np.pi ** 0.5) ** 0.5 \
            * np.exp(1j * self.p * (x0 - self.x)) * np.exp(- (x0 - self.x) * (x0 - self.x) / (2 * self.σ))

    def LoadMatrices(self):
        # Build matrix A and B (given above) for your particlar problem.
        # Call them A and self.B
        self.A = np.zeros((self.nSpacial, self.nSpacial), dtype = complex)

        # Set boundary conditions and write out the Hamiltonian terms
        self.A[0, 0], self.A[-1, -1] = 1.0, 1.0
        for i in range(1, self.nSpacial - 1):
            self.A[i, i - 1] = 1 / (4 * self.dx * self.dx)
            self.A[i, i] = (complex(0, 1 / self.dt) - 1 / (2 * self.dx * self.dx) - self.V(self.x[i]) / 2)
            self.A[i, i + 1] = 1 / (4 * self.dx * self.dx)

        self.B = np.zeros((self.nSpacial, self.nSpacial), dtype = complex)
        # Since I want the edges to be bound such that the wave packet has to keep reflecting back and forth
        # inside, I leave the top left corner and bottom right corners set to 0

        # And I adjust the middle terms to account for the regular B terms of the time-dependent Schrodinger eq
        for i in range(1, self.nSpacial - 1):
            self.B[i, i - 1] = - 1 / (4 * self.dx * self.dx)
            self.B[i, i] = (complex(0, 1 / self.dt) + 1 / (2 * self.dx * self.dx) + self.V(self.x[i]) / 2)
            self.B[i, i + 1] = - 1 / (4 * self.dx * self.dx)

        # Matrices A and B are sparse: Most of their entries are zero
        # and the only non-zero entries are along the main diagonal and the
        # upper and lower diagonal.  We can use this to our advantage and speed up
        # the solution process.  To that end, we'll assemble an array that
        # stores only the diagonal terms of A, and call is self.ab for later use.

        ud = np.insert(np.diag(self.A, 1), 0, 0) # upper diagonal
        d = np.diag(self.A) # main diagonal
        ld = np.insert(np.diag(self.A, - 1), - 1, 0) # lower diagonal
        # simplified matrix
        self.ab = np.matrix([ud, d, ld])
        return None

    def Animate(self, tMax : float):
        from numpy import dot, linspace, conjugate, real
        from scipy.linalg import solve_banded

        counter = 0
        t = 0

        while t < tMax:
            normalize = sum(self.Ψ * conjugate(self.Ψ)) * self.dx

            b = dot(self.B, self.Ψ)
            self.Ψ = solve_banded((1, 1), self.ab, b)

            plt.plot(self.x, real(1 / normalize * conjugate(self.Ψ) * self.Ψ), 'r.-')
            plt.plot(self.x, [self.V(x) for x in self.x])
            plt.ylim(- 0.1, 1)
            plt.draw()
            plt.pause(1e-6)
            plt.clf()

            counter += 1
            t += self.dt

        plt.close()
        return None

L = 5.0
nSpacial = 500
x0 = 0.5
dt = 1e-1
σ = 1e-1
p = 1.0

ψ = CrankNicolson(nSpacial = nSpacial, x0 = x0, dt = dt, σ = σ, p = p, L = L, high = True)

# Display the WaveFunction and the potential energy in the system to ensure that the code is working properly
plt.plot(ψ.x, ψ.Ψ.real)
plt.title('WaveFunction Amplitude vs Position')
plt.xlabel('x')
plt.ylabel('Ψ(x, 0)')
plt.show()

plt.plot(ψ.x, [ψ.V(x) for x in ψ.x])
plt.title('Potential Energy vs Position')
plt.xlabel('x')
plt.ylabel('V(x)')
plt.show()

print(f"Energy of the WaveFunction : {ψ.E:.3f}")
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors
has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (
Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors
has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (
Intel(R) AVX) instructions.
```



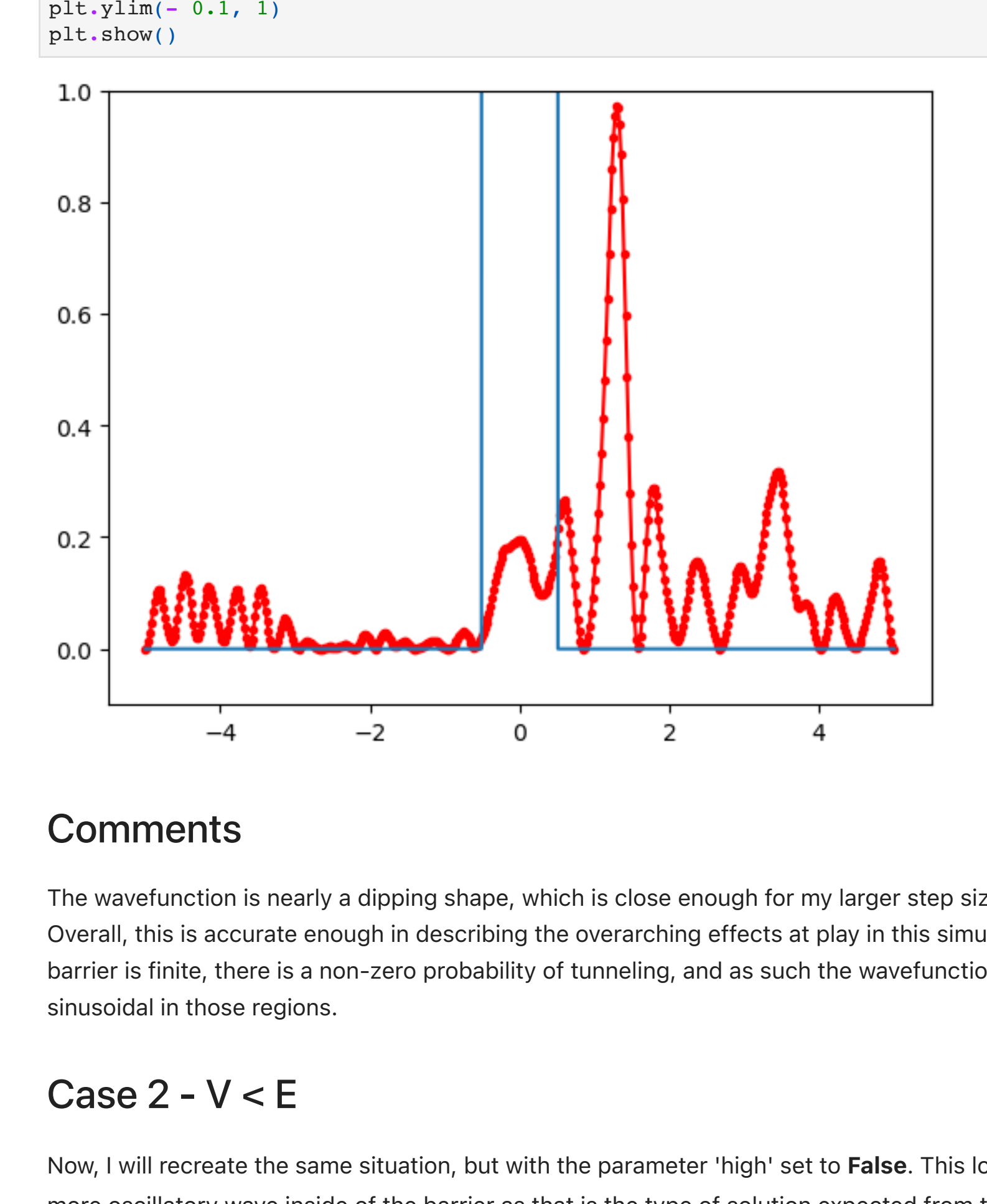



```
Energy of the WaveFunction : 25.500
```

In [2]:
```
%matplotlib

ψ.LoadMatrices()
ψ.Animate(tMax = 60.0)
```

```
Using matplotlib backend: <object object at 0x7f9db00668b0>
```

In [3]:
```
%matplotlib inline

from numpy import dot, linspace, conjugate, real
from scipy.linalg import solve_banded

normalize = sum(ψ.Ψ * conjugate(ψ.Ψ)) * ψ.dx

b = dot(ψ.B, ψ.Ψ)
ψ.Ψ = solve_banded((1, 1), ψ.ab, b)

plt.plot(ψ.x, real(1 / normalize * conjugate(ψ.Ψ) * ψ.Ψ), 'r.-')
plt.plot(ψ.x, [ψ.V(x) for x in ψ.x])
plt.ylim(- 0.1, 1)
plt.show()
```



Comments

The wavefunction is nearly a dipping shape, which is close enough for my larger step sizes I have defined for time and space. Overall, this is accurate enough in describing the overarching effects at play in this simulation with $V > E$. Since the potential barrier is finite, there is a non-zero probability of tunneling, and as such the wavefunction exists on both sides of the barrier and is sinusoidal in those regions.

Case 2 - V < E

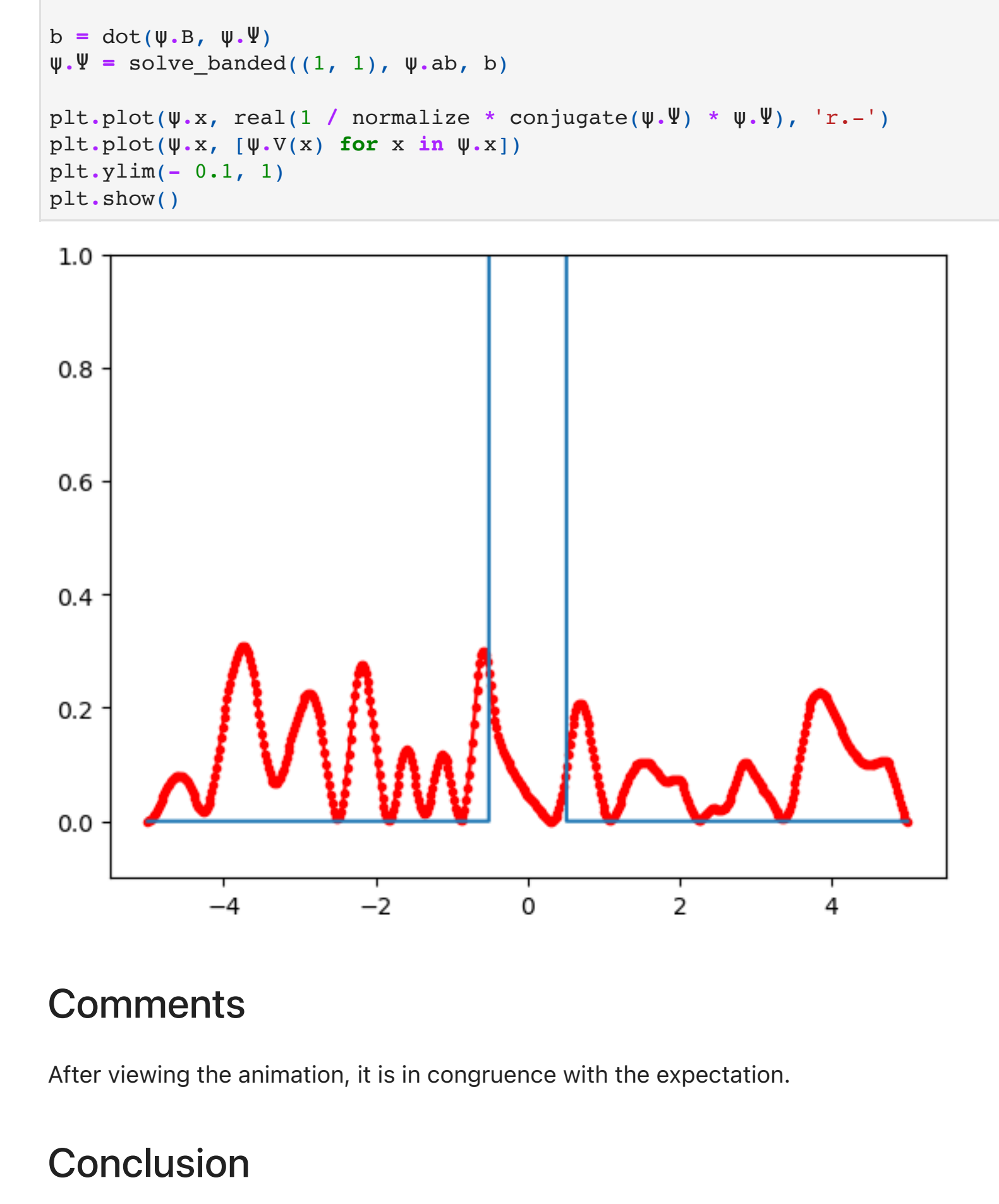
Now, I will recreate the same situation, but with the parameter 'high' set to **False**. This lowers $V = \frac{E}{2}$. In this case, I expect to see a more oscillatory wave inside of the barrier as that is the type of solution expected from the Schrodinger equation in that region.

In [4]:
```
# Try making the same animation, but with V small so that the barrier has
# less effect, and as such we say that the wave function has an oscillatory
# solution inside of the barrier, rather than exponential decay
ψ = CrankNicolson(nSpacial = nSpacial, x0 = x0, dt = dt, σ = σ, p = p, L = L, high = False)
```

In [5]:
```
%matplotlib

ψ.LoadMatrices()
ψ.Animate(tMax = 60.0)
```

```
Using matplotlib backend: MacOSX
```

In [6]:
```
%matplotlib inline

normalize = sum(ψ.Ψ * conjugate(ψ.Ψ)) * ψ.dx

b = dot(ψ.B, ψ.Ψ)
ψ.Ψ = solve_banded((1, 1), ψ.ab, b)

plt.plot(ψ.x, real(1 / normalize * conjugate(ψ.Ψ) * ψ.Ψ), 'r.-')
plt.plot(ψ.x, [ψ.V(x) for x in ψ.x])
plt.ylim(- 0.1, 1)
plt.show()
```



Comments

After viewing the animation, it is in congruence with the expectation.

Conclusion

The Crank-Nicolson method has proven to preserve the probability amplitudes quite amazingly well through this numerical time evolution of a type of 'diffusion equation' problem. The results are very interesting and this is certainly one of my personal favorite projects from this class. The quantum mechanics of the wavefunction are demonstrated very well and it is interesting to see how the Schrodinger equation describes particles from a purely wave-like nature in this simulation. Though I am not plotting the wavefunction, the time evolution of its absolute square (probability ditribution) does reflect underlying changes which are occuring at the wavefunction level.