Universidad Carlos III de Madrid

Wave Quantum Mechanics

t-Matrix python script guidelines

Daniel Sarcanean

MSc in Quantum Technologies and Engineering

;

Chapter 1

Introduction

In the present python script we look after a computational method that covers the t-matrix calculation for a simple uni-dimensional piece-wise potential, $V(x)$. Initially formulated by Peter C. Waterman in the 70s, its reliability and easier approach compared to other analytical methods for evaluating well potentials stands out through its mathematical elegance. For instance, the general solution for the transfer matrix method has the form of:

$$\Psi_b = \mathbf{M}(x_a, x_b)\Psi_a, \tag{1.1}$$

where $\mathbf{M}(x_a, x_b)$ is nothing else than the matrix resulting from adding up all the scattering terms that form the potential's region.

Chapter 2

Implementation

Although there is a `.md` appended to the code in git-hub explaining the code's fundamentals, I am an advocate that a good code does not need permanent comments to be understood, and so I will try to explain it in-depth in the present document just in case anything was not clear from its own documentation.

The main function that launches hierarchically all the others comes given by

```
def transferParameters(xa, xb, amplitude):
        initial_amplitude = amplitude
        finalMatrix = np.array([[1, 0], [0, 1]])
        for i in range(xb - xa):
                nextMatrix = tMatrixPositiveEnergy(i, i+1, amplitude)
                finalMatrix = np.matmul(nextMatrix, finalMatrix)
                amplitude = (np.linalg.det(finalMatrix) / finalMatrix[1,1]) *
                ↪  amplitude
                if energy < potentials[i]:
                        break
                elif energy > potentials[i] and energy < potentials[i+1]:
                        break
                else:
                        continue
        transmitance = np.linalg.det(finalMatrix) / finalMatrix[1,1]
        reflectance = finalMatrix[1,0] / finalMatrix[1,1]
        print(f'''
                Welcome to the t-Matrix calculator!

                ---------------------------------------------
                -             Your chosen output            -
                -          should have been plotted         -
                ---------------------------------------------

                Your initial amplitude is {initial_amplitude:.4f},
                and your transmitted and reflected amplitudes are
                {amplitude:.4f} and {amplitude*reflectance:.4f} respectively.
                In terms of probabilities, your transmitance is
                 ↪  {np.abs(transmitance)**2:.4f}
                and your reflectance is {np.abs(reflectance)**2:.4f}. Relating
                 ↪  this to your
                amplitude's probability you have {np.abs(initial_amplitude)**2 *
                 ↪  np.abs(transmitance)**2:.4f} for the
                Transmitance and {np.abs(initial_amplitude)**2 *
                 ↪  np.abs(reflectance)**2:.4f} for the Reflectance.

        ''')
        return finalMatrix
```

Herein, the aim was to create an effective `for` loop, so one only launches the calculations through $x_a \rightarrow x_b$, and nothing else. A `break` argument is implemented, so if the particle travels through a well/step where $E < V$ we directly assume that there is no propagation (although it is possible to make a tunneling correlation from this matrix). Anyways, the function finishes calculating both the last possible matrix (from the addition of every single step and *plateau*), and printing both the transmitance and reflectance (without taking into account the last potential[1]). To complement this function, a pair-to-pair calculation is done iteratively through the `for` loop. We can break it down into:

```
def tMatrixPositiveEnergy(xa, xb, amplitude):
        length = positions[xb] - positions[xa]
        A = amplitude

        if energy < potentials[xa]:
                k = np.sqrt(2 * me * (potentials[xa] - energy) / hbar**2)
                matrix = np.array([
                [0, 0],
                [0, np.exp(-k*length)]
                ])
                x = np.linspace(positions[xa], positions[xb+1], 200)
                plt.plot(x, energy + A*np.exp(-k*(x)), color='red')

        elif energy > potentials[xa] and energy < potentials[xb]:
                k1 = np.sqrt(2 * me * (energy - potentials[xa]) / hbar**2)
                k2 = np.sqrt(2 * me * (potentials[xb] - energy) / hbar**2)
                matrix = np.array([
                [0, (k1+1j*k2) / 2 * k1],
                [0, (k1-1j*k2) / 2 * k1]
                ])
```

[1]This is not exactly an impediment of the script, but a flaw from having a piece-wise potential. Its particularities will be commented at the Usage chapter.

```
                x = np.linspace(positions[xa], positions[xb+1], 200)
                plt.plot(x, energy + A*np.exp(-k2*(x - positions[xa])),
                 ↪  color='red')

        elif potentials[xa] == potentials[xb]:
                k = np.sqrt(2 * me * (energy - potentials[xa]) / hbar**2)
                matrix = np.array([
                [np.exp(1j*k*length), 0],
                [0, np.exp(-1j*k*length)]
                ])
                x = np.linspace(positions[xa], positions[xb], 200)
                #plt.plot(x, energy +(energy -
                 ↪  potentials[xb])*np.exp(-1j*k*(x)), color='green')
                plt.plot(x, energy + A*np.exp(1j*k*(x- positions[xa])),
                 ↪  color='red')

        else:
                k1 = np.sqrt(2 * me * (energy - potentials[xa]) / hbar**2)
                k2 = np.sqrt(2 * me * (energy - potentials[xb]) / hbar**2)
                matrix = (0.5)*np.array([
                [1 + k2/k1, 1 - k2/k1],
                [1 - k2/k1, 1 + k2/k1]
                ])

        return matrix
```

Where we see how each possibility is treated: a first step with $E < V$, a *plateau* with $E < V$, a *plateau* with $E > V$ and a step with $E > V$, and so for each of the cases there is a generated matrix that is given through their wavevector k components and length. Additionaly, a practical visual implementation has been introduced, so one can lastly see what it is being executed. The calculations are done following the exposed theoretical approach given in class, except for the cases where $E < V$, for which a real valued damped wave has been proposed as a solution of $\Psi(x)$.

Finally, the code executes a whole graphic environment where the axis can be visualized, as well as the piece-wise potential, and the "calculated" wave functions.

```
def graphicSolution(potentials, positions):
        for i in range(len(potentials)-1):
                if potentials[i] == potentials[i+1]:
                ax.hlines(y=potentials[i], xmin=positions[i],
                 ↪  xmax=positions[i+1])
                else:
                ax.vlines(x=positions[i], ymin=potentials[i],
                 ↪  ymax=potentials[i+1])
                ax.hlines(y=energy, xmin=positions[i], xmax=positions[i+1],
                 ↪  linestyle='--', color='green')
        plt.ylabel("Potential (hartrees)")
        plt.xlabel("Position (bohr radius)")
        plt.legend(frameon=False, fontsize=9)
        plt.show()
```

Chapter 3

Usage

To begin with the usage of the script, one can question why the units of the axis are so exotic. And so, to answer this, it has to be cleared out that over the whole script **atomic units** have been used. Why? Due to their simplicity and bigger computational friendliness.

Related to this clarification, several parameters (but not too many) can be tweaked in-code in order to perform the calculation one wants to. Below one can see the constants, from which the energy and amplitude (A) are recommended changing to desire.

```
###----Constants----###

plt.rcParams["figure.dpi"] = 300
fig, ax = plt.subplots(figsize=(10,5))
me =  1 # SI: 9.11E-31 kg
hbar =  1 / (2 * np.pi) # SI: 6.626E-34 / (2 * np.pi) J · s
energy = 12 / 27.21 # --> convertion to hartrees from eV
A = 0.1 # incident amplitude

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

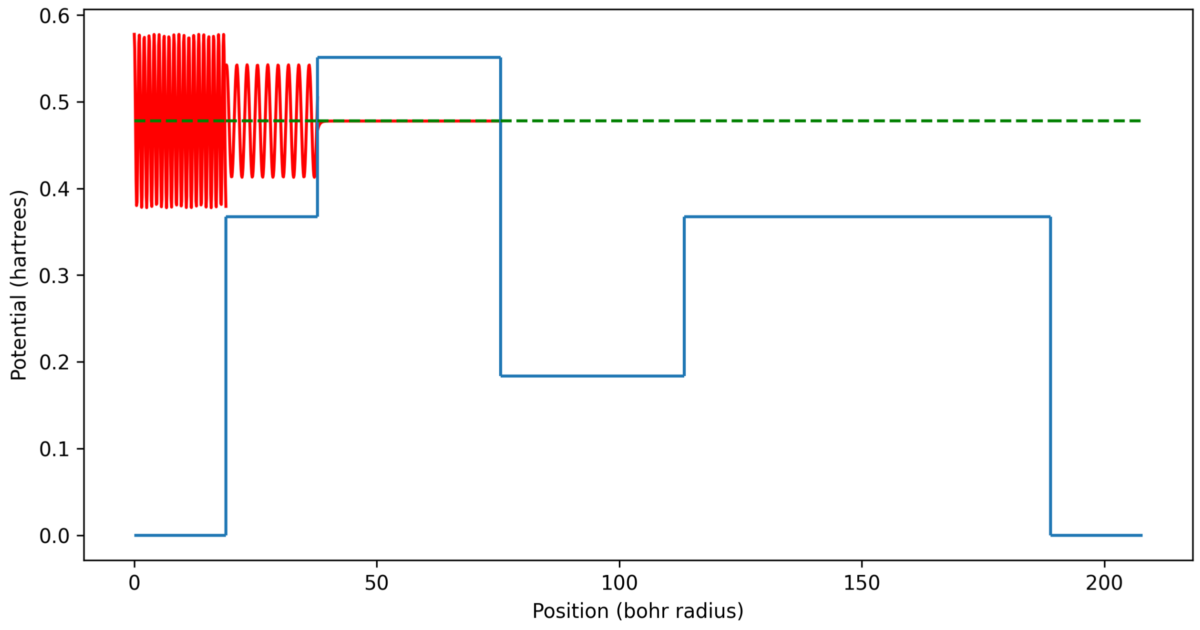
Apart from this, to launch the t-matrix method, one has to be in possession of a piece-wise potential database that can be read off a `.csv`. The program already covers this, and if you add to the same directory a `potentials.csv` file that is fashioned in the following manner

```
Potentials (eV);Positions (nm)
0;0
0;1
10;1
10;2
15;2
15;4
5;4
5;6
10;6
10;10
0;10
```

```
0;11
```

then it will run without a doubt. Of course, consider that, given a list, you have to know, as commented before, that it is not possible to evaluate the last potentials because "you have nothing to compare them to". To solve this one possible approach is to extend in 1-value your database, and add a newer *plateau* that keeps this potential, and so you will see the exact solution for the parameters you seek to know about.

Let us see an example. Given $E = 13$ eV, and the piece-wise potential appended above. Which is going to be the transmitance and reflectance if we evaluate over all the indexes? So, we begin by recalling the function `transferParameters(0,11,A)` from $0 \to 3$, because we have indexed the potentials in this notation (take care: `xa` and `xb` are not positions, but indexes!). We are given the plot below:



from which we get the values `Transmitance=0` and `Reflectance=0.123`, and hence we see that the value $T + R = 1$ is not conserved. *Why?* To fulfill this, one has to be sure that its function will trespass the whole barrier, this is, that $E > V$ at any point. If not, the algorithm will cut the whole function at the barrier where it vanishes, in fact taking into account its proper components (and so resembling this null transmitance and positive reflectance), but without normalizing them in function of the potential, as $M_0 \neq M_{V_{\max}}$. Now, what would happen if instead we used an $E = 20$ eV? Lets find out!

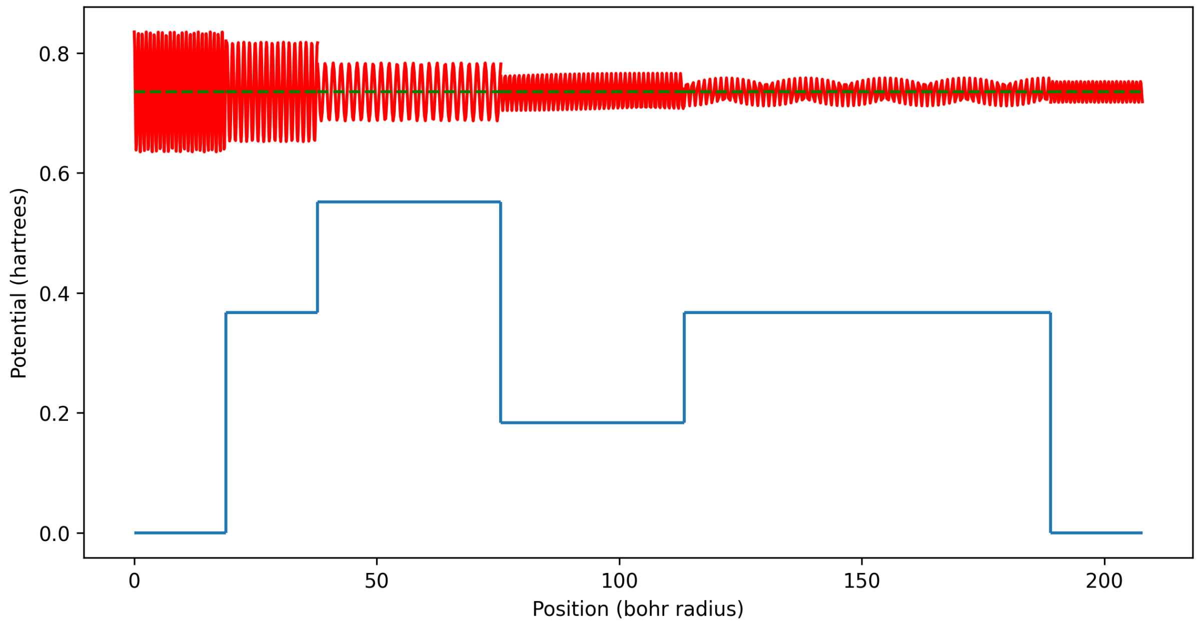
If we append the text output, we can see that the results are quite convincing:

```
Welcome to the t-Matrix calculator!

---------------------------------------------
-           Your chosen output              -
-         should have been plotted          -
---------------------------------------------
```

```
Your initial amplitude is 0.1000,
and your transmitted and reflected amplitudes are
0.0154-0.0043j and -0.0062+0.0012j respectively.

In terms of probabilities, your transmitance is 0.8436
and your reflectance is 0.1564. Relating this to your
amplitude's probability you have 0.0084 for the
Transmitance and 0.0016 for the Reflectance.
```


Potential (hartrees)
Position (bohr radius)