Quantic Measures Diary

Manu Canals

2019

Week Apr 1st - Apr 8th

- **UNITS.** In order for the used values to be of an acceptable order, $10^{-1} \sim 10^{2}$, the factor $m \equiv mass/\hbar^2$ in Schrödinger's equation has to be of this very same order. To do so, the units of the different magnitudes are the following:

Energy	eV
Time	fs
(factor) m	eV^{-1}Å^{-2}
Length	Å
Wave Function	Å$^{-\frac{1}{2}}$

Table 1: Units of the different magnitudes. Wave function's units have been introduced later, so it appears wrongly in some of the following points.

- **Eigenparam.** Checking (passed) if the sum of every component squared of the evects given by the eigh function is 1 (used in trapezoids formula when integrating).
- **Eigenparam** Checking if the eigenvalues and vectors are actually solution of the time independent Schrödinger's equation (used as an example an electron in a harmonic potential added up to a gaussian one). See figure 1.

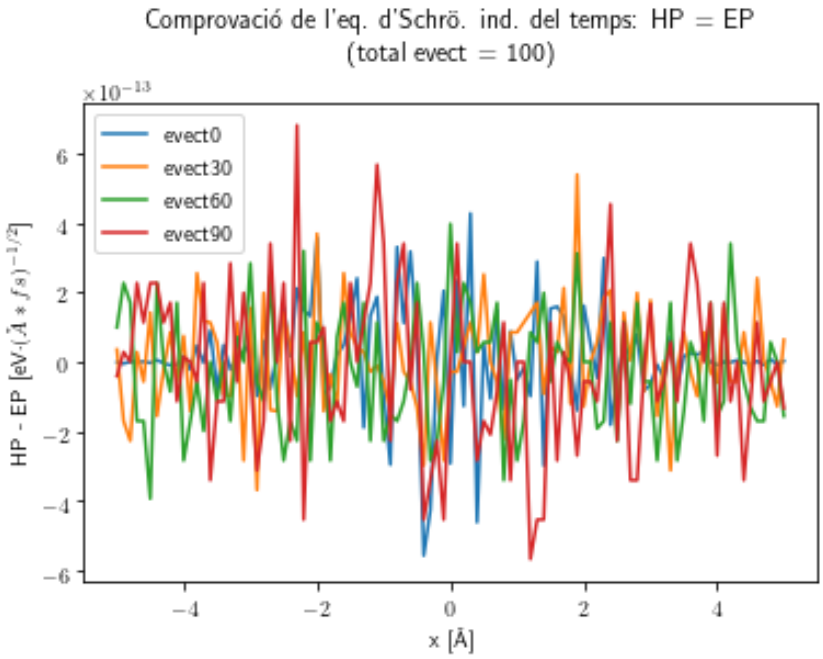


Figure 1: When looking at the difference we see that the results from eigh are solution of the TISE, since the difference shown here is of the order 10^{-13}, negligible (machine precision).

- **Comp.** With this subroutine we get the components of the given vector in a given basis. To check if it actually does, we sum up the linear combination of the components with the basis vector and check if get the original vector back.

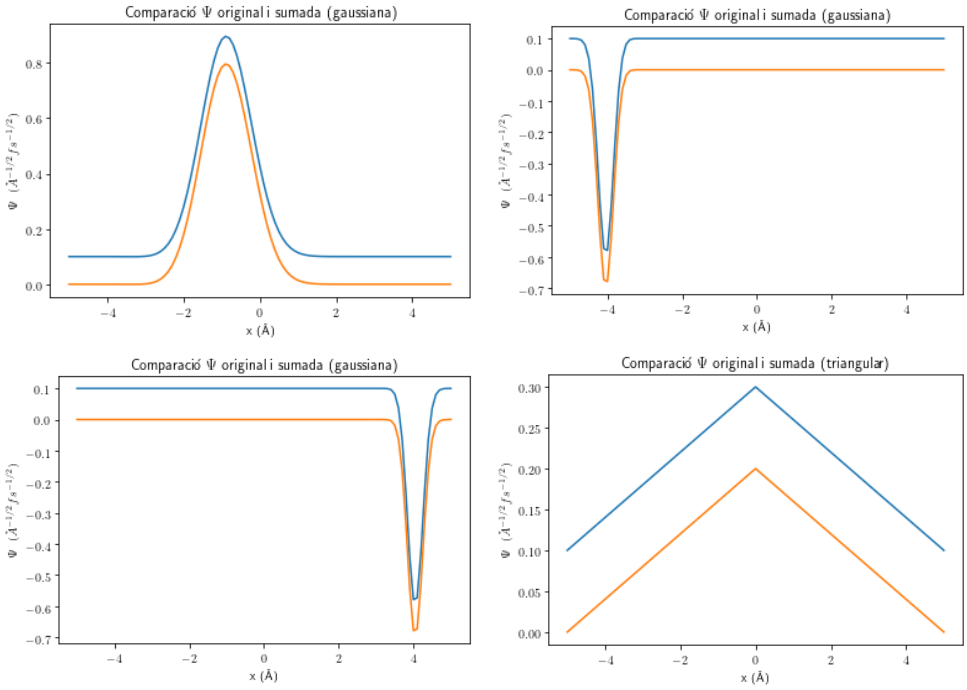
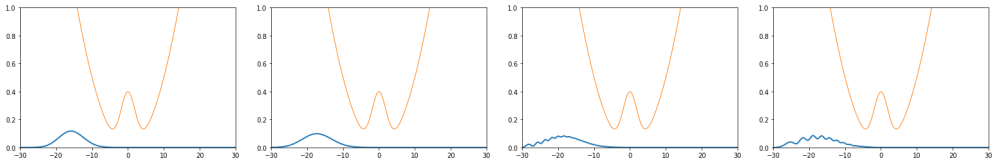


Figure 2: Original wave function (blue) and the summed up (orange). The original has been moved +0.1 units (those of psi's).

- **Evect evolution.** Defining the vector to evolve as one of the eigenvectors (evect), we can check how there's **no** evolution.
- **Wave function's units.** Since they have to be normalized, the integration in one spatial dimension has to be dimensionless.
- **Dimensions.** On of the first obstacle appears when evolving the wave function. Sometimes it gets to feel the influence of the walls (illustrated in figure 3). If it happens or not depends on the energy of the wave function (the expected value rather) and the potential of the problem. We want to avoid this influence since the walls *shouldn't be there*, they exist only for making the discretization possible.



(a) Frame #4 **(b)** Frame #6 **(c)** Frame #8 **(d)** Frame #10

Figure 3: Wall's influence when the main structure of the wave approaches the borders (labels omitted for illustration, they are the same as in figure 2)

To avoid this, we need to keep the structure from getting to the borders. Once the potential (**arbitrary** choice) and the initial wave function are fixed we could study the classical returning points. By doing this we could then choose where to put the walls to avoid the influence. This could be also seen as fixing the walls (order of magnitude of the x) and restraining the $\langle E \rangle$ and potential values, keeping the returning points centered.

Since we will be interested in changing the initial wave function and potentials, we shall **fix the positions of the walls: a and b**:

$$\boxed{a = -10\text{Å} \ ; \ \ b = +10\text{Å}}$$

Now we need to find the N we are going to use. The criteria here will be the quality image (and maybe the computing time in the eigenparam function) [1]. See figure 4.

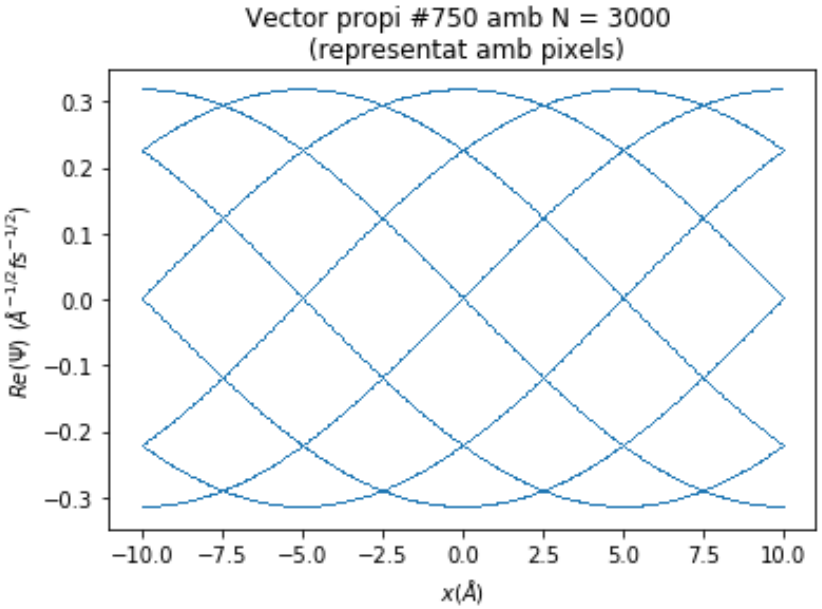


Figure 4: One of the eigenvectors (number 750) when using a discretization of N = 3000 intervals. Plotted with pixels. If you use lines, instead of connecting the two closer points and drawing this image we see here, as we should expect since using a lot of points with pixels should draw the same line used when plotting with lines, it draws a super dens sinusoidal wave. I think it's caused due to plotting complex function on to real values. But we shall not exploit more this fact since it is more a curiosity than a problem.

This study with pixels is actually pointless right now (although we will leave it here) since we need first to define the image we are going to plot (this is the wave function as well as the potential). To do so we have to find a way to keep the returning points centered (not too centered because the wave will then not be able to move). This we can do by restraining the parameters related to $\langle E \rangle$ like momentum and sigma. The different potentials we will use (if we use any different potential shapes) must have similar values, in order to maintain the conditions on the $\langle E \rangle$'s parameters constants.

Week Apr 8th - Apr 15th

- **Beginning with KIVY.** Although we ended last week talking about conditions on the problem's walls, potentials and wave functions, we are going now to move everything we did to a 'kivywise' kind of code. This is done now because we don't want to lose sight on the main goal here, making an app. Furthermore, kivy's syntax is most likely to force us to define our functions and general code structure (as we shall see next) in a certain way. Its better to at least have the wide view, even though we might jump some steps, in order to build everything properly and not having to change it later when moving it to kivy. Let's study then how kivy works.
  So far, this is what we can tell about kivy's structure. **Two files are needed**: one with python code (name.py) and the other with kivy code (name.kv). Its very important that these two, **they have**

[1] Something really weird happens when plotting some evect each time with more and more N, with points '.' or pixels ',' and no lines.

the same name and must be in the same folder, this makes it easier to connect the two of them when executing. The first one will contain all the function's and classes (we will specify it later how to put them together later). Here we write everything that *does* stuff: computing evolution, eigenparam, components, plotting, animating... In the second one (kivy file) you build the structure of the app in the sense of making the interface: placing everything in the screen, making buttons, sliders ... tools for the user. The python file will then use the information obtained by the app (contained in the kivy file) to run the functions and the *computing engines* in general. This button and sliders' variables in the kivy file **also exist in the python file**!
Let's dive more into the python file structure. The use of classes is essential here. Two main classes will be defined: a first one with all the computing tools, and a second one with only one function definition inside: the function, named 'build', will only need to return the first class (*1st_class_name()*). The first class has to inherit from some specific Kivy class, imported like a module. Which class depends on how you want to use the tools inside. In this case its going to by a BoxLayout since we want to put our tools, like plots or buttons, in different *boxes* in the screen (specific places with certain size, position, color...). The name of the second class has to be *1st_class_name_App*, the App part is the key here[2]. At the end of the file the second class will be executed by *nameApp().run()*.
Moving now to the kivy file, the name of the first class has to appear like this: *<1st_class_name>*. Below it, using indentation[3], we introduce all the elements of the app. Each element can have an id and the properties of the element can be used and changed in the python file using its *name of functions used like name.func (can't remember the name right now)*. So if we want to now the value of some slider we'll do: *slider_id.value()*.
This is the general structure of any Kivy app. Now we want to animate a plot[4]. This we will do by first assigning the canvas to some box with *box_id.add_widget(canvas_name)*. Then using the module Clock from Kivy, the function *Clock.schedule_interval(function, interval)* calls the function after the 'interval' time has passed (in seconds). This two lines, along with several definitions and starting computations, will be written inside the function *__init__*, which is executed when calling the class (since its returned by the second main class, which again is executed when the file is, the class will be called when the file is executed). In our case, the called function in the loop is the plot of $\psi(t)$. So the function will compute the value of $\psi(t)$ and drawing it. To do the first, the already written functions like eigenparam and comp will have to be executed before starting the loop, or every time the potential changes (for both functions) or the initial wave function changes (this only affect the comp function). Drawing is explained in the next point.

- **Drawing in Kivy.** We will do it the following way[5]:

  1. Importing *Figure* from *matplotlib.figure*. This is the object of the whole plot (we can call it canvas too I think).
  2. Creating a figure with *fig_name = Figure()*
  3. Since we will plot the potential and the wave function we generate twins (subplots) inside the figure.
  4. To pass the canvas to Kivy app we have to create an especial object importing it from Kivy (*FigureCanvasKivyAgg*)
  5. Setting everything ready (axes, labels, range, ...). The plot's data is going to be updated to plot_data, obejct.
  6. When the canvas is ready (or whenever, may work too), we assaign it to the box with *add_widget* as we said before.
  7. The only thing left to do is updating the data inside the plotting function in the loop and drawing.

- **Actual writting.** All this structure is set in the files with a lot of comments. The next step will be only to introduce the functions and start putting widgets/elements into our app.

[2] I'm not completly sure about what names have to match each other in order to make the app work. What I'm explaining here is what works for me and what I learned when beginning with Kivy. It's the first picture of kivy's syntax I got clear.
[3] Identation used always when defining something inside an object, like a button, a box, a grid, etc.
[4] See next point for the plotting object.
[5] Following these steps while looking at the code make it easier to see how everything is written.

Week(s) Apr 15th - Apr 22nd & Apr 22nd - Apr 29th

- **Easter holidays.** No meetings these weeks.

Week Apr 29th - May 6th

- **Adapt functions.** Since we are using classes, to enter the functions from eigenparam in the main class we will make them with no arguments and passing only self (most of the time).
- **Sequential function argument.** The loop function has to have a parameter dt but **it is not the real time**, it's only the real interval time. To get the real time we are now using *Clock.get_time()*.
- **Computing time evolution.** We have to use the following expression. We will you use two steps to compute it.

$$Basis : \{\varphi_n(n); n = 1, \dots, N+1\}$$

$$Initial\ wave\ function : \psi(x,0) = \sum_{n=1}^{N+1} c_n \varphi_n(x)$$

$$\psi(x,t) = \sum_{n=1}^{N+1} c_n \varphi_n(x) e^{-\frac{iE_n t}{\hbar}}$$

We can get this combination with the matrix product:

$$\begin{pmatrix} \varphi_1^1 & \dots & \varphi_i^1 & \dots & \varphi_{N+1}^1 \\ \vdots & & \vdots & & \vdots \\ \varphi_1^j & \dots & \varphi_i^j & \dots & \varphi_{N+1}^j \\ \vdots & & \vdots & & \vdots \\ \varphi_1^{N+1} & \dots & \varphi_i^{N+1} & \dots & \varphi_{N+1}^{N+1} \end{pmatrix} \begin{pmatrix} c_1 e^{-\frac{iE_1 t}{\hbar}} \\ \vdots \\ c_i e^{-\frac{iE_i t}{\hbar}} \\ \vdots \\ c_{N+1} e^{-\frac{iE_{N+1} t}{\hbar}} \end{pmatrix} = \begin{pmatrix} \psi^1(t) \\ \vdots \\ \psi^i(t) \\ \vdots \\ \psi^{N+1}(t) \end{pmatrix} = \psi(x,t)$$

The first matrix is the one we get from eigenparam, *evect*, so we only need to build the second one and then doing the product.

  - **Time evolution matrix:** when measuring, from the three factors in the time evolution expression, only the components change. So building a matrix with the exponential factor and the eigenvectors, and then a component vector, the time evolution would become much easier: just multipling the matrix with the components, regardless if measures are made, we will not have to build any matrix (components come from *comp*). To do so we have to find a way of building this first matrix considering the parameter time has to go inside, **without having to build the matrix itself again** (transposing and all, just giving the time value). Defining a function with parameter time to build this matrix would be pointless since we are going to call this function inside the loop and the point is to have the matrix ready before the loop starts, in order to make the computation easier [6].

- **Evolution test.** Given an eigenvector it doesn't evolve, as expected. The norm has been checked too: it remains constant **but** the value isn't exactly 1, it depends on the N number. Using $N = 100$ precision reaches 10^{-4} and using $N = 1000$ then goes to 10^{-6}. This value **remains constant** all the way down to 10^{-12}, the weird thing here is only the value. Actually, it also depends on the initial wave function shape, and this at the same time depends on the potential shape. If the wave gets closer to the edges the precision will be worse.
- **Random pick.** When measuring and picking the new (mean) position value we can use *np . random . choice (mesh , p = [prob])*. Selects a random point from the mesh with probabilty distribution given by p. There's no need to multiply it by deltax, given that the probabilty will remain proportional anyway. One thing we do have to take care of is that the actual sum has to

[6] This is just a thought, maybe it's not that important (timewise) to build this matrix (transpose and all perhaps doesn't take that much time to do).

equal 1 (it's needed for the numpy subroutine). If we had multiplied the probabilty by deltax it would have sum up to one, but not all of the times. The subroutine needs a high precision in this result and as we have seen before, the norm of the wave function not always has the best precision. Lucky, we have decided not to multiply it by deltax, and balancing this and fixing the precision problem has the same solution: multiplying the probability array by a constant. This constant is the actual sum value.

- **Resetting after measure.** *Clock . get _ time ( )* gets the total time of plotting and we need the time from the beggining of the time evolution, that is, after measuring. To do that we substract certain value called *rst_time*, the time when we reset the time count. When measuring or in general whenever we have to start from zero the time evolution we set this variable to the current time, *Clock . get _ time ( )*.

- **Wave function object.** The psiev and psi0 arrays are going to be, from now on (we are establishing it here), **row vectors** and **not** column vectors.

This...

$$self.psi0 = np.zeros(self.N + 1)$$
$$self.psiev = np.zeros(self.N + 1)$$
$$\begin{pmatrix} 1 & 2 & 3 & \dots \end{pmatrix}$$

not this...

$$self.psi0 = np.zeros(self.N + 1, 1)$$
$$self.psiev = np.zeros(self.N + 1, 1)$$
$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \end{pmatrix}$$

- $\langle E \rangle$ **remains constant between measures.** We have computed $\langle E \rangle$ for each frame and we could check it remains constant. We have done this by changing *comp()* so if *True* is passed (this is actually a kwarg that defaults to *do_psi = False*) computes the components of the actual *psiev* and stores the results in the variable *compev*.

Week May 6th - May 13th

- **Compacting operations.**
  - **Eigenparam: normalizating *evect*.** To multiply each column of a 2D numpy array by each number of a 1D numpy array we can do a loop, but to avoid looping (with large run times) **it's equivalent** to do a normal mulitplication $*$. This is not the case with numpy matrix or using *np.dot*. This will by used, for example, when multiplying *evect* by the normalization factor in *eigenparam*.
    We can really see the improvements: when using $N = 1000$ with the old method, the computer had problems opening the app's window (stayed in a black window for 5 to 10 seconds). After the change, you can not even see any black window before opening the normal app's window.
  - **Components.** Here we use the following compact computation instead of the previous one:

$$\Psi = np.zeros(self.N + 1) = [\Psi^0, \dots, \Psi^N]$$
$$Basis\ (stored\ in\ evect) : \varphi_0, \dots, \varphi_N\ where\ \varphi_i = [\varphi_i^0, \dots, \varphi_i^N]$$
$$C = \begin{pmatrix} (\varphi_0^0)^* \Psi^0 & \dots & (\varphi_j^0)^* \Psi^0 & \dots & (\varphi_N^0)^* \Psi^0 \\ \vdots & & \vdots & & \vdots \\ (\varphi_0^i)^* \Psi^i & \dots & (\varphi_j^i)^* \Psi^i & \dots & (\varphi_N^i)^* \Psi^i \\ \vdots & & \vdots & & \vdots \\ (\varphi_0^N)^* \Psi^N & \dots & (\varphi_j^N)^* \Psi^N & \dots & (\varphi_N^N)^* \Psi^N \end{pmatrix}$$

We get C using the following compact formula (already checked this is true):

$$C = \left((evect^*)^T * \Psi\right)^T$$

Where T means transpose and * conjugate. Also, *evect* is the usual matrix with eigenvectors. Finally, the component's array follows from:

$$comp = deltax * \left[sum(C, axis = 0) - C[0, :]/2 - C[-1, :]/2\right]$$

In *sum*, axis = 0 means adding up only columns (each column is added up).
We did all of this in order to avoid looping explictly. Summing up, we can compute the components in just two lines of code.
In this case, this difference with the run time using the old method and the new one is considerable. Changing the discretization, N, and plotting the time between frames we compare the two methods [7]. For just three cases the difference is clear enough:

N	**Old method [s]**	**New method [s]**
1000	0.286	0.07
2000	0.985	0.14
3000	2.12	0.36

Table 2: Time between frames when projecting using differents methods. The average time when comp() isn't called is 0.06 seconds. In the first case $N = 1000$, for the new method, this run time is slightly above the average time so no delay can be aprecciated. All this tests have been runned using 1/60 as interval argument in $Clock.schedule\_inteval$.

– **Eigenparam: getting eigen values and vectors.** What old method did was: first to build the hamiltonian H looping through the diagonals and then calling $eigh(H)$ from *np.linalg*. Finally, the normalization process was called, this has been already discussed. Here we tried two approaches to speed up $eigenparam()$[8]:

1. **Changing the way we build H.** Instead of using loops, we tried using $np.diag(array, k)$. It builds a matrix with the k'th diagonal being 'array' ($k = 0$: Main, $k < 0$ Lower, $k > 0$ Upper). We build these arrays using the *potential* array for the main diagonal, and $np.full$ for the off-set diagonals. The results weren't the expected.

N	**Old method [s]**	**New method [s]**
1000	0.012	0.020
2000	0.027	0.054
3000	0.037	0.150

Table 3: Run times for different ways of building up H. In this case, looping was more effective. We even tried using *diagflat* instead of *diag*. It improved diag's results but didn't beat looping run times.

2. **Changing *eigh*.** Since we already tried improving how to build H and normalization, the only thing left to check is *eigh*. Searching new methods of diagonalizaing for a tridiagonal matrix we found *eigh_tridiagonal* from *scipy* module. We import at the beggining *linalg* from *scipy*, *eigh_tridiagonal* is an attribute from it. This function even avoids builng the H matrix, only the main diagonal and one of the off_set diagonals need to be given. We set *check_finite = False* for more speed, this avoids checking for NaN's. The results were remarkable:

[7] When calling the *comp*() function, between those frames the app will run the *comp*() function and the plotting function, so we will see a change in the time between frames.
[8] Now the run time will be checked using the module *timeit* and its attribute $default\_timer$

N	*np.eigh* [s]	sp.linalg.eigh_tridiagonal [s]
1000	0.30	0.12
2000	2.25	0.40
3000	6.57	0.95

Table 4: Run times for different ways of computing the eigenvalues and eigenvectors.

- **Computing psiev.** We tried building the comp*exp vector using $*$ and then using *np.dot* to calculate psiev and surprisingly the old method goes faster. I supose its because we were using matrix objects that gives a lot of speed when doing the matrix product. Mixing it, we build compexp using $*$ and reshaping to matrix and then do the matrix product. Looking at the real interval times we see this goes slightly faster than when we used a comprehension list. After we got the column vector with psiev, we were using again a comprehension list to unpack the values in form of an array (needed for the input when seting the new data in the plots). Trying to instead reshaping to a row vector (although still being a matrix object), transforming it to an array object and **then** taking out the first index (due to matrix indexing) we got the same result. Checking for run time, the lalatter method proved to be considerably faster. We test this after fixing N to 800 (explained in one of the following points). The results in this case were (in seconds):

  $$t_{old} \approx 4 \times 10^{-3} \qquad t_{new} \approx 5 \times 10^{-5}$$

  So we change now and use the latter explained method.

- **Potentials traps.** When removing the explicit large potentials values from the potential in the edges, it seems that no change occur. The evolution looks the same. It's like the walls are already there without explicitly putting them. One thing I noticed, is that $\langle E \rangle$ is significantly smaller when 'there are no walls'. This is reasonable, but in the evolution sense no difference can be easly appreciated.

- **Potential and psi objects.** They both are 1D arrays and are created using *pot_init* and *psi_init*.

- **New *psi0_init*.** Before we were using the following expresion to initialize *psi0*:

  $$psi0 = \sqrt{\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}} e^{-ip_0 x}$$

  It is the analytic expression for a normalized guassian distribution when squared its absolute value. Although this expression **is** normalized, maybe when integrating the actual array (its absolute square) we don't get 1, since it depends on whitch points we get from *mesh*. And we would have to normalize again to get a normalized **array**. So, intead of wasting time with a complex expression like the previous one, we are going to initialize first with only the part that gives shape (the exponentials) with no multiplying constants and then normalizing it. The remaing expressions would be:

  $$psi0' = \sqrt{e^{-\frac{(x-\mu)^2}{2\sigma^2}}} e^{-ip_0 x} \rightarrow C^2 =^* trapezoids\ integration^* \rightarrow psi0 = \frac{psi0'}{C}$$

  This method is even quicker than the previous one. Its a win win change. Furthermore, this allow us not having to normalize again when measuring.

- **Dirac's sigma.** The sigma after measures its going to be *dirac_sigma*. It has to be the smallest possible in order to resemble a dirac's delta. If it's too small the evolution is too fast post measurement and with too many 'interferences' (run a test to see this). We set here this *dirac_sigma* to be:

  $$dirac\_sigma = 0.2$$

  This sigma gives more and more energy to the post-measure wave function as we make it smaller and smaller. And we want to keep the wave function's energy low, orders of the centered potential, so it doesn't have the reutrning points to far from the middle.

- **THE POTENTIAL.** We begin testing different potentials with the usual combination of harmonic and gaussian potentials [9]. Changing the gaussian parameters (for $k = 1$) we see that the wave function doesn't really 'bounces' further away, so it never reaches the walls. As expected, for the gaussian potential tends to zero when approaching the walls.
  We are going to change the harmonic parameters without the gaussian potential. We find that the wall's influence starts to kick in **from** $k = 0.4$ **and below**.

- **Fixing N.** To acomplish this, first we have to discuss where does N influence. Normalization doesn't depend on N. What does depends is the frame rate. If N is too high the plotting time per frame is significantly big and you see the animation with 'lag'[10]. Considering this fact, $N = 1500$ **becomes an upper limit**.
  But what we want to find is the minimum N so all functions don't take too much time to compute (keeping a good and smooth plotting). Keeping the smoothness of the plot, **we can get it low to** $N = 800$. At this point, not even the gaussian potential with $\sigma = 0.17$ looks sharp nor with edges. About run times, opening the tab takes no time and measuring also causes no problem.

- **Directly compute only truncated values.** Using the optional argument *select = 'v'* and *select_range = (min_eval, max_eval)* from *eigh_tridiagonal*, it returns only eigenvalues with value inside the given range and its related eigenvectors! Since we are talking about the Hamiltonian, its eigenvalues will be energy values. This means we are setting a limit on the energy values. This looks perfect for keeping our wave functions centered.

- **Truncated components.** Since we now don't have all evects to compose the given psi0, when adding up the liniear combination using truncated components and the corresponding evects, we don't get exactly the same psi0 back. Has the same shape (except for some minor vibrations) but the norm isn't the one psi0 had (1). Even though this norm is not exactly 1, it is conserved allong the evolution! So we have to normalize when computing these components.

- **Improvements when truncating.** Using a max eval of 200. About run times. Computing the eigevalue and vectors the time is reduced to half. Not a huge improvement, but it is one. When computing, run time goes down by a factor of 10. Talking about evolution, some vibrations appear in the new psi, but they are not really fast nor sharp. But in the other hand, when the old psi felt the wall and started vibrating really fast, now this psi tends to avoid this vibration (untill ac ertain point). We now actually are more isolated from the walls.

[9] The factor before this combination during these test have been always 20.

[10] We could fins a way to make this plotting run time smaller but I don't think it's possible to do much better than *draw()* does. Righ now we will consider plotting with *draw()*