UNIVERSITAT DE BARCELONA
Quantum Physics and Astrophysics Department
QuantumLabUB

PRÀCTIQUES EN EMPRESA

Physics Degree

Popularizing Quantum Physics: Quantic Measures

Author
Manu CANALS CODINA

Supervisors
Montserrat GUILLEUMAS MORELL
Bruno JULIÁ DÍAZ

March 2019 - January 2020

Contents

Introduction

This report presents the work done as part of the course *Pràctiques en empresa*, from the physics degree of the Universitat de Barcelona, in the period between March 2019 and December 2019. The project has been supervised by Prof. Montserrat Guilleumas Morell and Prof. Bruno Juliá Díaz.

This report's structure has three main sections: this one, **introduction**, presenting the project and the methodology followed; **project development**, where all the work done is explained, and finally; **conclusions**.

1 Motivations and goals

This course, *pràctiques en empresa*, was of incredible interest for me. It offered a closer way of getting first contact with the 'outside world', that is, outside classes and laboratories within the degree.

Being more than half-way through the degree, I had the impression of having no knowledge at all about any working possibilities in the physics subject.

By choosing this course, I expected to get to know about internships and working offers, as well as the way physicist work outside a classroom.

2 Project

After deciding to course *pràctiques en empresa* I started looking for internships offers and I tried a few (like *Quantic* in the BSC, but sadly they didn't work out) before Bruno reached their students. He and Muntsa offered us collaborating with QuantumLabUB.

Three undergraduates students finally got into this project: Arnau Jurado, Marina Orta and myself. Each contributing with our own project to QuantumLabUB.

2.1 QuantumLabUB

Initiated by the students Daniel Allepuz and Jan Albert Iglesias in February 2018 under the supervision of Muntsa and Bruno, QuantumLabUB is a project build by undergraduate students developing applications with the following goals in mind:

- Popularizing quantum physics
- Illustrating realistic quantum mechanical problems
- Working under the Github platform, in the QuantumlabUB repository [1].

Consequently, we had to present an idea for an application involving a realistic quantum problem with some kind of interaction for the popularizing purpose. This is, either show different problems with a wide variety of cases (Arnau and Marina's projects), or build a game on top of a certain problem (this particular project).

2.2 Initial idea

The initial idea was the following: playing around a **wave function evolution in 1D** and **taking measures** over it.

More precisely, computing the evolution of a particle inside an infinite box with a certain potential inside, implementing measures on the position at any time and building some goal for the player about the result. Such goal would be forbidden zones where the player tries to avoid having its measure result, by waiting the appropriate time to do the measure.

The application would show the evolution of the probability density and a button to make the measures. And on top of that, a system indicating the forbidden zones and the gaming mode, would create the final game.

Popularizing the concepts of the fundamental randomness concerning a measure and the corresponding behavior of its probability density (the wave function that is) were the aim of this application.

2.3 Final result

Here, I will show the final result for comparison with the initial idea.

I end up having a multiple screen system: with a starting screen, a game and demos.

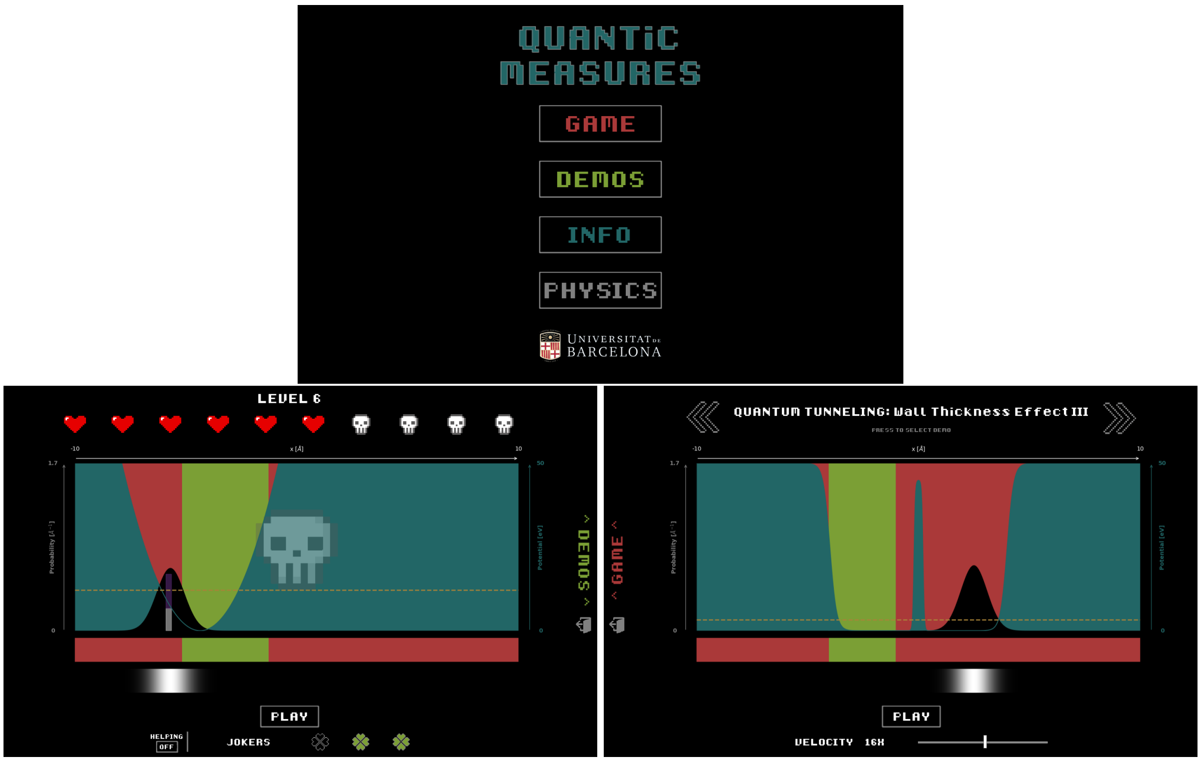


Figure 1: Screens in the project's final result.

First, a starting screen with information pop-ups and buttons leading to the other two screens.

Secondly, a gaming screen with the initial idea of the game: harmonic potential where the particle moves through changing every level, a lives' system for building the game with some jokers included.

Finally, a illustrating screen with demos on it. A list of cases of special interest has been prepared and set for the player to move through. These cases involve quantum tunneling effect and behaviors of a free particle inside a box.

As a first conclusion of the project, the final result did include the initial idea, as well as having an explicit way of showing certain interesting cases. Moreover, the interface really make the application an easy to see game.

3 Methodology

The group methodology (meetings) and my own (documentation).

3.1 Meetings

The current group working in the QuantumLabUB project were Bruno and Muntsa supervising, and the three of the undergraduates students: Arnau, Marina and myself. We met once a week, almost one hour, every week during this internship period (with a few exceptions).

Each student had a different project. During the meetings each one of us present our progress during the week, doubts and questions, and discussed future ideas for the project. We cross collaborated with the others projects by giving our feedback.

I found this meetings greatly productive. Apart from solving questions, exhibit my progress made me prepare actual data and tables to show. Therefore, while typing code and trying out ideas during the week, I started keeping results of the various changes I did.

3.2 Documentation

As previously stated, while developing the application I had to document my progress. As a result of this, I worked with two essential documents along this project. This memory comes mainly from those.

For the annotations while working, I wrote a *to do/done* list. This turn out to be really helpful, writing every thought and future ideas down allowed me to well define them and to mark a working path to follow. This document was to be used only for me.

For presentations in the meetings and deeply explaining my progress, I wrote a *diary*. Here, thoughts and ideas from the first list where expanded and explained for everyone to follow. This file is actually uploaded in Github as well [1], in my repository inside QuantumLabUB.

As part of the QuantumLabUB way of cross collaborating (along with the meetings) we worked with the already mentioned platform Github. There, version-controlled projects can be uploaded as open source. Weekly, following the meetings, we uploaded our files there.

3.3 Calendar

We didn't work with a strict time table. Only the weekly one-hour meeting was required. So the hours were when they best fit us, to always (or nearly always) have some progress to present in the meeting.

I kept track of the hours in a separated document. I worked in two major periods of time separated by summer.

- First period: from the beginning (March 7th) until final exams (middle May). Partial hour count: **117 h**.

[1] *Diary* contains all the progress until middle October, when I started sketching this report and no writing was further done on *Diary*.

- Second period: [2] after summer (October 3rd) up to the writing of this report (December endings). Partial hour count (only this period): **150 h**.

The total (counting the smaller middle period, see footnote) adds up to: **291 h**.

Since Github keeps track of the amount of code uploaded as well, this is the Github's code counting distribution for my user in the QuantumLabUB project.

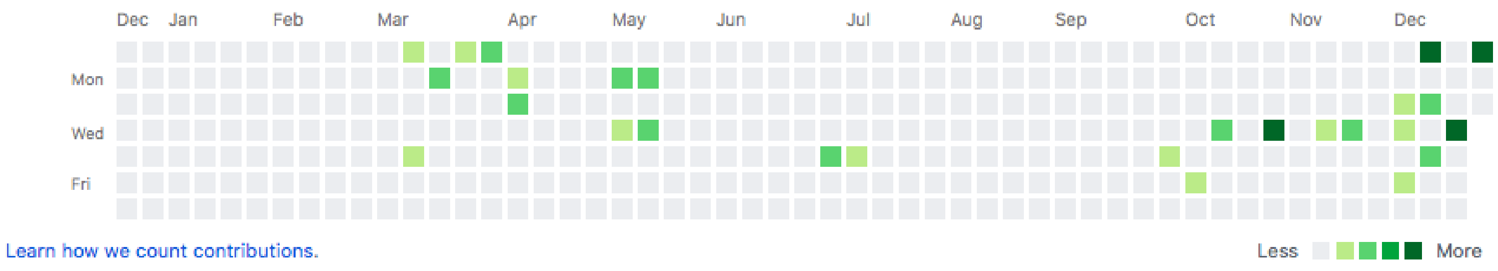


Figure 2: Github's code count for my user. As expected for my own hour count. Only shows amount of lines uploaded, hour counting accounts for more continuous work.

[2]Between exams and summer we met a couple of weeks, working **24 h** in this smaller period.

Project development

This is the second main section, and the body of the report. Here all the work done is presented in a general chronological way.

Bearing in mind that the application basic content consists of *showing the evolution of a wave function* and *building a game on top*, this section structure is as follows: **preparation**, any previous work done before even typing any code [3]; **solving the problem** of the evolution of a wave particle and measuring it, all the way on to the first creation of a game; **designing the game**, where the basic game is transformed into a playable game, and; **code management** after finishing the application.

Time wise: the two first sections correspond to work done before summer (section 5.4 was done after final exams), and the last two after summer.

4 Preparation

Before even typing any code, certain work was carried out.

4.1 Python

Although I had just coursed computational physics (using Fortran77), I actually had to go over programming with Python all over again. I did have the fundamentals, however, the code structure for programming an application required a deeper understanding of Python.

For the first days I rawly revised Python documentation [2] as well as the book used in the first year of the degree [3].

The following Kivy section will go through how an application is built in a general way. Nevertheless, here I will point out a couple of details that are essential for this code.

First, the whole code is build around class definitions, which then Kivy uses on its own, and some inheritance. Brand-new concepts for me back then.

Basically, creating a class in Python defines a new custom object, with its own variables (attributes) and functions (methods). The class just defines the object, and then an actual object can be created by *calling* that class. Each of this objects is called an instance of that class.

This part of creating the actual objects it's done by Kivy, so in the Python code there is only class definitions. This may look weird at first, I only have one executing line in the whole python file (runs the application), but classes are later used in the Kivy file (explained in the following section).

Inheritance would just be creating objects based on other objects, the new one (child class or subclass) will have the same attributes and methods as the original one (parent or super class) in addition to the defined inside the child class.

Secondly (this now concerns my particular case), these methods aren't defined just like usual functions. Since they will only act on objects of their own kind (class), the only argument passed is *self*. It contains every variable associated to the class.

Therefore, the methods are defined to be just transformations, actions on its own variables (attributes). For this reasons, this methods would be hard to use with objects that are not of its own class.

[3]This section might not be strictly chronological.

4.2 Kivy

Kivy, in their own words, is an open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.

Not at the same time I did with Python, but I did also go over its documentation (mainly application programming interface elements, API) [4] and consulted several times a recommended book I found online introducing Kivy [5].

Here I review the basic way to create an app. To do so using Kivy, the most simplest app, is as easy as running its basic App class.

```
>>> from kivy.app import App
>>> App().run()
```

This already creates an application! It's just a window visually, but has event managing ready, could be exported to multiple platforms (phones and tablets for instance)...

The *only* things left to make the app are: to define the functions to be used in the app and the widgets that are going to use them.

The way functions can be defined for the app to use is by overriding one of App's methods, build().

First, a class is defined with these functions (methods). Second, in an App's child the build method is overridden to return this class. Now the child app is run and the application has access to those methods (although doesn't use them for now).

```
>>> from kivy.app import App
>>> class CustomExample():
...     #Define custom methods
>>> class AppChildExample(App):
...     def build(self):
...         return CustomExample()
>>> AppChildExample().run()
```

Finally, the last thing left to make an actual app is to define the widgets: what widgets and what do they do (could use defined methods).

This happens in a different file, the Kivy file, which is written on its own syntax. Basically there, for every class defined in the Python file, a widget tree can be built. This is simply an indented list declaring (thus creating) the widgets related to that class (and its properties) the app is going to have.

Following the previous example, a possible Kivy file could be ('#' work as comments):

```
<CustomExample>:
    #Cross referencing
    #python file variables: kivy file variables

    python_var_name: kivy_var_name

    Button:
        id: kivy_var_name
        text: 'This button uses a method \n
               from CustomExample class.'
        on_press: root.custom_method()
```

With special naming the two files are used together (managed by App): if the Kivy file the app has to use is named *ExampleUB.kv*, the App's child class has to be identically named but adding App, *ExampleUBApp* in this case.

By doing this, the app runs using this couple of files. Moreover, inside both files, there can be references to variables in the other files (App class takes care of that too). In the example, the id for the button in the Kivy file, named *kivy_var_name*, is going to exist in the Python file, named *python_var_name*.

By defining classes in a Python file, listing the widgets in a Kivy file and calling the method *run()*, an app is created. That easy.

Needs to be said that I figured this clear picture of Kivy's workings with tones of printings, readings and failings, and this may not even be the actual way Kivy works, but works in this current app.

4.3 Github

As well as learning how to use Kivy and Python (again), we had to work using Github's platform to share the code as open source. It is an extremely useful way of doing programming projects.

It works with a system of branches, forks and merges. In a nutshell: each of us had our own fork where we uploaded our changes, and weekly (when having substantial progress) we would ask permission to merge our fork with the main branch (managed by Bruno). If conflicting changes would appear, the merge could not go through.

Github has several advantages and I will not discuss them all right now. Although, amongst them I have already talked about its code line counting and to mention, keeps the history of the uploaded files (for later comparison and possible returning to old versions of the project).

4.4 LaTeX

Finally, and possibly the tool that I have used the most along the degree since I learned it, I started writing documents in LaTeX.

From the beginning, I was suggested by Bruno to start doing the presentations in LaTeX. Since then, all my documents, before written using Word, were in LaTeX.

As usual, I had to go through a certain learning period, where I even went to a small introductory session. But finally, I am writing this document in LaTeX, like the two other documents I talked about and many more (like the CV for example).

5 Solving the problem

Solving the problem consists of two parts: computing the evolution of a wave particle and implementing a measure system.

With this two pieces worked out (and optimized), the most basic game can be build.

5.1 Evolution

Problem to be solved: evolution of a particle's wave function $\Psi(x,0)$ (in the position space) inside an infinite box with certain potential inside, $V(x)$. This is, find $\Psi(x,t)$.

For the entire project, the so called particle is considered to be an electron. This only affects the value of the particle mass and the choice of units. With the goal of having numbers of reasonable order while calculating ($10^{-1} \sim 10^{2}$), the factor $mass/\hbar^2$ is made to be of this very same order when choosing unit system. The first choices considering $\hbar$ were:

Energy	eV	$\hbar = 0.6582\, eV fs$
Time	fs	

Looking at the electron mass and the mentioned factor:

$$0.511 \quad \frac{MeV}{c^2} = 0,0568 \quad \frac{eV fs^2}{A^2} \quad \Big| \quad factor \approx 0.1311 \quad eV^{-1}A^{-2}$$

Where A stands for angstrom (LaTeX rendering issues). Unit system chosen:

Energy	Time	Length
eV	fs	Å

Table 1: Unit system chosen for this project.

Resolution

No numerical resolution is needed here. Being the potential time - independent, the evolution of a wave function follows from the *time - independent Schrödinger equation*:

$$\mathbf{H}\psi_n(x) = E_n\psi_n(x) \qquad where \qquad \mathbf{H} = -\frac{\hbar^2}{2m}\nabla_x^2 + V(x). \tag{1}$$

With $\psi_n(x)$ and E_n solving this equation, they are the eigenvectors and eigenvalues of $\mathbf{H}$. The evolution problem is solved with:

$$\Psi(x,t) = \sum_{n=0} c_n\psi_n(x) exp\left(-\frac{iE_n}{\hbar}t\right), \tag{2}$$

where c_n are the components of $\Psi(t = 0)$ (known) in the basis formed by all eigenvectors. Consequently, c_n are computed using:

$$c_n = \int \psi_n^*(x)\Psi(x,0)dx. \tag{3}$$

Basically, it all comes down to know how do eigenvectors evolve. They just *rotate* (in the complex plane) since they are only multiplied by one complex exponential.

Therefore, expressing $\Psi(x,0)$ as a weighted sum (linear combination) of these eigenvectors will yield its evolution simply by *rotating* each eigenvector.

Computation

Clearly, the convoluted part comes from finding the eigenvectors and values of $\mathbf{H}$ in (1). Numerical methods can be used to solve the components integral.

Working with analytical objects like ψ and $\mathbf{H}$ is what makes the problem harder. With discretizing the position space, solving (1) becomes much easier.

Accordingly, discretizing the positions (from a to b [Å] with N points), Ψ and ψ become N arrays. For $\mathbf{H}$, approximating the second derivative, becomes a tridiagonal $N \times N$ matrix.

$$\frac{\partial^2 f_i}{\partial x^2} \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2} \rightarrow (\mathbf{H}\phi)_i \equiv a\phi_{i-1} + b\phi_i + a\phi_{i+1}$$

$$a \equiv -\frac{1}{2m^*\Delta x^2} \qquad b \equiv \frac{1}{m^*\Delta x^2} + V_i \qquad with \qquad m^* \equiv \frac{m}{\hbar^2}$$

Thus, applying it to any vector (like Ψ of ψ) would just be a matrix product:

$$\mathbf{H}\phi = \begin{pmatrix} b & a & 0 & \dots & \dots & 0 \\ a & b & a & & & \vdots \\ 0 & a & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & a & 0 \\ \vdots & & & a & b & a \\ 0 & \dots & \dots & 0 & a & b \end{pmatrix} \begin{pmatrix} \phi_1 \\ \vdots \\ \phi_i \\ \vdots \\ \phi_N \end{pmatrix} = \begin{pmatrix} b\phi_1 + a\phi_2 \\ \vdots \\ a\phi_{i-1} + b\phi_i + a\phi_{i+1} \\ \vdots \\ a\phi_{N-1} + b\phi_N \end{pmatrix}$$

Once $\mathbf{H}$ has this form, getting its eigenvectors and eigenvalues is a simple algebra problem, implemented in many Python libraries. In this application I use the *eigh* method from *linalg* in *numpy*.

Once the basis is found, to get the components of the initial known $\Psi(x, 0)$, the integral is solved with trapezoid integration:

$$c_n = \int_a^b \psi_n^* \Psi(x, 0) dx \equiv \int_a^b p(x) dx = \left\{ p(x_i) \equiv p_I \right\} = \Delta x \left(\sum_{i=0}^{N} p_i - \frac{p_0}{2} - \frac{p_N}{2} \right). \qquad (4)$$

In conclusion: given a t, $\Psi(x, t)$ can be computed knowing the basis and the initial components.

- **Truncating.** When projecting some Ψ in one of these basis (taking its components) one could ask what are these components values.

  For a usual Ψ gaussian shape in a double well, discretizing with 100 points (thus 100 vectors in the basis) the real part of the components is:

  Clearly, the wave function only projects onto a few vectors of the basis (form the 10th component out of 100, the values are negligible). Considering this fact, computing the last components could be avoided in order to speed up the computation, and therefore the animation.

  Avoiding this final components is going to be called *truncating* the wave function and is going to be implemented for now.

  Checking the norm of the truncated wave function:

  Around the 20th component, the difference gets to negligible orders. And around the 40th it reaches machine precision.

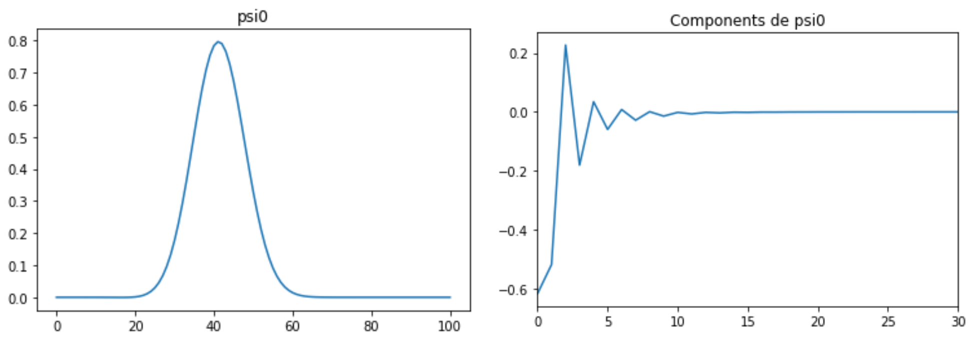


Figure 3: *Right*: real part of the components of the (*left*) wave function. The axis below indicates indices (total discretization of 100 points). The vertical axis for *left* is the probability distribution [Å^{-1}], and for *right*, only the comparison matters, not the value itself.

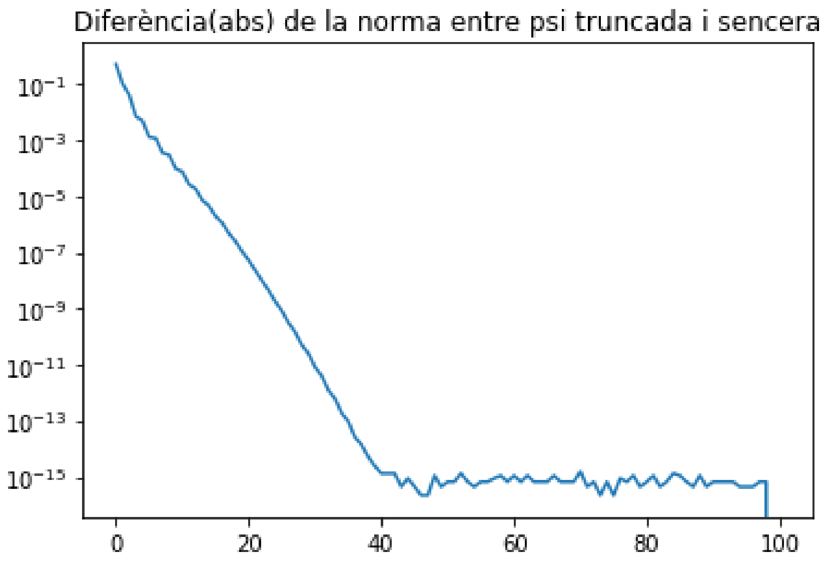


Figure 4: Difference with the whole and truncated wave functions's norm. The horizontal axis refers to axis (100 total discretization).

Implementation

Until now, the time evolution has been computed but not shown. The first animations were done using matplotlib's *animation*, creating gif's.

They are done with a repeatedly called function that computes $\Psi(x,t)$ every frame (not in advanced). Eigenbasis and components are computed before animation. Later, in every frame, the exponential factor is introduced.

This animation process followed here is going to remain along all the project (although not the gif's).

Checking's

After correctly running the first animation, some checking was carried out. In part thanks to the visual aid of the gif's.

- **Eigenvectors and values.** The eigenvectors and values were checked to actually solve the Schrodinger's equation. Figure 5.

  The equation is indeed solved, since the difference is negligible.

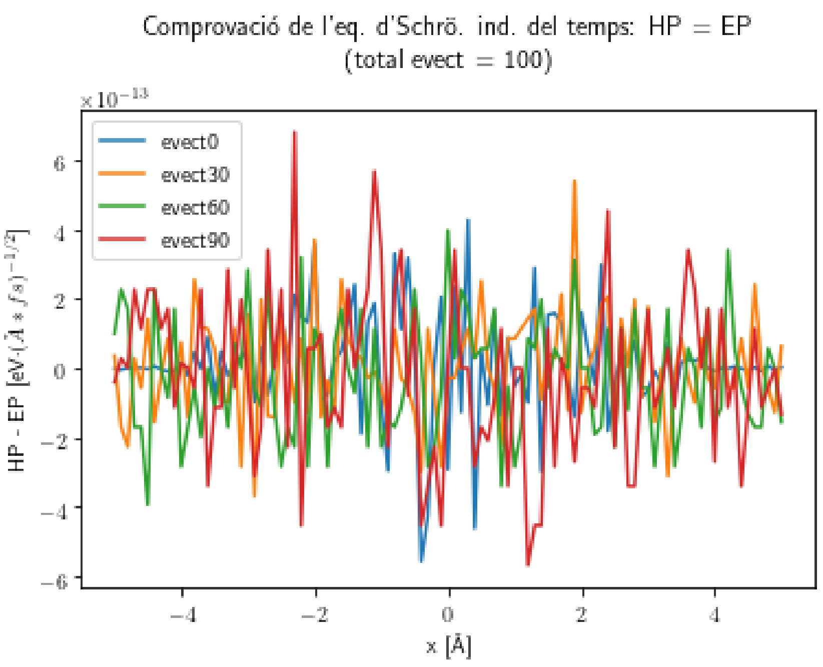
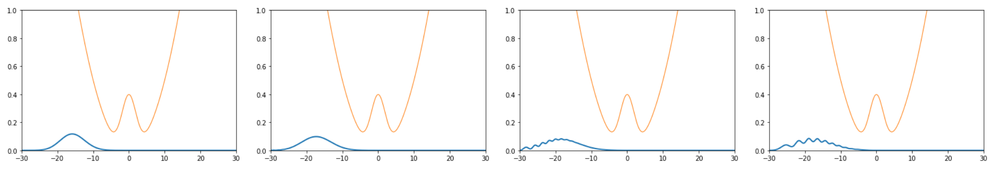


Figure 5: Difference between $\mathbf{H}\psi_i$ and E_i. By Schrödinger's equation, should yield null.

- **Eigenvector's evolution.** Since eigenvectors only rotate in the complex plane, its magnitude isn't changed, plotting its absolute value (what's actually always plotted) should appear invariant. Creating a gif this is checked.
- **Dimensions.** When the wave function approaches the walls it *feels* its influence and a messy pattern emerges along the evolution. This should be avoided in order to make the game fluent and easy to see.



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

Figure 6: Wall's influence on the wave function when it approaches them.

With these checking's, the evolution problem is solved: $\Psi(x,t)$ can be computed and animated (the general frame method). Introducing measures is the next step.

5.2 Measures

Now that the evolution problem is solved, $\Psi(x,t)$ can be calculated, the measures mechanics can be introduced: a button taking a random pick of the instant wave function (measure).

For buttons, Kivy is needed. Therefore, plotting and animation in Kivy has to be worked out. For that, every function already defined has to be implemented as a method inside a class, so its arguments now will just be *self*, as discussed in Python section 4.1.

Kivy plotting

As also discussed in the Kivy section 4.2, it works with two files: one with the widgets (buttons, layouts, sliders...) and one with the functions (to pass to the widgets or not).

Before explaining plotting, two concepts must be introduced.
One of those widgets, used the most, is the BoxLayout: basically creates an arrange of boxes in the application window where other widgets can be placed (in those boxes). Two separate steps: creating a figure and drawing it.
Usual matplotlib plotting works in the following way: creating a **figure** (axis, data, colors...) and then drawing it with a **renderer** (usually called canvas).

With this in mind, plotting in Kivy in this application works in the following way: in the Python file a matplotlib figure is created but not drawn, this same figure is then passed to a box in Kivy, to its renderer more specifically, which in turn draws the figure.
So a matplotlib figure is created as usual, but is drawn by the renderer in Kivy.
The resulting application and the first version of the application created during this project is the following:

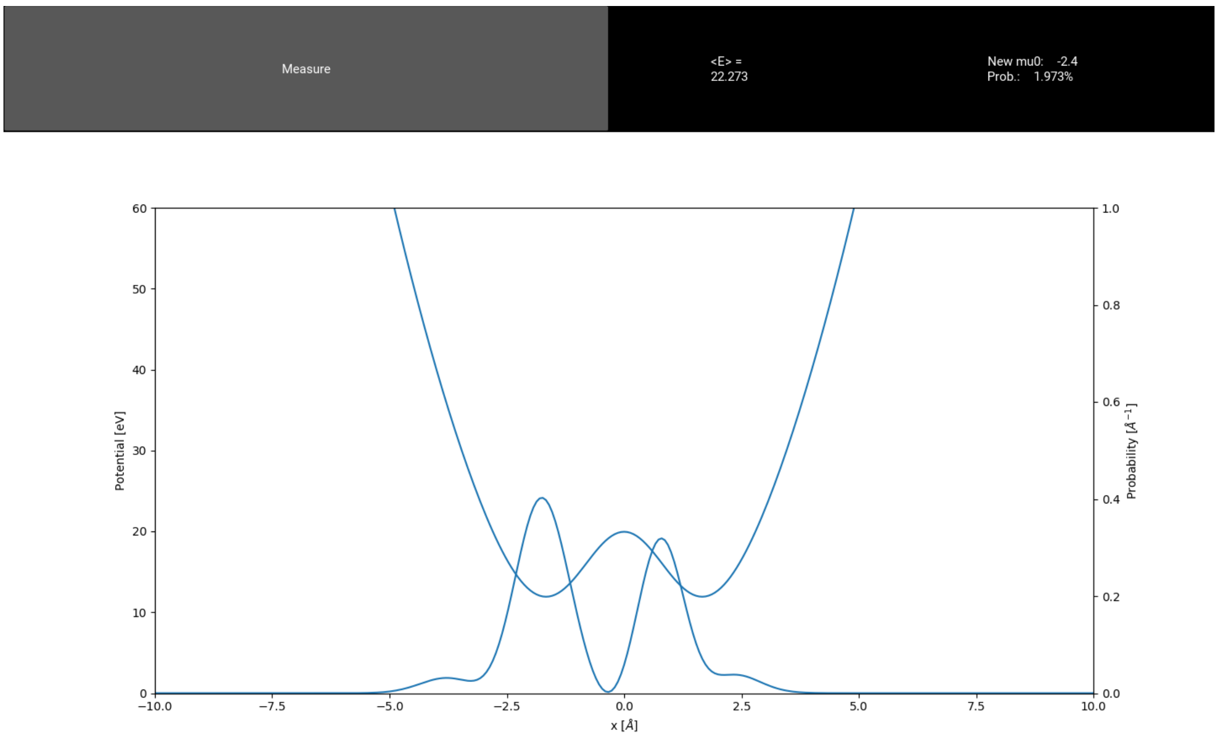


Figure 7: First version of the application created during this project. A simple BoxLayout with 4 boxes: 3 on the top, one for the measure button and two for the information labels (energy and measuring results), and 1 below. The latter is the box the matplotlib figure is passed to.

Kivy animation

Kivy animation works the same way the previous gif's were created , where animation occurred by repeatedly calling a function that computed $\Psi(x,t)$ in that frame time and drew it.
The repeated calls creating gif's where done by matplotlib's *animation*. In Kivy, the same role is played by *Clock*'s method *schedule_interval*: it sets Kivy to call a method every certain time, in a parallel way, while everything else keeps being executed (code isn't stuck there).

The given method will: get the time via *Clock.get_time*, compute $\Psi(x,t)$ with the matrix product in (5), update data and draw.

$$\begin{pmatrix} \varphi_1^1 & \cdots & \varphi_i^1 & \cdots & \varphi_{N+1}^1 \\ \vdots & & \vdots & & \vdots \\ \varphi_1^j & \cdots & \varphi_i^j & \cdots & \varphi_{N+1}^j \\ \vdots & & \vdots & & \vdots \\ \varphi_1^{N+1} & \cdots & \varphi_i^{N+1} & \cdots & \varphi_{N+1}^{N+1} \end{pmatrix} \begin{pmatrix} c_1 e^{-\frac{iE_1 t}{\hbar}} \\ \vdots \\ c_i e^{-\frac{iE_i t}{\hbar}} \\ \vdots \\ c_{N+1} e^{-\frac{iE_{N+1} t}{\hbar}} \end{pmatrix} = \begin{pmatrix} \psi^1(x,t) \\ \vdots \\ \psi^i(x,t) \\ \vdots \\ \psi^{N+1}(x,t) \end{pmatrix} = \psi(x,t) \tag{5}$$

Comments on this current method: *Clock.get_time* gets the time from the launch of the application, it's going to get messy and changed in the future. The matrix product is done by loops using list comprehensions, which take considerable time to execute.

Nevertheless, the animation is already greatly working. Running some tests, an eigenvector does not evolve, as expected, and the norm remains constant.

A problem with the norm emerge, although constant, is not identical to 1. Has high precision (for $N = 1000$ reaches 10^{-6}) and conserved, but as shall be seen, not quite enough.

This problem is solved just by renormalizing (the norm is conserved, this is the important part).

Measures implementation

The measure will be a call to a function, between frames, where all its consequences will be carried out. This measure function will:

- **Pick new position.** With the instant probability given by the absolute value squared of $\Psi(x,t)$, a random new position is picked using *numpy.random*'s method *choice*.

  Here, due to that small difference with 1 when checking the norm, Ψ has to be renormalized before passed to the choosing method.

- **Resetting Ψ.** After a measure is done, the wave function collapses onto the resulting eigenvector. In the position space, this case, it is a Dirac's delta. This new $\Psi(x,0)$ is approximated with a sharp gaussian packet.

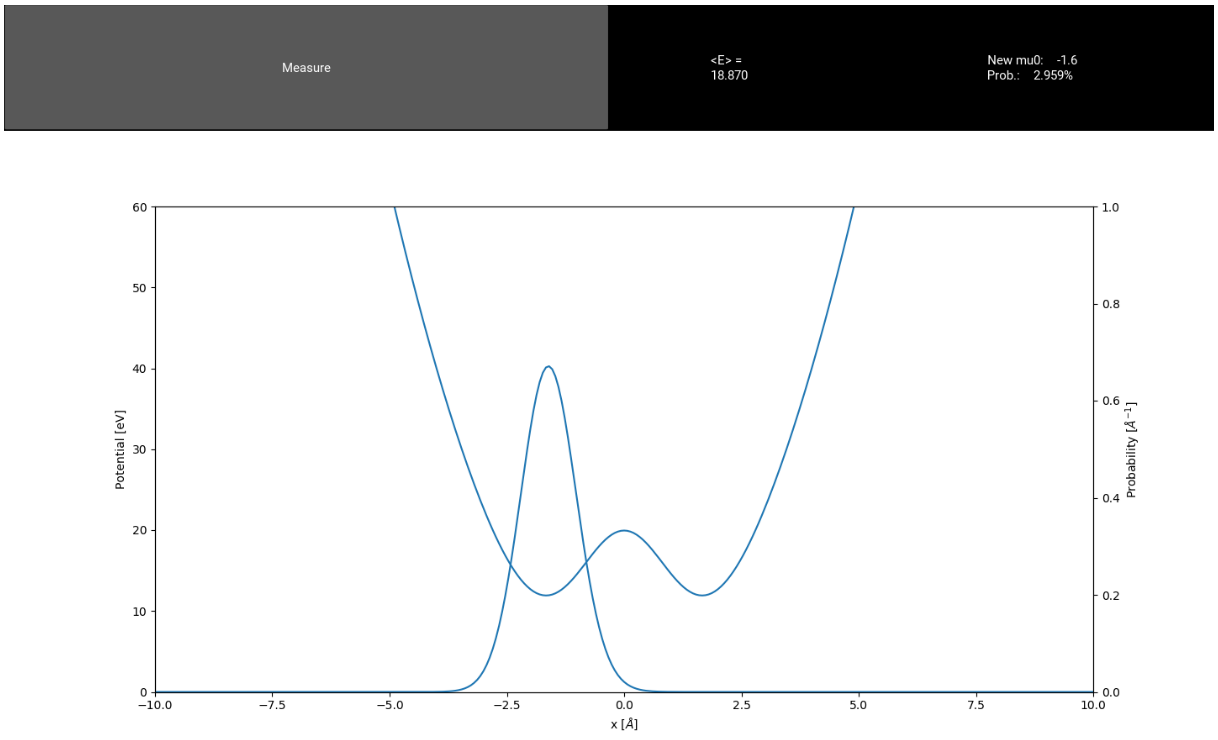


Figure 8: Initial wave function after every measure, a sharp gaussian packet centered on the result of the measure.

- **Components** Once the new $\Psi(x,0)$ is calculated, it starts evolving again. For that, components are calculated again (basis remains the same if the potential doesn't change).
- **Time reset.** This time reset is done by subtracting the time of the moment of the measure to the time that gets *get_time* (this is why it will get messy later on with various measures).

When calling this function Kivy, which is set to call the animation function repeatedly, waits until this measure is done before calling it.

Basically, when anything is called during the animation loop, the next call to the animation function is queued last.

This is why, when doing to much between frames, a drop in the frame rate can be seen. For this very same reason, the computation time has to be the smallest as possible, there are heavy visual effects if it is not taking care of.

5.3 Computation time

Before getting into it, a way of checking the actual computation time is needed. I used the *timeit* module, getting the time before and after a computation.

When reducing the computation time, there are some factors that are important to focus on. Here I will review: excessive looping, efficient implemented methods and truncating.

Excessive looping

Most of my loops were while: normalizing eigenvectors, calculating components and building **H**.

- **Normalizing eigenvectors.** Calculating each normalization factor and then multiplying each eigenvector by it was done with two loops. Changing it to simple $*$ multiplication, implemented by Python itself, changed the computation speed remarkably.

  Since when running the app one of the first things called were *eigenparam*, the time when opening the app changed after using $*$. With $N = 1000$ and 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.

- **Calculating components.** Before, explicit loops and list comprehension (almost equivalent) were used. In the previous mindset of using already implemented operations to multiply, the components are finally calculated using:

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

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

$$C = \left((evect^*)^T * \Psi\right)^T \tag{6}$$

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

$$comp = deltax * \left[sum(C, axis = 0) - C[0,:]/2 - C[-1,:]/2\right] \tag{7}$$

  In *sum*, axis = 0 means adding up only columns (each column is added up).

  Only two lines are then used to calculate the components! Lines containing (6) and (7).

  The difference in this case can be seen looking at the times between frames.

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 different 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 appreciated. All this tests were run using 1/60 as interval argument in *Clock.schedule_interval*.

- **Building H**. Instead of using loops, I 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). I 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. I even tried using *diagflat* instead of *diag*. It improved diag's results but didn't beat looping run times.

In the view of these results, looping is kept when building **H**. This would mean that the methods used already have loops on them, not implementing a more efficient way of doing it.

Efficient implemented methods

Looking for more specific methods to use the way the eigenvectors and values are found changed, as well as the way $\Psi(x,t)$ is computed.

- **Eigenvectors and eigenvalues.** Taking advantage of the shape of **H**, tridiagonal, *eigh_tridiagonal* can be used, from *scipy.linalg*. This even avoids building **H**, since only the diagonals have to be passed.

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.

The results really are an improvement over the older way.

- **Computing $\Psi(x,t)$.** Here I was using a matrix to do the *evect* and *comp*exp* product, which is really fast. But actually building the matrix can be done faster by building an array and turning it into a matrix. Doing so using $N = 800$, times doing the calculation improved:

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

Truncating

The last method used to try improving computation times is the already explained truncating of the wave function, which simply avoids computing some of the components.

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 this concerns the Hamiltonian, its eigenvalues will be energy values. This means a limit on the energy values is set.

With a max eval of 200, computing the eigenvalue and vectors the time is reduced to half. Not a huge improvement, but it is one.

When considering the whole computation, 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 (until a certain point). Actually the wave function is more isolated from the walls.

Implementing these improvements and plotting both the truncated and the whole wave function, the second version of the application results in:

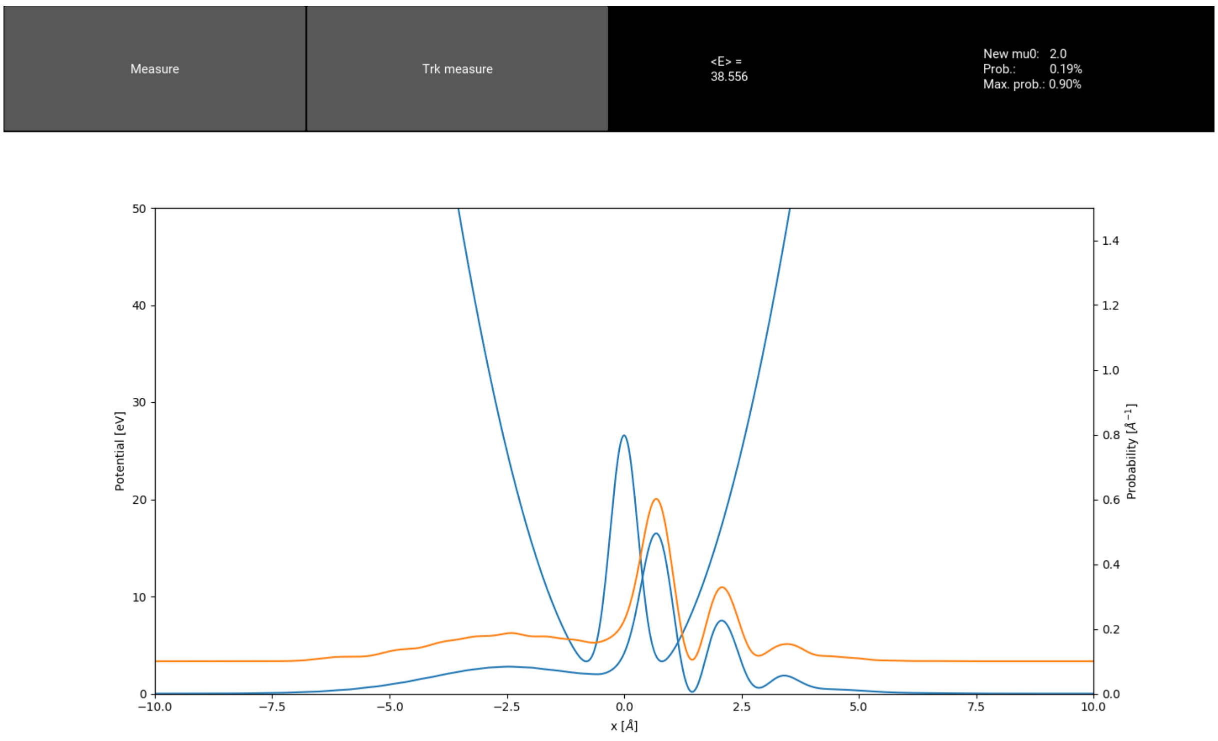


Figure 9: Second version of the application created during this project. Additionally contains the truncating alternative for comparison. If this way had to go through, only the truncated wave function would be shown.

5.4 Basic game

Included in this section (5) for a chronological reason: it contains all the work done before summer. Until now everything done before exams. This particular section was done in the short period between exams and summer.

After significantly improving computation times, introducing the truncated wave function wasn't worth the difference and the messy code. So it was commented out and eventually definitely taken out of the code.

At this point, only two steps remain to get the the first basic game: controlling plotting velocity and game mode (color zones).

Controlling plotting velocity

Here is where controlling the time with *get_time* gets uncontrollably messy. To avoid explaining the mess necessary to change velocity using *get_time* for just removing it later, I will jump forward to the final time control used.

Instead of using *get_time*, my own time parameter is used: a variable t that every frame gets added some quantity dt, that can be multiplied by some factor to change velocity.

$$t_{new} = t_{old} + vel\_factor * dt$$

Pausing and playing is then controlled with a conditional statement controlled by a variable called *pause_state*, which is True of False.

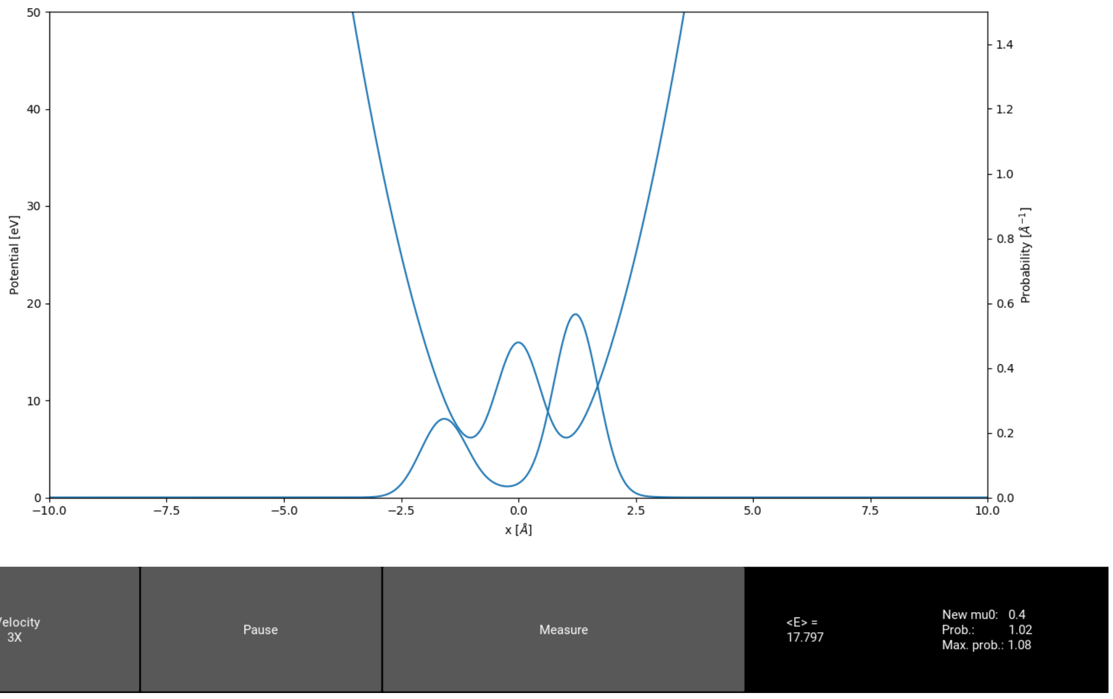


Figure 10: Third version of the application, velocity controlling buttons added and truncating removed.

Game mode

As proposed as the initial idea, the game would include color zones, including some gaming system (points for now) and different levels.

- **Color zones.** Some positions intervals have to be indicated as the forbidden zones. Red and green areas are painted indicating this zones.

  This painting is done with matplotlib's *fill_between*. It loops (not to many indices) some positions that denote the intervals.

  On top of this, filling colors mix if put on top of each other, I wanted to avoid that. So psi's fill (supposed to be black) doesn't have to mix with the background red and green colors. For that, the zones fill changes every frame.

  A general change in the coloring followed this implementation. Black background is set.

- **Levels.** One big trouble I ran into while starting the game (and one that will follow until almost the end) is how to create different levels.

  Creating a level consists of giving the potential's parameters and the position of the red and green zones. A file was created, with this numbers in each line. Potential is always a double well.

- **Gaming system.** First of all, it has to be checked if the level was passed or not.

  While painting the zones an array with the red zone points is created. Checking if the new position is in or not is simply done with an *in* statement when measuring.

  Therefore, the measure function changed. After changing it, it did:

- Take new position
- Check level and count points accordingly
- Pass level (read from file): new potential (thus new eigenbasis) + new zones
- Reset psi
- New components (in the new eigenbasis)
- Time reset

After implementing these changes:

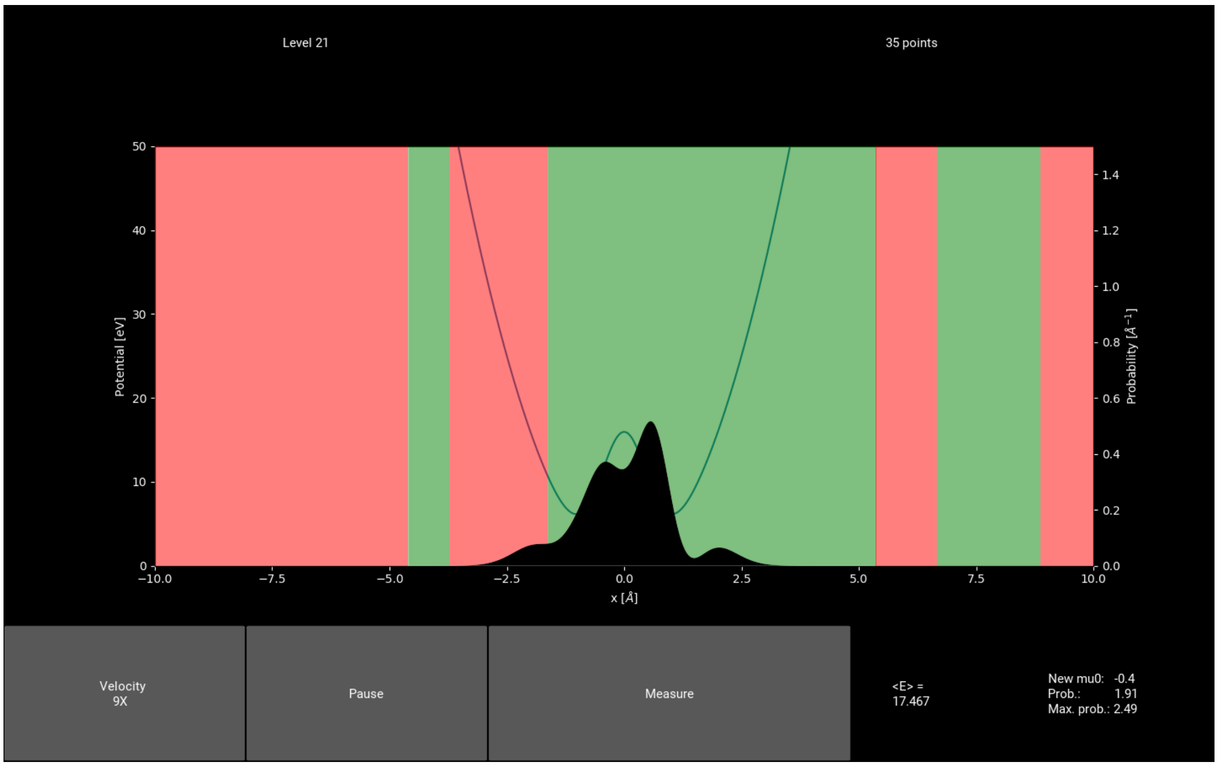


Figure 11: Fourth version of the application, first most basic game.

With the plotting velocity controlled and the points and color zones implemented, **this is already a game**, although not a very good one.

This version is the last one in the period previous to summer. The only thing left is to make of this game, a good game (not trivial).

6 Designing the game

To do a playable game out of the most basic one done in July, two major improvements had to be done: visual features and game content.

Visual features include: visualization of the result of the measure (indicate where it is, whether it was in or out..) and basic aesthetics (buttons distribution, styles, colors...). This two visual improvements will be done throughout this section.

With the game content, I am referring mainly to designing playable levels. As it shall be seen, the content finally splits up into a game part and an illustrating part.

6.1 First visual features

Contradicting this section introduction for this particular feature, the first addition after summer was the use of the keyboard in the game (which is not either visual feature nor content).

Later though, gray map and choice of colors is presented, following the visual features discussed in the introduction.

Spacebar

Kivy includes keyboard management in the following way. An instance of the keyboard can be created *requesting* the keyboard (one of Kivy's *Window* methods).

This instance is an object that registers every event involving the keyboard (keys pressed and released). It is an object that yields events. Using the function *bind* it is possible to bind this events to a custom callback.

In my case, a general *on_keyboard_down* callback is passed: from the event it gets certain default arguments like which key is pressed, among others.

Finally, if the key pressed is space, the *measure* function is called. The space button is now bound to measuring.

One minor problem occurred when playing the game. A click in the app's window released the keyboard for some reason. The temporary solution I worked out was introducing a button named *KB* that could be clicked to request again the keyboard if lost.

Gray map

First real visual feature was an improvement on the visualization of the wave function itself.

Although the wave function was represented by the distribution probability in the main plot, maybe the concept of it meaning the probability of a particle being there wasn't clear.

The initial idea was to draw 'particle' images fading where there was low probability.

Since the drawings should change every frame, computation time drop heavily as well as the images overlapping didn't give the desired result.

A gray map version of the probability was the final feature. Choosing properly the interpolation method, the computation times were just fine.

Choice of colors

The application mainly had 4 colors: red, green (zones) and black and grey (outside the plot).

A careful choice of the red and green colors was made. The default colors didn't quite convinced me.

Consulting some web pages online and the possible combinations of red and green, I end up choosing the colors using an on online palette [6] . The final choice was a triad of red, green and blue (for the potential line).

After these first visual features were included, the current version turn out to be as shown in Figure 13.

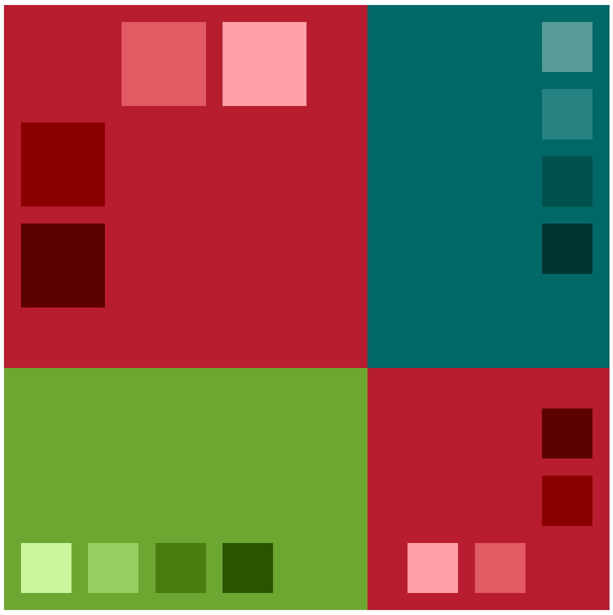


Figure 12: Color choice for the red and green zones, and potential (blue).

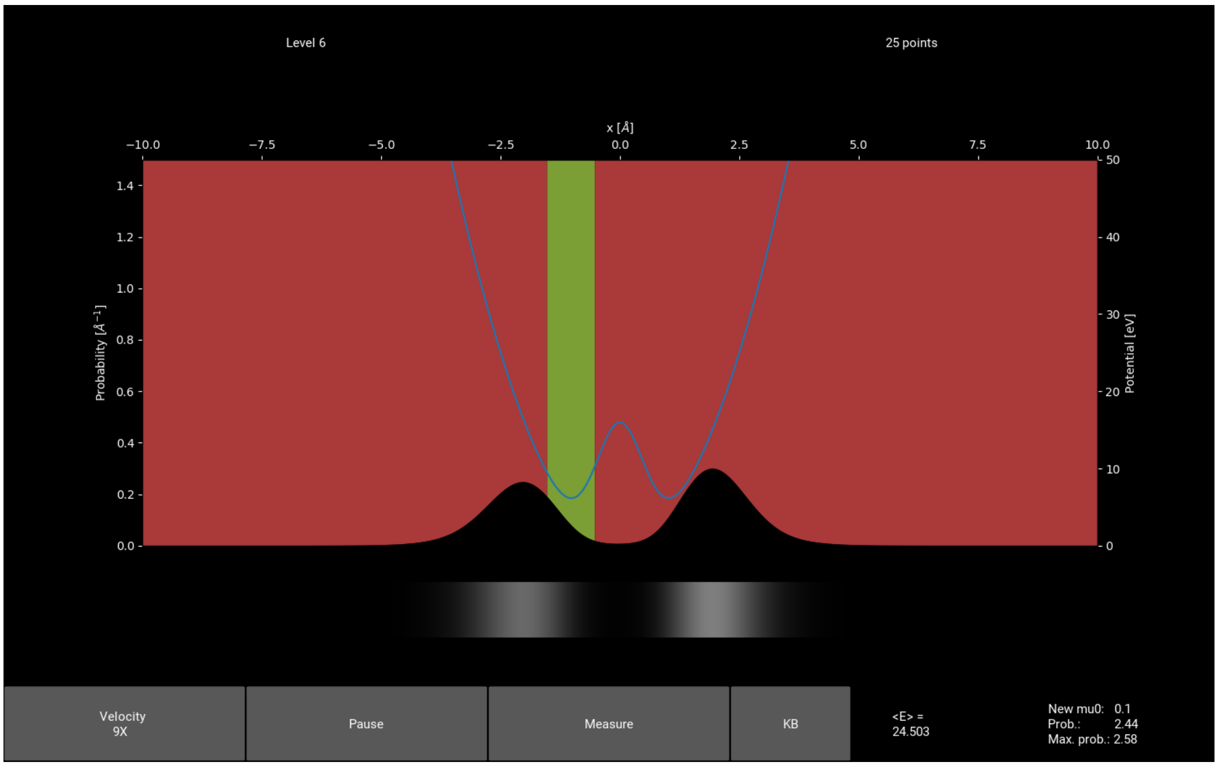


Figure 13: Fifth version of the application: spacebar, gray map and color introduced.

6.2 First content creation

The first attempt for improving the quality of the game content were focused on changing the game mode (currently points and levels) and creating a variety of playable levels, which turn out not being trivial.

Game mode

New game mode: **lives system**.

This lives system worked as follows: only pass level if result appears in the green zone, every miss makes the player loose a life (out of certain initial maximum amount). If no lives are left, it is game over. The goal for the player is to try and go as far as possible.

Visually, hearts representing lives were introduced on top, which were replaced with skulls when lost. A game over pop-up was implemented as well.

Consequently, a restart button had to be introduced.

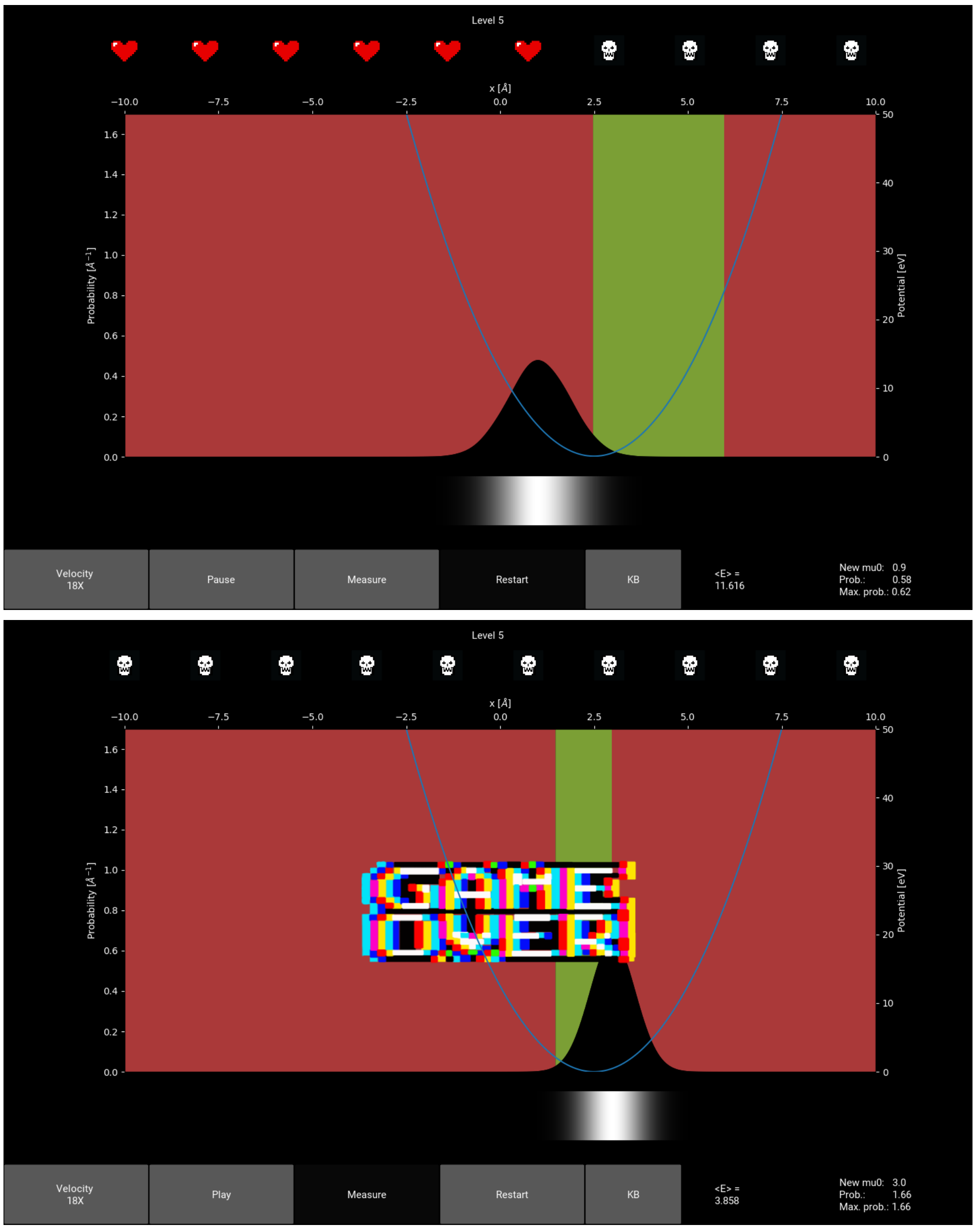


Figure 14: Sixth version of the application: lives game mode included (and consequently a restart button). *Bot*: pop-up image when the game is over.

Playable levels

Saying playable levels I am referring to:

- Zones appearing where the wave function can actually reach (impossible levels otherwise).
- Avoid wave functions not moving (in the middle of a well). The wave function should have always a considerable energy.

Since the initial wave function after a measure is random, levels can not be exactly written in advance to fulfill this two items. No prediction can be exactly made on how will the wave function behave.

Even though, the first attempt here were to include different potential shapes. Since we are using a settings file to create the potential, it had to include what kind of potential it is creating.

Then, inside a code, checking some key word from the file, different potentials are created.

During this experimentation of what kind of potential are good for a game (I tried double wells, triple wells, harmonic and Wood-Saxon) I end up classifying the potentials between ones were quantum special cases were easy to see, and ones that were made simply to play.

The illustrative ones were displayed first and later the gaming ones. Even though, the game wasn't playable at all. Illustrative levels were too slow and some times imposible to pass, and still had only a few gaming levels. But it was an important first step.

This distinction is important, it is part of the final structure of the game.

6.3 Second visual features

Here, a first visualization on the measure result was include and some changes in the main plot aesthetics.

Visualization of the measure

In the previous version of the game, for a player not used to working with the application, it was hard to realize where the measure appeared (it happen to fast).

In order to correct that, an indicator of the position result was introduced. It was two arrows (with no head) in the resulting position: one white showing the height of the wave function were the result appeared, and another purple indicating the top of the wave function on that moment.

While indicating where the position was, a feeling for the relative probability of the actual result was shown.

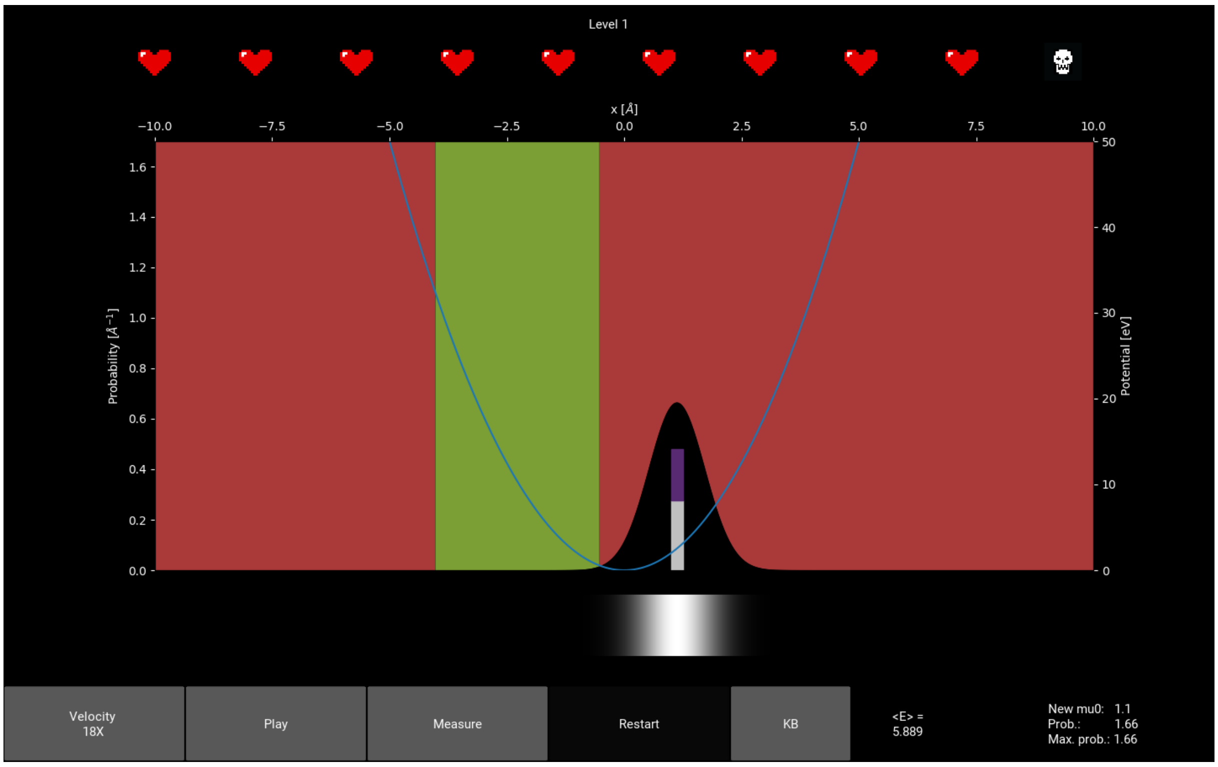


Figure 15: Seventh version of the application: visualization of the measure introduced.

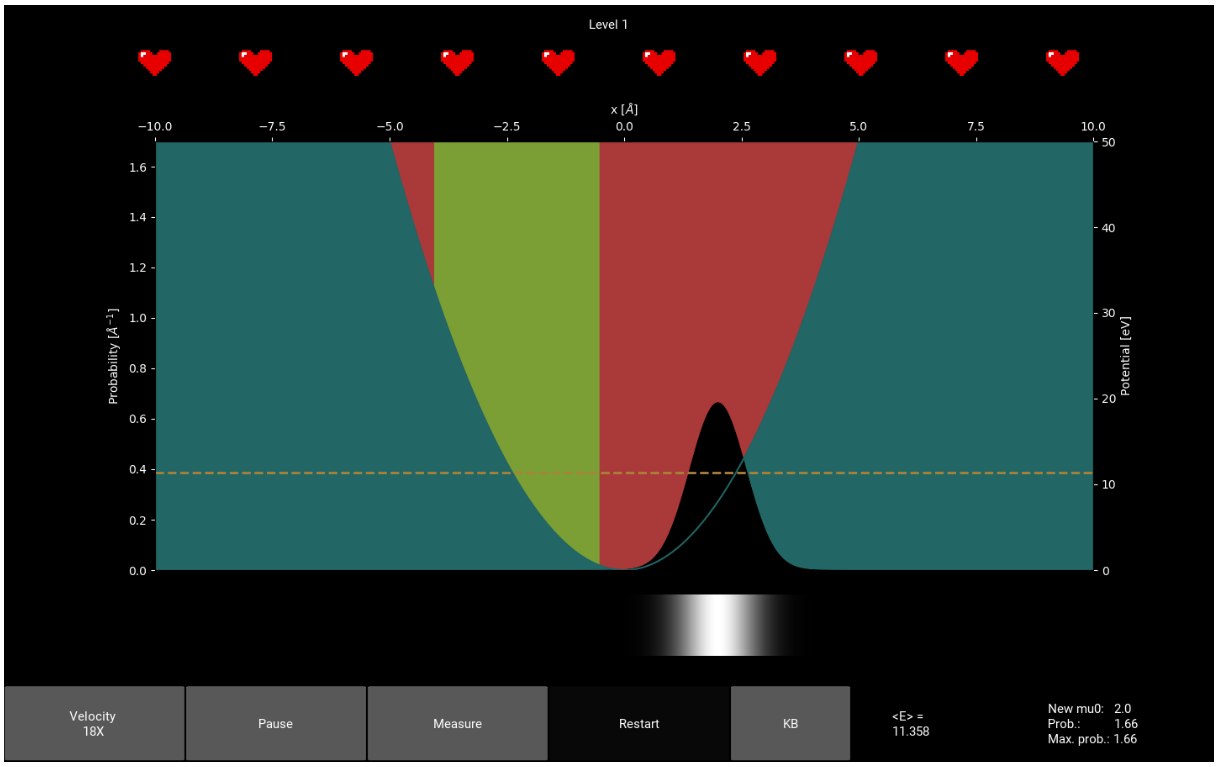
In the figure 16 the bar is indicating where the new position appeared (the new psi is centered there as can be seen). The total height of the bar indicates the the height of the most probable result in the moment of the measure, and the white height indicates the height of the actual result (height as probability in a visual sense).

Main plot aesthetics

This is, a change in the potential plot (it currently is a line poorly seen) and the energy information.

The potential was filled with the already chosen color (blue). Considering the mixing of the fills, the actual fill had to start from Ψ to the potential line (checking which is on top) and then, the color zones fill all the way up to the top.

The energy information was set to be a dashed line. Its value can be checked looking at the potential axis.



```
<E> =
11.358
```

```
New mu0:  2.0
Prob.:      1.66
Max. prob.: 1.66
```

Figure 16: Eighth version of the application: main plot aesthetics.

6.4 Birth of the final game structure

Centering again on the creation of playable levels together with managing levels gave birth to the final game structure.

By then, although I had been trying from the introduction of levels to create decent playable levels (with many potential shapes and entertaining enough) no real progress was done.
A change on the approach had to be made.

As commented in the previous content section 6.2, a difference in the use of the levels had began to appear: some more game oriented and others with illustrative purposes.
Since the only playable and easy to manage potential was the harmonic, I restricted the gaming levels to harmonic potentials only. The rest was exclusively for illustrating purposes.
One trouble I ran into with those illustrative levels was that some were impossible to pass, so a way of passing levels had to be introduced as well (an *skip* button).

Therefore, a final decision was made, the application will split into two parts: the illustrative levels along with level management constitute the illustrative part, and the harmonic levels the gaming part.

- **Helping mode** Also introduced in the middle of this change of approach, was the *helping mode*: the new position wasn't picked randomly if helping was activated, it directly was the most probable one (the peak of the wave function).

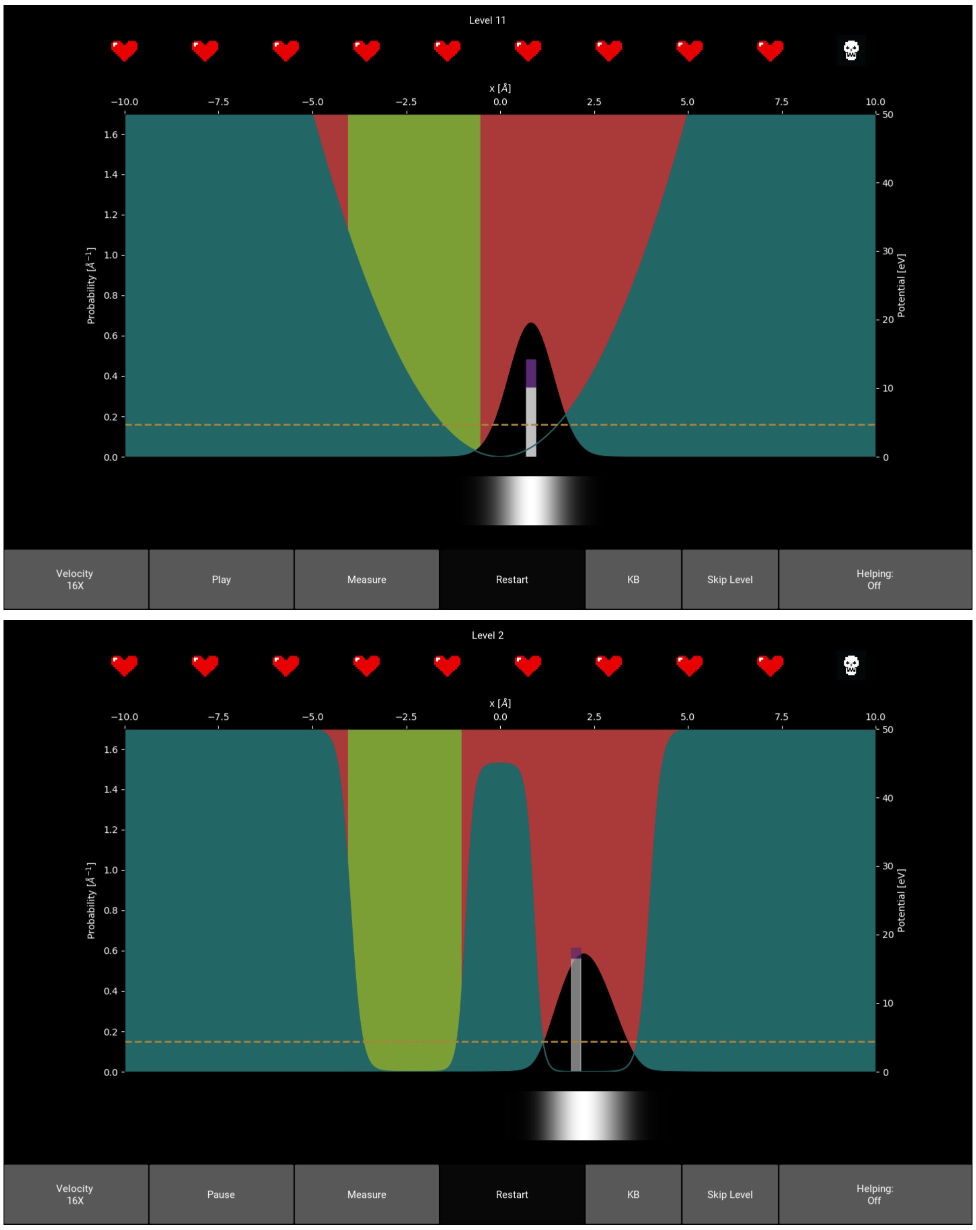


Figure 17: Ninth version of the game. Skipping levels option included, and definite separation between game and illustrative levels done. *Top*: Example of gaming level. *Bot*: Example of illustrative level (quantum tunneling). Here the skip option is mandatory.

Since the levels were considerer to be split in to kinds, a separation in the application had to be made: in the next section this is explained.

6.5 Screen System

In order to create a clear distinction between the gaming levels and the illustrating ones I turn to a Kivy 'widget', ScreenManager, which allows to implement a screens system in the application.

Before jumping into its inner workings, the main goal was to get three screens:

- **Starting**. Initial screen when running the app. title and UB logo will be included, as well as buttons leading to the other screens. On top of that, an information pop-up will be introduced here.

- **Gaming**. Will contain the game only (basically what we had already) and buttons for changing screens as well.
- **Illustrating**. Will contain the illustrating levels. The lives system will no longer be here. Buttons for changing the screen will be included here as well.

Introducing screens caused some major changes in the code.
Each screen has to be contained in a different class. Then, another class, *ScreenManager*'s child, will have access to every screen (thus its classes). *ScreenManager* is an already implemented class in Kivy.
The application is then build with this child.
I had to almost copy paste all my code to create the gaming screen and illustrating screen (they are almost the same). This doubled my code size.

On top of creating these new classes, **initiation problems** (when did each screen initiate and hence the corresponding widgets started to exist, and thus its ids), **looping control** (although one screen wasn't shown, its loop calling the drawing didn't stop, *ciao* computation times), **keyboard bindings** to the correct screen, **Kivy file** new structure and **checking again run times** had to be dealt with.
These will be briefly reviewed, since it more technical than conceptual and everything is properly indicated in the commented code.

- **Initiation problems.** The first drawing and the first problem resolution was previously done when initiating the application. At that moment the Kivy id's (like the box where the plot was drawn) already existed.

  When building the app with a difference class (*ScreenManager*'s child) the box didn't started existing until later on. So, everything done in the previous init, was done in a *fake* initiation called when everything did exist, a *pseudo_init*.
- **Looping control** and **keyboard bindings**. Although screens sometimes aren't showed, they do not cease to exist: the not shown screen may keep the keyboard bound to itself and might not even stop running (useless).

  So when transitioning between screens this had to be taken care of: requesting keyboard to the new screen happened and the loop started, as well as the previous screen loop stoped.
- **Kivy file**. Remember that Kivy had a widget tree for each class. Before there was only one, but now there are several. Each class needs to have an individual tree. This change, doubled or tripled our code in the Kivy file.

The last big problem needs to be commented on its own. I really did put many hours into solving this problem.

Plotting run times. Since a lot of widgets were introduced, I had to take care of the run times again. On the contrary, the application wouldn't be fluent at all, which would make the game useless.
Checking what took the most time to execute between frames, table 5, clearly drawing takes the most time.

Evolution computation	Psi	Fill	Gray map	Draw
0.005	0.001	0.006	0.001	0.055
0.007	0.0015	0.006	0.001	0.07

Table 5: In seconds, run times for each process between frames when paused. Top with default window, bot with full screen window, the difference.
This values were calculated before introducing screens and before introducing some general changes, so they will not be the same as later results shown. The important part is the comparison between drawing times and computation times.

Therefore, to increase frame rate I had to focus in drawing. After a considerable search, I found several references in what object of the plot takes more time plotting. Turns out that drawing the axis is *incredible* expensive! See table 6.

To avoid drawing them every frame I carry out another exhaustive search about matplotlib way of drawing [4] . It actually exists a method to separate the artists (each plotting object painter, roughly explained) and command only a few of them to draw.

The problem is that the one drawing the plot in my application *isn't matplotlib*, Kivy actually does the drawing! And it doesn't offers the possibility to separate matplotlib's artist.

So another solution was to be found (the axis really were messing with the frame rate). Since the plotting is for popularizing purposes, the next solution could be done: drawing *fake* axis using arrows and annotations.

The results actually accomplish the goal of increasing the frame rate, see table 6, and in a noticeable and remarkable way.

	paused normal	playing normal	paused fullscreen	playing fullscreen
No axis	1/60	1/30	1/30	1/20
Real axis	1/20	1/15	1/15	1/12
Fake axis	1/40	1/25	1/25	1/17

Table 6: In seconds, run times for each different axis method. The maximum to ever aim to is 1/60, where application does the minimum (paused and no axis). Faking the axis gives an intermediate result between no axis and real ones.

Finally, the design of every screen is to be presented. For each one, there is two versions, since this last part was done in the last two weeks where two versions were done, one for each week.

In the last of these versions, the final fanciness design was done.

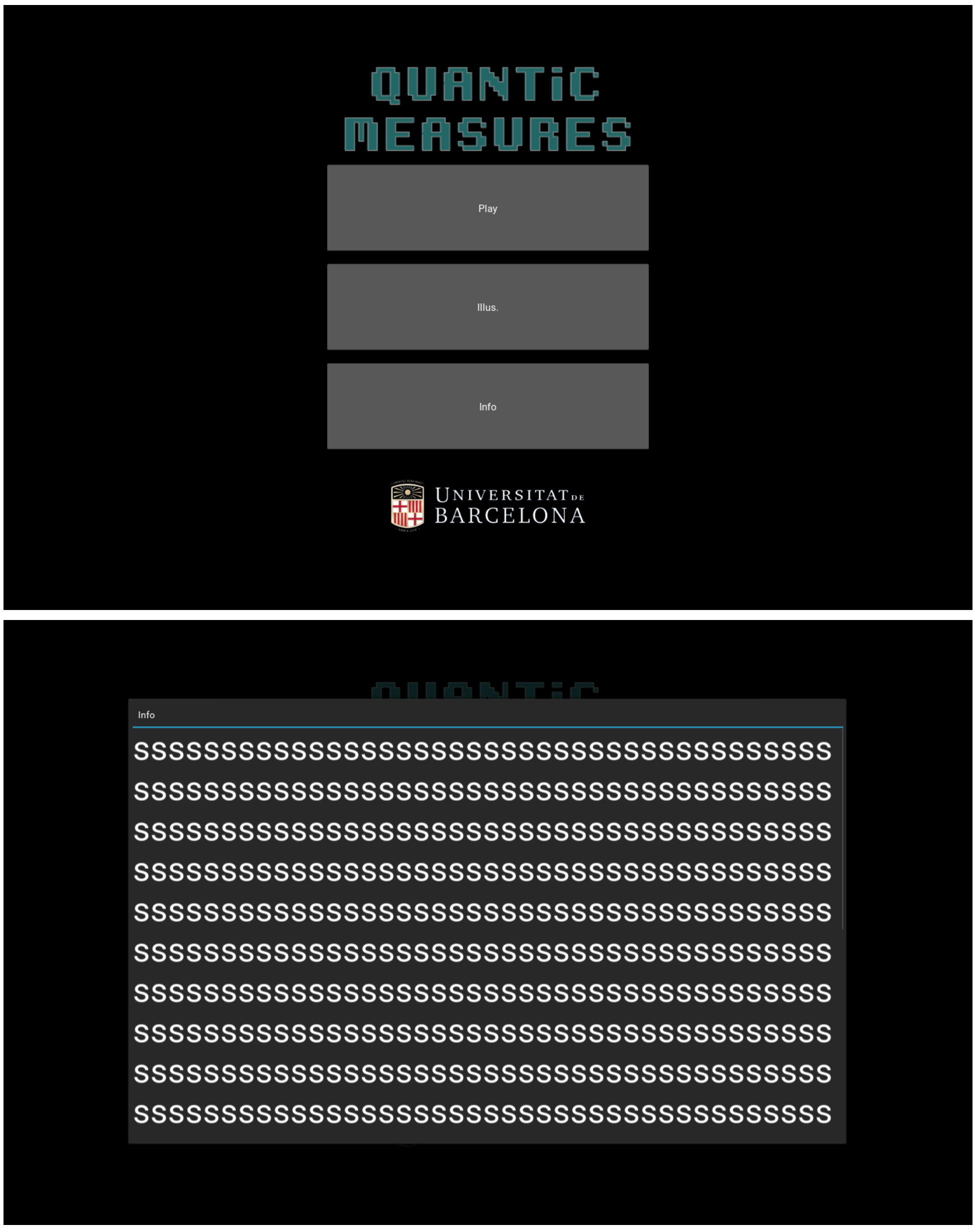
Starting

This screen is to include the title image, UB logo, some information displaying and buttons leading to the other screens.

This screen is the simplest one obviously, contains no animation.

In the **first** version, the layout sketch was implemented in a basic way. It include a pop-up where game information would be displayed.

[4]No concrete references can be made here, since I cross checked several online sites until forming the clear picture.



ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss

Figure 18: Tenth version of the game: starting screen. Includes buttons to other screens, an info pop-up (*bot*), title image and UB logo.

Two comments on this screen.

First, the choice of font for the title image (chosen from an online site [7]) was the final font chosen for the entire application. This font is called 8 bit madness.

Second, although looks like not too much happens in this screen, in this screen initiation is happening the first draws and problem solving of the other two screen (calls to *pseudo_init* functions). As mentioned before, these had to be done in the proper moment, when the screens actually existed.

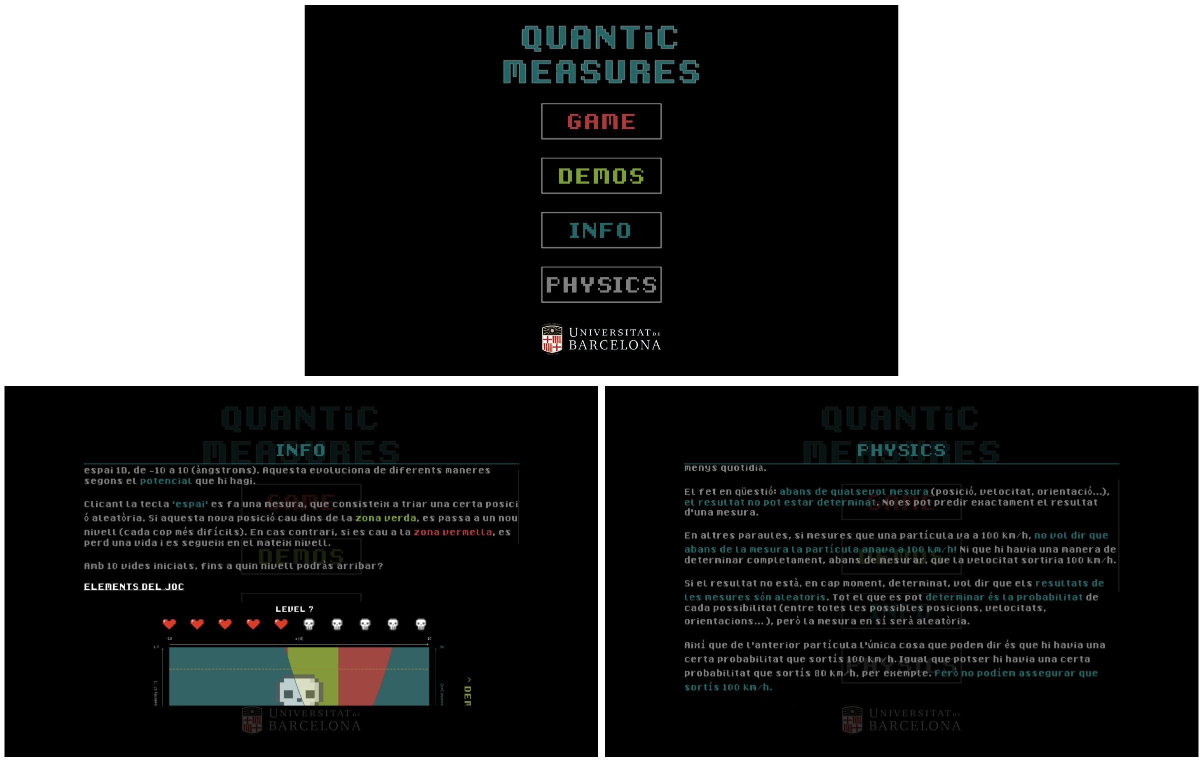
In the **second** version, only the information display was changed.

A new pop-up was introduced. With these two pop-ups the information was separated in game info and physics info.

The initial idea was to write the text and possible images in a pdf and display it in the pop-up, but Kivy doesn't have an stable implementation for displaying pdf directly.

Following this change, the general font was changed as well. For the accents in some of the words (the text is in Catalan) another font was used, VT323.

Both fonts had to be downloaded an registered in the Kivy label base.



espai 1D, de -10 a 10 (àngstroms). Aquesta evoluciona de diferents maneres segons el potencial que hi hagi.

Clicant la tecla 'espai' es fa una mesura, que consisteix a triar una certa posició aleatòria. Si aquesta nova posició cau dins de la zona verda, es passa a un nou nivell (cada cop més difícils). En cas contrari, si es cau a la zona vermella, es perd una vida i es segueix en el mateix nivell.

Amb 10 vides inicials, fins a quin nivell podràs arribar?

menys quotidia.

El fet en qüestió: abans de qualsevol mesura (posició, velocitat, orientació...), el resultat no pot estar determinat. No es pot predir exactament el resultat d'una mesura.

En altres paraules, si mesures que una partícula va a 100 km/h, no vol dir que abans de la mesura la partícula anava a 100 km/h! Ni que hi havia una manera de determinar completament, abans de mesurar, que la velocitat sortiria 100 km/h.

Si el resultat no està, en cap moment, determinat, vol dir que els resultats de les mesures són aleatoris. Tot el que es pot determinar és la probabilitat de cada possibilitat (entre totes les possibles posicions, velocitats, orientacions...), però la mesura en sí serà aleatòria.

Així que de l'anterior partícula l'única cosa que podem dir és que hi havia una certa probabilitat que sortís 100 km/h. Igual que potser hi havia una certa probabilitat que sortís 80 km/h, per exemple. Però no podíem assegurar que sortís 100 km/h.

Figure 19: Eleventh and final version of the game: starting screen. Font and coloring changed, another pop-up included: one explains the game, the other the physics of it.

One third of the final app is over.

Gaming

This screen is to include the whole game, as well as buttons leading to the other screens.

As of last version (ninth, in figure 17) four major changes are included in this **first** version (of the last two): color zones under the plot, fading images, random level generator and jokers.

- **Color zones under the plot**. To ease the visualization of the forbidden zones, another color zones plot has been included under the main plot. Shown in figure 20.

  Since previously the color zones were behind the potential fill, and also could lead to probability of actually taking a measure in the 2D plane, this new addition was necessary.

- **Fading images**. In order to continue improving the visualization of the measure, new indicators had to be included.

  This time, with the goal to indicate whether a level is passed or not. Whenever a measure is taken (thus a level is passed or not) an image pops up and fades away: an skull if missed and an arrow up if passed (a 'level up' text in the second and final version).

  This was done by resetting the image's alpha inside the repeatedly called function (like with the arrow indicating the position).

- **Random level generator**. A way of generating automatically random levels.

  This is one of the most happy moments along this project. After maybe two whole months stuck with defining good levels, going through splitting the application in two and all, I finally found a way (having an harmonic potential helped a lot) of creating infinite many levels.

  The core of this generator is the following. Remembering that creating a level is actually just giving the parameters to the potential (harmonic has two: aperture and center) and the forbidden zones interval, the generator creates an interval of values for each parameter and randomly picks each one.

  The key here is to carefully define these intervals. To make playable levels three rules had to be followed: limiting the new wave function energy, keeping the wave function far from the walls and making the zones appear where the wave function has low E (other wise maybe an impossible level). With these rules decent levels are created.

  Playing with the probability distribution when picking the random parameter, the levels become more than decent.

  This system works together with the new way of defining the color zones: only one green zone with a middle position and width. The width decreases when passing levels (as well as the speed of the game).

- **Jokers**. Introducing the option of skip a level. This is done simply by allowing the player to use the skip function a limited number of times (3).

  Doing this, another problem was kind of solved: when the wave function appears in the middle of the well, it doesn't move from there an it is almost impossible to reach any green zone on the side [5]. Using these jokers here, the problem is avoided (more than solved).

The first version of the final two shown in figure 20.

[5]When playing with the probability distribution in the level generator this problem is solved, reducing the probability of the wave function appearing in the middle of the well.

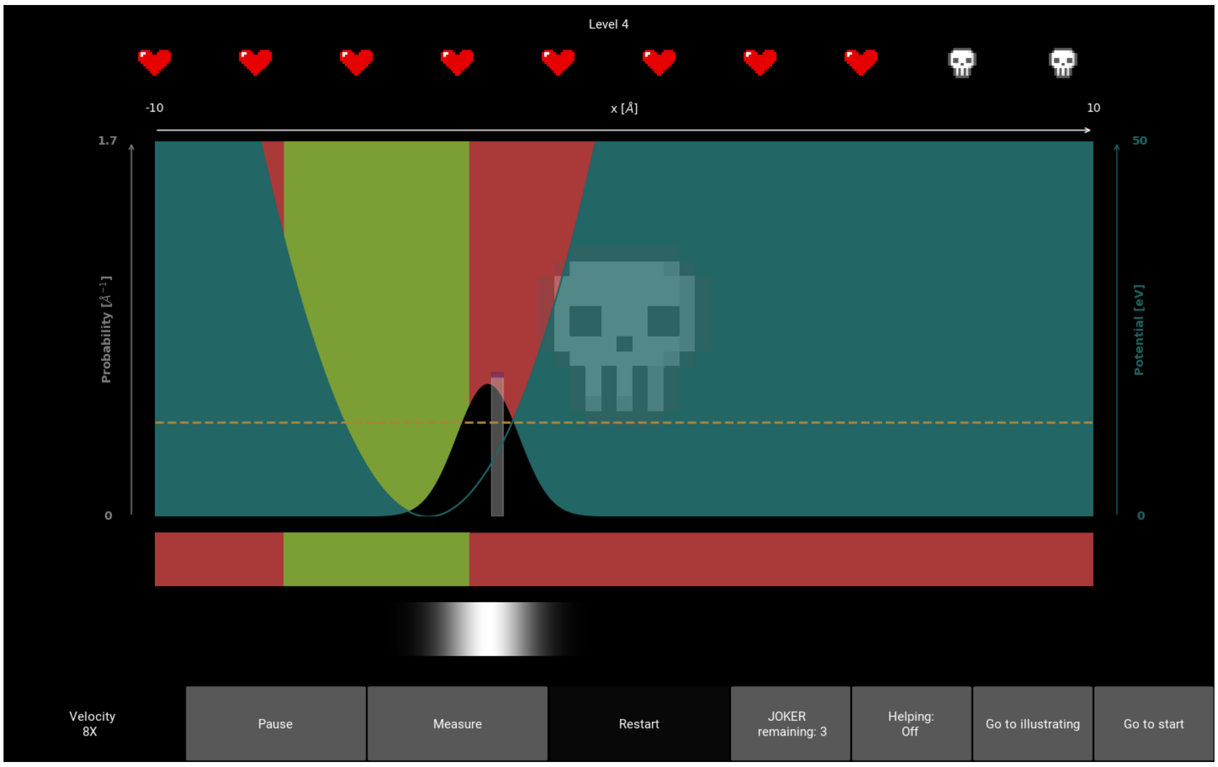


Figure 20: Tenth version of the application: gaming screen. Includes fading images, color zones under the plot, level generator (not appreciated here) and jokers.

As the **second** and final version of the gaming screen, only few things were left to do: removing buttons as much as possible to clean the interface and changing the general font.
Final result shown in figure 21

Illustrating

This screen is to include the demos and a way of moving through levels, as well as buttons leading to the other screens.

As of last version (ninth, in figure 17) three major changes are included in this **first** version (of the last two): color zones under the plot (exactly the same as gaming screen), moving through levels system and clicking issue.

- **Moving through levels system**. As discussed previously in the section 6.4, a way of moving through levels is needed for showing illustrative cases.

  First thing to do, was to actually read the whole settings file and keep it in a list. Therefore, moving through levels would be just moving through a list. Then, skip level just read different indices of that list, and a next and previous button could be implemented.

  On top of these two ways of moving through levels, a way of going directly to the desired level was implemented with a DropDown widget.

  This is just a list of buttons to select the level. Although it wasn't easy to implement, it could actually be done (each selection read an index of that list with the settings).

- **Clicking issue**. The mentioned problem with requesting the keyboard and loosing it (section 6.1)was finally solved.

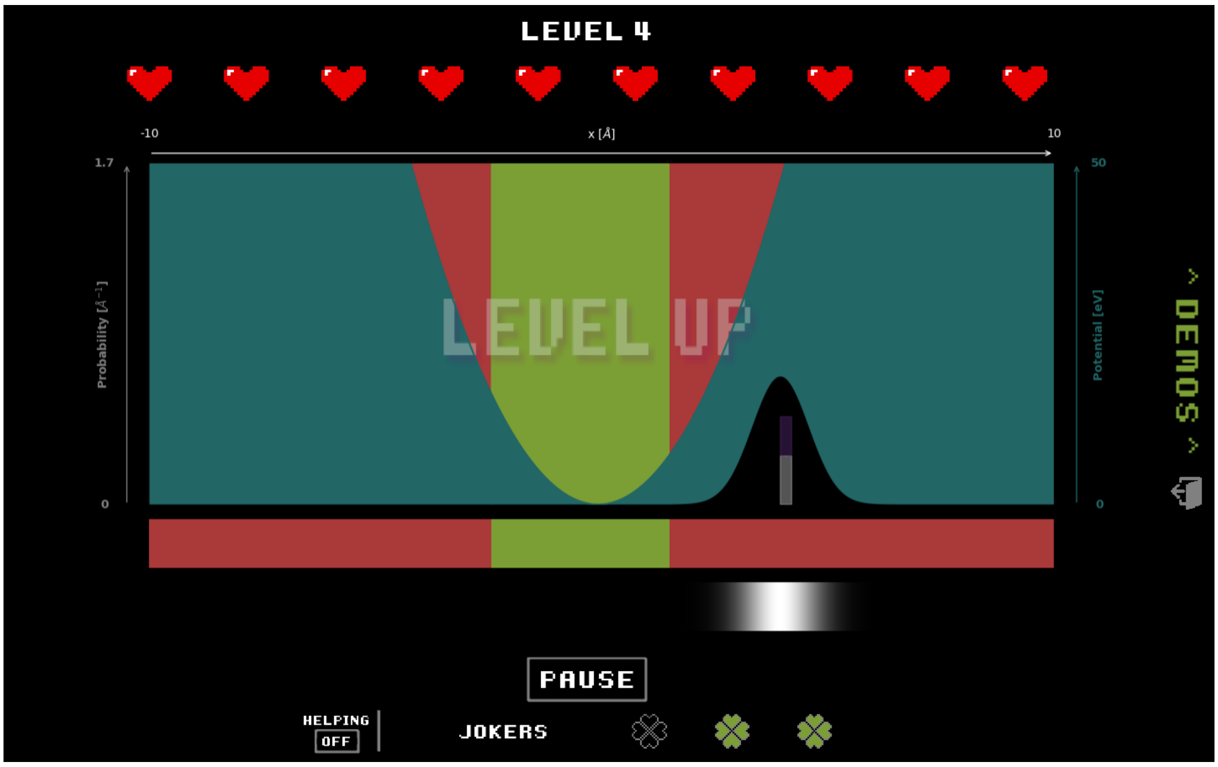


Figure 21: Eleventh and final version of the application: gaming screen. Buttons were removed as much as possible and the general font changed.

I still don't know why clicking anywhere in the plotting box releases the keyboard. What I didn't realized back then was that the figure passed to Kivy was a widget itself, an for this reason it had event management included.

Therefore, the way the problem was fixed it binding the same event that unbound the keyboard, with an extra request. Now, each time the box is clicked, the keyboard is released and then requested again.

With this problem solved, the KB buttons wasn't necessary anymore.

The **first** version of the final two shown in figure 22.

The **second** and final version included a change in the general font as well as the reorganization of the buttons, as shown in the figure 23.

7 Code management

The application is already finished, but some work has to be done with the code: cleaning, ordering and commenting.

The first two basically consists of a review of the actual code, and looking for deprecated pieces of code or possible optimizations. It is not even necessary to expose it here. The commenting on the other hand, was an intense work.

I followed Python's Enhanced Proposal (PEP) indications. PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment.

The lines I followed commenting my code came from PEP 257 [8] and other sources [9].

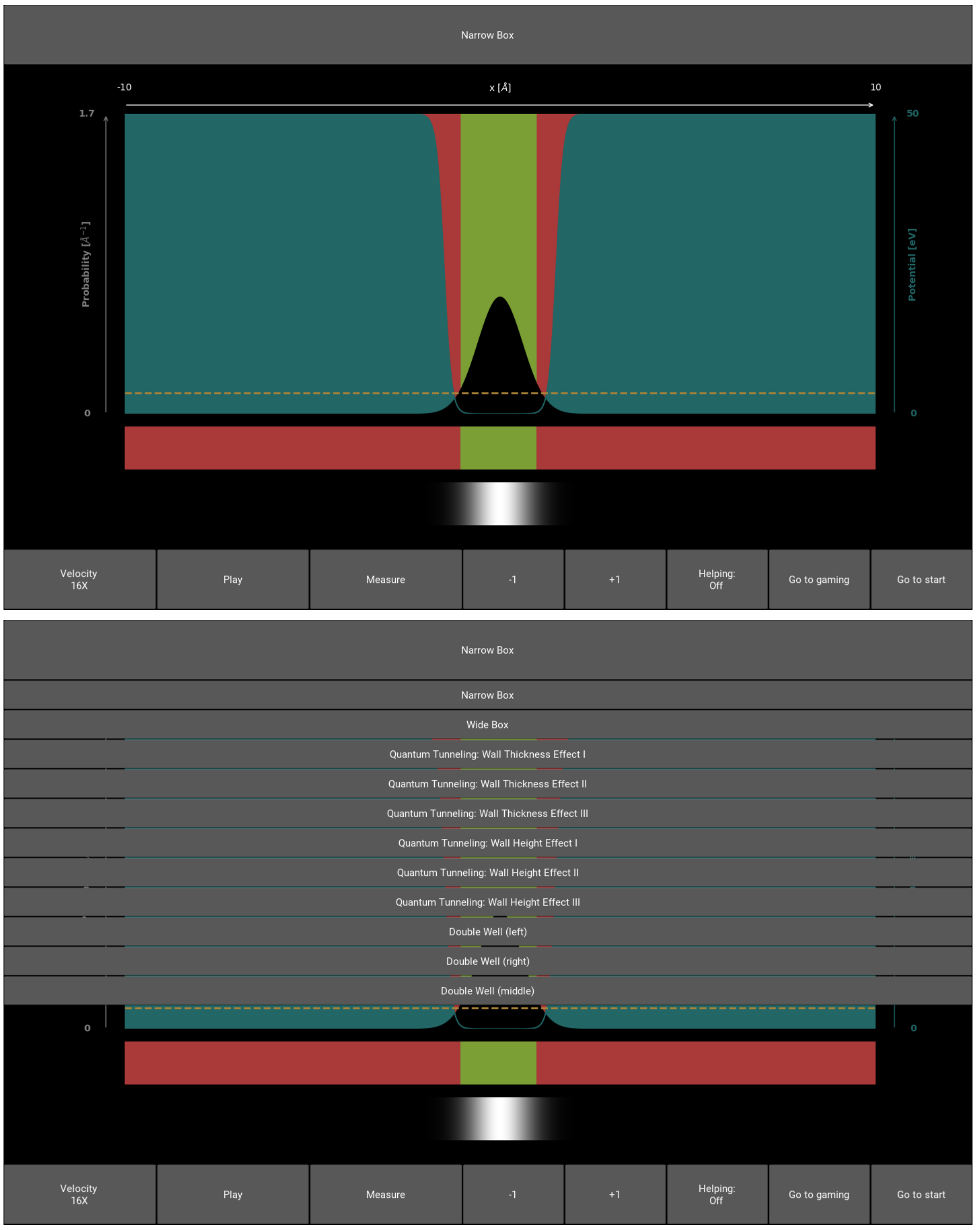


Figure 22: Tenth version of the application: illustrating screen. Includes level management and the KB button has been removed. *Bottom*: here the dropdown where levels can be selected is shown.

The supposed final version of the code was uploaded to GitHub on December 22nd, 2019.

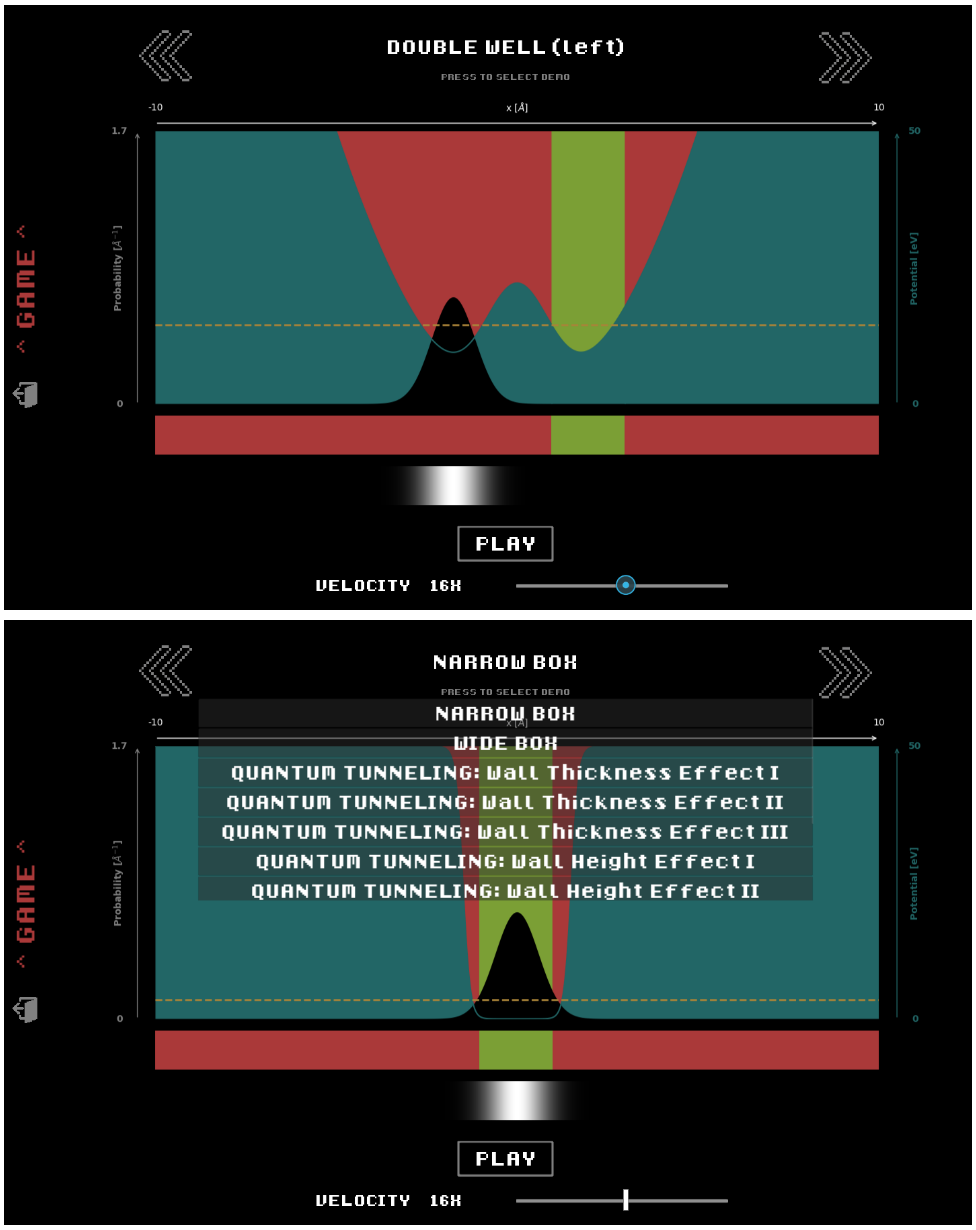

DOUBLE WELL (left)
PRESS TO SELECT DEMO
-10
x [Å]
10
1.7
Probability [Å⁻¹]
0
50
Potential [eV]
0
GAME
PLAY
VELOCITY 16X
NARROW BOX
PRESS TO SELECT DEMO
NARROW BOX
WIDE BOX
QUANTUM TUNNELING: Wall Thickness Effect I
QUANTUM TUNNELING: Wall Thickness Effect II
QUANTUM TUNNELING: Wall Thickness Effect III
QUANTUM TUNNELING: Wall Height Effect I
QUANTUM TUNNELING: Wall Height Effect II
PLAY
VELOCITY 16X

Figure 23: Eleventh and final version of the application: illustrating screen. Most of the buttons have been reorganized. The new dropdown is shown in the *bottom* image.

Conclusions

8 Exhibitions

This project will be presented in a couple of exhibitions.

- **YoMo**. It is an interactive fair of science and technology that translates school understandings from the classroom to the real world through experiences.
- ***13a Festa de la Ciència***. It is an initiative of the *Ajuntament de Barcelona* to promote the scientific knowledge and to involve the citizens with its progress.

9 Valuation

Throughout this project I learned a lot of valuable lessons.

It work as a first individual physics project (with the help of the rest of the group of course) of large duration.

It make me learn how to adapt to situations where I got the idea but not the tools. I found myself in a lot of situations where I knew nothing about how to make what I wanted, but searching for it and finally doing it. I think this is one of the most valuable lessons I learned.

During this project I also learned how to keep track of my progress and organize myself.

References

[1] QuantumLabUB Github repository. (*Master: brunojulia*)
`https://github.com/brunojulia/quantumlabUB`

[2] Python documentation.
`https://docs.python.org/3/index.html`

[3] J. M. Gómez, R. Graciani, M. López, X. Luri, S. Estradé, A. Rosell. 2016. *"Programació en Python. Per a físics i enginyers"*. Universitat de Barcelona, Barcelona. 281 pp [online] Available at: `http://37.187.177.141/v1.2/fitxers/fitxer-1525468369` [Accessed March 9th 2019]

[4] Kivy documentation, API Reference.
`https://kivy.org/doc/stable/api-index.html`

[5] Dusty Phillips. Apr 2014. *"Creating Apps in Kivy."*. O'Reilly Media, Sebastopol. 188 pp [online] Available at: `https://github.com/pd-Shah/kivy/blob/master/Dusty%20Phillips-Creating%20Apps%20in%20Kivy_%20Mobile%20with%20Python-O%27Reilly%20Media%20(2014).pdf` [Accessed April 8th 2019]

[6] Paletton, color choosing.
`http://paletton.com/#uid=3000u0kllllaFw0g0qFqFg0w0aF`

[7] Font Squirrel.
`https://www.fontsquirrel.com`

[8] PEP 257, Docstring Conventions.
`https://www.python.org/dev/peps/pep-0257/#one-line-docstrings`

[9] Real Python. Writing Comments in Python (Guide).
`https://realpython.com/python-comments-guide/#how-to-write-comments-in-python`