Assignment 6 on Python: Simulations

March 6, 2018

1 Introduction

Python is very good at simulating models. In this assignment, you will simulate a tube light.

2 The Tube Light Model

We use a 1-Dimensional model of the tubelight. A uniform electric field is present, that accelerates electrons. Electrons are emitted by the cathode with zero energy, and accelerate in this field. When they get beyond a threshold energy E_0, they can drive atoms to excited states. The relaxation of these atoms results in light emission. In our model, we will assume that the relaxation is immediate. The electron loses all its energy and the process starts again.

Electrons reaching the anode are absorbed and lost. Each "time step", an average of N electrons are introduced at the cathode. The actual number of electrons is determined by finding the integer part of a random number that is "normally distributed" with standard deviation of 2 and mean 10.

The problem is to plot the light intensity as a function of position after the process has reached steady state. As you probably know, there are "dark spaces" and we will try to find them in our simulation.

The algorithm is given and you have to fill in the steps with code.

- We create a simulation universe. The tube is divided into `n` sections. In each time instant, `M` electrons are injected. We run the simulation for `nk` turns. The electrons are unable to excite the atoms till they have a velocity of `u0`. Beyond this velocity, there is a probability `p` in each turn that a collision will occur and an atom excited. Note that the electron's velocity reduces to zero if it collides.

```
n=100  # spatial grid size.
M=5    # number of electrons injected per turn.
nk=500 # number of turns to simulate.
u0=5   # threshold velocity.
p=0.25 # probability that ionization will occur
```

- Write code to get user update of these values. Look at the usage of sys.argv and use it for this task.
- Create **vectors** to hold the electron information. The dimension should be nM (this is an over estimate but that is ok). You need vectors for
  - Electron position `xx`
  - Electron velocity `u`
  - Displacement in current turn `dx`

  Create them initially with zeros in them. This will be explained later.
- We need to accumulate information as part of the simulation. Three vectors are needed for that, to hold

- Intensity of emitted light, `I`
- Electron position, `X`
- Electron velocity, `V`

Each turn, we record all electron positions and velocities in these arrays. If they had a collision, we also record that as emitted light.

We do not know the length of these arrays. So we create them as lists and extend them as required. This is slow, but more flexible. To create an list `a`, just use `a=[]`.

- Loop over the turns

```
for k=1:nk
  <<block>>
end
```

Note that "<<block>>" is asking LyX to collect all the codes defined within sections named <<block>>= ... @ and put them within the for loop.

This is an extremely convenient feature of literate programming. It means that you can put your code as it suits your report, but the code is collected properly as required by the compiler. You now have to define the code that will go into the for loop.

- Find the electrons present in the chamber.

  Remember we defined vectors of dimension nM. So not all the entries in the vectors represent electrons. How many of these entries should we use?

  The answer is that if an electron is in the chamber, its position must satisfy $0 < x < L$, where $L = n$ for this simulation. We simplify things by assuming that anytime the electron reaches $x = L$, it is reset to $x = 0$.

  So, if an entry has zero x position, that electron has not yet been injected. Only $x > 0$ entries correspond to electrons within the chamber.

  We do this by finding all those electrons whose position is greater than zero. Use the `where` command to do this:

```
ii=where(xx>0);
```

`ii` is a vector containing the indices of vector `xx` that have positive entries.

- Compute the displacement during this turn, assuming that the Electric field creates an acceleration of 1. Hence, the displacement of the i^{th} electron is given by

$$dx_i = u_i \Delta t + \frac{1}{2} a (\Delta t)^2 = u_i + 0.5$$

  Use vector notation to do this instead of a for loop.

  Note: Since you only want to move electrons that exist, you must act only on those electrons you found in the previous part. Python makes this easy. You just have to code:

```
dx[ii]=u[ii]+0.5
```

  which says "add 0.5 to each velocity and store it in displacement for those indices belonging to vector `ii`". You need to use this trick throughout this assignment.

- Advance the electron position and velocity for the turn. The equations are

$$\begin{aligned} x_i &\leftarrow x_i + dx_i \\ u_i &\leftarrow u_i + 1 \end{aligned}$$

- Determine which particles have hit the anode (their positions would be beyond n). Use the `where` command for this. Set the positions, displacements and velocities of these particles to zero. No for loops!

- Find those electrons whose velocity is greater than or equal to the threshold.

- Of these, which electrons are ionized? This is a little tricky. So I am giving the code. Assume that `kk` is the vector of indices corresponding to energetic electrons. Create a uniformly distributed random vector *of length* `len(kk)` and find those indices that are less than `p`, the probability of a collision.

```
ll=where(rand(len(kk[0]))<=p);
kl=kk[ll];
```

  The first line does just what I said above - it creates the random vector and uses `where` to locate those entries that are less than or equal to `p`. I use `len(kk[0])` since `kk` is a list and I want the first array in the list. Now `ll` is a vector of indices that tells us the indices in vector `kk`. Not much use by itself. We want the electron indices. That is what the second line does. `kl` now contains the indices of those energetic electrons that will suffer a collision.

- Reset the velocities of these electrons to zero (they suffered an inelastic collision). The collision could have occurred at any point between the previous x_i and the current x_i. So roll another die and determine the actual point of collision and update the `xx` array suitably:

$$x_i \leftarrow x_i - dx_i\rho$$

  Here ρ is a random number between 0 and 1. Do this using vector operations.

  Note: Actually the positions should not be distributed uniformly. The collision could have occurred at any *time* in $(k-1)\Delta t < t < k\Delta t$. But the electrons are accelerating, and uniformly distributed in time does not mean uniformly distributed in space. In addition, the electron will move and accelerate after the collision. Can you work out a better code than the above?

- We also need to do another thing. The excited atoms at this location resulted in emission from that point. So we have to add a photon at that point. To add these photons to the `I` vector, use the following code

```
I.extend(xx[kl].tolist())
```

  We are extending the list after converting the array $xx[kl]$ to a list. This is a slow process, since Python may need to allocate a new, larger vector and copy over the old one. So it should be done only when we cannot know the size of the vector ahead of time.

- Now inject `M` new electrons. First determine the actual number of electrons injected as

```
m=randn()*Msig+M
```

  This calculates the number injected this turn by rolling a "normally distributed random number", multiplying it by the standard deviation and adding the mean value (both are specified on the command line)

  But where do these injected electrons go in the `xx` array? We have to find the unused indices in the electron vector and add the new electrons there. But, what if the array is full? This should not happen, but it theoretically could. So add either add the electrons or the number of slots available, whichever is smaller. `Note:` For these electrons, initialize the position to

$$x_i = 1$$

  So the tubelight stretches from 1 to n. A position of 0 means an unused slot. Search for that using `find` to find the unused slots.

- Again find all the existing electrons. This time, add their positions and velocities to the `X` and `V` vectors. We are finding these indices twice in the loop. Can you optimize by finding only once? i.e., can you use the same index vector at the beginning of the next loop?

- The loop is done. At the end of the run, we will have `I`, the intensity vector, `X` the position vector and `V` the velocity vector. We first plot the "electron density", i.e., the number of electrons between i and $i+1$. We can do this by generating a population plot of `X`. To do this, look up `hist`. The plot should go into window 0. (look up `figure`)

- Now plot a population plot of the light intensity. Again, use `hist`. The plot should come in window 1.

- Plot the "electron phase space": for each electron plot a "X" corresponding to $x = x_i$, $y = v_i$.

- We also want to print out the Intensity as a table. For this we need the data. This data is returned by the `hist` function when it plots the histogram. What is returned is a "tuple" with three elements.

  – The first is an array of population counts

  – The second is the bin position.

  – The third (which we do not use) is a list of the rectangles that are used to build up the histogram.

  The second actually gives the dividing positions between bins, and so has a dimension one greater than the population array. Convert to mid point values by:

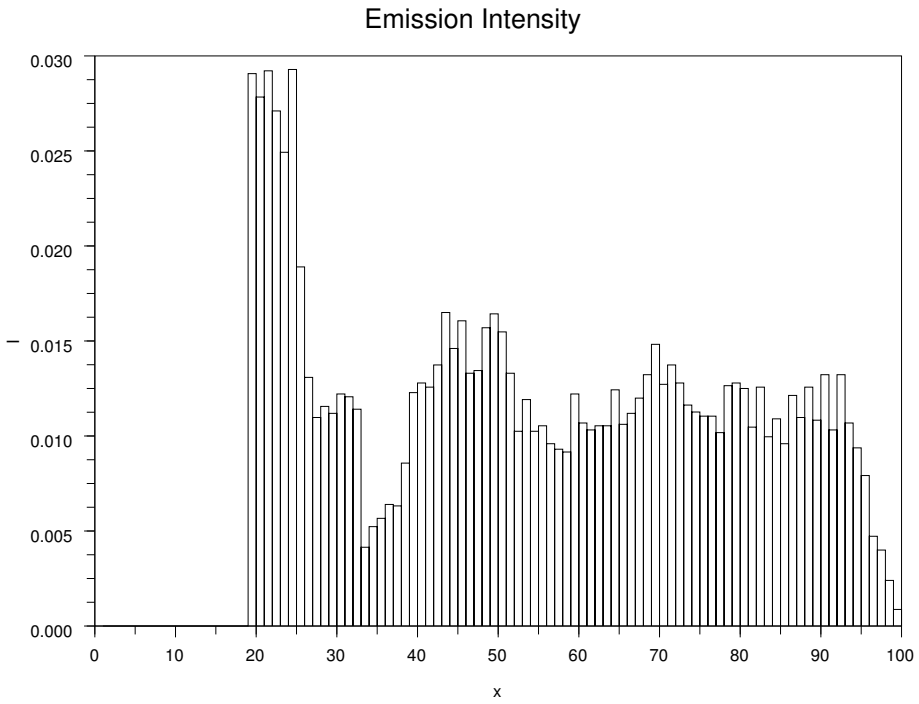
  ```
  xpos=0.5(bins[0:-1]+bins[1:])
  ```

  This averages the vector containing left positions of all the bins and the vector containing the right positions of all the bins. Now the dimensions will agree. We only need to print out the two arrays in the following format:

  ```
  Intensity data:
   xpos       count
  binval1   population1
  binval2   population2
  ...
  binvalN   populationN
  ```

You can vary the parameters and see the conditions under which dark spaces appear (these are regions which emit less light). You can also see the effect of changing the gas (varying the threshold).

For example, using $u_0 = 7$ and $p = 0.5$, we obtain the following emission graph



The region upto 20 is where electrons are building up their energy. Beyond that is a region where the emission decays, representing the fewer energetic electrons that reached there before colliding. At 40 is the next peak. But this is a diffuse peak since the zero energy location of different electrons is different.

A fairly complex simulation, yet it was done in just a few lines of Python..