Universidad Nacional de Colombia - sede Manizales
Facultad de Ciencias Exactas y Naturales
Curso: Informática 3.

Repositorios

Michael López Parra

Presentado a:

Cristian Elías Pachón Pacheco

024 de marzo del 2023

---

1. **Objetivos**
   - ❖ Crear y simular diferentes sistemas en el entorno python usando estructuras de datos y matplotlit para graficar.

2. **Sistema de Electrones.**

En esta simulación se busca crear una celda de electrones, en la cual se incluirán dentro de ella electrones de manera aleatoria, siguiente a esto usaremos el paso Montecarlo para cambiar de posición uno de esos electrones aleatorios y calcular la energía total del sistema que se verá reflejada en una gráfica de energía vs temperatura.

Las líneas de código se explican de la siguiente manera:

Primero se importa numpy que será la creadora de estructura de datos y arreglos y matplotlib con lo que se graficará, luego se definen las constantes a usar durante todo el código:

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   k=1
5   r = 1
6   T = 0.01
7   q = 1
8   elec_Circunferenciales = 3
9   elec_Internos = 2
10
```

Siendo k la constante de Coulomb, r nos ayudará a crear el sistema inicial, T será la temperatura, q la carga de los electrones, los electrones circunferenciales serán la cantidad de electrones de la celda y los electrones internos los que estarán contenidos en ella.

Ahora se definirán las distintas funciones que se usaron en el código:

```
11  def crear_Estado_Inicial():
12      angulo = 6.28/(elec_Circunferenciales)
13      x_out = [r*np.cos(ang) for ang in np.arange(0, 6.28, angulo)]
14      y_out = [r*np.sin(ang) for ang in np.arange(0, 6.28, angulo)]
15      x_in = [np.random.random() - 0.5 for i in range(elec_Internos)]
16      y_in = [np.random.random() - 0.5 for i in range(elec_Internos)]
17      return x_out,y_out,x_in,y_in
18
```

Con esta función se crea el estado inicial, es decir se crea las parejas ordenadas de cada electrón que componen la celda, distanciados cierto ángulo para mantener simetría, esto responde a las líneas 12, 13 y 14.

Las siguientes dos líneas crean los pares ordenados de los electrones aleatorios dentro de la celda.

```
19  def dibujar_sistema(x_out,y_out,x_in,y_in):
20      plt.figure()
21      plt.plot(x_out,y_out, "ro")
22      plt.plot(x_in, y_in, "bo")
23      plt.gca().set_aspect("equal")
24      plt.xlim(-1.1, 1.1)
25      plt.ylim(-1.1, 1.1)
26      plt.grid()
27      plt.show()
28
29  x_out,y_out, x_in,y_in = crear_Estado_Inicial()
30  dibujar_sistema(x_out,y_out,x_in,y_in)
31
```

Con esta función se hará visible en un plano cartesiano el estado inicial, con los electrones de las celdas de color rojo y los electrones internos de color azul.

```
32  r=[]
33  for i in range(len(x_out)):
34      r.append((x_out[i], y_out[i]))
35  for i in range(len(x_in)):
36      r.append((x_in[i], y_in[i]))
37
38  print( "======", r)
```

En lo anterior se creó un vector r para almacenar todas las parejas ordenadas, tanto de los electrones de las celdas como los internos, esto para manejar más fácilmente sus ubicaciones y cada electrón por separado.

```
39  def distancia(r1, r2):
40      return ((r1[0]-r2[0]) ** 2 + (r1[1]-r2[1])**2) ** 0.5
41
42  from itertools import combinations
43  def calcular_energia_total():
44      sumEnergias = 0
45      combinaciones = list(combinations(r, 2))
46      for r_ in combinaciones:
47          r1, r2 = r_[0], r_[1]
48          #print(r1,"vs", r2, "distancia =>", distancia(r1,r2))
49          sumEnergias += k*q*q / distancia(r1,r2)
50      #print("final",sumEnergias)
51      return sumEnergias
52
```

La funcion de distancia contiene la fórmula para la distancia entre dos puntos y la función de calcular energía total almacena la energía de todas las combinaciones posibles entre los electrones del sistema.

```
57  def metropolis(ran_int_position):  #cambio aleatorio de la posicion de un electron interno
58      r.pop(ran_int_position)
59      x_change = np.random.random() - 0.5
60      y_change = np.random.random() - 0.5
61      r.append((x_change,y_change))
62      x_in.pop(int(ran_int_position-elec_Circunferenciales))
63      y_in.pop(int(ran_int_position-elec_Circunferenciales))
64      x_in.append(x_change)
65      y_in.append(y_change)
66      calcular_energia_total()
67
```

En la función metrópolis se toma un electrón y se elimina del vector que contiene todas las parejas ordenadas para luego eliminar de la lista de las posiciones X e Y las componentes de ese electrón y agregar unos nuevos valores para ellos creando un nuevo electrón y calcular la energía de ese sistema.

```
69  def paso_montecarlo(T):
70      for i in range(elec_Internos):
71          ran_int_position=np.random.randint(0,len(r))
72          while ran_int_position < elec_Circunferenciales:
73              ran_int_position=np.random.randint(0,len(r))
74          else: pass
75          metropolis(ran_int_position)
76      #dibujar_sistema(x_out,y_out,x_in,y_in)
77
```

Para la función paso Montecarlo se implementó un ciclo para que cambiara de posición los electrones tantas veces como electrones internos halla, el segundo ciclo se implementa para que se escoja una posición del vector de electrones correspondientes a los internos, y para cada cambio se implementa metrópolis.

La simulación consiste en ir cambiando la temperatura del sistema y ver cómo va cambiando su energía. Para eso definimos cuantos paso Montecarlo serán necesarios, la mayor temperatura que alcanzará el sistema y la menor temperatura, y a que razón de cambio se irá reduciendo.

```
78  amount_mcs = 1000
79  T_high=5
80  T_low=0.01
81  step=-0.1
82
```

Se prosigue a crear un arreglo para las temperaturas y para las energías y así manejar los datos de manera más concisa.

```
90  temps = np.arange(T_high, T_low, step)
91  energies = np.zeros(shape=(len(temps), amount_mcs))
```

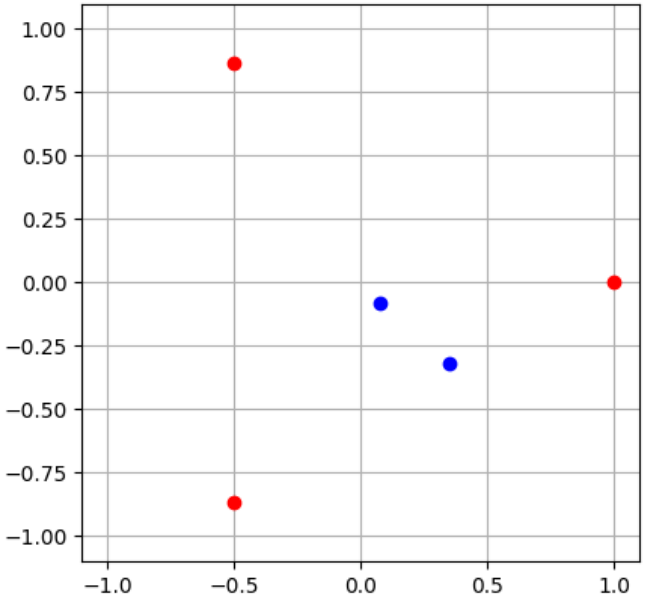
Para crear la simulación se recorrerá todo el arreglo de las temperaturas para aplicar el paso Montecarlo a cada una de las diferentes temperaturas y aplicar todos los pasos Montecarlo antes definidos, además de eso se llenará el arreglo de energías con componente en X la temperatura y en Y la energía.

```
95  for ind_T, T in enumerate(temps):
96      for i in range(amount_mcs):
97          paso_montecarlo(T)
98          energies[ind_T, i] = calcular_energia_total()
```

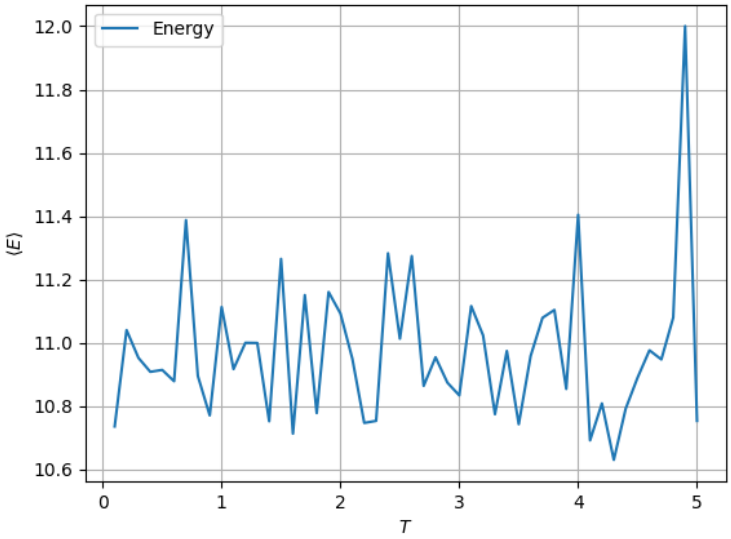
Finalmente buscamos graficar el comportamiento de lo antes mencionado para lo que se crea un nuevo arreglo con las energías distribuidas uniformemente:

```
101  tau = amount_mcs // 2
102  energy_mean = np.mean(energies[:, tau:], axis=1)
103
104  plt.figure()
105  plt.plot(temps, energy_mean, label="Energy")
106  plt.legend()
107  plt.xlabel(r"$T$")
108  plt.ylabel(r"$\left<E\right>$")
109  plt.grid()
110  plt.show()
```

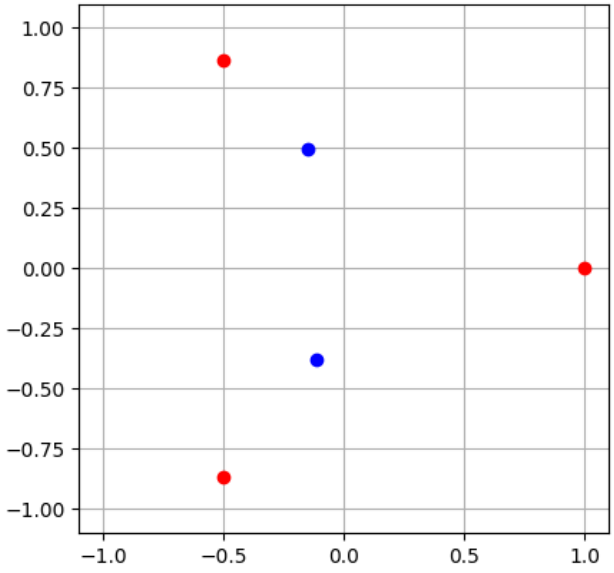
Al aplicar estas líneas de código se obtiene el estado inicial con 3 electrones de celda y dos electrones internos:



Se obtiene la gráfica de energía contra temperatura dando un comportamiento aleatorio e imposible de descifrar, con lo cual se necesitarán nuevos factores para controlar mejor la simulación.



Y el sistema final luego de los cambios realizados:



Analizando la gráfica y el estado final se puede observar el gran pico de energía alcanzado al final de la simulación, se infiere que este se debe a la cercanía que hay entre los dos electrones superiores y la gran pérdida de energía por la separación de unos de los electrones de celda.

3. **Simulación de celdas solares.**

   En esta simulación se busca diseñar el comportamiento de las celdas solares y cuanta corriente pueden generar gracias a los rayos solares y las propiedades de sus materiales.
   Este código se compone de varios módulos para así facilitar su comprensión, en los cuales se encuentran las constantes físicas, el espectro solar, las propiedades de los materiales tanto como los IP y los IN, las funciones de apoyo y la simulación.
   Cabe recalcar que el módulo de simulación importa los módulos de las propiedades IP y IN además del módulo de fórmulas de apoyo, que este a su vez importa los módulos de espectro solar y el módulo de constantes físicas, además del módulo de espectro solar que importa una base de datos que conforma todo el espectro solar.
   Aclarado lo anterior se explicará la función simulación, la cual recibe un voltaje determinado y aplica todas las funciones de apoyo, con el fin de llegar a la corriente generada en esa celda.

```python
def simulation(voltage):        #this function return j_cell of the solar cell
    # Built-in potential ------------------------------------------------------------
    V_bi = Vbi(Eg_n, Eg_p, chi_n, chi_p, N_d, N_a, Nc_n, Nc_p, Nv_n, Nv_p, ni_n, ni_p)

    # Depletion region --------------------------------------------------------------
    x_n = region(N_a, N_d, eps_n, eps_p, N_d, N_a, V_bi, voltage )
    x_p = region(N_d, N_a, eps_n, eps_p, N_d, N_a, V_bi, voltage )

    if x_n > w_n:  # Condition in case the depletion region exceeds the width of the material
        x_n = w_n
        x_p = w_n * (N_d / N_a)

    # Photocurrent ------------------------------------------------------------------
    dj_p    =  dJp(s_p, L_p, D_p, w_n, x_n, alpha_1, Ref, Trans)
    dj_n    =  dJn(s_n, L_n, D_n, w_p, x_p, alpha_2, w_n, alpha_1, Ref, Trans)
    dj_scr  =  dJscr(x_n, x_p, w_n, alpha_1, alpha_2 , Ref, Trans)
    j_ph    =  Jph(dj_n, dj_p, dj_scr)
    # Saturation current density J0 -------------------------------------------------
    j0_p    =  J0pn(D_p, p_0, L_p, s_p, w_n, x_n)
    j0_n    =  J0pn(D_n, n_0, L_n, s_n, w_p, x_p)
    j_0     =  J0(j0_n, j0_p)
    # Saturation current density J00 ------------------------------------------------
    j_00    =  J00(x_n, x_p, ni_n, ni_p, L_p, L_n, D_p, D_n)
    # Dark current density ----------------------------------------------------------
    j_dark  =  Jdark(j_0, j_00, voltage)
    # Cell current density ----------------------------------------------------------
    j_cell  =  j_ph - j_dark
    return j_cell

```

Para esta simulación se usarán 10 celdas solares para reducir el tiempo de la misma trabajándolas como una lista por conveniencia, se definirán las mismas constantes anteriores, cuantas veces se simulará. Voltaje mayor, voltaje menor y que tanto se varía el voltaje, además de los arreglos para cada voltaje y los arreglos vacíos para la corriente y la potencia.

```python
celdas=10
cells=[]
for u in range(celdas):
    cells.append(u)


amount_mcs = 1000
vol_high=0.5375
vol_low=0
step= -(vol_high-vol_low)/100

temps = np.arange(vol_high, vol_low, step)
current = np.zeros(shape=(len(temps), amount_mcs))
potential = np.zeros(shape=(len(temps), amount_mcs))

```

Se procede a llenar los arreglos para la corriente y para la potencia, simulando para cada celda solar variando el voltaje.

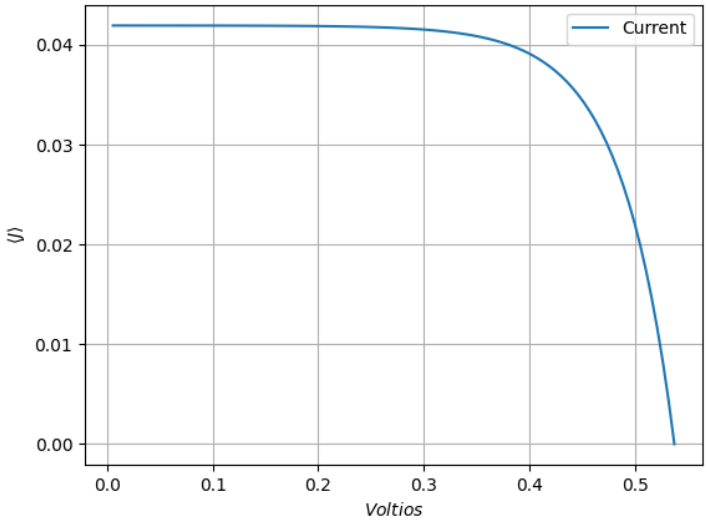
```python
for a in range(len(cells)):
    for ind_v, voltage in enumerate(temps):
        for i in range(amount_mcs):
            current[ind_v, i] = simulation(voltage)
            potential[ind_v, i] = simulation(voltage)*voltage

```

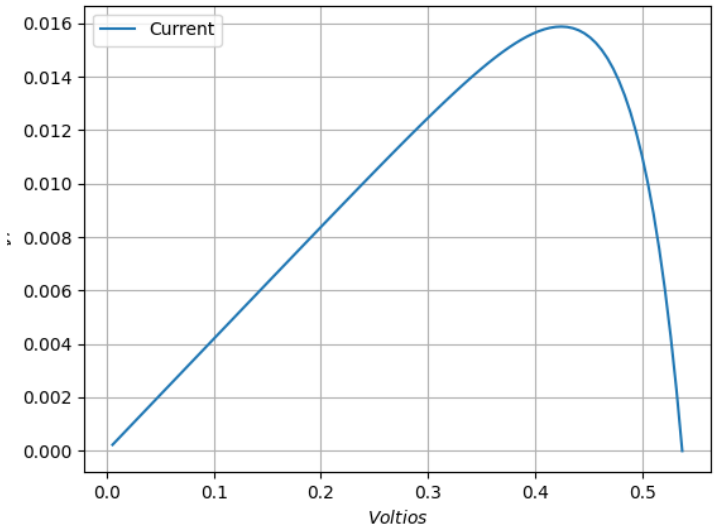
Finalmente, para graficar el comportamiento creamos dos arreglos tanto para la corriente como para la potencia y replicamos el código para las gráficas.

```
69  tau = amount_mcs // 2
70  current_mean = np.mean(current[:, tau:], axis=1)
71  potential_mean = np.mean(potential[:, tau:], axis=1)
72
73  plt.figure()
74  plt.plot(temps, current_mean, label="Current")
75  plt.legend()
76  plt.xlabel(r"$Voltios$")
77  plt.ylabel(r"$\left<J\right>$")
78  plt.grid()
79  plt.show()
80
81  plt.figure()
82  plt.plot(temps, potential_mean, label="Current")
83  plt.legend()
84  plt.xlabel(r"$Voltios$")
85  plt.ylabel(r"$\left<J\right>$")
86  plt.grid()
87  plt.show()
```

Al correr el programa obtenemos la gráfica de corriente contra voltaje:



Y la gráfica para potencia contra voltaje:



De la gráfica de potencia se puede observar un punto máximo que corresponde a la potencia de 0.0159W y al voltaje de 0.4251V haciendo uso de la ecuación de potencia hallamos la corriente en ese punto:

$$P_{mpp} = J_{mpp} * V_{mpp}$$

Dando una corriente de 0.03741

Además de la gráfica de corriente se puede determinar la corriente cuando el voltaje es cero y el voltaje cuando la corriente es cero siendo respectivamente 0.0424A y 0.5375V siendo estos J_{sc} y V_{oc}.

Estos valores los reemplazamos en la fórmula de factor de llenado:

$$FF = \frac{J_{mpp} * V_{mpp}}{J_{sc} * V_{oc}} = 0.6967$$

Y en la fórmula de eficiencia:

$$\eta = \frac{J_{sc} * V_{oc} * FF}{P_{mpp}} = 1$$

DE lo anterior se puede inferir que se simuló un caso ideal donde no existe la perdida de energía.