RELATORIA 2 – INFORMATICA III

Objetivos

- Familiarizarse con los métodos de simulación de diversos fenómenos físicos y dispositivos.

- Implementar técnicas numéricas para ejecutar simulaciones de carácter aleatorio a sistemas de muchos cuerpos.

- Interpretar los resultados arrojados en cada caso.

Simulaciones:

Sistema de electrones por Montecarlo

En esta simulación se buscó encontrar la configuración en las posiciones de los electrones tal que la energía potencial total del sistema fuera la mínima encontrada en cierto número de iteraciones Montecarlo, el sistema estaría constituido por dos grupos de electrones, los inmóviles que formarían un circulo dentro del cual se situarían los móviles.

Inicialmente se creó el sistema con un estado inicial, para esto se utilizó la librería matplotlib que nos permitió tener una perspectiva visual del sistema para comprender mejor lo que sucedería, con ayuda de dicha librería se creó un espacio en el cual se situaron los electrones como se muestra en la figura 1, para esto se creó una función llamada **crear_Estado_Inicial** que sitúa los electrones inmóviles con ayuda de funciones trigonométricas como seno y coseno, mientras que para los electrones móviles se permitió que el programa tome una posición al azar dentro de un rango de -0.5 y 0.5, para luego devolver listas con las posiciones en x e y de cada electrón.

```python
e_out = 100
e_in = 150

def crear_Estado_Inicial():
    angulo = 6.28/e_out
    x_out = [r*np.cos(ang) for ang in np.arange(0, 6.28, angulo)]
    y_out = [r*np.sin(ang) for ang in np.arange(0, 6.28, angulo)]
    x_in = [np.random.random() - 0.5 for i in range(e_in)]
    y_in = [np.random.random() - 0.5 for i in range(e_in)]
    return x_out,y_out,x_in,y_in
```

Figura 1. Función para posicionar los electrones en un estado inicial.

Posteriormente se graficaron cada uno de los puntos para tener una base visual del sistema, para esto se utilizó la función **dibujar_sistema** que se muestra en la figura 2, la cual recibe las listas de posiciones generadas en la función **crear_Estado_Inicial.**

```python
def dibujar_sistema(x_out,y_out,x_in,y_in):
    plt.figure()
    plt.plot(x_out,y_out, "ro")
    plt.plot(x_in, y_in, "bo")
    plt.gca().set_aspect("equal")
    plt.xlim(-1.5, 1.5)
    plt.ylim(-1.5, 1.5)
    plt.grid()
    plt.show()
```

Figura 2. Función para graficar el sistema.

Luego de esto se crearon dos listar que contienen a su vez listas de dos elementos con el punto correspondiente a cada electrón, estas listas de nombraron como **r_in** y **r_out** como se muestra en la figura 3, separar los dos grupos de electrones de esta manera permitirá más adelante manipularlos de manera que se cumplan las condiciones de la simulación.

```
r=[]
for i in range(len(x_out)):
    r.append([x_out[i], y_out[i]])

for i in range(len(x_in)):
    r.append([x_in[i], y_in[i]])

r_out = r[:e_out]
r_in = r[e_out:]
```

Figura 3. Creación si separación de las listas contenedoras de posiciones.

Para las energías se utilizó la lista **r** para poder calcular las distancias entre todas las combinaciones posibles de los electrones agrupándolos de a dos, posteriormente se recorrieron todas estas combinaciones para calcular la energía total del sistema en un estado especifico, tal como se muestra en la figura 4 con la función **calcular_energia_total.**

```
def distancia(r1, r2):
    return ((r1[0]-r2[0])**2 + (r1[1]-r2[1])**2)**0.5

def calcular_energia_total():
    sumEnergias = 0
    combinaciones = list(combinations(r, 2))
    for r_ in combinaciones:
        r1, r2 = r_[0], r_[1]
        sumEnergias += k*q*q/distancia(r1,r2)
    return sumEnergias
```

Figura 4. Función para calcular la energía total del sistema en un estado determinado.

Una vez se tienen todas las funciones necesarias para determinar un estado es necesario proceder a crear las funciones que permitan cambiarlo, como se muestra en la figura 5 para esto se creó la función **new_state** que elige un punto (o electrón) móvil aleatorio y le asigna una nueva posición aleatoria a cada una de sus componentes, pero esta vez limitada a un rango más amplio, de -1.5 a 1.5, esto

para evitar que los electrones se vayan a infinito en su búsqueda de la energía menor.

```python
def new_state(r_in):
    random_el = r_in[np.random.choice(range(len(r_in)))]
    random_el[0] = np.random.random()*1.5 - 0.75
    random_el[1] = np.random.random()*1.5 - 0.75
    return r_in
```

Figura 5. Función para cambiar el estado del sistema.

Ya con todas las funciones necesarias para ejecutar la simulación de manera correcta se implementó el algoritmo **metrópolis** tal como se muestra en la figura 6, en el cual se calcula la energía total para un estado inicial, posteriormente se introduce un cambio al sistema y se vuelve a calcular la energía total, si esta energía nueva es menor que la anterior se acepta el cambio, si por el contrario es mayor se evalúa la probabilidad de que se acepte el cambio a partir de comparar un valor aleatorio entre 0 y 1 con el factor de Boltzmann ya que se modela el sistema como un conjunto canónico, dados que esta condición no se cumpla el estado volverá a ser el primero establecido.

```python
def metropolis(nuevoEstado, T):
    estado = nuevoEstado
    old_E = calcular_energia_total()
    estado = new_state(r_in)
    new_E = calcular_energia_total()
    deltaE = new_E - old_E
    if deltaE <= 0:
        pass
    else:
        if np.random.uniform(0, 1) <= np.exp(-deltaE/(k*T)):
            pass
        else:
            estado = nuevoEstado
```

Figura 6. Implementación del algoritmo metrópolis a la simulación.

El método Montecarlo se implementó como se muestra en la figura 7, con una función que retorna las nuevas posiciones como listas para poder graficar el siguiente estado, en esta función se ejecutarán las iteraciones como ejecuciones del algoritmo metrópolis.

```
def monte_carlo_step(T):
    for i in range(len(r_in)):
        nuevo = new_state(r_in)
        newx_in = []
        newy_in = []
        for punto in r_in:
            newx_in.append(punto[0])
            newy_in.append(punto[1])
        metropolis(nuevo, T)
    return newx_in, newy_in
```

Figura 7. Implementación de la técnica Montecarlo.

Por último, se simuló el sistema ejecutando Montecarlo determinado número de veces y guardando las energías asociadas a cada iteración en una lista llamada **energías**, luego se graficó el sistema para la última iteración que corresponde a la menor energía hallada en la cantidad de veces que se ejecutó el programa tal y como se muestra en la figura 8.

```
amount_mcs = 1000
energies = np.zeros(shape = amount_mcs)
for i in range(amount_mcs):
    newx_in, newy_in = monte_carlo_step(T)
    energies[i] = calcular_energia_total()

dibujar_sistema(x_out, y_out, newx_in, newy_in)
```

Figura 8. Simulación.

Curvas características de una celda solar:

Para esta simulación se utilizó una guía en la cual se especificaron las ecuaciones y los parámetros a calcular, en total se utilizaron seis módulos los cuales se exponen a continuación y un archivo csv.

Constantes físicas:

Este archivo contiene los valores numéricos y las unidades de las constantes físicas involucradas en las ecuaciones tales como.

- Constante de Boltzmann
- Carga del electrón
- Temperatura ambiente
- Permitividad eléctrica del vacío
- Velocidad de la luz
- Constante de Planck
- Masa del electrón

Espectro solar:

En este archivo se encuentra el código encargado de leer el archivo “data.csv” el cual contiene los datos de las longitudes de onda e intensidades presentes en el espectro de radiación solar. Además, en este archivo se separan solo la parte del espectro de interés para obtener el flujo fotónico como se muestra en la figura 9.

```
lamb  = np.array(spectrum.loc[(spectrum['Wvlgth nm'] <= 1107.0), 'Wvlgth nm'])  # nm
I_AM15  = np.array(spectrum.loc[(spectrum['Wvlgth nm'] <= 1107.0), 'Global tilt  W*m-2*nm-1'])
photon_flux = I_AM15 * lamb / (q * 1240) / 10000   # 1 / (cm² * nm)   #esta es la variable de interes
```

Figura 9. Sección del espectro solar de interés y flujo fotonico.

A partir de los datos almacenados en la variable **I_AM15** se puede obtener la potencia lumínica incidente integrando, como se muestra en la figura 10.

```
I_AM15_  = np.array(spectrum['Global tilt  W*m-2*nm-1'])
P_inc = (integrate.simps(I_AM15_) / 10000) * 1000   # mW / cm²
```

Figura 10. Calculo de la potencia incidente.

Funciones:

En este archivo se almacenaron las funciones encargadas de calcular cada una de las ecuaciones necesarias para llevar a cabo la simulación, estas ecuaciones se dividen en las que están asociadas al material tipo N y al tipo P ya que se está hablando de una unión entre estos dos tipos de materiales las ecuaciones en cuestión describen las siguientes características de estos materiales.

- Potencial integrado
- Región de deflexión de los materiales tipo N y tipo P
- Densidades de corriente de los materiales tipo N y tipo P
- Corriente debido a portadores generados en la zona de carga
- Corrientes de saturación
- Densidades de corrientes por difusión bipolar
- Densidad de corriente total

Propiedades de los materiales tipo N y tipo P

En este archivo están contenidas todas las constantes necesarias para calcular las ecuaciones mencionadas en el apartado anterior, estas a diferencia de las físicas que son generales están más asociadas a cada material, en este caso existen en forma escalar y también arreglos debido a la cantidad de valores necesarios o a su dependencia con otro parámetro.

Simulación:

En este archivo se encuentra una función encargada de ejecutar las funciones alojadas en el módulo **Funciones** la cual retorna el valor de la densidad de corriente total, a partir de este valor y proporcionando diferentes valores para el voltaje se pudo obtener la gráfica de J – V para esta celda utilizando la librería **matplotlib** tal como se muestra en la figura 11, en la cual se crearon dos arreglos numpy, uno para el voltaje (rango de números con np.arange()), y otro para la densidad de corriente (matriz unidimensional de ceros con una longitud de 100).

```
voltage = np.arange(0, 1, 0.01)
jcell = np.zeros(shape = 100)
```

Figura 11. Arreglos que contienen los datos a graficar.

Posteriormente se asignó un valor de densidad de corriente a cada uno de los valores de voltaje generados con el arreglo tal y como se muestra en la figura 12.

```
for valor in range(len(voltage)):
    jcell[valor] = simulation(voltage[valor])
```

Figura 12. Creación del arreglo contenedor de los datos para la densidad de corriente.

Con ayuda de **matplotlib** se graficaron estos dos conjuntos de datos para obtener la gráfica de la figura 13, de la cual se extraerán parte los parámetros característicos.

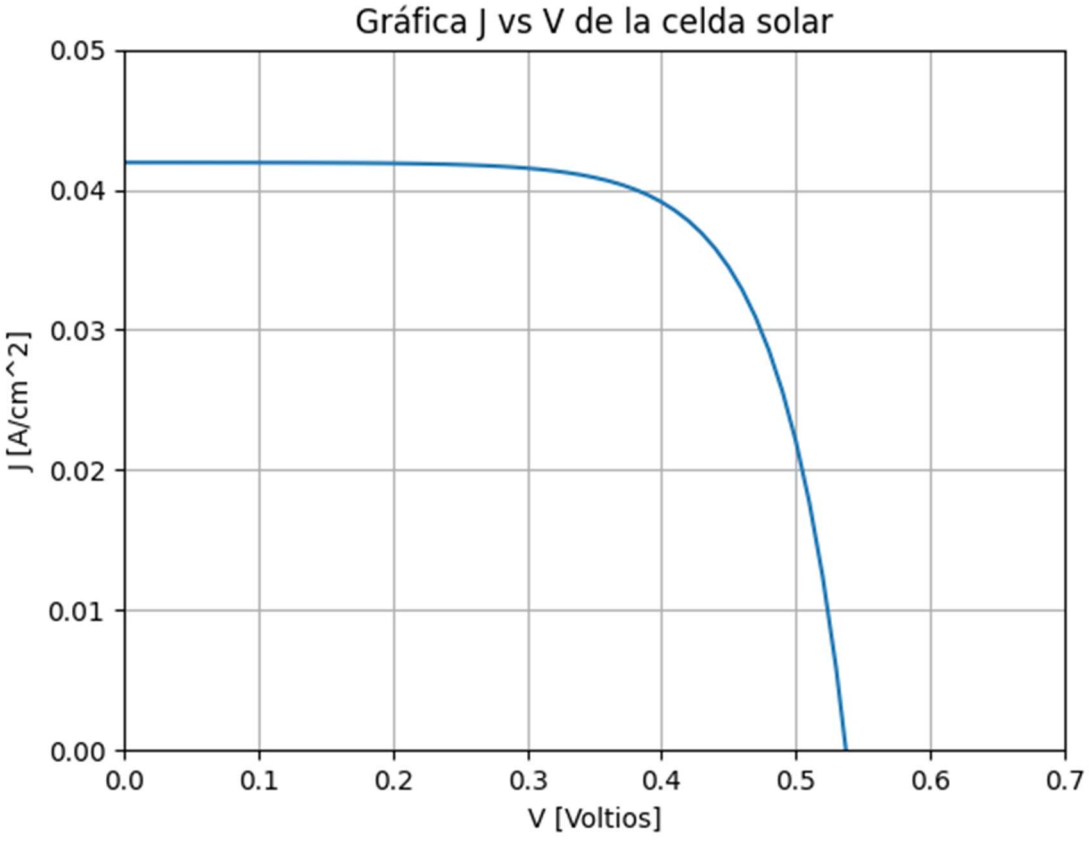

Figura 13. Grafica J – V simulada para la celda solar.

Para graficar la potencia en función del voltaje se multiplican las componentes de los dos arreglos generados en la figura 11 y se obtiene la figura 14.

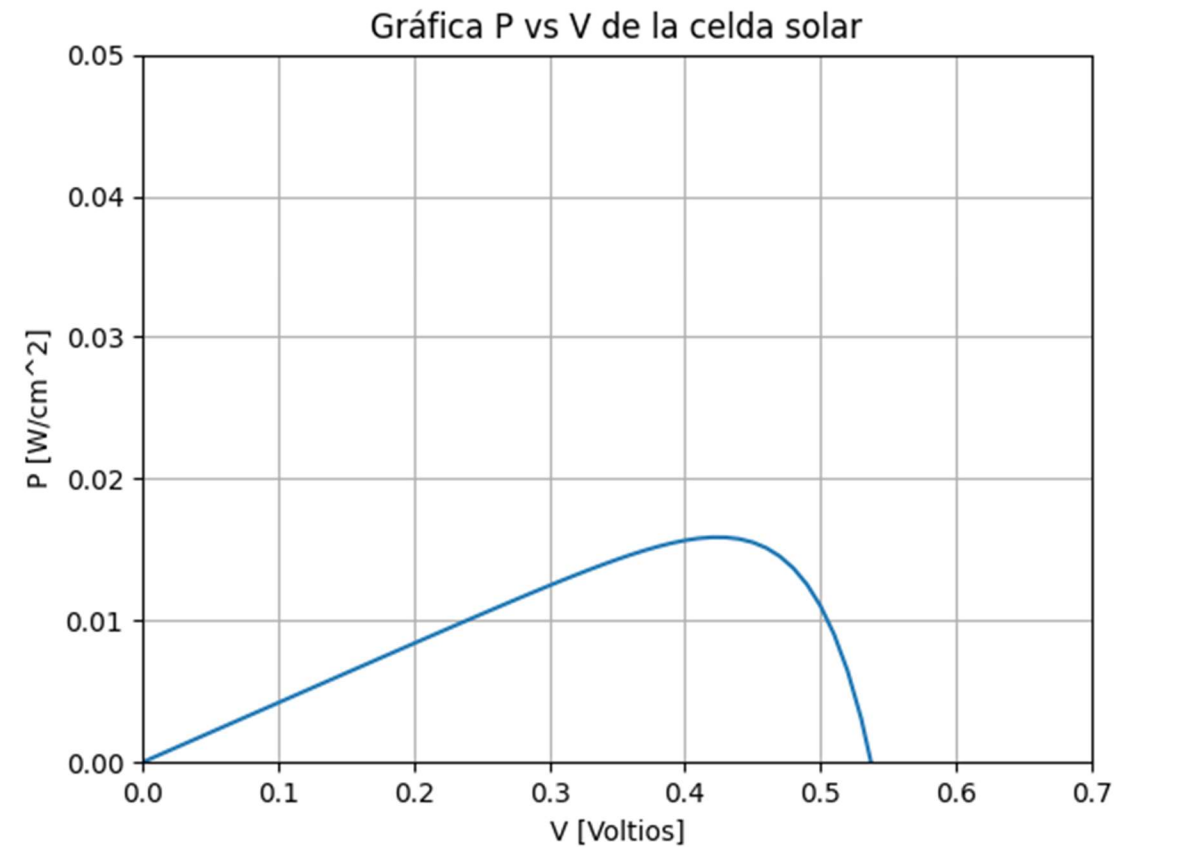

Figura 14. Gráfica P – V simulada para la celda solar.

A partir de las gráficas de las figuras 13 y 14 es posible caracterizar la celda calculando la potencia máxima (P_{MPP}), el factor de llenado (FF) y la eficiencia (n), los cuales se muestran a continuación.

Potencia máxima (P_{MPP}):

$$P_{MPP} = V_{MPP} * J_{MPP} \qquad (1)$$

Donde V_{MPP} es el voltaje asociado a la potencia máxima, y J_{MPP} es la densidad de corriente asociada a este voltaje, se calcularon como se muestra en la figura 15.

```
maxima_p = max(potencia)
v_mpp = voltage[list(potencia).index(maxima_p)]
j_mpp = jcell[list(voltage).index(v_mpp)]
p_mpp = j_mpp*v_mpp
```

Figura 15. Calculo de la potencia máxima.

En la figura 15 la v_mpp se encontró buscando en el arreglo de **potencia** el índice asociado a la potencia máxima, de manera similar se encontró j_mpp pero esta vez buscando dicho elemento en el arreglo de **voltage,** el valor obtenido para este parámetro es de aproximadamente $\mathbf{0.0159}\ \boldsymbol{W/cm^2}$.

Factor de llenado (FF):

$$FF = \frac{P_{MPP}}{J_{sc} * V_{oc}} \qquad (2)$$

Para hallar este valor es necesario encontrar el corte de la gráfica de J – V con el eje horizontal, para esto se utilizarán funciones de numpy que permiten encontrar un valor aproximado, estas son:

np.sign(): Esta función permite recorrer un arreglo pasado como argumento y devuelve un 1 si el elemento es positivo y un -1 si el elemento es negativo, en este caso devolverá un arreglo de 1 y -1 según sea el caso.

np.diff(): Esta función recorre un arreglo y calcula la diferencia entre el elemento i y el elemento i + 1, de manera que en este caso funciona para encontrar el elemento en el que los datos pasan de valores positivos a negativos ya que esta diferencia para el caso del arreglo generado con la función anterior es de -2-

np.where(): Devuelve un arreglo que contiene los elementos que cumplen cierta condición, en este caso recorrerá el arreglo de 0 (considerado como False) hasta encontrar el elemento distinto de 0 (considerado como True) y lo tomará.

Esta cadena de funciones se utilizó como se ve en la figura 16, donde el 0 entre corchetes al final de la línea permite solo elegir el primer elemento del arreglo arrojado por la función el cual contiene también su tipo como segundo elemento, en cuanto a J_{sc} se encontró simplemente extrayendo el elemento asociado a un voltaje 0 en el arreglo de la densidad de corriente total.

```
corte = np.where(np.diff(np.sign(jcell)))[0]
v_0c = voltage[corte][0]
j_sc = jcell[0]
ff = p_mpp/j_sc*v_0c
```

Figura 16. Calculo del factor de llenado.

Ejecutando el código anterior de obtiene un valor para el factor de llenado de aproximadamente **0.2** siendo este un valor adimensional.

Eficiencia:

$$n = \frac{P_{MPP}}{P_{inc}} \qquad (3)$$

En este caso este cálculo se obtiene directo al dividir el valor de la potencia máxima encontrada anteriormente con el valor de la potencia lumínica incidente calculada en el módulo **EspectroSolar**, este parámetro tiene un valor de aproximadamente **0.015 %**, la cual es bastante baja para un dispositivo de este tipo.

Luego de calculados estos parámetros se obtuvo la gráfica completa.

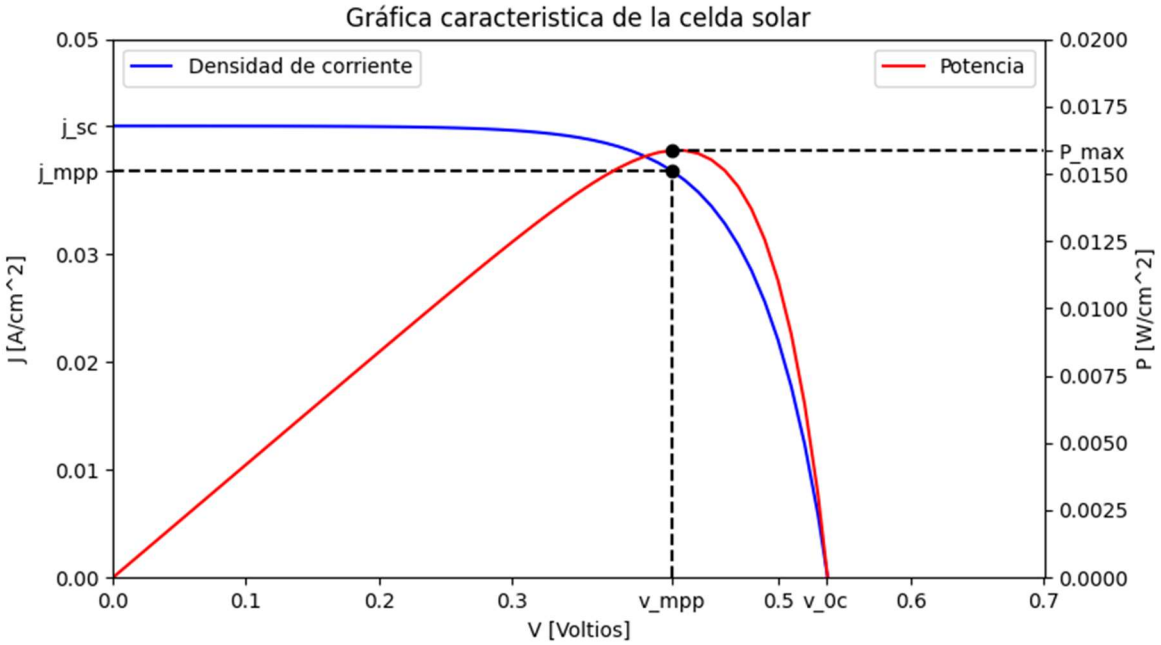

Figura 17. Gráfica característica de la celda solar.