Vincent Andrews

Computational Physics: Final Project

Import Libraries:

In [96]:
```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
import math
from tabulate import tabulate
from scipy.integrate import odeint
from scipy.integrate import solve_ivp
import scipy
import sympy as sp
from math import cos,sin,exp,sqrt, pi
from random import random, seed
from scipy.constants import Boltzmann
from scipy.interpolate import interp1d
from scipy.fft import fft, dct
from scipy.fftpack import fft, ifft
from scipy.signal import savgol_filter
```

Problem 1: Determine the electric potential through a wire by doing integration things.

In [43]:
```
epsilon = 8.854187e-12  # F/m
charge_density = 1e-6    # C/m
observation_point = np.array([0, 3])  #observation point (xobs, yobs)

#parametrization of the wire
def x(s):
    return s

def y(s):
    return s**4

#function to calculate the distance between two points
def distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

#midpoint rule integration
def midpoint_rule_integration(func, a, b, num_points):
    h = (b - a) / num_points
    integral = 0

    for k in range(1, num_points + 1):
        sk = a + (k - 0.5) * h
        integral += h * (func(sk) / distance(observation_point, np.array([x(sk), y(sk)

    return integral

#electric potential using midpoint rule
num_points = 1000  # Adjust the number of points as needed
electric_potential = (1 / (4 * np.pi * epsilon)) * midpoint_rule_integration(lambda s:
print("Electric potential = ", electric_potential, "Volts")
```

```
Electric potential =  6312.395581155265 Volts
```

Problem 2: Bunch of snakey wirey current bois, need to find the currents through each wire using Ohm's law.

Part a: find current through all five of the wires

In [4]:
```
# B = left side of equations derived using Kirchoff's law
B = np.array([[9.5, -2.5, 0, -2, 0],
              [-2.5, 11, -3.5, 0, -5],
              [0, -3.5, 15.5, 0, -4],
              [-2, 0, 0, 7, -3],
              [0, -5, -4, -3, 12]])

# right side of equation is column vector
vector = np.array([12, -16, 14, 10, -30])

# solve system of equations using my bestie, np
X = np.linalg.solve(B, vector)

# five equations, five unknowns
i1, i2, i3, i4, i5 = X

print("Current values through each wire in Amps (A)")
print(f"i1 = {i1} A")
print(f"i2 = {i2} A")
print(f"i3 = {i3} A")
print(f"i4 = {i4} A")
print(f"i5 = {i5} A")
```

```
Current values through each wire in Amps (A)
i1 = 0.11624444005378028 A
i2 = -3.927550828630771 A
i3 = -1.1880526815445926 A
i4 = -0.5384003739560795 A
i5 = -4.667097165933372 A
```

part b: find current through 5 ohm resistor in the diagram

In [6]:
```
i_5ohm = i2 - i5
print(f"Current through 5ohm: {i_5ohm} A")
```

```
Current through 5ohm: 0.7395463373026008 A
```

Problem 3: Perform all of those methods on different intervals and tabulate results

In [10]:
```
def func(x):
    return math.exp(x) -2 -0.01/x**2 +(2e-6)/x**3

def dfunc(x):
    return math.exp(x) + 0.01/x**2 - 0.000006/x**4

#part a: bisection method
def bisection_method(f, a, b, tol=1e-6, max_iter=100):
    iter_count = 0
    while (b - a) / 2 > tol and iter_count < max_iter:
        c = (a + b) / 2
        if f(c) == 0:
```

```
            break
        elif f(c) * f(a) < 0:
            b = c
        else:
            a = c
        iter_count += 1

    root = (a + b) / 2
    return root, iter_count

#part b: newton raphson method
def newton_raphson(f, df, a, b, tol=1e-6):
    x = (a+b)/2 #midpoint
    diff = np.inf
    iter_count = 0
    while diff > tol:
        f_val = f(x)
        df_val = df(x)
        x_new = x - f_val / df_val
        diff = abs(x_new - x)
        iter_count += 1
        x = x_new
    return x, iter_count

#part c: ridders method
def ridders(f,a,b,max_itrs=200,tol=1e-6):
    # Ensure that x0 < x1
    if a < b:
        x0 = a
        x1 = b
    else:
        x0 = b
        x1 = a

    # Compute the function at the two endpoints and make sure they bracket
    # a root
    f0 = f(x0)
    f1 = f(x1)
    if f0 == 0.0:
        return x0
    elif f1 == 0.0:
        return x1
    if f0*f1 > 0.0:
        raise Exception("Starting points for Ridders' rule must bracket root")

    # Main Ridders' method loop
    for i in range(max_itrs):
        # Find the midpoint of the interval
        x2 = (x0+x1)/2.0
        f2 = f(x2)

        # Apply Ridders' formula to get x3
        x3 = x2 + (x1-x2)*(f2/f0)/np.sqrt((f2/f0)**2-(f1/f0))
        f3 = f(x3)
        # print(i,x3)

        # Check to see if we can return: we took a very small step or
        # our latest guess evaluates to 0
        if abs(x3-x2) < tol or f3 == 0.0:
            return (i - 1), x3
```

```
        # Bracket the root as tightly as possible
        if f3*f2 < 0.0:
            # The root lies between x2 and x3, so set that as the interval for
            # the next iteration
            x0 = x2; f0 = f2
            x1 = x3; f1 = f3
        else:
            # f3 and f2 have same sign, so x3 and x2 don't bracket the root.
            # Check whether f0 or f1 is the other endpoint
            if f3*f1 < 0.0:
                # x3 and x1 bracket the root, so replace x0 with x3
                x0 = x3; f0 = f3
            else:
                # x3 and x0 bracket the root, so replace x1 with x3
                x1 = x3; f1 = f3

        #print("Exceeded iteration limit without solution")
    return None

#define the four intervals
A = 0.01
interval_1 = [A,3.0]
interval_2 = [A,9.0]
interval_3 = [A,27.0]
interval_4 = [A,81.0]

#I know the way i did this is really ghetto but i am running on fumes rn <3
#going to do all the bisection bois first
bi_root_1, bi_itr_1 = bisection_method(func, A, 3.0)
bi_root_2, bi_itr_2 = bisection_method(func, A, 9.0)
bi_root_3, bi_itr_3 = bisection_method(func, A, 27.0)
bi_root_4, bi_itr_4 = bisection_method(func, A, 81.0)

#then i guess its newty time
newt_root_1, newt_itr_1 = newton_raphson(func, dfunc, A, 3.0)
newt_root_2, newt_itr_2 = newton_raphson(func, dfunc, A, 9.0)
newt_root_3, newt_itr_3 = newton_raphson(func, dfunc, A, 27.0)
newt_root_4, newt_itr_4 = newton_raphson(func, dfunc, A, 81.0)

#saving the rattiest, stinkiest, ugliest one for last
B = 0.010021 #need to change interval to bracket the root
rid_root_1, rid_itr_1 = ridders(func, B, 3.0)
rid_root_2, rid_itr_2 = ridders(func, B, 9.0)
rid_root_3, rid_itr_3 = ridders(func, B, 27.0)
rid_root_4, rid_itr_4 = ridders(func, B, 81.0)


# create table: im sure i did this really stupidly but hey it prints
data = [["1", bi_root_1, bi_itr_1, newt_root_1, newt_itr_1, rid_itr_1, rid_root_1],
        ["2", bi_root_2, bi_itr_2, newt_root_2, newt_itr_2, rid_itr_1, rid_root_1],
        ["3", bi_root_3, bi_itr_3, newt_root_3, newt_itr_3, rid_itr_1, rid_root_1],
        ["4", bi_root_4, bi_itr_4, newt_root_4, newt_itr_4, rid_itr_1, rid_root_1]]
# columns of table
cols = ["Interval", "Bi root", "Bi itr", "Newton's root",
             "Newton's itr", "Ridder's root", "Ridder's iterations"]
print("                                  Results From Root-Finding Methods \n")
print(tabulate(data, headers=cols))
```

```
                                Results From Root-Finding Methods

  Interval    Bi root    Bi itr    Newton's root    Newton's itr    Ridder's root
Ridder's iterations
----------  ---------  --------  ---------------  --------------  ---------------  --
------------------
         1   0.703204        21         0.703205               6         0.703205
6
         2   0.703205        23         0.703205               9         0.703205
6
         3   0.703206        24         0.703205              18         0.703205
6
         4   0.703205        26         0.703205              45         0.703205
6
```

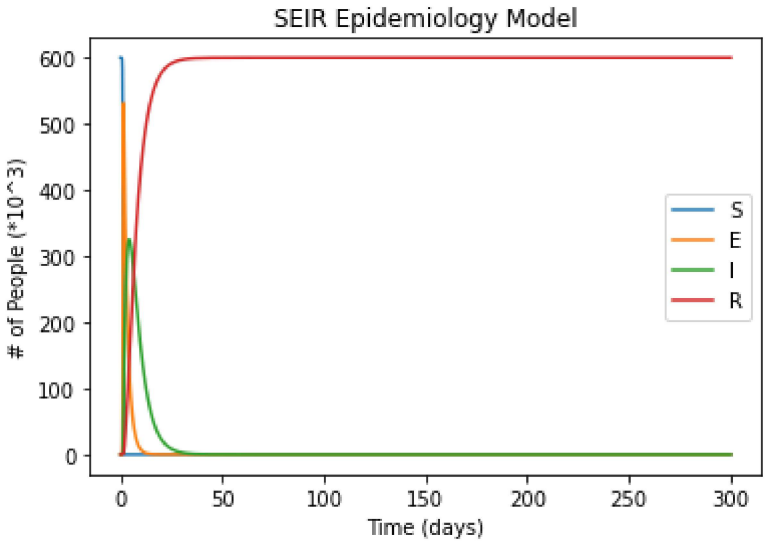
Problem 4: SEIR epidemiology model

Part a: plotting populations change over time using SEIR model

In [114…

```
def seir_f(t, y, beta, sigma, gamma):
    s, e, i, r = y
    return np.array([-beta * i * s,
                     -sigma * e + beta * i * s,
                     -gamma * i + sigma * e,
                     gamma * i])

# try some parameter values
beta = 0.514
sigma = 0.50
gamma = 0.20
t = list(range(1,301))

sol = solve_ivp(seir_f, [0, 300], [599.99, 2e-4, 0, 0],
                args=(beta, sigma, gamma))
plt.figure(figsize=(10, 6))
fig = plt.figure(); ax = fig.gca()
curves = ax.plot(sol.t, sol.y.T)
plt.ylabel("# of People (*10^3)")
plt.xlabel("Time (days)")
plt.title("SEIR Epidemiology Model")
ax.legend(curves, ['S', 'E', 'I', 'R']);
```

```
<Figure size 720x432 with 0 Axes>
```



Part b: get max infection number and day that it occurs

In [20]:
```
infected = list(sol.y.T[:,2])
print("The max number of infected people is:", max(infected))
index = infected.index(max(infected))
print("This occurs at:", index, "days")
print("It looks like its around:", 15, "days on the plot" )
# 133 days makes more sense maybe? but the plot says 15 - the plot is probably wrong
```

```
The max number of infected people is: 325.5038907198857
This occurs at: 132 days
It looks like its around: 15 days on the plot
```

Part c: Derive the growth rate k from the information provided

The two infected populations are given by:

$$I_1 = I_0 e^{kt_1} and I_2 = I_0 e^{kt_2}$$

Setting these equal to each other and dividing $\frac{I_1}{I_2}$, the I_0 cancels out and we have:

$$\frac{I_1}{I_2} = \frac{e^{kt_1}}{e^{kt_2}}$$

Flipping this and solving for the constant of proportionality, k:

$$\frac{I_2}{I_1} = kt_2 - kt_1$$

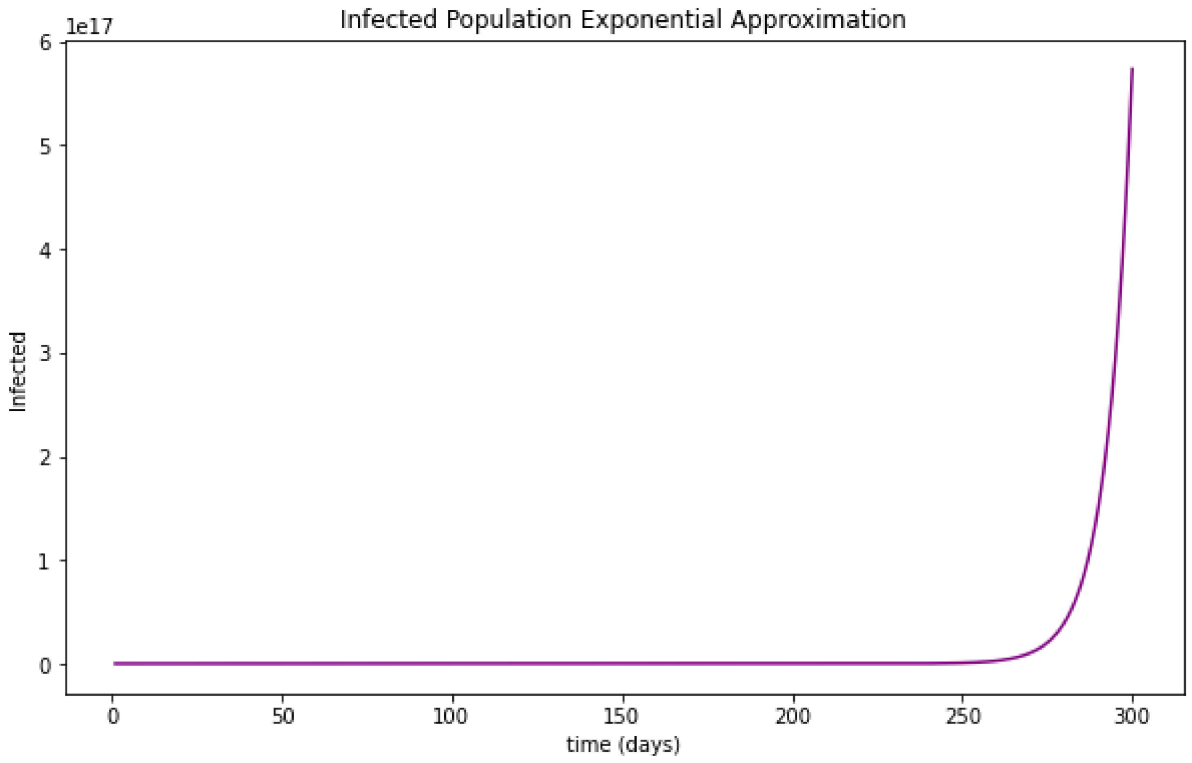
$$k = \frac{I_2}{I_1}(t_2 - t_1)^{-1}$$

Part d: plot that function

In [112…
```
t = list(range(1,301))
I0 = 2
```

```
k = 0.13399 # from using equation at two points
def I(k, t):
    i = I0 * exp(k * t)
    return i

values = []
for i in t:
    v = I(k,i)
    values.append(v)

plt.figure(figsize=(10, 6))
plt.title("Infected Population Exponential Approximation")
plt.plot(t, values, color = 'purple')
plt.xlabel("time (days)")
plt.ylabel("Infected")
```

Out[112]:
```
Text(0, 0.5, 'Infected')
```



Problem 5: Blackbody Problem, assuming T = 5000 K for all steps

Part a: Determine the value of the Blackbody function for different wavelengths

In [77]:
```
T = 5000  # in K
h = 6.67e-34
k = Boltzmann
c = 3e8
wavelengths = [0, 400e-9, 800e-9, 1200e-9, 2000e-9, 4000e-9]

def planck_function(wavelength):
    return (2 * h * c**2) / (wavelength**5) * (np.exp((h * c) / (wavelength * k * T))

B_vals = [planck_function(w) for w in wavelengths if w != 0]
results = [[0,math.inf],[4e-07,5393160.112747346], [8e-07,337072.51014520554],
           [1.2e-06,66582.22483826213], [2e-06,8629.056259717263], [4e-06, 539.3160162
```

```
cols2 = ["wavelength (nm)", "B_lambda"]
print(tabulate(results, headers=cols2))
B_vals = [math.inf] + B_vals
```

```
  wavelength (nm)          B_lambda
-----------------  ----------------
                0               inf
             4e-07         5.39316e+06
             8e-07    337073
             1.2e-06   66582.2
             2e-06      8629.06
             4e-06       539.316
```

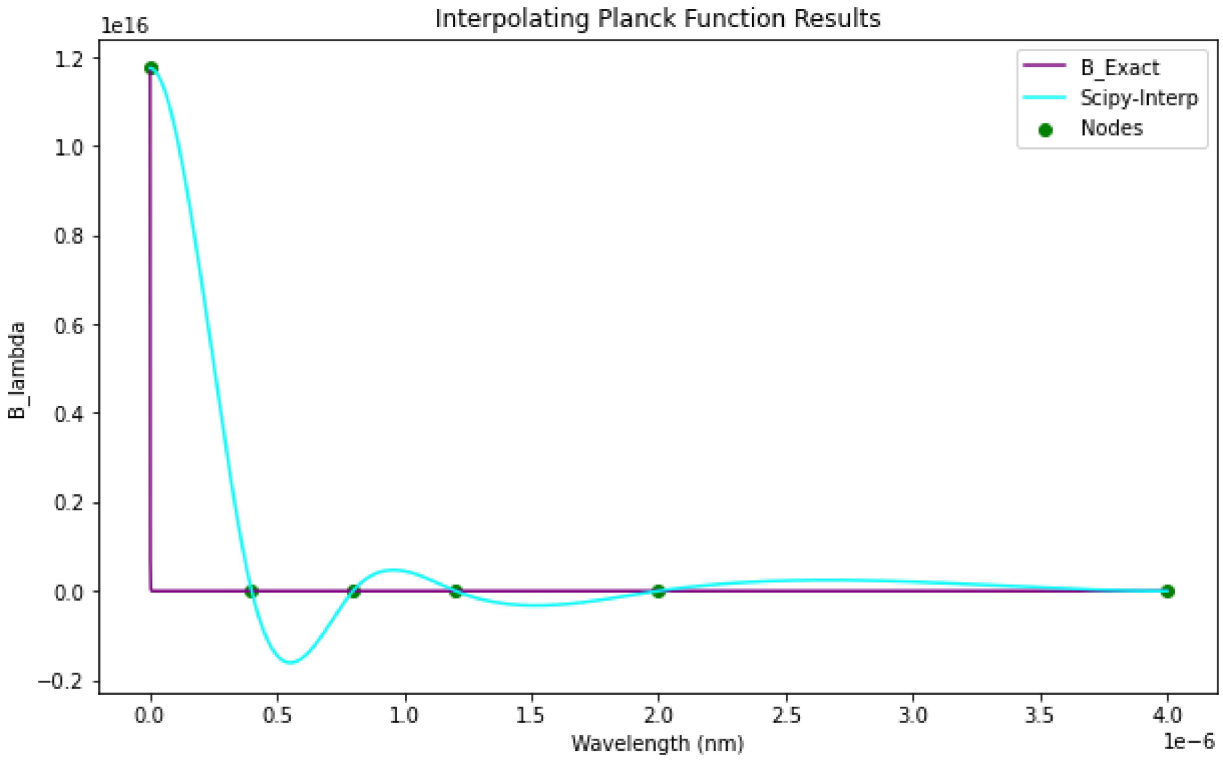
Part b:Generate a cubic spline fit to the Planck function using the six nodes found

In [111…

```
# step size for interpolation
step_size = 1e-9
a = wavelengths[0]
b = wavelengths[5]
points = int((b - a)/ (step_size))

lambdas = np.linspace(a, b, points + 1)
B_exact = []
cubic_interp = []

# pop the first element bc the Planck function does not work for lamaba --> 0
B_vals.pop(0)
wavelengths.pop(0)

# maybe append another point to interpolate ?
wavelengths = [1e-9] + wavelengths
B_vals = [planck_function(1e-9)] + B_vals

# doing interpolation things
interp = scipy.interpolate.CubicSpline(wavelengths, B_vals, bc_type = "clamped")
for i in lambdas:
    B_exact.append(planck_function(i))
    cubic_interp.append(interp(i))

# plotting results
plt.figure(figsize=(10, 6))
plt.plot(lambdas, B_exact, label = "B_Exact", color ='purple')
plt.plot(lambdas, cubic_interp, label ="Scipy-Interp", color ='cyan')
plt.scatter(wavelengths, B_vals, color = "green", label ="Nodes")

plt.title("Interpolating Planck Function Results")
plt.ylabel("B_lambda")
plt.xlabel("Wavelength (nm)")
plt.legend()
```

Out[111]:

```
<matplotlib.legend.Legend at 0x1ccf8a93790>
```



Part c: calculate erros of cubic spline fit

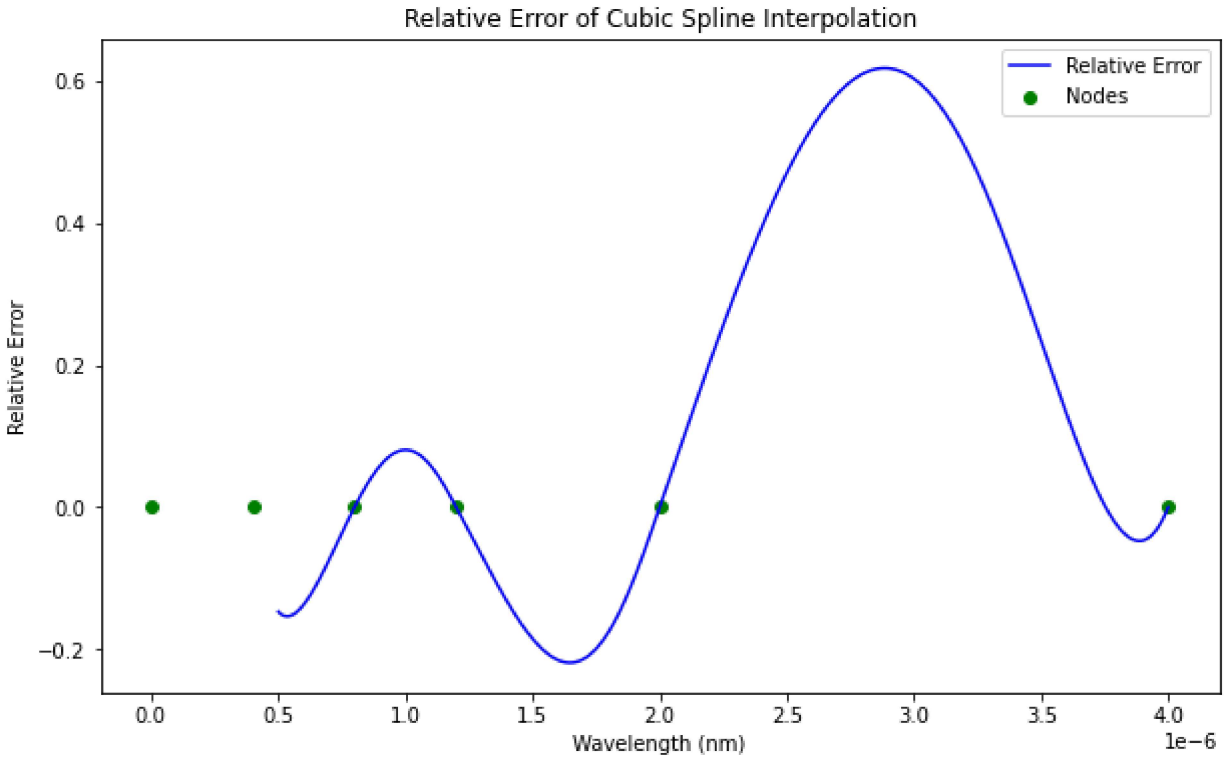
In [110…

```
def relative_err(interp, true):
    rel_err = []
    for i in range(len(interp)):
        rel_error = (interp[i] - true[i])/true[i]
        rel_err.append(rel_error)
    return rel_err

# compute relative error
errs = relative_err(cubic_interp, B_exact)
zeros = np.zeros(6) # err is 0 at nodes

plt.figure(figsize=(10, 6))
plt.plot(lambdas[500:], errs[500:], color = 'blue', label = "Relative Error")
plt.scatter(wavelengths, zeros, color = "green", label ="Nodes")
plt.title("Relative Error of Cubic Spline Interpolation")
plt.ylabel("Relative Error")
plt.xlabel("Wavelength (nm)")
plt.legend()
# green nodes are where relative error is 0
```

```
C:\Users\vince\AppData\Local\Temp\ipykernel_12292\662364571.py:4: RuntimeWarning: inv
alid value encountered in double_scalars
  rel_error = (interp[i] - true[i])/true[i]
C:\Users\vince\AppData\Local\Temp\ipykernel_12292\662364571.py:4: RuntimeWarning: div
ide by zero encountered in double_scalars
  rel_error = (interp[i] - true[i])/true[i]
```

Out[110]:

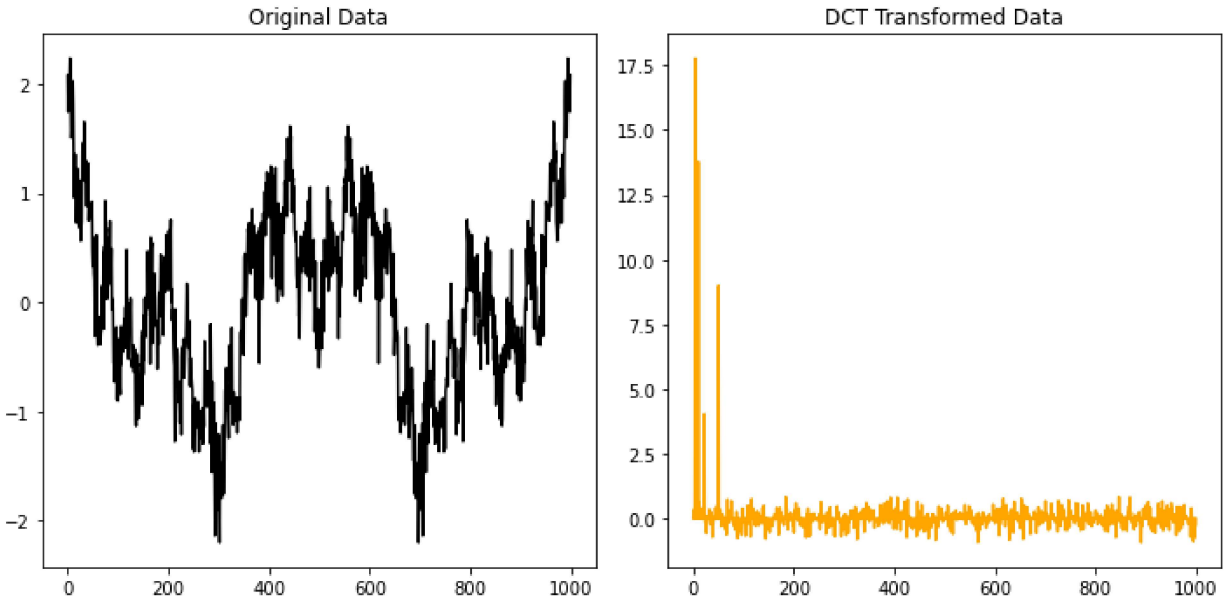
```
<matplotlib.legend.Legend at 0x1ccf89b2ca0>
```



Problem 6: Fourier Transform stuff

Parts a and b: doing things

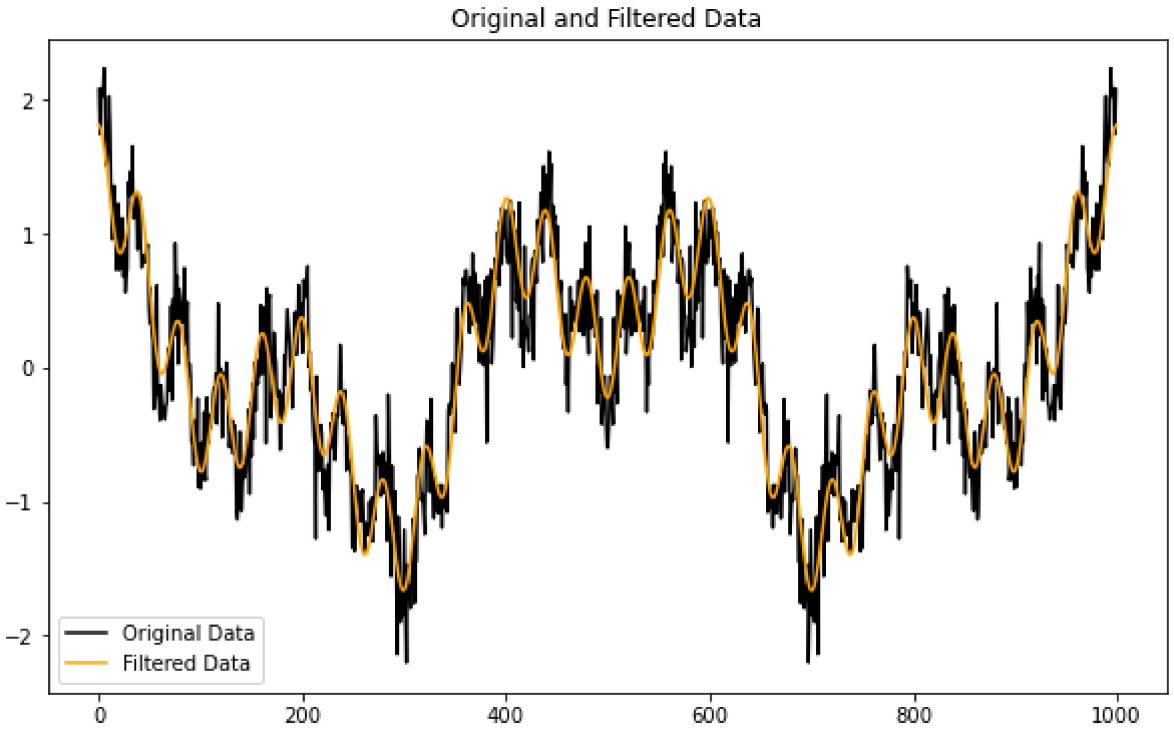
In [97]:
```
#data = np.loadtxt("C:\Users\vince\Downloads\hw8_prob4_data.txt", float)
#cs = scipy.fft.dct(ys)

file_path = r'C:\Users\vince\Downloads\hw8_prob4_data.txt'
with open(file_path, 'r') as file:
    lines = file.readlines()
    data = [float(line.strip()) for line in lines]

dct_result = dct(data, type=2, norm='ortho')

# Plot the original and transformed data
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(data, color = 'black')
plt.title('Original Data')

plt.subplot(1, 2, 2)
plt.plot(dct_result, color = 'orange')
plt.title('DCT Transformed Data')

plt.tight_layout()
plt.show()
```



Part c: Perform a manual filter to get rid of the noise by setting all coefficients equal to 0 unless they have a magnitude of at least 100. Create a new data curve by performing the inverse cosine transform on the filtered coefficients, and plot it on top of the original data

In [109…

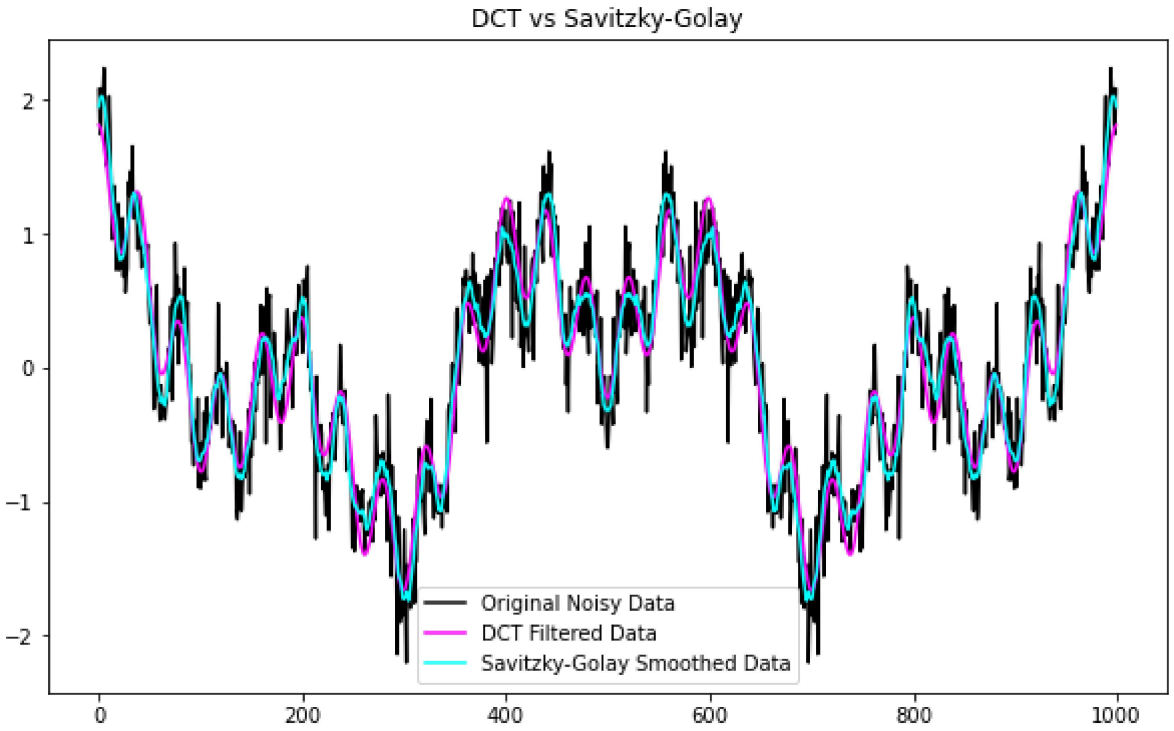
```
coefficients = fft(data)

# Set small coefficients to zero
coefficients[np.abs(coefficients) < 100] = 0

# Perform Inverse Fourier Transform
filtered_data = ifft(coefficients)

# Plot the original and filtered data
plt.figure(figsize=(10, 6))
plt.plot(data, label='Original Data', color='black')

# Filtered Data
plt.plot(filtered_data.real, label='Filtered Data', color='orange')

plt.title('Original and Filtered Data')
plt.legend()
plt.show()
```



Part d: Savy boi

In [102…

```
coefficients = fft(data)

# Set small coefficients to zero using DCT filtering
coefficients[np.abs(coefficients) < 100] = 0

# Perform Inverse Fourier Transform for DCT filtering
dct_filtered_data = ifft(coefficients).real

# Smooth the noisy data using Savitzky-Golay filter
window_size = 21 # Adjust as needed
poly_order = 3   # Adjust as needed
smoothed_data = savgol_filter(data, window_size, poly_order)

# Plot the data
plt.figure(figsize=(10, 6))

# Original Noisy Data
plt.plot(data, label='Original Noisy Data', color='black')

# DCT Filtered Data
plt.plot(dct_filtered_data, label='DCT Filtered Data', color='magenta')

# Savitzky-Golay Smoothed Data
plt.plot(smoothed_data, label='Savitzky-Golay Smoothed Data', color='cyan')

plt.title('DCT vs Savitzky-Golay')
plt.legend()
plt.show()
# i love where the legend placed itself
```



Problem 7: Gas particle collision problem

Part a: get R1-R2 in terms of the impact parameter, b:

The impact parameter forms right triangle with the vectors R1 and R2; therefore, we can apply pythagorean theorem: $c^2 = a^2 + b^2$ to get the vectors in terms of the impact parameter.

$$R_1 - R_2 = a^2 + b^2 = c^2,$$

And since the radius of the particle is the same: $c^2 = 4r^2$. Now we can solve for the length, a:

$$a = \sqrt{(4r^2 - b^2)},$$

Plugging this back into $c^2 = a^2 + b^2$ with the new expression for a: $\sqrt{(4r^2 - b^2)} + b^2 = 4r^2$, so our final expresion for $R_1 - R_2$ is:

$$R_1 - R_2 = \sqrt{(4r^2 - b^2)}\hat{x} - b^2\hat{y}.$$

In [36]:
```
# part a continued
# now, using R1-R2 to create functions for computing the velocity after a collision

def v1_final(v1, v2, R):
    v1 = np.asarray(v1)
    v2 = np.asarray(v2)
```

```
    R = np.asarray(R)
    return (v1 - ((v1 - v2)@(R)) / (np.linalg.norm(R)**2) * (R))

def v2_final(v1, v2, R):
    v1 = np.asarray(v1)
    v2 = np.asarray(v2)
    R = np.asarray(R)
    # bc math: R2-R1 is just negative R1-R2
    return (v2 - ((v1 - v2)@(R)) / (np.linalg.norm(R)**2) * (-R))
```

Part b: Generate 1000 particles with the speeds indicates and compute the initial average speed and rms speed.

In [37]:
```
# generate those particles
speeds = np.zeros(1000)
speeds[:10] = 5000 #m/s
speeds[10:] = 100 #m/s

# average speed
average_speed = np.mean(speeds)

# rms speed
vrms = sqrt(sum(speeds**2) / 1000)

print("Initial Average Speed:", average_speed, "m/s")
print(f"Initial rms: {vrms:.2f} m/s")
```

```
Initial Average Speed: 149.0 m/s
Initial rms: 509.80 m/s
```

Part c: Perform a million random collisions between the particles, updating the new speed to the speeds list

In [38]:
```
for i in range(1_000_000):
    # randomly select theta1 and b. theta2 is always = 0
    theta1 = np.random.uniform(0, 2 * pi)
    theta2 = 0
    b = np.random.uniform(-2, 2)

    # choose two lucky particles & get their initial speed
    particle1, particle2 = np.random.choice(range(999), 2)
    spd1, spd2 = speeds[particle1],speeds[particle2]

    # vecloity vectors of particles
    v1_vec = [spd1*cos(theta1), spd1*sin(theta1)]
    v2_vec = [spd2*cos(theta2), spd2*sin(theta2)]

    # compute this boi
    R1R2 = [-sqrt(4 -b**2), -b]

    # call functions to get new particle speeds
    v1_speed = v1_final(v1_vec, v2_vec, R1R2)
    v2_speed = v2_final(v1_vec, v2_vec, R1R2)

    # update list with new speeds before next iteration
    speeds[particle1] = np.linalg.norm(v1_speed)
    speeds[particle2] = np.linalg.norm(v2_speed)
```

Part d: Compute new average speed and rms after the collisions are finsihed

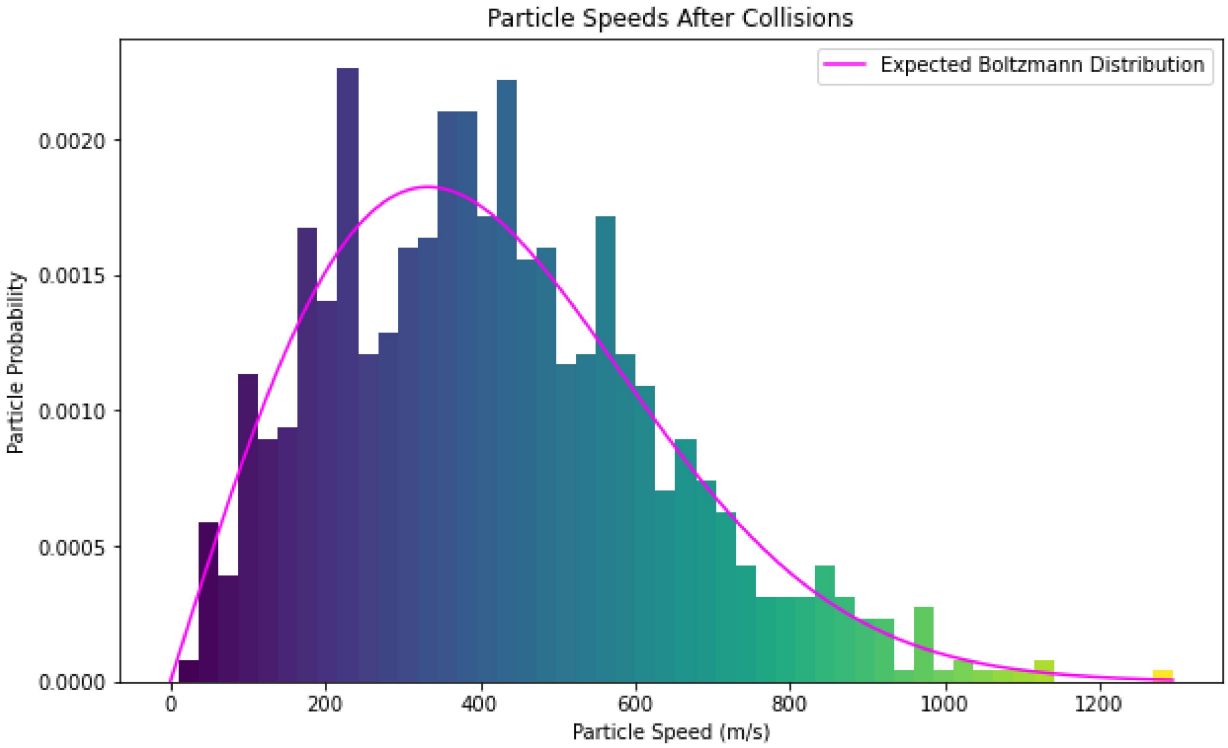
In [39]:
```
final_speeds = speeds
average_speed = np.mean(final_speeds)
vrms = sqrt(sum(final_speeds**2) / 1000)

print("Final Average Speed:", average_speed, "m/s")
print(f"Final rms: {vrms:.2f} m/s")
```

```
Final Average Speed: 419.12787759649973 m/s
Final rms: 470.48 m/s
```

Part e: Plotting time

In [115…
```
# plotting histogram
cm = plt.cm.get_cmap('viridis')
plt.figure(figsize=(10, 6))
N, bins, patches = plt.hist(final_speeds, bins=50, density = True)
plt.xlabel('Particle Speed (m/s)')
plt.ylabel('Particle Probability')
plt.title('Particle Speeds After Collisions')
bin_centers = 0.5 * (bins[:-1] + bins[1:])
col = bin_centers - min(bin_centers)
col /= max(col)
for c, p in zip(col, patches):
    plt.setp(p, 'facecolor', cm(c))

# plotting boltzman over histogram
kB = Boltzmann  # Boltzmann constant
m = 5.310e-26  # kg
T = (mass * vrms**2) /(2*kB)

# Plot Boltzmann distribution for comparison
spds = np.linspace(0, max(final_speeds),1000)
boltz = (((m * spds) /(kB * T))) * (np.exp((-m * spds**2) /(2 * kB * T)))
plt.plot(spds, boltz, 'magenta', label='Expected Boltzmann Distribution')
plt.legend()
```

Out[115]:
```
<matplotlib.legend.Legend at 0x1ccf8aa3e20>
```



The probability distribution follows the expected curve, so the particles are thermalized after the collisions occur and follow the general distribution of a Boltzmann plot. With more collisions, the data would likely match the expected curve better.

I hope you like the color gradient histogram, I felt an unecessary need to do it because it sounded cool but was very annoying to make.