P346 Computational Physics lab
Project Report

From Theory to Practice: Applications of Numerical Calculus in Multidisciplinary Problems

Submitted by
Gayatri P
(2211185)
Integrated M.Sc. (3rd year)
School of Physical Sciences

Under the guidance of
Dr. Colin Benjamin
School of Physical Sciences
NISER

November, 2024

Abstract

Numerical Calculus and Differential Equations have wide range of applications in various different fields. Here, we have presented five such practical applications by solving problem statements as described in *Numerical Methods: An Inquiry Based Approach with Python*, Sullivan 2020.

The first two problems involve solving ordinary and partial differential equations, in particular, the heat equation, to find the ideal wall thickness of an adobe house and visualise population growth and diffusion. The last three problems use concepts of numerical differentiation and integration to perform edge detection in images, calculate total water discharge through a dam and measure emission lines strengths of a galaxy spectrum.

All the detailed code used in this project (along with simulational videos) can be found on this website.

Contents

1 | Hunting and Diffusion

In this problem, we consider a population diffusion model which incorporates hunting. Firstly, we study the population size and stability in the presence of a hunting agent. Secondly, we analyze the same problem by incorporating spatial diffusion of the population. Finally, we also look at some scenarios where the terrain is non-uniform, resulting in a non-uniform population distribution and analyse the solution at different timestamps.

1.1 | Modelling population size

Let u be a function modelling a mobile population living in an environment with a growth rate of $r\%$ per year with a carrying capacity of K. The typical equation that governs the size of the population is,

$$\frac{du}{dt} = ru\left(1 - \frac{u}{K}\right) \tag{1.1}$$

Now assume that hunters harvest $h\%$ of the population per year. The above equation can account for this by adding an additional hunting term.

$$\frac{du}{dt} = ru\left(1 - \frac{u}{K}\right) - hu \tag{1.2}$$

Eq. 1.2 is a first-order nonlinear ordinary differential equation. The parameter space for u can defined from $u \in [0, \infty)$, where the carrying capacity, $K \in \mathbb{R}^+$. We will see that after solving the differential equation the u actually only goes from 0 to K. Both $r, u \in [0, 1]$, as they represent the fraction of population growing/getting hunted. Since this is an initial value problem, we need to provide some population size at time $t = 0$. Assuming we need at least 2 organisms to reproduce, we provide $u(0) = 2$. However the solution will not be much different with any other small value of $u(0)$.

We will now discuss how to numerically solve this differential equation. Note that the units of r and h throughout this section is $[\text{Time}]^{-1}$ and u and K are unitless. For simplicity, we won't be explicitly mentioning these units moving forward.

1.1.1 | Theoretical Approach

The simplest method to solve a differential equation of the form $u' = f(u, t)$ is Euler's method. Here, we discretize time into steps of Δt. Given some initial conditions u_i, we approximate the next iteration as,

$$u_{i+1} = u_i + f(u_i, t_i)\Delta t \tag{1.3}$$

However, with large enough time steps, the error (of $O(h^2)$ in this case) accumulates, and the system quickly deviates from the actual solution. The implicit Euler's method however, tries to stabilize the solution by using

$$u_{i+1} = u_i + f(u_{i+1}, t_{i+1})\Delta t \tag{1.4}$$

However, while this is more numerically stable, this method undershoots the solution. For example, if we were solving this for a harmonic oscillator problem, the implicit Euler's method would appear to 'leak' energy.

The Runge-Kutta method tries to fix the problem by taking a middle ground between the explicit and implicit Euler's methods to somewhat cancel out the deviations. Mathematically, the Runge-Kutta second order (RK2) method takes k_1 and k_2 as follows, as takes an average of the two values to calculate u_{i+1}.

$$k_1 = f(u_i, t_i), k_2 = f(u_i + k_1, t_i + \Delta t) \tag{1.5}$$

$$u_{i+1} = u_i + \frac{dt}{2}(k_1 + k_2) \tag{1.6}$$

Notice that the value we plug into k_2 is the same value we calculate for explicit Euler's method. Hence, here, the slope of the function at both increments is taken into account to calculate u_{i+1}.

This algorithm can be further improved by taking two extra weights to obtain the RK4 method. Here, we also consider the point $t_i + \Delta t/2$ and take the weighted average of four different slope values.

$$\begin{aligned} k_1 &= f(u_i, t_i) \\ k_2 &= f\left(u_i + k_1 \frac{\Delta t}{2}, t_i + \frac{\Delta t}{2}\right) \\ k_3 &= f\left(u_i + k_2 \frac{\Delta t}{2}, t_i + \frac{\Delta t}{2}\right) \\ k_4 &= f\left(u_i + k_3 \Delta t, t_i + \Delta t\right) \\ u_{i+1} &= \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{1.7}$$

Where the weights for each k value come from the Taylor series derivation. RK4 is good enough for our purposes with a total accumulated error of the order $O(h^5)$.

1.1.2 | Results

In Fig. 1.1, we have implemented forward Euler, Runge-Kutta 2 and Runge-Kutta 4 methods for a specific case where $r = 0.5, h = 0.25$, $K = 100$ and $u(0) = 2$. The real solution with these parameters was obtained analytically[1] as $u(t) = \frac{50e^{t/4}}{24+e^{t/4}}$, which is also plotted in the figure.

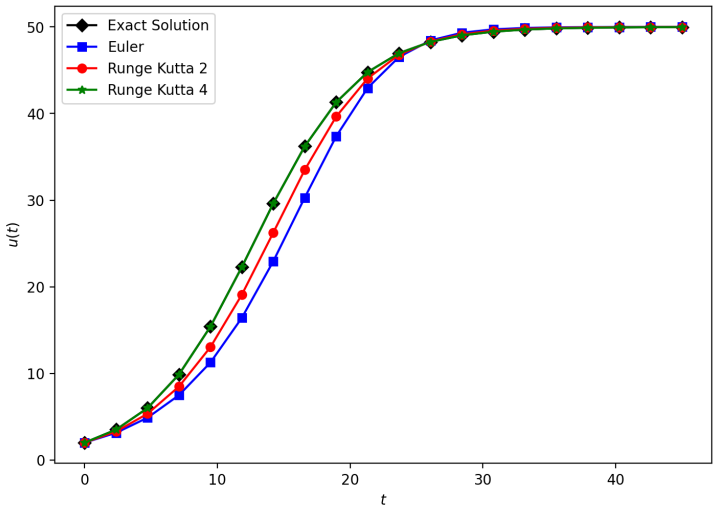
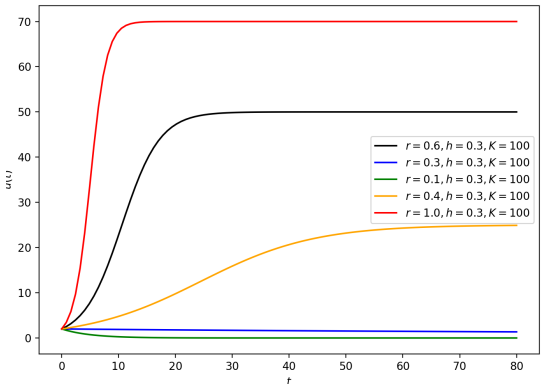


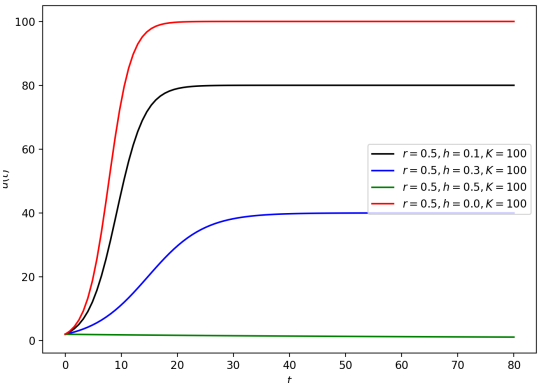
Figure 1.1: Comparison of solutions obtained using 3 different methods compared with the real solution, with the same step sizes. Here, we can see that the RK4 method almost entirely overlaps the real solution, and RK2 is the second most accurate solution.

In every case, we can see that the population size saturates after a certain amount of time at a much lower carrying capacity due to hunting. Solutions obtained using RK4 method for some other variations of parameters are shown below.

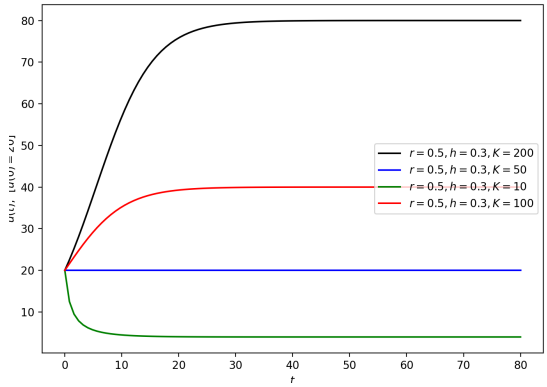
[1] Source.



(a) With h and K kept constant, the time for saturation decreases and the saturated population increases with an increase in growth rate.



(b) With r and K kept constant, the population at saturation decreases with an increase in hunting rate as expected. When $h = 0$, the population saturates at its carrying capacity.



(c) With r and h kept constant, the population size at saturation varies proportionately with K. If $K < u(0)$, the population dies off eventually.

Figure 1.2: Solutions obtained for Eq. 1.2 by systematically varying its parameters.

1.2 | Modelling the spread of individuals in a population

Now, we want to model the spread of the mobile population. Before setting up the simulation, we will have 3 basic assumptions –

1. Food is abundant in the entire environment.
2. Individuals in the population like to spread out so that they don't interfere with each others' hunt for food.
3. It is equally easy for the individuals to travel in any direction in the environment.

We can model the diffusion of the species through the environment by adding a diffusion term to Eq. 1.2 and converting it into a partial differential equation, where u is a function of t, x and y.

$$\frac{\partial u}{\partial t} = ru\left(1 - \frac{u}{K}\right) - hu + D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{1.8}$$

Here, $D > 0$ is the diffusion coefficient, indicating the ease of diffusion. To simplify the process we consider the 2D domain $(x, y) \in [0, 1] \times [0, 1]$ for the spatial part of this equation. Here, K specifies the carry capacity per unit area.

Eq. 1.8 can be solved using methods described in section 2. Here, we will be using the forward (or explicit) Euler method to calculate first and second derivatives. Since we are performing the calculation for every point in the grid, it will be much less computationally expensive than using higher order methods.

1.2.1 | Theoretical Approach

We can implement Eq. 1.8 by discretising space and time into Δx and Δt. Here we keep $\Delta x = \Delta y$ for simplicity. The equation can thus be rewritten as,

$$\frac{u_{i,j}^{(t+1)} - u_{i,j}^{(t)}}{\Delta t} = r u_{i,j}^{(t)} \left(1 - \frac{u_{i,j}^{(t)}}{K}\right) - h u_{i,j}^{(t)} + D \left(\frac{u_{i+1,j}^{(t)} - 2u_{i,j}^{(t)} + u_{i-1,j}^{(t)}}{\Delta x^2} + \frac{u_{i,j+1}^{(t)} - 2u_{i,j}^{(t)} + u_{i,j-1}^{(t)}}{\Delta y^2}\right) \tag{1.9}$$

Where i, j represents the spatial coordinates for t^{th} time snapsot. Here, we have used the finite difference approximation for second derivative to calculate the spatial diffusion part.

$$\frac{d^2 f}{dx^2} = \frac{f(x+1) - 2f(x) + f(x-1)}{\Delta x^2} \tag{1.10}$$

Now, we can rearrange to get $u_{i,j}^{(t+1)}$ as,

$$u_{i,j}^{(t+1)} = u_{i,j}^{(t)} + \Delta t \left[(r-h)u_{i,j}^{(t)} - \frac{r(u_{i,j}^{(t)})^2}{K} + \frac{D}{\Delta x^2}\left(u_{i+1,j}^{(t)} + u_{i-1,j}^{(t)} + u_{i,j+1}^{(t)} + u_{i,j-1}^{(t)} - 4u_{i,j}^{(t)}\right)\right] \tag{1.11}$$

To implement boundary conditions for the spatial coordinates, we can add the Dirichlet boundary condition, $u|_{x,y=0} = 0$. However, one can argue that the physical interpretation of the Dirichlet boundary condition, i.e. that the population dies off near the physical constraints of the boundary, does not make much sense. Hence, here we will apply the Neumann boundary condition, $\frac{\partial u}{\partial n}\big|_{x,y=0} = 0$. This makes sure that the population can spread evenly, even near the boundary points.

To implement Neumann boundary conditions, one can simply put

$$u_{i=0} = u_{i=1} \text{ and } u_{i=n} = u_{i=n-1} \tag{1.12}$$

for both x and y coordinates. This makes sure that their derivative at the boundary goes to zero.

1.2.2 | Implementation

Here we have used the `NUMBA` module which generates optimized machine code from pure Python to speed up the computation process.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import PillowWriter
from matplotlib import cm
import numba
from numba import jit

# Initialise space grid with 50 random spawns
n = 100
i = 5 # to avoid spawing any population near the edge
coords = np.array([(np.random.randint(i, n-i), np.random.randint(i, n-i)) for _ in range
    (50)])
init_population = np.zeros((n, n))
for x, y in coords:
    init_population[x,y] = 2 # arbitrary initial value

# diffusion constant
D = 0.01

# set the dimensions of the problem
x = 1
dx = 0.05
dt = 0.0001 # such that D*dt/dx**2 < 1/4
```

```
times = 252000 # number of iterations
times_snapshot = 3600 # total number of snapshots
f = int(times/times_snapshot)

population_frames = np.zeros([times_snapshot, 100, 100])
population_frames[0] = init_population
population_density = np.zeros(times) # keeps track of the average population density

# Solving the PDE
# Set up numba function
@numba.jit("(f8[:,:,:], f8, f8, f8)", nopython=True, nogil=True, fastmath = True,
    parallel=True)
def solve_pde(environment, K, r, h):
    cs = environment[0].copy() #current state
    length = len(cs[0])
    density = np.zeros(times)
    density[0] = np.average(cs) # average population density
    cf = 0 # current frame

    for t in range(1, times):
        ns = cs.copy() # new state

        # Since only iterate spatially from 1 to n-1
        # the algorithm by design is implementing Dirichlet BCs
        for i in range(1, length-1):
            for j in range(1, length-1):
                growth = dt*((r-h)*cs[j][i] - (r*cs[j][i]**2)/K)
                diffusion = D*dt/dx**2 * (cs[j+1][i] + cs[j-1][i] +\
                                          cs[j][i+1] + cs[j][i-1] -\
                                          4*cs[j][i])
                ns[j][i] = cs[j][i] + diffusion + growth

        # Implementing Neumann BCs
        ns[:,0] = ns[:,1] # left boundary
        ns[:,-1] = ns[:,-2] # right boundary
        ns[0,:] = ns[1,:] # top boundary
        ns[-1,:] = ns[-2,:] # bottom boundary

        density[t] = np.average(cs)
        cs = ns.copy()
        if t%f==0: # take snapshot
            cf = cf + 1
            environment[cf] = cs

    return environment, density

# Setting up the parameters
K, r, h = 1, 0.9, 0.2

# Get population snapshots and population size over time and plot
population_frames, population_sizes = solve_pde(population_frames, K, r, h)
plt.plot(np.linspace(0, times*dt, times), population_sizes)
plt.xlabel('Time (s)')
plt.ylabel('Total Population')
plt.show()

# generate an animation of the simulation over time
def animate(i):
    ax.clear()
    im = ax.contourf(population_frames[10*i], 100, levels=np.linspace(0,np.max(
    population_sizes),50))
    plt.title(f't = {10*i*f*dt:.2f} sec')
    ax.set_xticks([])
    ax.set_yticks([])
    return fig,

fig, ax = plt.subplots(figsize=(8,6))
fig.colorbar(im, ax=ax)
ani = animation.FuncAnimation(fig, animate,
                              frames=359, interval=50)
ani.save('simulation.gif', writer='pillow', fps=30)
```

Listing 1.1: Code to simulate the hunting and diffusion model

1.2.3 | Results

Here are a few snapshots obtained in the simulation. The full simulation animations can be found here. As expected from the previous section, the population (initially spawned at random points) diffuses slowly until it reaches saturation.

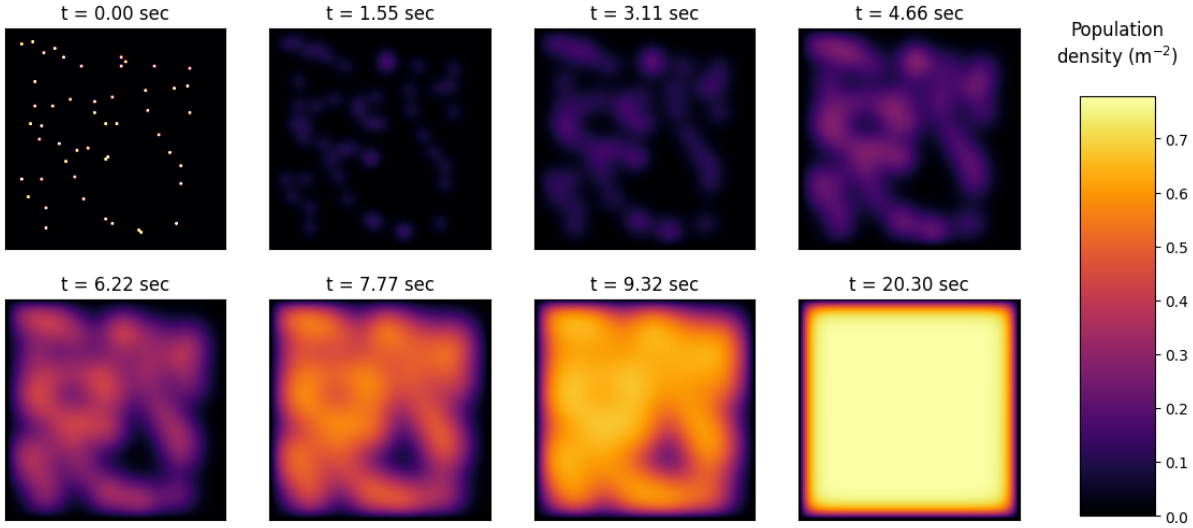


Figure 1.3: Snapshots of the environment at different times with Dirichlet Boundary Conditions ($r = 0.9, h = 0.2$)

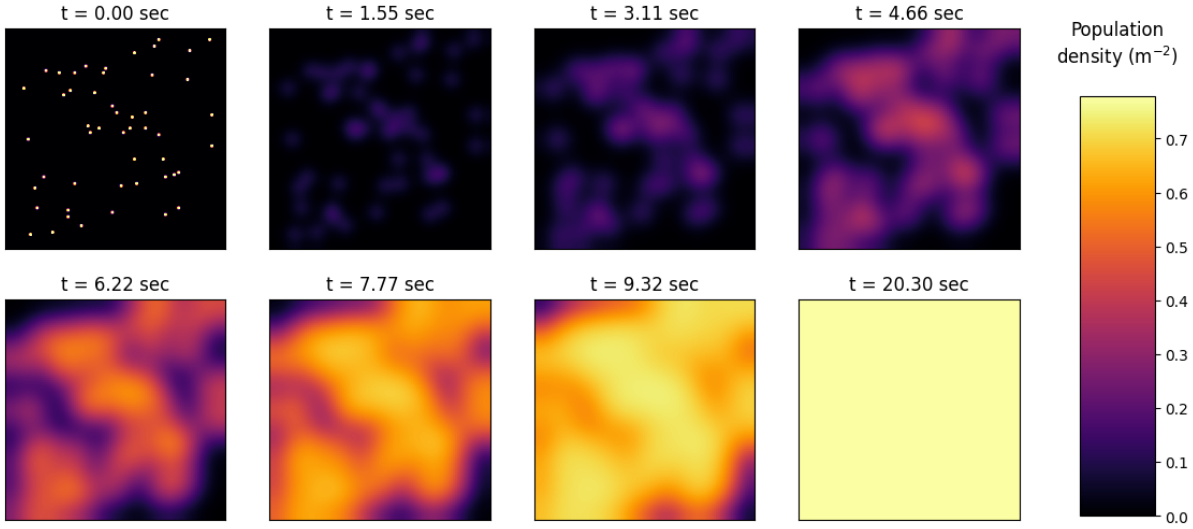


Figure 1.4: Snapshots of the environment at different times with Neumann Boundary Conditions ($r = 0.9, h = 0.2$)

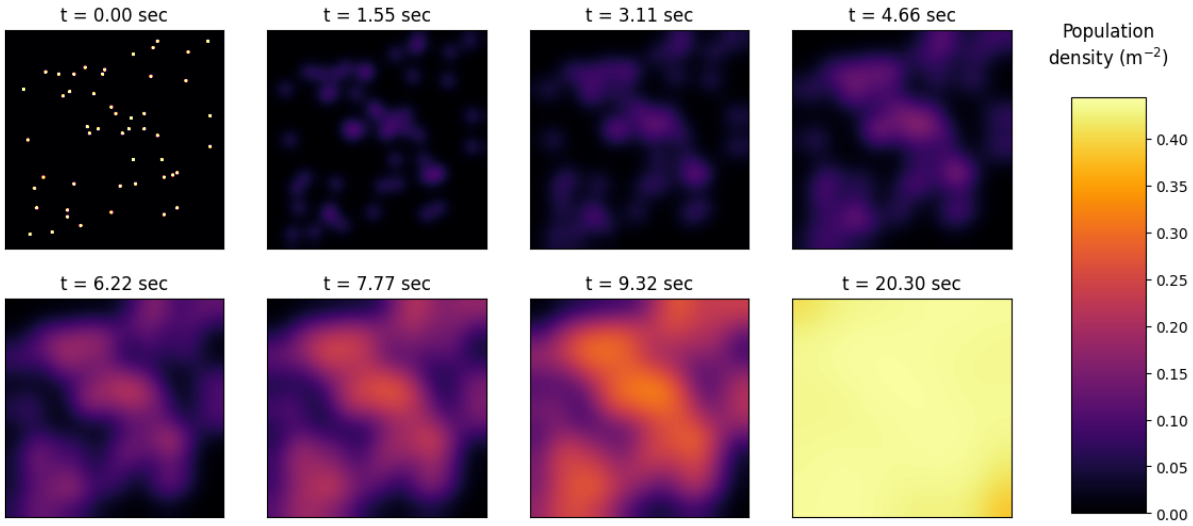


Figure 1.5: Snapshots of the environment at different times with Neumann Boundary Conditions $(r = 0.9, h = 0.5)$

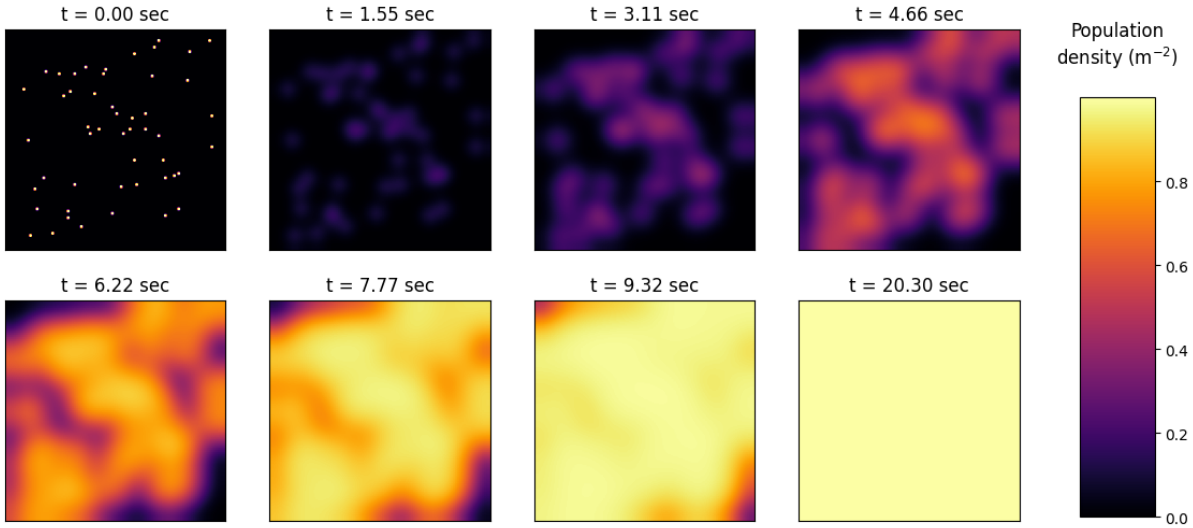


Figure 1.6: Snapshots of the environment at different times with Neumann Boundary Conditions $(r = 0.9, h = 0)$

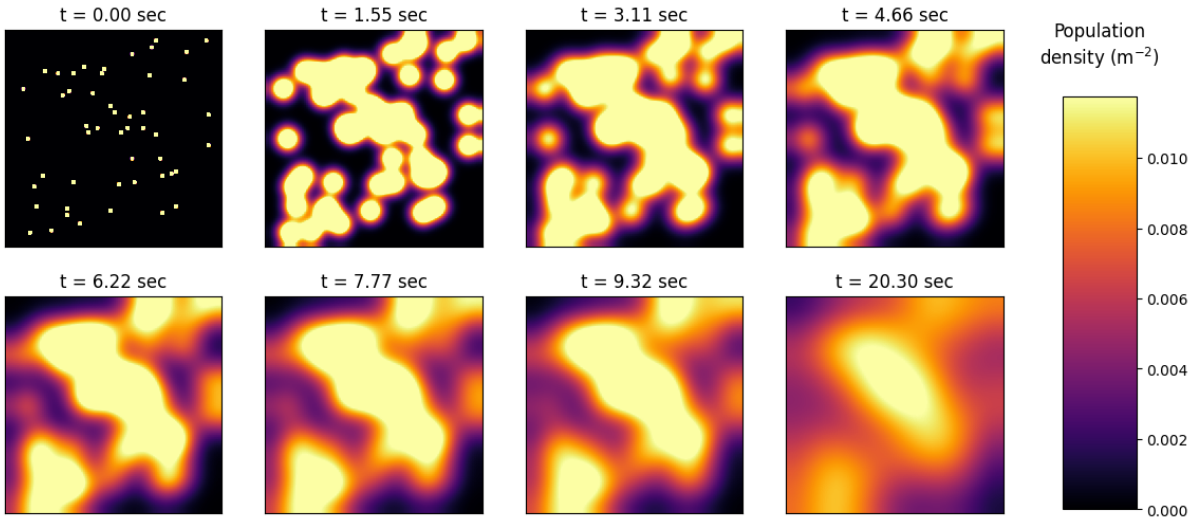


Figure 1.7: Snapshots of the environment at different times with Neumann Boundary Conditions $(r = 0.9, h = 0.9)$

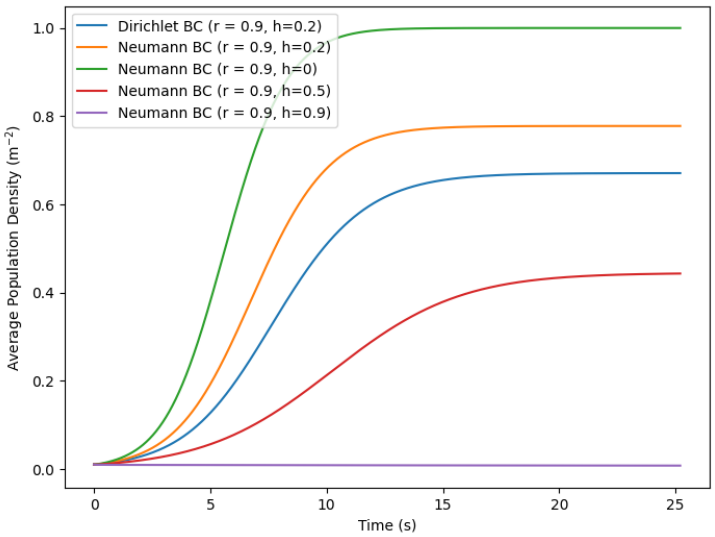


Figure 1.8: The average population density as a function of time for above-shown simulations.

1.3 | Modelling the spread of individuals for a rough terrain

The third assumption in the previous section assumes a smooth terrain where it is equally easy for individuals to travel in any direction. However, this is rarely the case, as the terrain may have various deformities which affect population diffusion. We will now try to model this mathematically.

For rough terrain, described by a 2D scalar function $T(x, y)$, the actual form of the spatial component of Eq. 1.8 becomes $\nabla \cdot T(x, y)\nabla u$. Which means,

$$\frac{\partial u}{\partial t} = ru\left(1 - \frac{u}{K}\right) - hu + \nabla \cdot (T(x, y)\nabla u) \tag{1.13}$$

1.3.1 | Theoretical Approach

We can choose any $T(x, y)$ that is positive in our domain and perform the operation $\nabla \cdot (T\,\nabla u)$ as,

$$\begin{aligned}\nabla \cdot (T\,\nabla u) &= \nabla \cdot \left(T\frac{\partial u}{\partial x}\hat{x} + T\frac{\partial u}{\partial y}\hat{y} \right) \\ &= \frac{\partial}{\partial x}\left(T\frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y}\left(T\frac{\partial u}{\partial y} \right) \\ &= \frac{\partial T}{\partial x}\frac{\partial u}{\partial x} + T\frac{\partial^2 u}{\partial x^2} + \frac{\partial T}{\partial y}\frac{\partial u}{\partial y} + T\frac{\partial^2 u}{\partial y^2} \\ &= T\,\nabla^2 u + \frac{\partial T}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial T}{\partial y}\frac{\partial u}{\partial y} \end{aligned} \tag{1.14}$$

Now consider only the x coordinate. We can approximate for a fixed y,

$$\frac{\partial T}{\partial x}\frac{\partial u}{\partial x} \approx \frac{T(x-dx)[u(x-dx)-u(x)] + T(x+dx)[u(x+dx)-u(x)]}{2\,dx^2} \tag{1.15}$$

$$T\,\nabla^2 u \approx \frac{u(x-dx)-2u(x)+u(x+dx)}{2\,dx^2} T(x) \tag{1.16}$$

Hence, for a fixed t the full divergence term becomes,

$$\begin{aligned}\nabla \cdot (T\,\nabla u)|_{(x,y)} = \frac{1}{2\,dx^2}\Big\{ & T(x,y)[u(x-dx,y)+u(x+dx,y)+u(x,y-dy)+u(x,y+dy)-4u(x,y)] \\ & + T(x-dx,y)[u(x-dx,y)-u(x,y)] + T(x+dx,y)[u(x+dx,y)-u(x,y)] \\ & + T(x,y-dx)[u(x,y-dx)-u(x,y)] + T(x,y+dx)[u(x,y+dx)-u(x,y)]\Big\}\end{aligned} \tag{1.17}$$

Or in terms of coordinates (i,j),

$$\begin{aligned}\nabla \cdot (T\,\nabla u)|_{i,j} = \frac{1}{2\,\Delta x^2}\Big\{ & T_{i,j}[u_{i-1,j}+u_{i+1,j}+u_{i,j-1}+u_{i,j+1}-4u_{i,j}] \\ & + T_{i-1,j}[u_{i-1,j}-u_{i,j}] + T_{i+1,j}[u_{i+1,j}-u_{i,j}] \\ & + T_{i,j-1}[u_{i,j-1}-u_{i,j}] + T_{i,j+1}[u_{i,j+1}-u_{i,j}]\Big\}\end{aligned} \tag{1.18}$$

$$\text{thus, } u_{i,j}^{(t+1)} = u_{i,j}^{(t)} + \Delta t\left[(r-h)u_{i,j}^{(t)} - \frac{r(u_{i,j}^{(t)})^2}{K} + \nabla \cdot (T\,\nabla u)|_{i,j}^{(t)} \right] \tag{1.19}$$

1.3.2 | Implementation

We can implement the terrain function in our environment using `np.meshgrid()` function. Here, we have modelled 4 terrains as shown below, but this method will work for any non-negative terrain function $T(x,y)$.

```
def gauss2d(x, y, cx=0.5, cy=0.5):
    # gaussian in domain [0,1] x [0,1]
    z = np.exp(-(x-cx)**2-(y-cy)**2)
    return z

def gauss2d_inv(x, y, cx=0.5, cy=0.5):
    # inverted gaussian function
    z = 1-np.exp(-(x-cx)**2-(y-cy)**2)
    return z

def twohills(x, y):
    # a combination of two gaussian hills
    z = 1.04-np.exp((-(x-0.2)**2-(y-0.3)**2)/0.15)-np.exp((-(x-0.8)**2-(y-0.7)**2)/0.1)
    return z

def ridge(x,y):
    # a terrain function that looks like a ridge
    return np.sin((x+3)*(y-0.5)**2)
```

```
n = 100
X    = np.linspace(0, 1, n)
Y    = np.linspace(0, 1, n)
X, Y = np.meshgrid(X, Y)

terrain1 = gauss2d(X,Y)
terrain2 = gauss2d_inv(X,Y, cx=0.2, cy=0.4)
terrain3 = twohills(X,Y)
terrain4 = ridge(X,Y)
```

Listing 1.2: Code to model the rough terrains

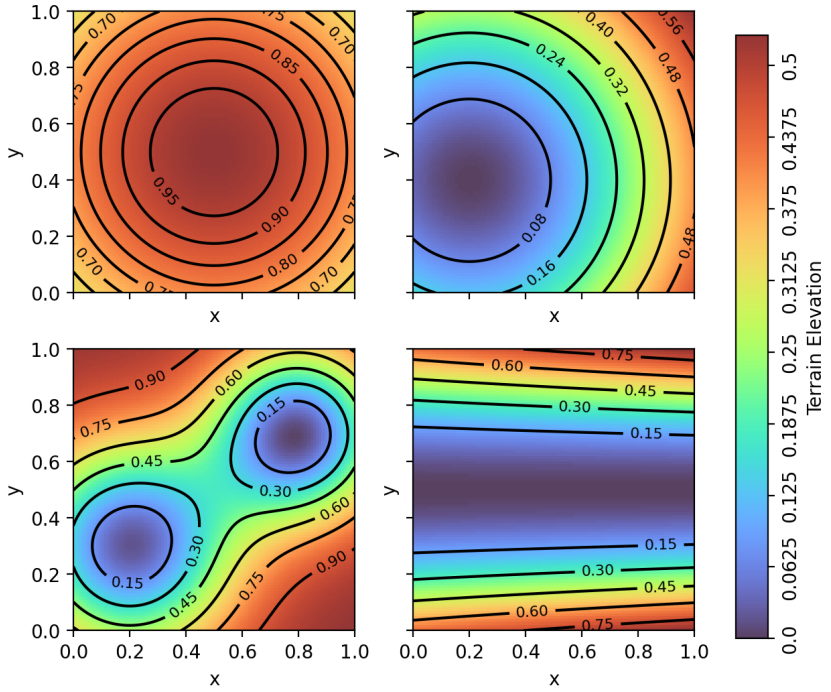


Figure 1.9: Contour Plots of the terrains functions we used for the environment

The only modification to the `solve_pde()` function defined earlier would be the addition of the divergence term.

```
@numba.jit("(f8[:,:,:], f8, f8, f8)", nopython=True, nogil=True, fastmath = True)
def solve_pde(environment, K, r, h):
    cs = environment[0].copy() #current state
    length = len(cs[0])
    density = np.zeros(times)
    density[0] = np.average(cs) # average population density
    cf = 0 # current frame
    D = terrain

    for t in range(1, times):
        ns = cs.copy() # new state

        for i in range(1, length-1):
            for j in range(1, length-1):
                growth = dt*((r-h)*cs[j][i] - (r*cs[j][i]**2)/K)
                diffusion = (dt/2*dx**2)* (D[j,i] *(cs[j, i-1]+cs[j, i+1]+cs[j-1,i]\
                                                    +cs[j+1,i]-4*cs[j,i]) +\
                                           D[j-1,i]*(cs[j-1,i]-cs[j,i])+\
                                           D[j+1,i]*(cs[j+1,i]-cs[j,i])+\
                                           D[j,i-1]*(cs[j,i-1]-cs[j,i])+\
                                           D[j,i+1]*(cs[j,i+1]-cs[j,i]))
                ns[j][i] = cs[j][i] + growth + diffusion

        # Implementing Neumann BCs
        ns[:,0] = ns[:,1] # left boundary
        ns[:,-1] = ns[:,-2] # right boundary
```

```
        ns[0,:] = ns[1,:] # top boundary
        ns[-1,:] = ns[-2,:] # bottom boundary

        density[t] = np.average(cs)
        cs = ns.copy()
        if t%f==0: # take snapshot
            cf = cf + 1
            environment[cf] = cs

    return environment, density
```

Listing 1.3: Modified PDE solver function

1.3.3 | Results

Here again, the population diffuses slowly until it reaches saturation. However, this time, the diffusion is not uniform and is dependent on the terrain function.

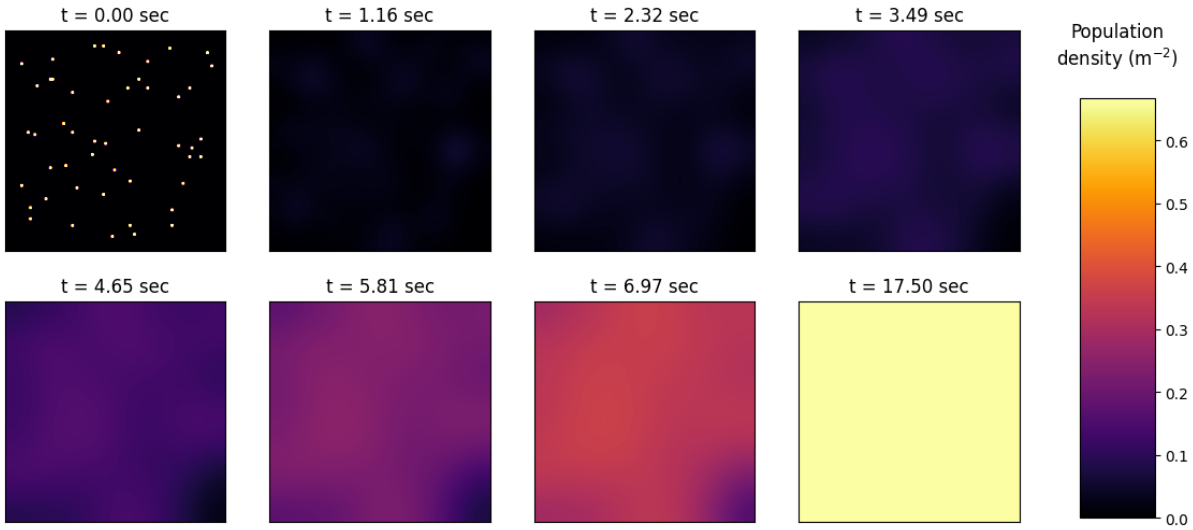


Figure 1.10: Snapshots of the environment at different times with a Gaussian Terrain ($r = 0.9, h = 0.2$)

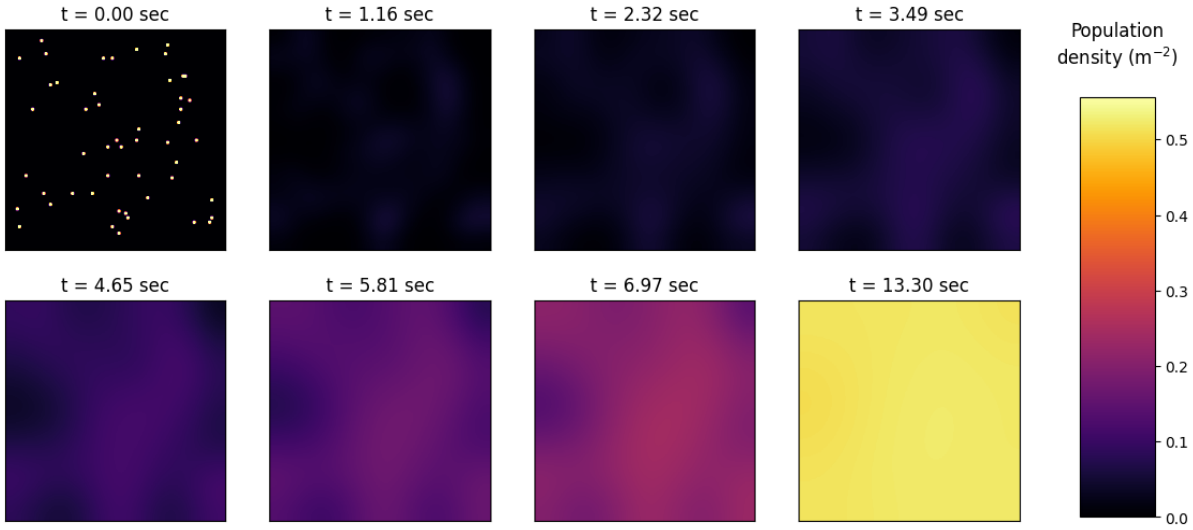


Figure 1.11: Snapshots of the environment at different times with a Gaussian Terrain ($r = 0.9, h = 0.4$). Here while the simulation looks similar to the one above the final saturated value is lesser due to the higher hunting rate.

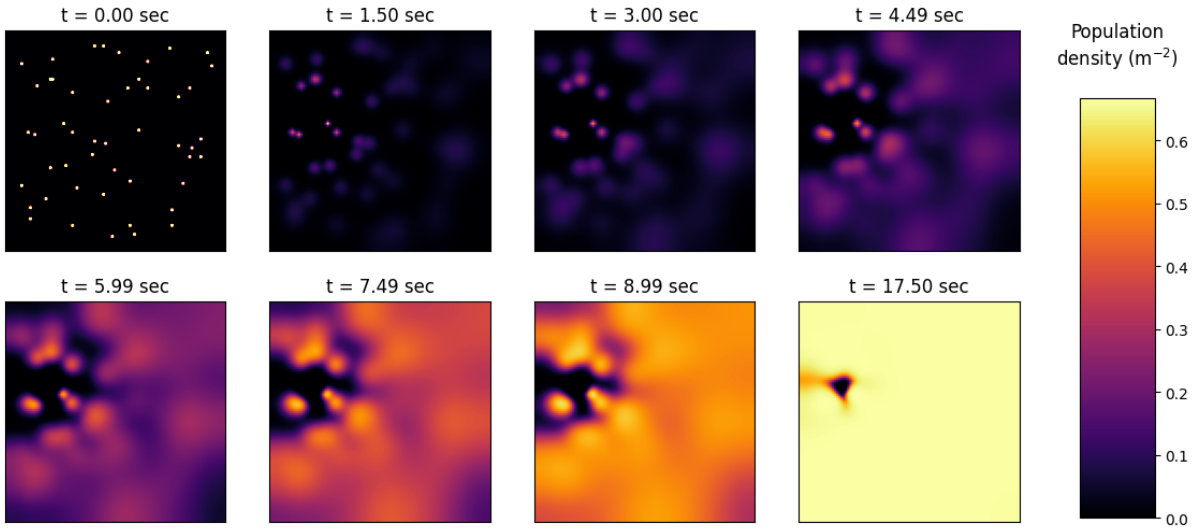


Figure 1.12: Snapshots of the environment at different times with an inverted Gaussian Terrain ($r = 0.9, h = 0.2$)

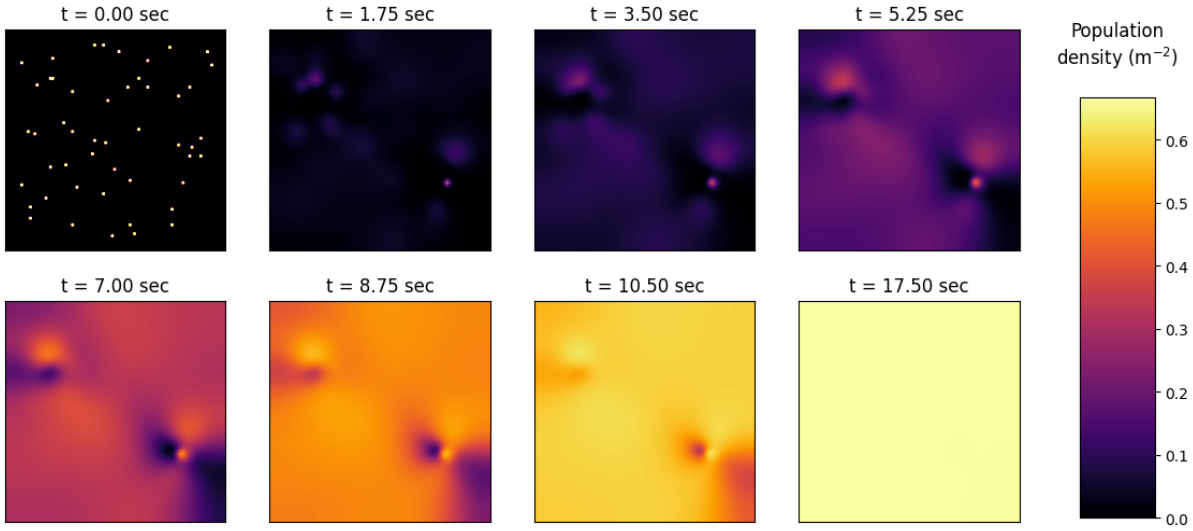


Figure 1.13: Snapshots of the environment at different times with a Gaussian Terrain with two peaks ($r = 0.9, h = 0.2$)

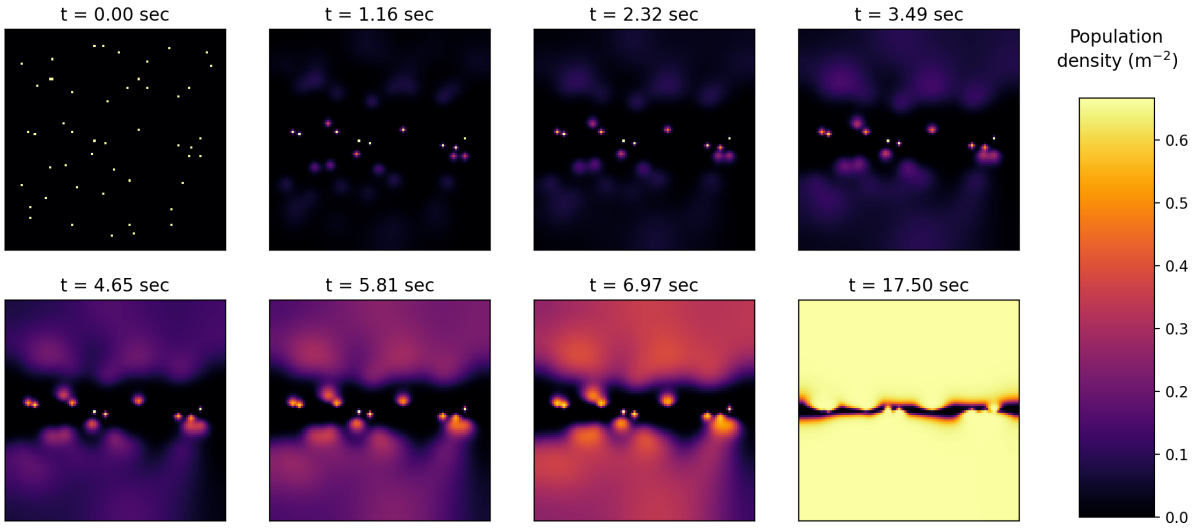


Figure 1.14: Snapshots of the environment at different times with the ridge-like terrain ($r = 0.9, h = 0.2$)

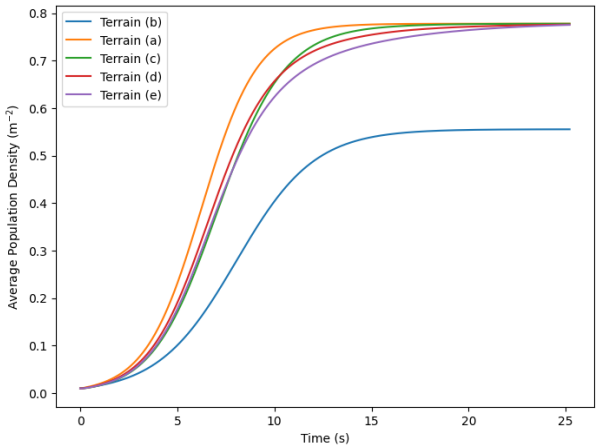


Figure 1.15: The average population density as a function of time for above-shown simulations in order. As you can see, all environments evetually saturate to the same value, except for the one with a lower K value.

The simulations videos for most of the simulations in Section 1.3.3 are available here.

2 | Heating Adobe Houses

Houses in desert climates are usually built using adobe. It is a construction material that uses soil (a mixture of clay, sand and water), stabiliser and binder as raw materials that are mixed and moulded to form sun-dried blocks. Due to this, adobe houses are known for their great thermal efficiency.

The thicker the adobe walls are, the better, as it will maintain a nearly constant inside temperature. However, it would also be more expensive to build. Here, we try to model the heat flow through adobe walls using the heat equation to find the optimum wall thickness. For this, we use synthetic data, which approximates the typical diurnal temperature variation in these climatic regions.

2.1 | Approximating Temperature Variation

Desert climates are known for their high variability in temperature throughout the day. For this project, I have used the temperature profile of Sukkur, Rajasthan, which is located around 200 km from the Thar Desert[2]. During winters, the temperature can vary from 9 °C to 16 °C and during summers the variation is from 29 °C to 45 °C. An approximate average of this is implemented in the project.

Using diurnal temperature variations models described in Parra-Saldivar and Batty 2005, the external temperature variation can be approximated as a sinusoidal curve, as shown in Fig. *2.1*.

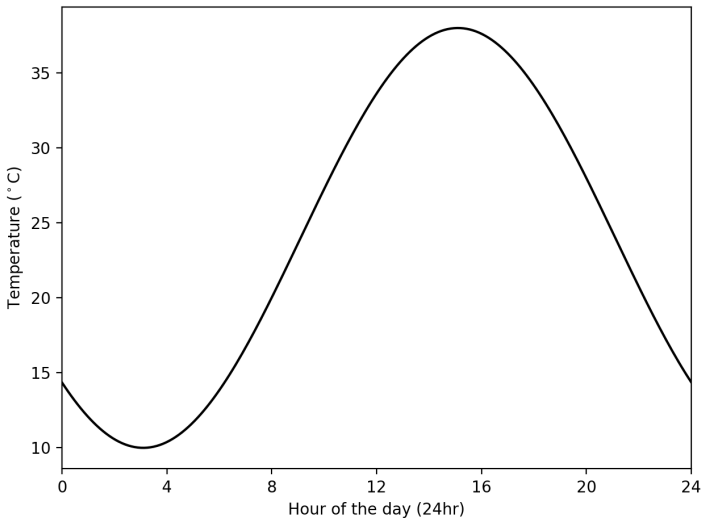


Figure 2.1: Dirunal temperature variation modelled for a day using a sinusoidal curve with equation: $y = 14\sin(\pi x/12 + 3.9) + 24$. The parameters we manually adjusted so that the minimum temperature ($\sim 10°$C) falls around 03:00 and the maximum ($\sim 37°$C) around 15:00 in the day.

2.2 | Theoretical Approach

The heat equation describes the flow of heat through a 3 dimensional space,

$$\frac{\partial T}{\partial t} = \frac{k}{c_p \rho}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) \tag{2.1}$$

where c_p is the specific heat of the adobe, ρ is the mass density of the adobe, and k is the thermal conductivity of the adobe. For simplicity we consider $D = \frac{k}{c_p \rho}$, whose standard value is around 0.27 mm^2/s for adobe materials.

2.2.1 | One Dimensional Case

Consider a infinitesimally thick rod of length L on which we will numerically solve the heat equation. We can discretise space and time into Δx and Δt respectively. The left end of the rod ($x = 0$) will be our

[2]Source: timeanddate.com

external wall, and hence its temperature will change as a function of time, as discussed in the previous section. The right end ($x = L$) will be the inner wall.

We can use the finite difference method to solve this partial differential equation. This is also called the forward time-centered space (FTCS) method. The iterative finite difference formula for the heat equation (using the approximation for the first and second derivtives) would be:

$$\frac{T_{(t+1,\,x)} - T_{(t,\,x)}}{\Delta t} = D\frac{T_{(t,\,x+1)} - 2T_{(t,\,x)} + T_{(t,\,x-1)}}{(\Delta x)^2} \tag{2.2}$$

Here we are using the FTCS method because it the least numerically intensive. Also, as the external temperature changes as a function of time, so does our boundary conditions (especially on the left edge). FTCS allows us to have flexible boundary conditions as opposed to other methods like backward difference or the Crank-Nicholson method.

On the right edge we will assume Neumann boundary conditions $\left(\frac{\partial T}{\partial n} = 0\right)$ as the heat flows into the air inside the house. Mathematically this means $T_{x=N} = T_{x=N-1}$.

2.2.2 | Extension to higher dimensions

The same finite difference approach can be similarly extended to 2 or 3 dimensions. However here we make a key simplification. Assuming the heat only flows from the external surface to the internal surface and that no significant heat flows into the ground or the roof, we can consider $\frac{\partial T}{\partial z} = 0$. We have reduced the dimension of our problem by 1.

Now, consider that the wall has a fixed width W and thickness L. Here, we assume that the entire external surface is at the same temperature. This is obviously not true since the Sun's inclination throughout the day can lead to non-uniform heating. We also ignore the possibility of windows or other heat outlets from the house. So, if the assumption is true, the heat should spread through the wall uniformly; hence, we can further reduce the dimension of our problem by 1. To put it more precisely, if you assume the wall to be stacked with N layers horizontally, each layer will have the same temperature and no temperature flow will occur within a particular layer (Fig. 2.2).

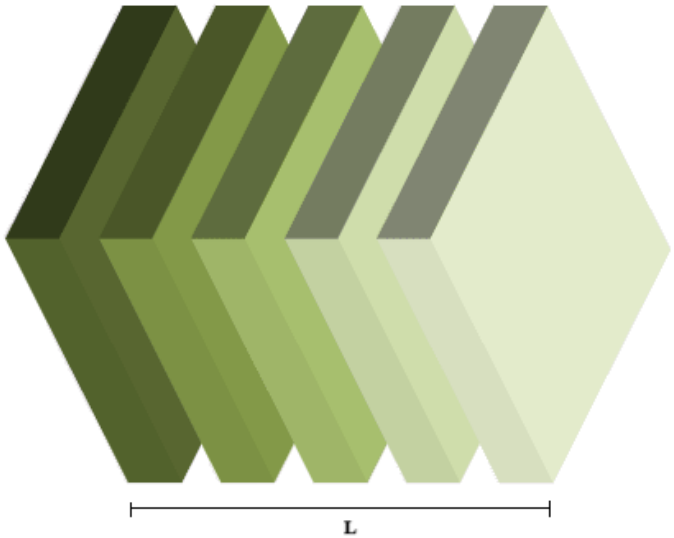


Figure 2.2: A model of a wall for demonstration, consisting of 5 layers, each with a fixed temperature

Hence, the optimum thickness obtained from our one-dimensional problem will be, at most, an upper limit if we relax some of the above assumptions (there are more heat sinks to consider).

2.3 | Implementation

The following Python code uses the `NUMBA` module, which generates optimized machine code to speed up the computation process.

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit, cuda

K0 = 273.15 # 0 Celsius in Kelvin

@numba.jit("f8(f8)", nopython=True, nogil=True, fastmath = True)
def external_temperature(sec):
    h = sec/3600
    h = h%24
    t = 15*np.sin(np.pi*h/12 + 3.9) + 24
    return K0 + t

### Defining the problem
alpha = 0.27 # mm^2/sec
days = 7
duration = 3600*24*days #seconds
nodes = 300 # space discretized into nodes

# initialize wall temperature as a gradient
# where initial inner wall is set at 25 degree Celsius
wall = np.linspace(external_temperature(0), 25+K0, nodes)

### Solving the heat equation
@numba.jit("(f8[:],f8,f8,f8)", nopython=True, nogil=True, fastmath = True, cache=True)
def solve_heat_eqn(init_state, duration, dt, dx):
    wall = init_state.copy()
    counter = 0
    inners = []

    while counter < duration :
        w = wall.copy()
        for i in range(1, nodes - 1):

            wall[i] = dt * alpha * (w[i - 1] - 2 * w[i] + w[i + 1]) / dx ** 2 + w[i]

        counter += dt
        wall[0] = external_temperature(counter)
        wall[-1] = wall[-2] # Neumann BC

        inners.append(wall[-1])

    return wall, np.array(inners)

# Solve heat equation for a particular wall thickness
def get_inner_temperatures(thickness):
    dx = thickness / (nodes-1)
    dt = 0.5 * dx**2 / alpha
    final, inners = solve_heat_eqn(wall, duration, dt, dx)
    return inners

variations = {}

# plotting $\Delta T$ vs thickness.
thicknesses = [T for T in range(170, 300, 10)]
for thickness in thicknesses:
    inner_temps = get_inner_temperatures(thickness)
    stable_region = int(len(inner_temps)/2)
    maxT = np.max(inner_temps[stable_region:])
    minT = np.min(inner_temps[stable_region:])
    variations[thickness] = maxT-minT

plt.figure(figsize=(9,7))
variations = dict(sorted(variations.items()))
plt.plot(variations.keys(), variations.values(), '-ko')
plt.ylabel(r'$\Delta T$ ($^\circ$C)')
plt.xlabel('Thickness (mm)')
```

Listing 2.1: Finding the closest possible emission line to match with from the database

2.4 | Results

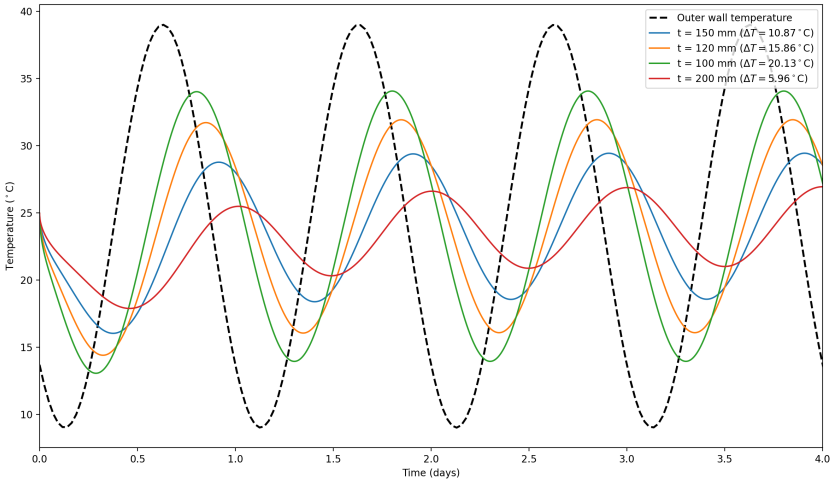


Figure 2.3: The inside temperature for different wall thicknesses compared to outside temperature. Note that as the thickness increases, not only does the temperature variation inside decrease, but also the time taken for the outside conditions to affect the inside of the wall increases.

The above two plots show the temperature of the inner wall as a function of time, with the temperature of the outer wall for comparison. We have given a few days to simulate the temperature variations to achieve a steady state.

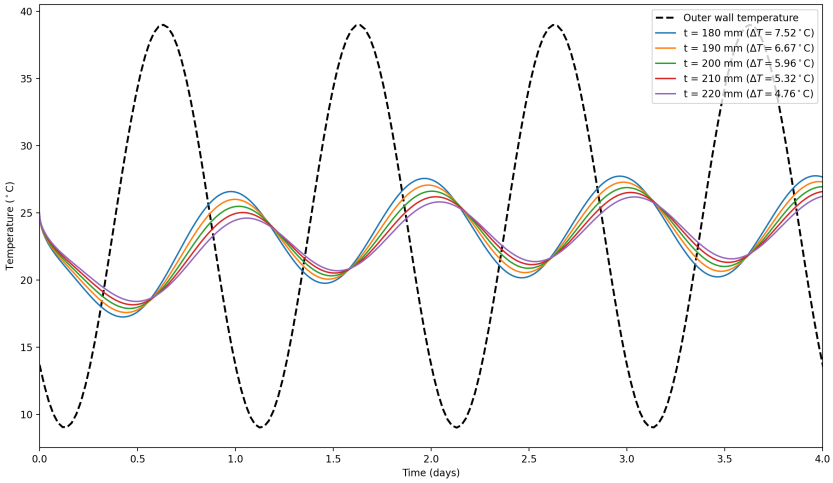


Figure 2.4: Similar to the above plot but for a higher value of wall thicknesses. The temperature variation becomes less and less extreme as the thickness increases.

Fig. 2.5 shows the temperature variation, ΔT, as a function of wall thickness. We can see the variation is not linear and slows down particularly after $\sim$ 200 mm (for Adobe).

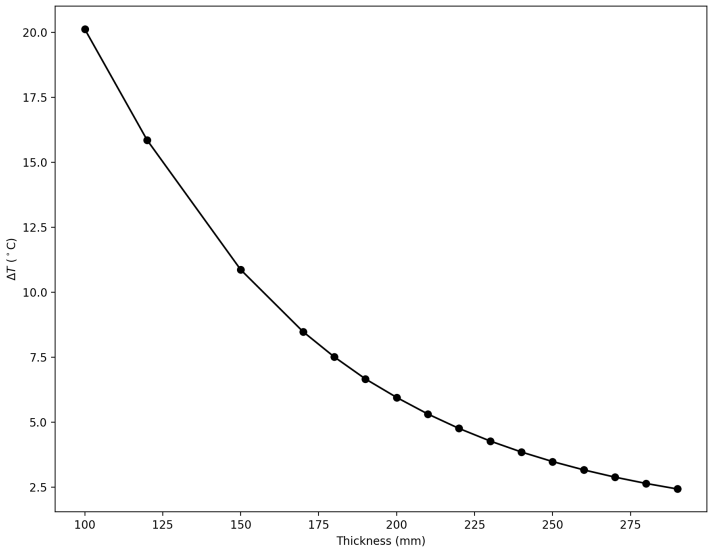


Figure 2.5: Final temperature variation ΔT vs wall thickness plot

2.5 | Discussion

According to England 2013, the comfortable range of temperature variation for human beings is around 3 to 4 °C (more precisely 20 to 24 °C). From Fig. 2.5, we can see that the temperature variations are in the comfortable range for wall thicknesses $\geq$ 220mm. So, to minimise the construction cost, 220mm would be the ideal wall thickness.

Due to the simplifications we made along the way, including reducing the dimensions by 2 and assuming uniform heating mean that the obtained value would be the minimum wall thickness required for adobe walls to remain at a comfortable temperature. Hence, this result broadly agrees with the values argued by Parra-Saldivar and Batty 2005, which are $\sim$ 340mm. (Note that the paper also takes into account multilayer walls and walls with windows).

3 | Edge Detection in Images

Edges in images are discontinuities in intensity. They usually represent the boundaries of objects or lighting present in the image. It is one of the first steps in object detection models.

Edges can be typically classified into Step, Ramp and Roof edges. A step edge represents an abrupt change in intensity, where the image intensity transitions from one value to another in a single step. A ramp edge describes a gradual transition in intensity over a certain distance rather than an abrupt change. A roof edge represents a peak or ridge in the intensity profile, where the intensity increases to a maximum and then decreases.

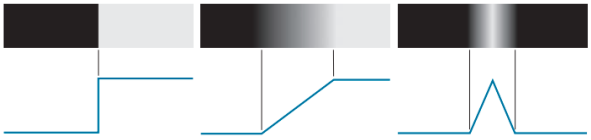


Figure 3.1: Types of edges found in an image. *Source: Digital Image Processing by R. C. Gonzalez & R. E. Woods*

Here, we will attempt to build an algorithm that can detect any of these edges using the principles of numerical differentiation.

3.1 | Algorithm and Theoretical Approach

The most straightforward algorithm for finding the edges in any image involves the following processes.

1. Conversion of an RGB image into a grayscale image. This is to flatten the 3-dimensional array into a 2-dimensional one to make it easier to work with. The standard formula for the conversion is

$$\text{Gray} = 0.3 \cdot \text{Red} + 0.59 \cdot \text{Green} + 0.11 \cdot \text{Blue} \tag{3.1}$$

   This formula closely represents the average person's relative perception of the brightness of red, green, and blue light.

2. Consider the greyscale image as a plot of a multivariable function $G(x, y)$ where the ordered pair (x, y) is the pixel location and the output $G(x, y)$ is the value of the grey scale at that point. Finding the gradient at each pixel in the grid grid represents the change in intensity at every pixel.

3. Fixing a threshold value (δ) and classifying all pixels with values $||\nabla G(x, y)|| > \delta$ as an edge.

3.1.1 | Gradient of a 2D Scalar Field

The gradient of a 2D scalar matrix will give us the overall change in intensity around every point. If G represents our scalar matrix, the gradient at any point (i, j) can be written as,

$$\nabla G \approx \left\langle \frac{G(x+1, y) - G(x-1, y)}{2}, \frac{G(x, y+1) - G(x, y-1)}{2} \right\rangle \tag{3.2}$$

where we used the central difference scheme for the first derivative with $h = 1$.

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} \tag{3.3}$$

However, pixels could be tightly packed in an image, and a point's immediate neighbours may not have enough contrast to truly detect edges. Furthermore, in Eq. 3.2, notice that we only use 4 of the 8 neighbors of the pixel (i, j). The algorithm could be even more precise if we could somehow include information about pixels further away than 1 pixel.

Using the Taylor series expansion,

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(\xi_3)h^3, \tag{3.4}$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(\xi_3')h^3 \tag{3.5}$$

$$\implies \frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{1}{3}f'''(x)h^2 + ... \tag{3.6}$$

By putting changing $h \to 2h$ and adding both the summations, we get (for $h = 2$)

$$f'(x) = \frac{8\left[f(x+1) - f(x-1)\right] - \left[f(x+2) - f(x-2)\right]}{12} \tag{3.7}$$

All the centred finite difference schemes will have an error of $O(h^2)$ associated with them.

One can apply a simple blur to the image before edge detection to reduce the fine noise by averaging every pixel value with its 8 neighbours.

3.1.2 | Second Derivatives

As we have seen in the previous section, the local maxima/minima in gradient values represent edge points. This means that at edge points, there will be a peak in the first derivative, and equivalently, there will be a zero crossing in the second derivative. Thus, edge points may be detected by finding the zero crossings of the second derivative of the image intensity.

The simplest way to find the second derivate of a scalar matrix is to find its Laplacian. Using the Taylor series expansion mentioned earlier, we can calculate the second derivate of a function as

$$f''(x) = f(x+h) + f(x-h) - 2f(x) \tag{3.8}$$

Thus, the two-dimensional Laplacian will be,

$$\nabla^2 G(x,y) = \frac{1}{h^2}\left(G(x,y+1) + G(x,y-1) + G(x-1,y) + G(x-1,y) - 4G(x,y)\right) \tag{3.9}$$

However, the problem with this approach is that even very small local peaks in the first derivative will result in zero crossings in the second derivative, making them extremely sensitive to noise.

The zero crossings can be found by multiplying a pixel value with its neighbour and checking if the product is < 0.

3.2 | Implementation

```
import numpy as np
import matplotlib.pyplot as plt

# RGB to grayscale conversion
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.3, 0.59, 0.11])

# Image blur
def blur(img):
    n = img.shape
    smooth = img.copy()
    # we ignore the edges as they are just 1 pixel
    for x in range(1, n[0]-1):
        for y in range(n[1]-1):
            smooth[x,y] = (img[x,y]+img[x-1,y]+img[x+1,y]+\
                           img[x,y-1]+img[x-1,y-1]+img[x+1,y-1]+\
                           img[x,y+1]+img[x-1,y+1]+img[x+1,y+1])/9
    return smooth

# A simple masking function for display
def mask(img, k=0.25):
    mk = np.where(img > k*np.max(img), 1, 0)
```

```
    return mk

# Gradient of G using the Nabla operator
def nablaG(G,x,y,h=1):
    if h == 1:
        delx = (G[x+1,y]-G[x-1,y])/2
        dely = (G[x,y+1]-G[x,y-1])/2
    elif h == 2:
        delx = (8*G[x+1,y]-G[x+2,y]-8*G[x-1,y]+G[x-2,y])/12
        dely = (8*G[x,y+1]-G[x,y+2]-8*G[x,y-1]+G[x,y-2])/12
    else:
        return (0, 0)
    return (delx, dely)

def gradient(img, h=1, file=False):
    grad = np.zeros((img.shape[0], img.shape[1]), dtype=float)
    for x in range(h, img.shape[0]-h):
        for y in range(h, img.shape[1]-h):
            g = nablaG(img, x, y, h=h)
            grad[x, y] = np.sqrt(g[0]**2+g[1]**2)
    return grad

# Calculate the Laplacian of a matrix
def laplacian(img, h=1):
    lap = np.zeros((img.shape[0], img.shape[1]), dtype=float)
    for x in range(h, img.shape[0]-h):
        for y in range(h, img.shape[1]-h):
            lap[x, y] = (img[x,y+1]+img[x,y-1]+img[x+1,y]+img[x-1,y]-4*img[x,y])/h**2
    return lap

# Find the zero crossings
def zero_crossings(img):
    mk = np.ones((img.shape[0],img.shape[1]))
    # zero crossings are given a value 0, the others 1.
    for x in range(1, img.shape[0]-1):
        for y in range(1, img.shape[1]-1):
            pix = img[x,y]
            if pix*img[x-1,y] < 0 or pix*img[x+1,y] < 0 or pix*img[x,y+1] < 0 or pix*img[
    x,y-1] < 0:
                mk[x,y] = 0
    return mk

# performing edge detection on a sample
img_rgb = mpimg.imread(f'data/pic.png')
img = rgb2gray(img_rgb)
grad = blur(gradient(img, h=2))
plt.figure()
plt.imshow(mask(grad, k=0.3), cmap = 'gray')
plt.show()
```

Listing 3.1: Code implementing edge detection techniques and related functions as discussed above

3.3 | Results

After calculating the corresponding gradient matrix, Figures 3.2 to 3.4 show edge detection performed on three different images using the first derivative approach (i.e. using the gradient matrix) compared with the industry standard *Canny* Edge Detection algorithm using `OpenCV`.

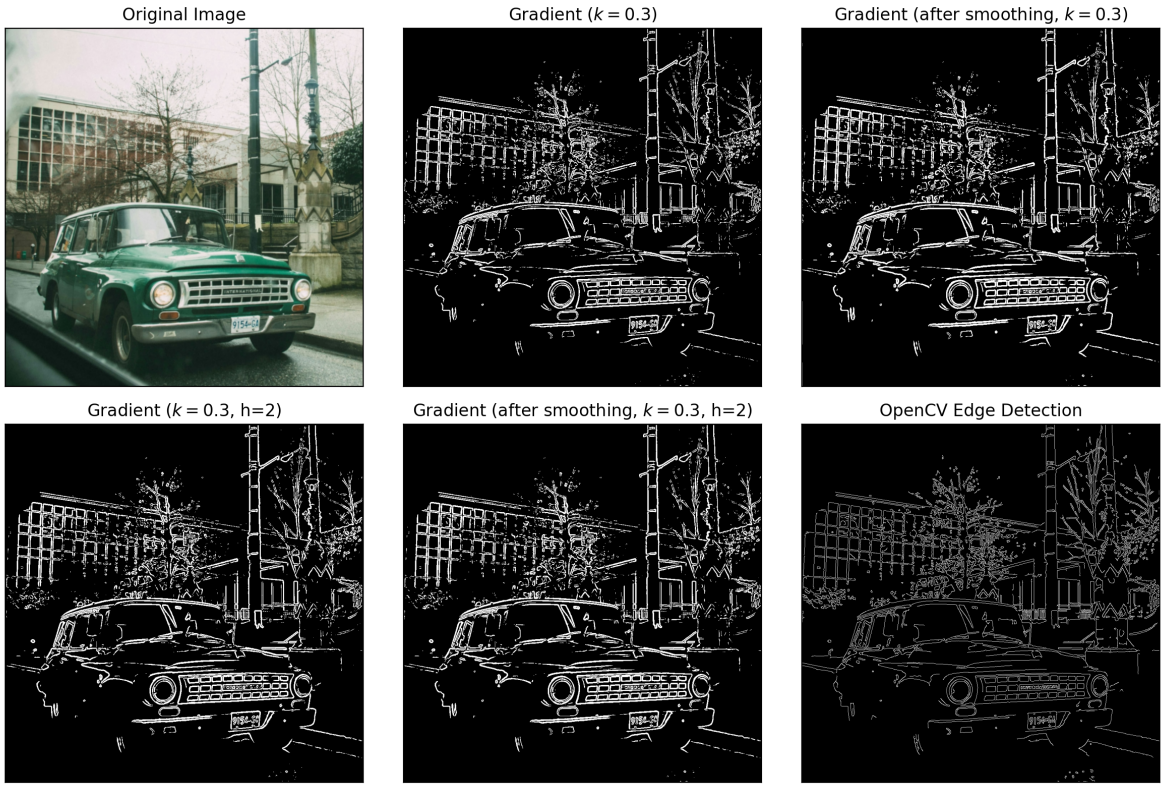


Figure 3.2: Edge detection results by altering various parameters as mentioned for an image

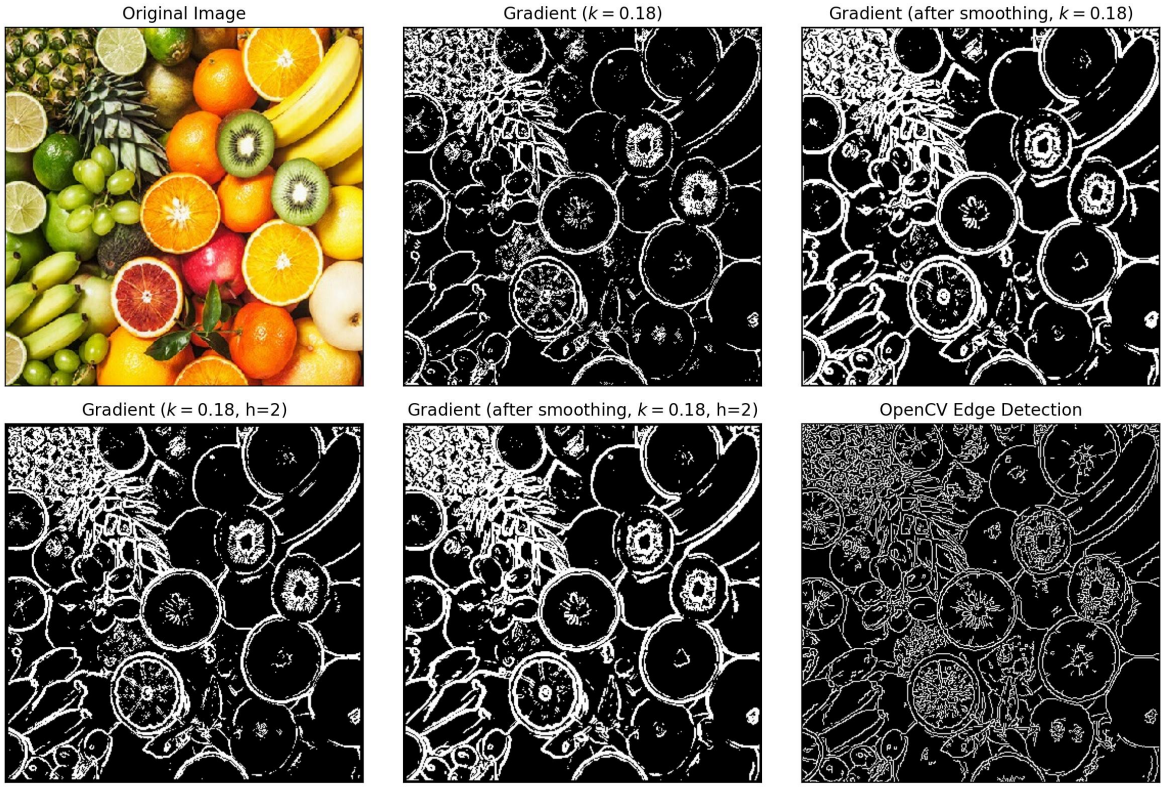


Figure 3.3: Edge detection results by altering various parameters as mentioned for an image

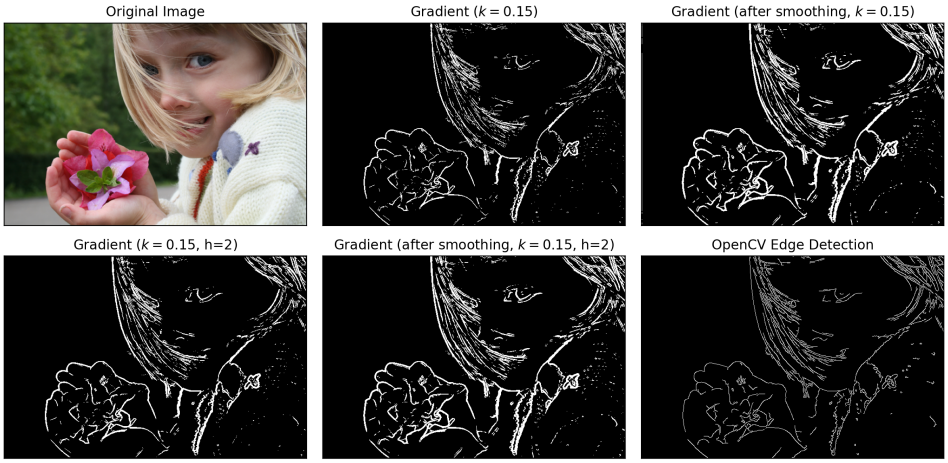


Figure 3.4: Edge detection results by altering various parameters as mentioned for an image

Figure 3.5 shows edge detection performed using the second derivative approach, i.e. finding the zero crossings of the Laplacian. These images were first passed through the `blur()` function.

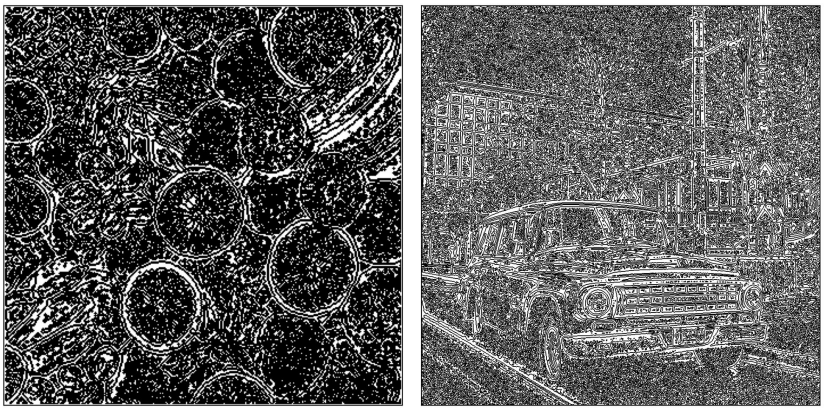


Figure 3.5: Zero crossings of the Laplacian for two of the images used above.

As you can see, these results are not as good as the ones we obtained with the first approach. This is due to the high amount of noise in the image creating a lot of local extremum points (even after smoothing), which get detected by the zero crossing algorithm.

3.4 | Discussion

In this project, we have explored different kinds of edge detection algorithms that use the principle of numerical differentiation. We have seen that the gradient approach (using first derivatives) works much better than the laplacian approach (using second derivatives) due to the high amount of local extremum points caused by noise. However, Laplacian is a very useful tool in blob detection and feature transformation algorithms, which are beyond the scope of this project.

We have also explored including more than the immediate neighbouring pixels in the calculation of derivatives. From the results, we can see that while this marginally reduces the noise in the edges detected, it also reduces the accuracy of the edges. A similar thing is seen when a blur filter is applied before the edge detection – the thickness of the edges increases.

The standard edge detection algorithm, on the other hand, tries to fix these shortcomings by calculating directional gradients separately and also using hysteresis to detect continuous edges.

Additionally, there are many ways to include 4 corner edges into our gradient. The *Sobel* and *Prewitt* operators are popular methods which use convolution filters to find the horizontal and vertical changes in intensity.

4 | Dam Integration

The USGS water data repository contains information about the water flow output of various dams in the USA. In this problem, given a year's worth of data on the flow rate of the dam (ft^3/sec) as a function of time, we are asked to calculate the total amount of water that has been discharged from the dam during that calendar year.

For this project, I have chosen the water flow data of a dam situated at Wolf River, Germantown, TN, USA[3] for the year 2023. The flow data plotted as a function of time is shown below.

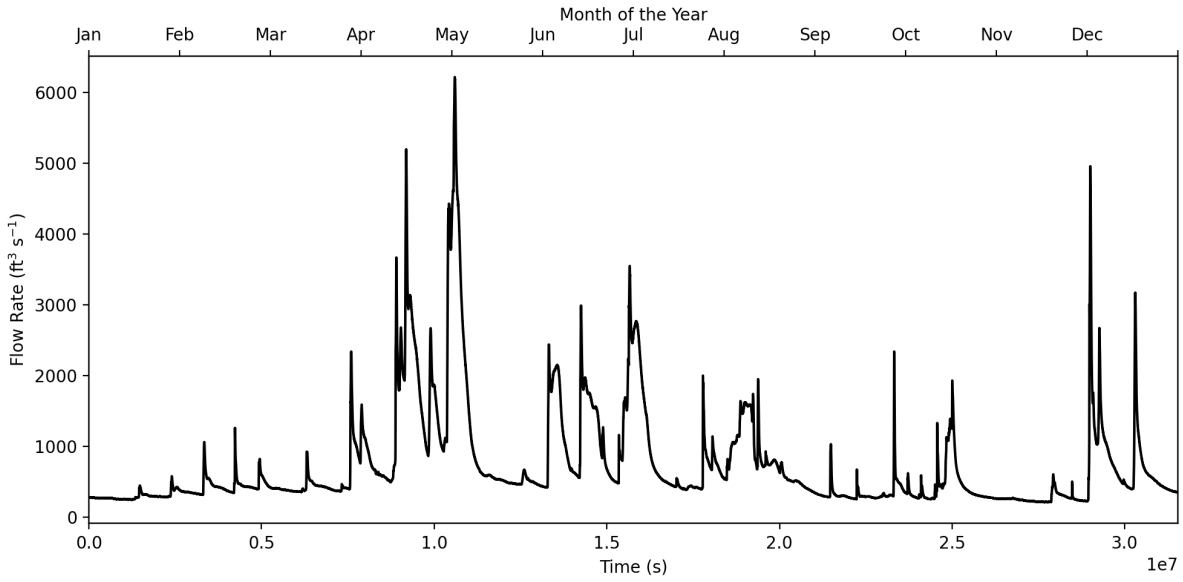


Figure 4.1: Flow-rate vs time along with the corresponding months for the year 2023

4.1 | Theoretical Approach

Let $f(t)$ be the flow-rate of the dam as a function of time. To calculate the total water discharged in a time interval from t_a to t_b, one can integrate $f(t)$ over that time.

$$\text{Water discharged } = \int_{t_a}^{t_b} f(t)dt \tag{4.1}$$

Let W be the total water discharged in the year. Therefore,

$$W = \int_{0}^{T} f(t)dt \tag{4.2}$$

where T is one year.

Our dataset consists of flow-rate of the dam at time stamps of 15 minutes. We have converted that to seconds to produce Fig. 4.1. Since $f(t)$ is a collection of samples and not a well-defined function, we must perform numerical integration to find W.

Let us discuss various ways of performing numerical integration of this discrete dataset.

4.1.1 | Mid-point Rule

The mid-point rule algorithm divides the integral into equal sub-intervals and approximates the f in each subinterval as $f(t)$. Then, it finds the area under the curve for each of those rectangular faces. This is one of the Newton-Cotes methods since it contains equally divided sub-intervals.

If h is the width of each sub-interval, the integral can be approximated as,

[3]Source: U.S. Geological Survey, Raw Data

$$w = \int_{t_a}^{t_b} f(t)dt \approx h[f(\xi_0) + f(\xi_1) + ... + f(\xi_{n-1})] \tag{4.3}$$

where there are n intervals and ξ_i is the mid-point of t_i and t_{i+1}.
We can use Taylor series expansion to find the error in the estimation. Consider a point t_i.

$$\begin{aligned} f(t) &= f(t_i) + (t - t_i)f'(\xi_i) \\ \int_{t_i}^{t_i+1} f(t)dt &= hf(t_i) + \frac{1}{2}h^2 f'(\xi_i) \quad [\because h = t_{i+1} - t_i] \\ \epsilon_i &= \frac{1}{2}h^2 f'(\xi_i) \end{aligned} \tag{4.4}$$

where ϵ_i is the absolute error in one single panel. For n many panels, we can find the total error as a sum of error contributions of all the panels.

$$\epsilon = \sum_{i=0}^{n-2} \frac{1}{2}h^2 f'(\xi_i) = \frac{n-1}{2}h^2 f'(\xi_i) = \frac{t_b - t_a}{2}hf'(\xi_i) \tag{4.5}$$

4.1.2 | Trapezoidal Rule

Another approximate method of calculating an integral is the trapezoidal (or trapezium) rule. Here, the area under the curve is approximated by a sum of n trapezia instead of rectangles.
For a single panel from t_a to t_b,

$$\int_{t_a}^{t_b} f(t)dt \approx \frac{t_b - t_a}{2}(f(t_b) + f(t_a)) \tag{4.6}$$

For n many panels,

$$\int_{t_a}^{t_b} f(t)dt \approx \frac{h}{2}(f(t_0) + 2f(t_1) + 2f(t_3) + ... + 2f(t_{n-1}) + f(t_n)) \tag{4.7}$$

All the points in the middle have double the weight because all of them get counted twice, once for the panel on the right and the other time for the panel on the left.

4.1.3 | Simpson's 1/3 Rule

We have seen that the trapezoidal method approximates the integrand by a straight line. One could argue that a better approximation can be obtained by approximating the integrand with an easily integrable nonlinear function. Simpson's 1/3 rule uses quadratic interpolation of data to do the same.
Consider three points t_1, t_2 and t_3, through which we have to interpolate a quadratic polynomial $p(t)$. It follows that

$$p(t) = \alpha + \beta(t - t_1) + \gamma(t - t_1)(t - t_2) \tag{4.8}$$

where α, β and γ are unknown constants evaluated from the such that polynomial passes through the points, $p(t_1) = f(t_1), p(t_2) = f(t_2)$ and $p(t_3) = f(t_3)$. These conditions yield

$$\alpha = f(t_1),\ \beta = \frac{f(t_2) - f(t_1)}{(t_2 - t_1)} \text{ and } \gamma = \frac{f(t_3) - 2f(t_2) - f(t_1)}{(t_3 - t_1)^2/2} \tag{4.9}$$

$$\implies \int_{t_1}^{t_3} f(t)dt \approx \int_{t_1}^{t_3} p(t)dt = \frac{(t_3 - t_1)/2}{3}[f(t_1) + 4f(t_2) + f(t_3)] \tag{4.10}$$

$$\tag{4.11}$$

where $(t_3 - t_1)/2 = h$. For a large number of panels, we can derive the composite Simpson's rule from the above equation as,

$$\int_{t_a}^{t_b} f(t)dt \approx \frac{h}{3}\left[f(t_a) + 4\sum_{i=2,4,6,..}^{n} f(t_i) + 2\sum_{i=3,5,7,..}^{n-1} f(t_i) + f(t_b)\right] \tag{4.12}$$

where $(t_b - t_a)/n = h$, n is the number of panels. The error in this estimation can be derived using Taylor series expansion as,

$$\epsilon_i = \frac{-1}{90}h^5 f^4(\xi_i) \implies \epsilon = \frac{-(t_b - t_a)}{180}h^4 f^4(\xi) \tag{4.13}$$

4.1.4 | Simpson's 3/8 Rule

This method uses cubic interpolation of data points to approximate the integrand. Since a third-order polynomial can be only determined from four points (i.e. three panels), the total integral for 1 panel is approximated to,

$$\int_{t_1}^{t_3} f(t)dt \approx \int_{t_1}^{t_3} p(t)dt = \frac{3(t_2 - t_1)}{8}[f(t_1) + 3f(t_2) + 3f(t_3) + f(t_4)] \tag{4.14}$$

Where the name 3/8 method comes from the 3/8 factor in the expression. For a large number of panels, we can derive composite Simpson's 3/8 rule as,

$$\int_{t_a}^{t_b} f(t)dt \approx \frac{3h}{8}\left[f(t_a) + 3\sum_{i=2,5,8,..}^{n-1}(f(t_i) + f(t_{i+1}) + 2\sum_{i=4,7,10,..}^{n-2} f(t_i) + f(t_b)\right] \tag{4.15}$$

where the interval is divided into n subintervals and n is a multiple of 3. h is the fixed time interval between two data points.

The error in this estimation can be derived using Taylor series expansion (Matthews 2004),

$$\epsilon_i = \frac{-3}{80}h^5 f^4(\xi_i) \implies \epsilon = \frac{-(t_b - t_a)}{80}h^4 f^4(\xi) \tag{4.16}$$

4.2 | Implementation and results

Before finding the total annual water discharge, let us first consider a small snippet of our data with a significantly sharp peak. We will first try to find the area under the curve in this domain.

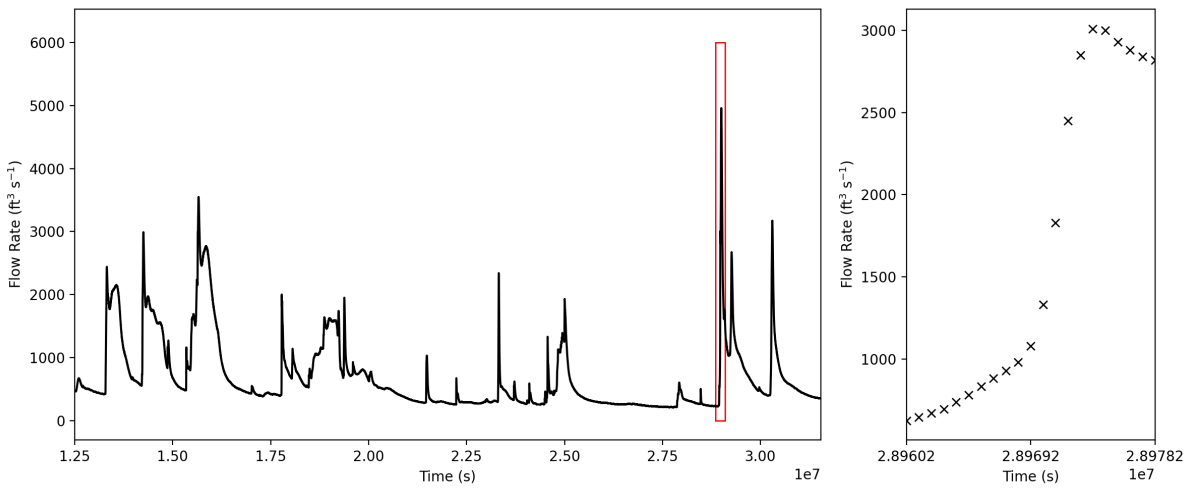


Figure 4.2: The right panel shows the zoomed-up version of the highlighted section with a very sharp peak we will be focusing on

Here we have found this particular region of interest using

```
roi = np.where((t>2.896e7) & (t<2.898e7))
f1 = flow[roi]
t1 = t[roi]
t1 = (t1-np.min(t1)) # changing the range of x from 0 to 1 for simplicity
t1 = t1/np.max(t1)
dx = t1[1]-t1[0]
```

Listing 4.1: Isolating the region of interest

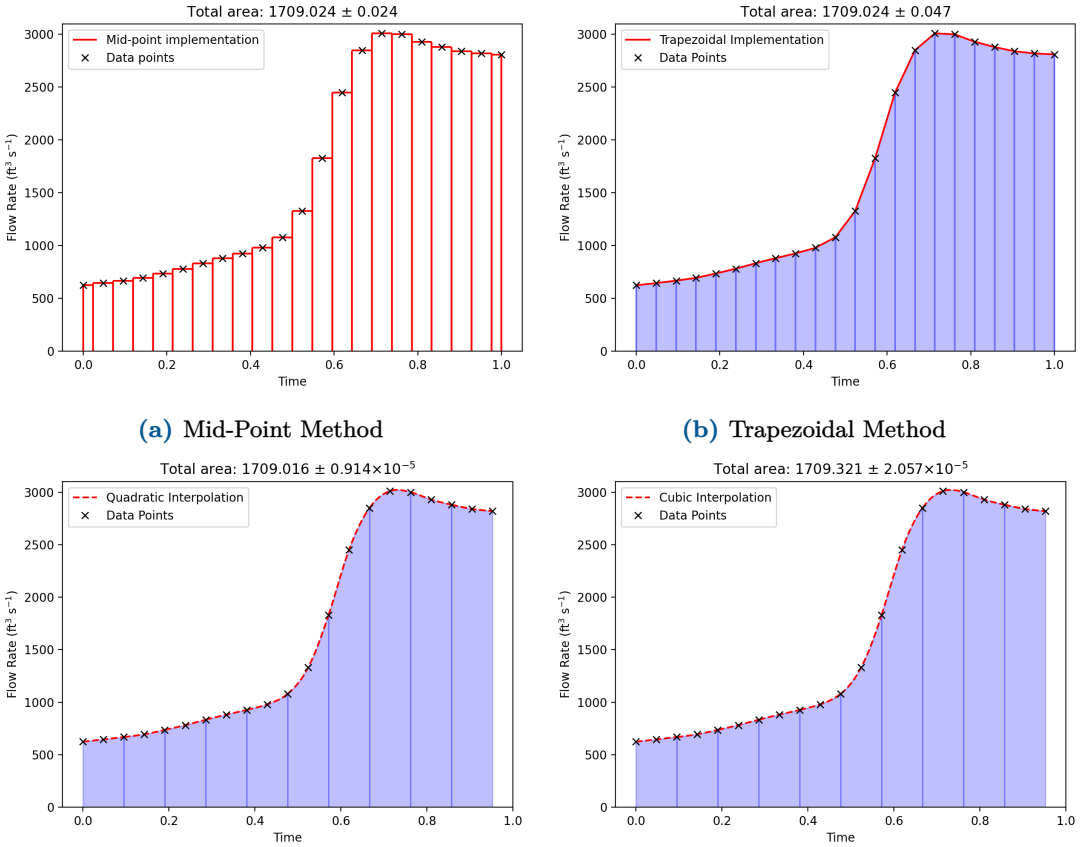
We have implemented the discussed integration methods in the following code snippet.

```
    import numpy as np
    import matplotlib.pyplot as plt
    from datetime import datetime
    from scipy.interpolate import interp1d

    """ Extract flow-rate information from the csv file"""
    dates, times, flow = np.loadtxt('project/data/water.csv', unpack = True, usecols =
    (2, 3, 5), dtype=object)
    flow = np.array(flow, dtype=float)
    t = np.zeros(dates.size)
    for i in range(dates.size):
        d = datetime.strptime(dates[i]+times[i], '%Y-%m-%d%H:%M')
        t[i] = d.timestamp()
    start_time = t[0]
    t -= start_time

    """ Calculate derivatives using central difference approach
    used in the calculation of error"""
    def d2dt(y,h=1):
        diffs = np.zeros(len(y))
        for x in range(2, len(y)-2):
            diffs[x] = (y[x+1]+y[x-1]-2*y[x])/(h**2)
        return diffs

    def d4dt(y,h=1):
        diffs = np.zeros(len(y))
        for x in range(2, len(y)-2):
            diffs[x] = (y[x+2]-4*y[x+1]+6*y[x]-4*y[x-1]+y[x-2])/h**4
        return diffs

    """ Integration functions """
    def mid_point_integration(x, y):
        half_step = (x[1]-x[0])/2
        res = 0
        res += y[0]*half_step
        for i in range(1,len(x)-1):
            res += y[i]*half_step*2
        res += y[-1]*half_step

        f2epsilon = np.max(abs(d2dt(y)))
        err = ((x[-1]-x[0]) * f2epsilon * dx**2)/(24)
        return res, err

    def trapezoidal_integration(x, y, dx):
        res = (dx/2) * (y[0] + 2*np.sum(y[1:-1]) + y[-1])
        f2epsilon = np.max(abs(d2dt(y)))
        err = ((x[-1]-x[0]) * f2epsilon * dx**2)/(12)
        return res, err

    def simpsons_13_integration(x, y, dx):
        # If there are an even number of samples, N, then there are an odd
        # number of intervals (N-1), but Simpson's rule requires an even number
        # of intervals. Hence we perform simpson's rule on the first and last (N-2)
        # intervals, take the average and add up the end points using trapezoidal rule
        if len(x) % 2 == 0:
            res = (dx/3) * (y[0] + 4*np.sum(y[1:-2:2]) + 2*np.sum(y[2:-2:2]) + y[-3])
            res += (dx/3) * (y[1] + 4*np.sum(y[2:-1:2]) + 2*np.sum(y[3:-1:2]) + y[-2])
            res /= 2
            res += 0.5*dx*(y[0] + y[-1])
        else:
            res = (dx/3) * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])
        f4epsilon = np.max(abs(d4dt(y)))
```

```
        err = ((x[-1]-x[0]) * f4epsilon * (dx**4))/(180)
        return res, err

    def simpsons_38_integration(x, y, dx):
        # If there are an N number of samples, then there are an (N-1)
        # number of intervals. Simpson's 3/8 rule requires an 3n number
        # of intervals. Hence in case of 3n-1 or 3n-2 intervals, we approximate the end
    points
        # similar to what we did for Simpson's 1/3 rule
        if len(x) % 3 == 0: # (n-1)%3 = 2
            res = y[0] + 3*(np.sum(y[1:-3:3])+np.sum(y[2:-3:3])) + 2*np.sum(y[3:-3:3])+ y
    [-3]
            res += y[1] + 3*(np.sum(y[2:-2:3])+np.sum(y[3:-2:3])) + 2*np.sum(y[4:-2:3])+
    y[-2]
            res += y[2] + 3*(np.sum(y[3:-1:3])+np.sum(y[4:-1:3])) + 2*np.sum(y[5:-1:3])+
    y[-1]
            res *= (3*dx/8) * (1/3)
            res += dx*(y[0] + y[-1])
        elif len(x) % 3 == 1: # (n-1)%3 = 0
            res = y[0] + 3*(np.sum(y[1:-1:3])+np.sum(y[2:-1:3])) + 2*np.sum(y[3:-1:3])+ y
    [-1]
            res *= (3*dx/8)
        elif len(x) % 3 == 2: #(n-1)%3 = 1
            res = y[0] + 3*(np.sum(y[1:-2:3])+np.sum(y[2:-2:3])) + 2*np.sum(y[3:-2:3])+ y
    [-2]
            res += y[1] + 3*(np.sum(y[2:-1:3])+np.sum(y[3:-1:3])) + 2*np.sum(y[4:-1:3])+
    y[-1]
            res *= (3*dx/8) * (1/2)
            res += 0.5*dx*(y[0] + y[-1])

        f4epsilon = np.max(abs(d4dt(y)))
        err = ((x[-1]-x[0]) * f4epsilon * (dx**4))/(80)
        return res, err

    """ Convert the domain to [0, 1] for simplicity in calculation of errors
    and then multiply by t_final (t_inital = 0)"""
    tmax = t[-1]-t[0]
    int_t = t/np.max(t)
    dx = int_t[1]-int_t[0]

    mid_point, max_error_mid_point = mid_point_integration(int_t, flow)
    trapezoidal, max_error_trapz = trapezoidal_integration(int_t, flow, dx)
    simpsons_13, max_error_simpsons13 = simpsons_13_integration(int_t, flow, dx)
    simpsons_38, max_error_simpsons38 = simpsons_38_integration(int_t, flow, dx)

    """ Multipy the integrand values by tmax and the error values by tmax*(dt)^2 and tmax
    *(dt)^4 for different integration methods
    as required. Here, since our data points are 15 minutes apart, dt is taken to be 900
    seconds."""
    print(f'Mid-point: {mid_point*tmax*1e-9:.3f} x 10^9 \pm {max_error_mid_point*tmax
    *(900**2)*1e-5:.3f} x 10^5 ft^3')
    print(f'Trapezoidal: {trapezoidal*tmax*1e-9:.3f} x 10^9 \pm {max_error_trapz*tmax
    *(900**2)*1e-5:.3f} x 10^5 ft^3')
    print(f'Simpsons 1/3: {simpsons_13*tmax*1e-9:.3f} x 10^9 \pm {max_error_simpsons13*
    tmax*(900**4):.3f} ft^3')
    print(f'Simpsons 3/8: {simpsons_38*tmax*1e-9:.3f} x 10^9 \pm {max_error_simpsons38*
    tmax*(900**4):.3f} ft^3')
```

Listing 4.2: Code implementing different integration techniques and their associated errors

Now, running our program on the region of interest, we can test the effectiveness of our code especially while integrating sharp features.



(a) Mid-Point Method **(b)** Trapezoidal Method

(c) Simpson's 1/3 rule with quadratic interpolation **(d)** Simpson's 3/8 rule with cubic interpolation

Figure 4.3: All mentioned integration techniques implemented for the small snippet with 22 data points.

4.3 | Discussion

We have employed 4 different methods to calculate the total water discharged annually. The results obtained are:

$$\begin{aligned}
\text{Using Mid-point method, } W &= (23.207 \times 10^9 \pm 2.167 \times 10^5)\ \text{ft}^3 \\
\text{Using Trapezoidal Rule, } W &= (23.207 \times 10^9 \pm 4.337 \times 10^5)\ \text{ft}^3 \\
\text{Using Simpson's 1/3 Rule, } W &= (23.207 \times 10^9 \pm 24.405)\ \text{ft}^3 \\
\text{Using Simpson's 3/8 Rule, } W &= (23.207 \times 10^9 \pm 54.910)\ \text{ft}^3
\end{aligned}$$

As we can see, the values obtained using different match quite well with each other, with varying levels of accuracy. Hence, we can say that our result is quite precise.

Since our flow-rate function is not a well-defined function, we cannot use methods like Gaussian or Lagrange quadrature, which are generally more precise. Here our abscissas are predetermined; hence, we can only use one of the Newton-Cotes methods.

5 | Galaxy Integration

A spectrum represents the intensity of light being emitted over a range of energies (i.e. frequencies). One can analyze the light from stars and galaxies using spectral gratings to study their features. A galaxy spectrum, in particular, will tell you about the types of stars the galaxy contains, the relative abundances of each type of star, and many more.

Galaxy spectra are typically characterized by a strong continuum component caused by the combination of a range of blackbody emitters spanning a range in temperature. However, stars are also surrounded by thin gas, which either emits or absorbs light at only a specific set of frequencies, causing spectral lines. Every chemical element produces a specific set of lines at fixed frequencies, so by identifying the lines, we can tell what types of atoms and molecules a star is made of. If the gas is cool, then it will absorb light at these wavelengths, and if the gas is hot, then it will emit light at these wavelengths.

The strength of each emission line (in W/m^2) is defined as the relative intensity of each peak across the associated frequencies. The problem at hand is to develop a process for analyzing galaxy spectra so as to determine the strength of each of the emission lines.

5.1 | Algorithm

The algorithm to analyse galaxy spectrum data to find the line strengths can be divided into two parts – (i) cleaning up raw data along with detection of emission lines and (ii) finding the relative strength of each emission line.

5.1.1 | Data Cleanup & Finding Emission Lines

To find emission lines, we will follow the following procedure. For demonstration, we will be working with spectra of NGC 1275.

Step 1: Continuum Subtraction
The continuum represents the sum of the blackbody radiation emitted by objects in the galaxy. To integrate the emission lines, we first need spectrum sans the continuum. The standard approach to this is to approximate a spectrum to a blackbody function using Chebyshev polynomials. However, since we are only dealing with a small section of the blackbody curve, we can roughly approximate it with a straight line. But depending on the spectrum, we can use different kinds of curve fittings.

```
def linregress(x, y):
    n = len(x)
    Sxx = x@x
    Sxy = x@y
    Sy = np.sum(y)
    Sx = np.sum(x)
    delta = n*Sxx - (Sx)**2
    slope = (n*Sxy - Sx*Sy) / delta
    intercept = (Sxx*Sy - Sx*Sxy) / delta
    return slope, intercept

plt.plot(frequency, intensity, 'k', label='Observed spectrum', alpha=0.5)

# using specutils package
spectrum = Spectrum1D(flux=intensity*u.Jy, spectral_axis=lambda_*u.angstrom)
g1_fit = fit_generic_continuum(spectrum)
y_continuum_2 = g1_fit(lambda_*u.angstrom)
flux = intensity/y_continuum_2
plt.plot(frequency, y_continuum_2, 'b--', label='Continuum fit using\nspecutils package')

# using least square fitting
slope, intercept = linregress(frequency, intensity)
y_continuum_1 = slope*frequency+intercept
plt.plot(frequency, y_continuum_1, 'r:', label='Continuum using Least-Square\nStraight
    line fit')

flux = intensity/y_continuum_1 # continuum subtracted intensity
```

Listing 5.1: Implementation of Linear Regression for straight line fitting compared with the standard specutils library generic continuum fit

Fig. 5.1 shows the straight line fit along with the Chebyshev polynomial fit (using **astropy**'s **specutils** package) on top of the observed spectrum.

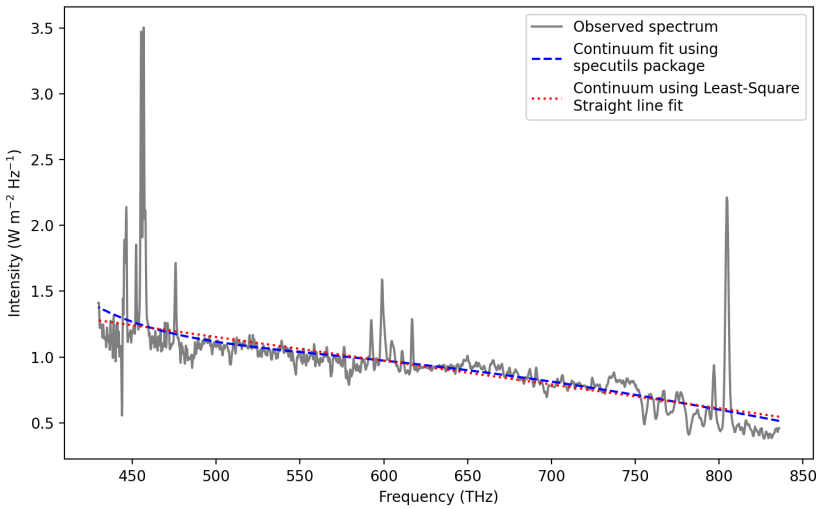


Figure 5.1: Continuum fit of the galaxy spectrum

The continuum subtracted spectrum is obtained by dividing the original spectrum by the fitted continuum. This approach preserves the features of the spectrum better than simple subtraction of the continuum from the original.

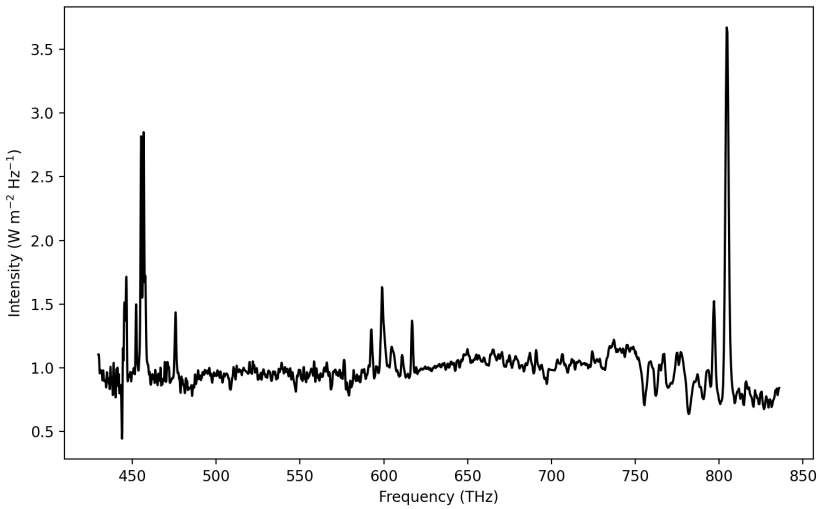


Figure 5.2: Continuum subtracted galaxy spectrum

Step 2: Smoothening the Spectrum
The background noise can be greatly reduced by applying a median filter over the spectrum. This makes line finding easier and less prone to error due to random spikes in the spectrum. The median filter acts by acting on each point,

$$f[i] = \text{median}(f[i-1], f[i], f[i+1]) \tag{5.1}$$

We can also increase the kernel of this median filter by including more neighbouring points in the above equation. However, that has the disadvantage of possibly distorting emission line strengths.

```
def medfiltt(x, k=3):
    n = len(x)
    xs = np.zeros(n)
```

```
    for i in range(n):
        a = i-k if i >= k else 0
        b = i+k if i <= n-k else -1
        xs[i] = np.median(x[a:b])
    return xs

flux_smooth = medfiltt(flux)
```

Listing 5.2: Implementation of median filter with kernel k

Step 3: Finding emission lines

The simplest method to find emission lines is to calculate the derivative at every point and classify the ones that are above a certain threshold value. However, due to the high amount of noise in the spectrum, we need to include as many neighbouring points as possible in calculating the first derivative.

Eqn. 3.3 is the first-order approximation of the first derivative using the central difference method, with an order $O(h^2)$ error. Using Richardson Extrapolation, one can increase the accuracy of this formula. Here, we have performed Richardson Extrapolation thrice to arrive at the 4th-order approximation of the first derivative.

Richardson extrapolation is given by,

$$G = \frac{2^p g(h/2) - g(h)}{2^p - 1} + O(h^{p+q}) \tag{5.2}$$

where G is the quantity we are after and $g(h)$ is the approximate quantity using a step size h. p and q represent the order of the leading error term and the increment in the order for the error terms after that, respectively. Using $p = 2$ and $q = 2$ in Eqn. 3.3,

$$f'(x) \approx \frac{8(f(x+h) - f(x-h)) - f(x+2h) + f(x-2h)}{12h} + O(h^4) \tag{5.3}$$

Again, if we perform the same procedure two more times,

$$f'(x) \approx \frac{15(f(x+h) - f(x-h)) - 9((f(x+2h) + f(x-2h)) + ((f(x+3h) + f(x-3h))}{12h} + O(h^6) \tag{5.4}$$

$$f'(x) \approx \frac{4}{5}(f(x+h) - f(x-h)) - \frac{1}{5}(f(x+2h) - f(x-2h)) + \frac{4}{105}(f(x+3h) - f(x-3h)) - \frac{1}{280}(f(x+4h) - f(x-4h)) + O(h^8) \tag{5.5}$$

```
def dydx(y):
    dy = np.zeros(len(y))
    for i in range(4, len(y)-4):
        dy[i] = (-y[i+4] + (4*280/105)*y[i+3] - 56*y[i+2] + 224*y[i+1] - 224*y[i-1] + 56*
    y[i-2] - (4*280/105)*y[i-3] + y[i+4])/280

    return dy
```

Listing 5.3: Implementation of the 4th order approximation of the first derivative

The inclusion of more neighbouring points in the derivative effectively minimises the effect of random spikes of noise in the data.

Practically, the peaks were found by setting a threshold value for the first derivative along with a threshold value for the continuum subtracted spectrum. In order to avoid really closely spaced peaks, we have also implemented a simple for loop to remove any peaks within $\sim$ 10 THz of each other.

```
threshold = 0.080
flux_smooth = medfiltt(flux, 1)
dy4 = dydx(flux_smooth, 4)
line_fs = frequency[np.where((dy4>threshold) & (flux_smooth>1.3))]
realines = []
i = 0
while i < len(line_fs):
    realines.append(line_fs[i])
    for j in range(i+1, len(line_fs)):
```

```
        if abs(line_fs[i] - line_fs[j]) <= 10:
            i += 1
    i += 1
```

Listing 5.4: Finding the peaks using a threshold value set manually

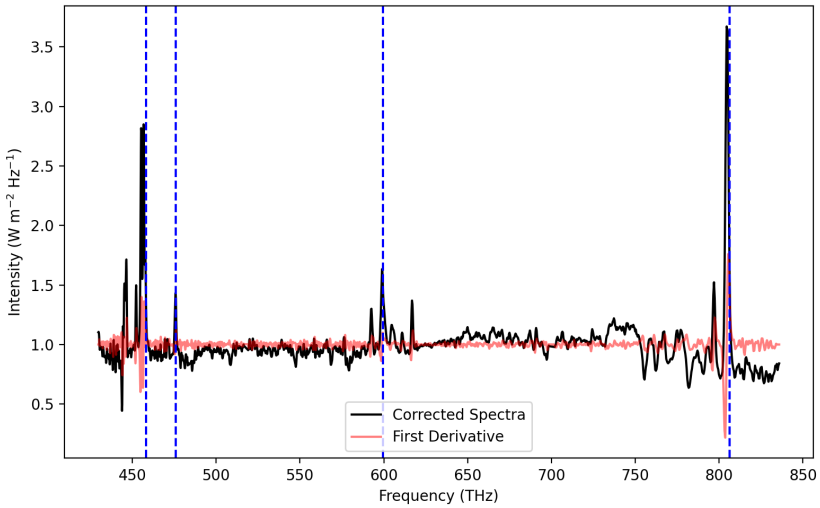


Figure 5.3: Major emission peaks detected for our observed spectrum. The red line indicates the first derivative of the continuum subtracted spectrum.

5.1.2 | Finding Line Strengths

Now, let us examine each peak closely.

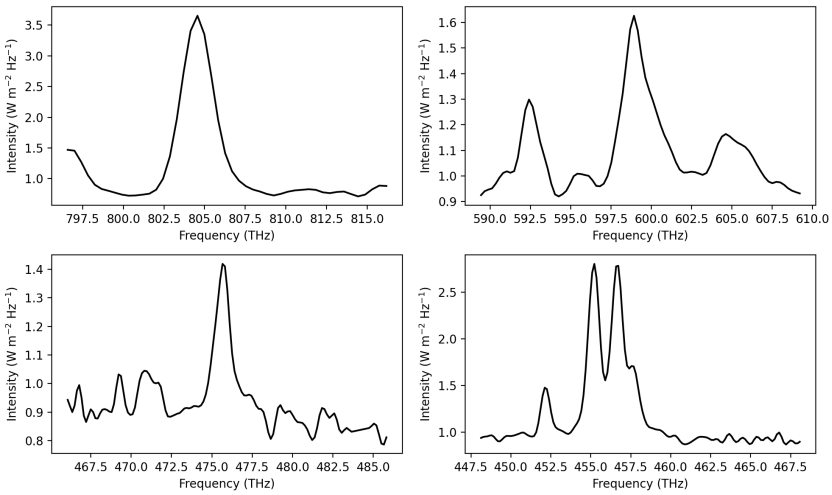


Figure 5.4: Zoomed in version of the detected peaks

The strength of each emission line is given by integrating its intensity with respect to frequency (in W/m^2).

$$\text{Line Strength} = \int_{\nu_0}^{\nu_1} I(\nu) d\nu \tag{5.6}$$

However, given the noisy nature of our spectrum, numerical integration has a high degree of error associated with it. Not to mention that one has to manually figure out the integration limits in the above equation for every peak.

A more precise way to find the line strength is by considering each emission peak as a Gaussian distribution. Now, by fitting a Gaussian function on each peak window, we can determine the line strength as the area under a Gaussian distribution, given by

$$\text{for any } g(\nu) = Ae^{-\frac{(\nu-\nu_0)^2}{2\sigma^2}}, \int_{-\infty}^{\infty} g(\nu)d\nu = A\sigma\sqrt{2\pi} \tag{5.7}$$

By using `scipy.curve_fit()`, we have fit Gaussian functions over every curve using our initial estimate.

```
from scipy.optimize import curve_fit

def gaussian(x, amplitude=1, mean=0, stddev=1):
    y = amplitude*np.exp(-((x-mean)**2)/(2*stddev))
    return y

for line in realines:
    window = np.where((frequency > line-10) & (frequency < line+10))
    xs = frequency[window]
    ys = flux_smooth[window] - 1

    plt.plot(xs, ys, 'k', alpha=1, label='Corrected Spectra')

    params, cov = curve_fit(gaussian, xs, ys, p0=[np.max(ys), line, 5])
    fit_x = np.linspace(xs[-1], xs[0], 200)
    fit_y = gaussian(fit_x, params[0], params[1], params[2])

    strength = params[0]*np.sqrt(params[2]*2*np.pi)
    strength_err = strength*np.sqrt(cov[0][0] + cov[2][2]/4)

    plt.plot(fit_x, fit_y, 'b-', label=f'Best fit Gaussian', alpha=0.5)
    plt.title(f'f = {params[1]:.2f} THz, Line Strength = ({strength:.3f} '+ r"$\pm$"+ f'
    {strength_err:.3f}) W/m'+r"$^{2}$", fontsize=10)
    final_lines[params[1]] = (strength, strength_err)

    plt.ylabel('Intensity (W m$^{-2}$ Hz$^{-1}$)')
    plt.xlabel('Frequency (THz)')
```

Listing 5.5: Finding the peaks using a threshold value set manually

The error in the estimation of A and σ from the covariance matrix is used to find the error in the line strength given by,

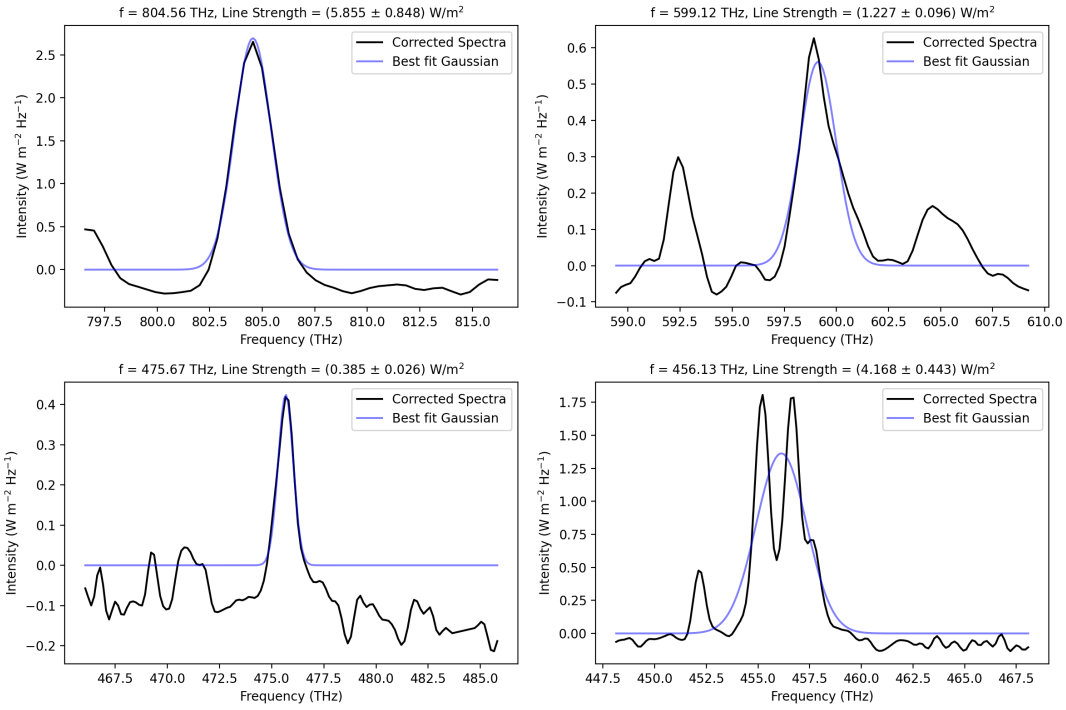


Figure 5.5: Gaussian function fitted on each emission line

The results are shown below.

$$\frac{\Delta\text{Line Strength}}{\text{Line Strength}} = \sqrt{\left(\frac{\Delta\sigma}{\sigma}\right)^2 + \left(\frac{\Delta A}{A}\right)^2} \tag{5.8}$$

As we can see here, the fourth panel consists of a double peak which cannot be estimated by a single Gaussian function. In such cases, we can manually change the estimation window by cutting off one of the peaks to estimate the other.

```
line1 = 455
line2 = 457

window = np.where((frequency > line-10) & (frequency < line+10))
xs = frequency[window]
ys = flux_smooth[window] - 1
plt.plot(xs, ys, 'k', alpha=1, label='Corrected Spectra')

# define the first window by cutting of the second peak
win1 = np.where(xs < line1+0.8)
params1, cov1 = curve_fit(gaussian, xs[win1], ys[win1], p0=[1.75, line1, 0.1])
win2 = np.where((xs > line2-1) & (xs < line2+0.4))
params2, cov2 = curve_fit(gaussian, xs[win2], ys[win2], p0=[2, line2, 0.5])

# define the second window by cutting of the first peak
fit_x = np.linspace(xs[-1], xs[0], 200)
fit_y1 = gaussian(fit_x, params1[0], params1[1], params1[2])
fit_y2 = gaussian(fit_x, params2[0], params2[1], params2[2])

# final curve fit is the addition of the two emission lines
plt.plot(fit_x, fit_y1+fit_y2, 'b-', label='Best fit Gaussian', alpha=0.5)
print('Area under fit1:', params1[0]*np.sqrt(params1[2]*2*np.pi))
print('Area under fit2:', params2[0]*np.sqrt(params2[2]*2*np.pi))
strength1 = params1[0]*np.sqrt(params1[2]*2*np.pi)
strength2 = params1[0]*np.sqrt(params2[2]*2*np.pi)
strength1_err = strength1*np.sqrt(cov1[0][0] + cov1[2][2]/4)
strength2_err = strength2*np.sqrt(cov2[0][0] + cov2[2][2]/4)

final_lines[params1[1]] = (strength1, strength1_err)
final_lines[params2[1]] = (strength2, strength2_err)
```

Listing 5.6: Dealing with double peaks

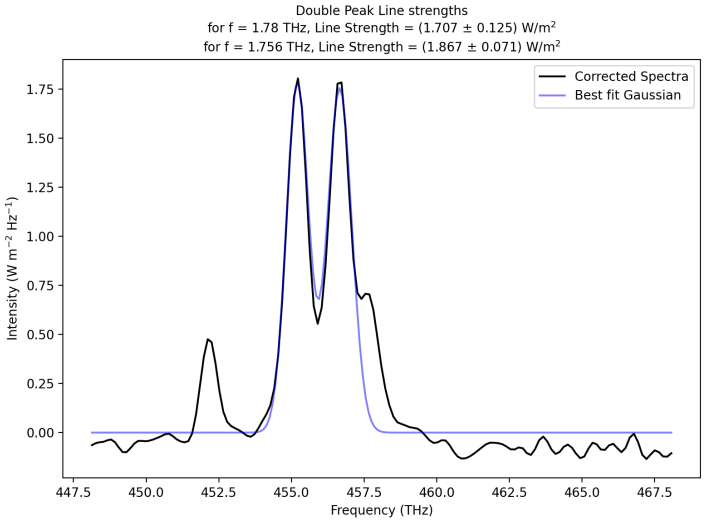


Figure 5.6: The double peak fitting

5.1.3 | Chemical Correspondence of Emission Lines

Now that we have all the emission peaks along with their respective line strengths (and their error bars), all we have to do is to match them against the literature data to find the corresponding ion/molecule causing

the emission. Here, we have used the Galaxy Emission Line database developed by Drew Chojnowski[4].

```
import pandas as pd

data = pd.read_csv('galaxy_emission_lines.csv')
print('Emission Lines Found:\n')

for line, strength in final_lines.items():
    line_w = f2w(line)
    ion = data.iloc[(data['lambda']-line_w).abs().argsort()[0]]['Ion']
    print(f'f = {line:.2f} THz ({line_w:.1f} AA), possibly due to {ion}\n  Line Strength
    = ({strength[0]:.3f} \pm {strength_err:.3f}) W/m^2')
```

Listing 5.7: Finding the closest possible emission line to match with from the database

5.2 | Results

The results are summarised in table 5.1 and Fig. 5.7

Emission frequency (THz)	Wavelength (nm)	Line Strength (W/m^2)	Species
804.56	372.62	(5.855 ± 0.861)	O II
599.12	500.39	(1.227 ± 0.106)	O III
475.67	630.25	(0.385 ± 0.035)	O I
455.20	658.59	(1.707 ± 0.203)	N II
456.66	656.49	(1.867 ± 0.110)	Hα

Table 5.1: Emission line result summary for galaxy NGC1275

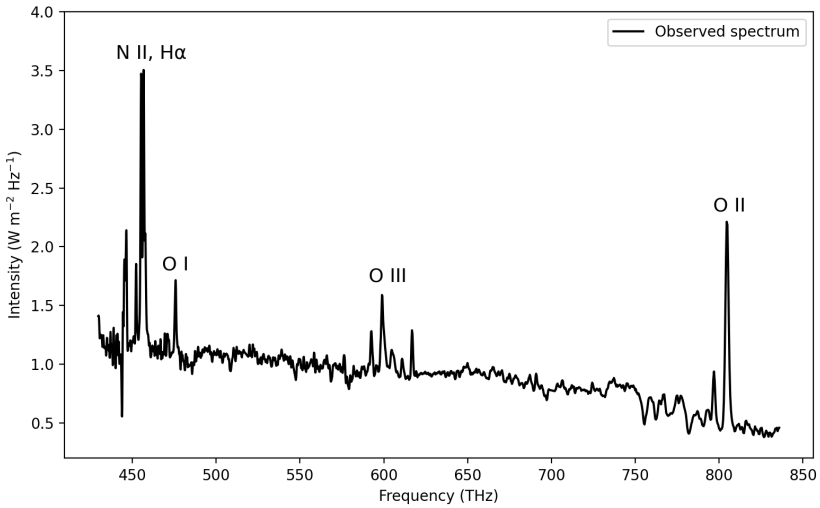


Figure 5.7: Emission lines detected for galaxy NGC1275

Note that by relaxing the threshold conditions, one can possibly find a few more low-strength emission lines. However, the accuracy will be low due to the high amount of noise in the spectrum.

Hence, we have successfully obtained line strengths and their corresponding chemical origin for for different spectral lines in the spectrum of galaxy NGC1275.

[4] Source.

References

England, Public Health (2013). *Minimum home temperature thresholds for health in winter - A systematic literature review*. Tech. rep.

Gezerlis, Alex (July 2023). *Numerical Methods in Physics with Python*. Cambridge University Press.

Jain, Ramesh, Rangachar Kasturi, and Brian G. Schunck (Jan. 1995). *Machine Vision*. McGraw-Hill Science, Engineering and Mathematics.

Kiusalaas, Jaan (Jan. 2013). *Numerical Methods in Engineering with Python 3*. Cambridge University Press.

Matthews, John H (2004). *Simpson's 3/8 rule for Numerical Integration*.

Parra-Saldivar, M. Luisa and William Batty (Sept. 2005). "Thermal behaviour of adobe constructions". In: *Building and Environment* 41.12. DOI: 10.1016/j.buildenv.2005.07.021.

Sullivan, Eric (Sept. 2020). *Numerical Methods: An Inquiry Based Approach with Python*.

Walker, Benjamin J. et al. (Oct. 2023). "VisualPDE: Rapid Interactive Simulations of Partial Differential equations". In: *Bulletin of Mathematical Biology* 85.11. DOI: 10.1007/s11538-023-01218-4.