PROBLEM SET 3: DISSUSION AND ADVECTION
DUE: 3 NOV. 2020

GEO2300: FYSISKE PROSESSER I GEOFAG

SIGURD SANDVOLL SUNDBERG

1. Problem 1: The Diffusive Climate Model.

We want to consider a diffusive climate model, where we will divide the earth into small slices along the latitude lines. Each section is 1° tall, and goes around the entire Earth. We essentially slice the Earth like an apple, in small slices, and assume that the temperature in each slice is the same.

1.1. **a).** Starting of with a simple climate model discussed in the compendium[1], and expanding it with a diffusive term, so our governing equation is

$$\frac{\partial}{\partial t}T = \frac{1}{\rho c_p H}\left(\frac{S}{4}(1-\alpha) - \epsilon\sigma T^4\right) + \frac{D}{R_E^2}\frac{\partial^2}{\partial\theta^2}T \tag{1}$$

where we have the following constants

- $\rho = 1\,[kg/m^3]$,
- $c_p = 1004\,[J/(K\,kg)]$,
- $H = 10^4\,m$,
- $S/4 = 350\cos(\theta) + 150\,[W/m^2]$,
- $\alpha = 0.7 - 0.45\cos(\theta)\,[-]$,
- $\sigma = 5.67\times10^{-8}\,[W/(m^2K)]$,
- $\epsilon = 0.6\,[-]$,
- $R_E = 6.37\times10^6\,[m]$,
- $d\theta = \pi/180$.

Writing out equation 1, using the FTCS scheme we have

$$\frac{T_j^{n+1} - T_j^n}{dt} = \frac{1}{\rho c_p H}\left(\frac{S_j}{4}(1-\alpha) - \epsilon\sigma\left(T_j^n\right)^4\right) + \frac{D}{R_E^2}\left(\frac{T_{j+1}^n - 2T_j^n + T_j^n}{d\theta^2}\right) \tag{2}$$

$$T_j^{n+1} = \frac{dt}{\rho c_p H}\left(\frac{S_j}{4}(1-\alpha)\right) - \frac{\epsilon\sigma dt}{\rho c_p H}\left(T_j^n\right)^4 + \frac{Ddt}{R_E^2 d\theta^2}\left(T_{j+1}^n - 2T_j^n + T_j^n\right) + T_j^n \quad s = (dtD)/(R_E^2 d\theta^2) \tag{3}$$

$$T_j^{n+1} = sT_{j+1}^n + (1-2s)T_j^n + sT_{j-1}^n + \frac{dt}{\rho c_p H}\left(\frac{S_j}{4}(1-\alpha)\right) - \frac{\epsilon\sigma dt}{\rho c_p H}\left(T_j^n\right)^4 \tag{4}$$

which can be rewritten on the follow form

$$T^{n+1} = \mathbf{A}T^n + F \tag{5}$$

where we have

$$F = q_1\left(\frac{S_j}{4}(1-\alpha)\right) - q_2\left(T_j^n\right)^4 \tag{6}$$

with $q_1 = \frac{dt}{\rho c_p H}$ and $q_2 = \frac{\epsilon\sigma dt}{\rho c_p H}$.

Date: November 2, 2020.

Matrix **A** is given by the following terms

$$sT_{j+1}^n + (1-2)T_j^n + sT_{j-1}^n. \tag{7}$$

Using that we have no flux at the poles, we have Neumann boundary conditions given by

$$\frac{\partial}{\partial\theta}T = 0, \quad T_0 = T_{-1} \wedge T_{n+1} = T_n \tag{8}$$

Studying the boundaries of our problem we have the following

$$T_0 = sT_1 + (1-2s)T_0 + sT_0 = sT_1 + (1-s)T_0 \tag{9}$$

$$\vdots \tag{10}$$

$$T_n = sT_n + (1-2s)T_n + sT_{n-1} = (1-s)T_n + sT_{n-1} \tag{11}$$

Our matrix **A** then looks as follows

$$\mathbf{A} = \begin{bmatrix} 1-s & s & 0 & \dots & \dots \\ s & 1-2s & s & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & 0 & s & 1-2s & s \\ \dots & \dots & 0 & s & 1-s \end{bmatrix} \tag{12}$$

Converting the FTCS scheme to Python code, could result in the following script

```python
def S(t):
    return 350 * np.cos(t) + 150  # W/m^2


def albedo(a, b, t):
    return a - b * np.cos(t)


def T_analytical(S, a, e):
    return (S*(1-a)/(e*sigma))**(1/4)


def setupArrays(a, b, n, t):
    # Setup of S_j and a_j
    S_j = np.zeros(n-1)
    a_j = np.zeros(n-1)
    for i in range(n-1):
        S_j[i] = S(t[i])
        a_j[i] = albedo(a, b, t[i])
    return S_j, a_j


def setupMatrixFTCS(n, s):
    # Setup the matrix A
    A = np.zeros([n-1, n-1])
    A[0, 0] = 1 - s
    A[0, 1] = s
    A[n-2, n-2] = 1 - s
    A[n-2, n-3] = s

    for i in range(1, n-2):
        A[i, i] = 1 - 2*s
        A[i, i - 1] = s
        A[i, i + 1] = s
    return A


def solverFTCS(A, n, rho, e, tEnd, Sj, aj, dt):
    # Forcing constant term
    F = q * Sj * (1-aj)*dt

    t = 0
    T = np.zeros(n-1)
    while t < tEnd:
        t += dt
        T = np.dot(A, T) + F - Q*(T**4) * dt
    return T


""" Following is the variable definitions both fixed and non fixed"""
# Variables
epsilon = 0.6  # Emmisivity
s = 0.4
n = 182  # Integration points

Pole_albedo = 0.7  # Albedo at the poles as solution to a - b*cos(
    theta).
Ecvator_albedo = 0.45  # Set to get the albedo at evcator

# Semi fixed variables
startThe = -np.pi/2  # start position in radian
endThe = np.pi/2  # end position in radian
dtheta = np.pi / 180  # theta[1] - theta[0] Steplength in radian
t_end = 140 * 10 * day_to_second  # 10*T_rad in seconds, end time

# Fixed variables
rho = 1  # kg/m^3
heat_cap = 1004  # j/kgK heat cap of air
H = 1e4  # m Height of the atmosphere
sigma = 5.67e-8  # W/m^2k
radiusE = 6.37e6  # m

# Defining fixed constants for the rest of the problem.
q = 1 / (rho * heat_cap * H)
Q = (epsilon * sigma) / (rho * heat_cap * H)

"""Following the the code that performs the calculation and creates
    the plots"""
if __name__ in "__main__":
    # Setup the problem
    theta = np.linspace(startThe, endThe, n-1)
    Sj, aj = setupArrays(Pole_albedo, Ecvator_albedo, n, theta)
    A = setupMatrixFTCS(n, s)

    Dlist = [1e6, 5e6, 1e7, 5e7]  # m^2/s
    # Dlist = [1e6]  # m^2/s
    for D in Dlist:
        # Setup non-constant constants
        dt = s * (radiusE * radiusE * dtheta * dtheta)/D  # time step

        # Solve the problem
        T = solverFTCS(A, n, rho, epsilon, t_end, Sj, aj, dt)

        P = {1e6: r"$1\times 10^{6}$", 5e6: r"$5\times 10^{6}$",
             1e7: r"$1\times 10^{7}$", 5e7: r"$5\times 10^{7}$"}
        plt.plot(np.rad2deg(theta), (T-273), '--',
                 label=rf"Plot for diffusivity D = {P[D]:s} $[m^2/s]$"
    )

    AnaT = T_analytical(Sj, aj, epsilon)
    plt.plot(np.rad2deg(theta), (AnaT-273), 'k',
```

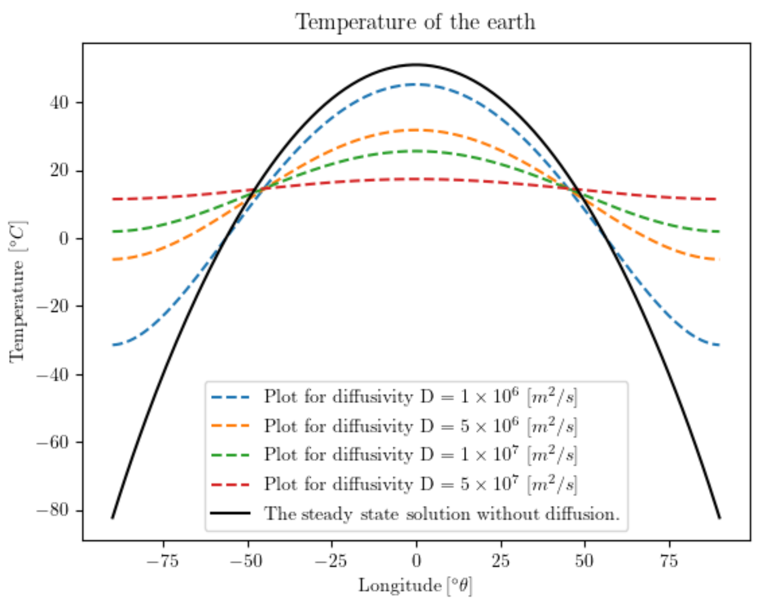

FIGURE 1. The case with no diffusivity, plotted against cases with diffusivity. Along the y-axis the temperature is shown and along the x-axis, longitude is shown. On the edges of the plot, we have the north and the south pole, whilst in the middle is the equator. Seen is that as diffusivity increases the difference between cold and warm areas diminish, and if diffusivity becomes high enough, there would be no visible difference.

```
                    label=r"The steady state solution without diffusion.")
```

Imports and plotting labels are left out.

Studying a model where we use different values for D, namely $D = \{1 \times 10^6, 5 \times 10^6, 1 \times 10^7, 5 \times 10^7, \}[m^2/s]$ and compare our results against the case without diffusivity, where we choose dt such that $s = 0.4$ and use a final time of 10 times $T_{\text{rad}} = 140\text{days}$[1] to achieve a steady state solution. Seen in figure 1 we have the four cases with increasing diffusivity, where the steady state solutions minimum and maximum value, gets closer the higher diffusivity becomes. This means that the temperature at the poles would be closer to that of the equator the higher diffusivity becomes.

Whilst in the zero diffusivity case, we have large difference in temperature between the poles and equator, in the highest diffusivity case we have almost the same temperature at the poles as we do at equator.

1.2. **b).** Considering different climate model where we look at how changes in the emissivity ϵ of the atmosphere changes the global temperature. As a baseline case we will use the diffusive climate model with $D = 10^7$ m/s. We want to find out how much we would need to change the emissivity to achieve a global average temperature increase of 5° Celsius.

We can adapt our program from part a), with only adding the following small modifications

```
if __name__ in "__main__":
    # Setup the problem
    theta = np.linspace(startThe, endThe, n-1)
    Sj, aj = setupArrays(Pole_albedo, Ecvator_albedo, n, theta)
    A = setupMatrixFTCS(n, s)

    D = 1e7  # m^2/s
    de = 0.002

    # Setup non-constant constants
    dt = s * (radiusE * radiusE * dtheta * dtheta)/D  # time step

    # Base value
    T = solverFTCS(A, n, rho, epsilon, t_end, Sj, aj, dt)
    c_TB = T-273.15

    while True:
        # epsilon -= de
        epsilon = 0.56  # This is for testing reasons.
        T = solverFTCS(A, n, rho, epsilon, t_end, Sj, aj, dt)
        c_T = T - 273.15
        difference = abs(np.mean(c_TB - c_T))
        if difference > 5:
            print(
                f"To achieve a 5 degree average temperature change the
 emmisivity needs to be: {epsilon:.3f}")
            break

    # Find whether the transformation is uniform or not
    uniform = abs(c_TB - c_T)
    mm = np.max(uniform) - np.min(uniform)
    print(f"The difference in maximum and minimum tempratures is {mm
:.3f}")
    plt.plot(uniform)
```

The rest of the function definitions are the same as for the a), only modifications the main part of the script was needed to adapt our code.

From this we can extract the following results

```
To achieve a 5 degree average temperature change the emmisivity
  needs to be: 0.560
The difference in maximum and minimum tempratures is 0.099
```

From which we can see, that to achieve a global warming of 5° Celsius, we would need to change the emissivity from 0.58 to 0.56. Also we have that the difference between the minimum and maximum values of the difference between the case with 0.58 and 0.56 emissivity, is only 0.1° Celsius. This is supported by figure 2, where we see a minimal difference between the equator and the poles. So we can fairly confidently that the increase in global temperature follows a uniform increase.

1.3. **c).** Going back to the diffusive climate model with emissivity 0.58, we want to study what a change in the albedo would result in. We will consider a case where the ice on the poles disappeared in the future, and see how this change would affect the global climate. Firstly the ice on the north pole, would result in more open water being visible to the Sun, and water has a low albedo. For the south pole we would have visible land which again have a higher albedo than ice.

We will adapt the albedo, to represent this, however not with accurate values. We will still use a small difference in albedo between the poles and equator, but much lower than our base case. From figure 3, we see the changed albedo, and see how much the temperature increase when we change the albedo. This is due to the angle at which the light hits the Earth surface at the poles compared to the equator, at a higher angle more light will be reflected, thus the albedo should be

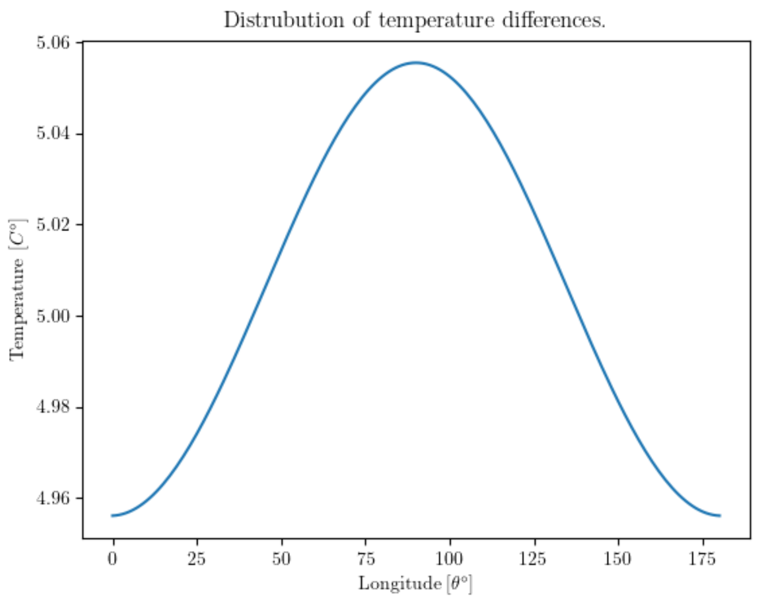

FIGURE 2. Difference between the emissivity 0.58 and 0.56 in degrees Celsius. We see that the transform seems fairly uniform, with minimal difference between the edges. It has the same shape as our non-diffusivity case, and represents a large increase, though minimal, at the equator compared to the poles.

higher at the poles even without ice. Even though we have the same albedo at the equator, we see a large increase in the equator temperature. This comes from the fact that diffusivity tries to average the global temperature, and as it is warmer at the poles, we would have less warm air would need to move from the equator to the poles to average the global temperature. We can see the change in temperature compared to our base case in figure 4, which resembles the non-diffusivity case for our climate model. We see that increase in temperature is greatest at the pole due to a large change in the albedo.

We can again reuse the same code as in a) with the following changes

```python
if __name__ in "__main__":
    # Setup the problem
    theta = np.linspace(startThe, endThe, n-1)
    A = setupMatrixFTCS(n, s)

    # Normal Albedo
    nPole_albedo = 0.7  # Albedo at the poles as solution to a - b*cos
    (theta).
    nEcvator_albedo = 0.45  # Set to get the albedo at evcator
    nSj, naj = setupArrays(nPole_albedo, nEcvator_albedo, n, theta)

    # Changed albedo
    Pole_albedo = 0.35  # Albedo at the poles as solution to a - b*cos
    (theta).
    Ecvator_albedo = 0.1  # Set to get the albedo at evcator
    Sj, aj = setupArrays(Pole_albedo, Ecvator_albedo, n, theta)

    D = 1e7  # m^2/s
```

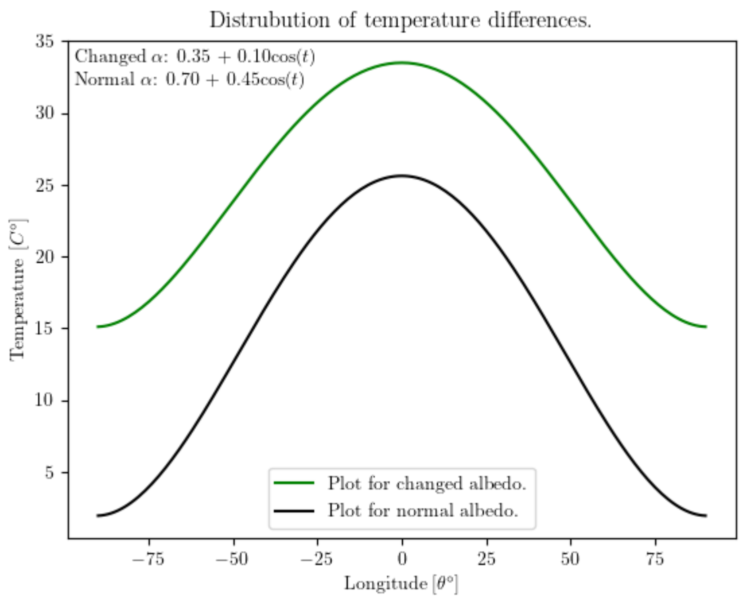

FIGURE 3. The base case, plotted against a case where we have less ice on the poles. Values are not accurate representation of actual albedo that would come from poles without ice, however this represents a case where the albedo at equator is the same, and less at the poles.

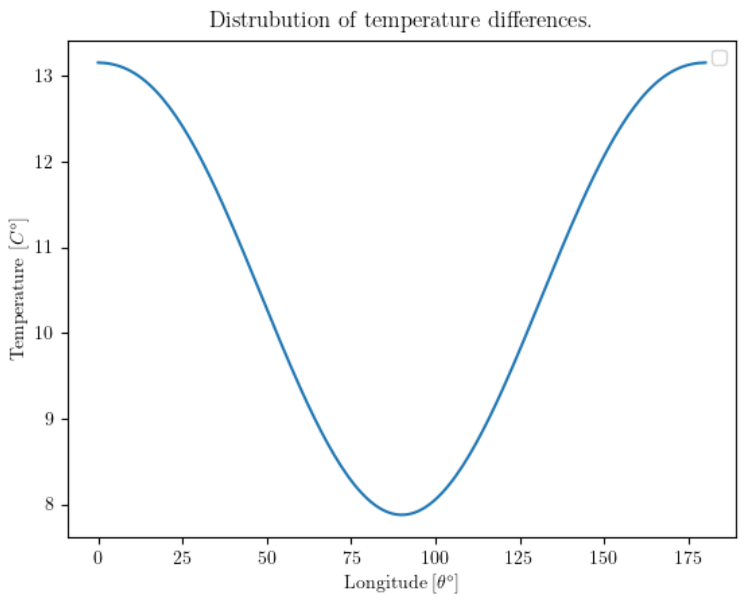

FIGURE 4. The change in temperature from the base case. We see a large increase in temperature at the poles and a smaller one at the equator. This is resembles the inverse of the non-diffusivity case for our climate model.

```
    # Setup non-constant constants
    dt = s * (radiusE * radiusE * dtheta * dtheta)/D  # time step

    # Base value
    T = solverFTCS(A, n, rho, epsilon, t_end, Sj, aj, dt)

    plt.plot(np.rad2deg(theta), (T-273), 'g',
             label=rf"Plot for changed albedo.")

    nT = solverFTCS(A, n, rho, epsilon, t_end, nSj, naj, dt)
    plt.plot(np.rad2deg(theta), (nT-273), 'k',
             label=rf"Plot for normal albedo.")

    c_nT = nT - 273.15
    c_T = T - 273.15
    uniform = abs(c_nT - c_T)
    mm = np.max(uniform) - np.min(uniform)
    print(f"The difference in maximum and minimum tempratures is {mm
    :.3f}")
```

From this we get that the difference between the maximum and minimum values of the global temperature change is

```
The difference in maximum and minimum tempratures is 5.278
```

from figure 4. In other words the temperature at the poles increase by roughly 5 degrees Celsius more at the poles compared to the equator.

1.4. **d).** Another way we could study this climate model is to look at how a different numerical method reproduces the results. Or looking at stability of two different algorithms, perform when we change the variable $s = (dtD)/(R_E^2 d\theta^2)$, which dictates our chosen value for dt.

The Crank-Nicholson(CN) scheme, is a combination FTCS scheme, and its implicit version iFTCS. The iFTCS scheme looks as follows

$$-sT_{j+1}^{n+1} + (1+2s)T_j^{n+1} - sT_{j-1}^{n+1} = \frac{dt}{\rho c_p H}\left(\frac{S_j}{4}(1-\alpha)\right) - \frac{\epsilon\sigma dt}{\rho c_p H}\left(T_j^n\right)^4 \tag{13}$$

keeping our forcing terms from equation 4 the same, thus only changing the diffusivity terms. This is due to us having no good way of representing $(T^{n+1})^4$.

Our CN scheme then becomes

$$-\frac{s}{2}T_{j+1}^{n+1}+(1+s)T_j^{n+1}-\frac{s}{2}T_{j-1}^{n+1} = -\frac{s}{2}T_{j+1}^{n+1}+(1-s)T_j^{n+1}+\frac{s}{2}T_{j-1}^{n}+\frac{dt}{\rho c_p H}\left(\frac{S_j}{4}(1-\alpha)\right)-\frac{\epsilon\sigma dt}{\rho c_p H}\left(T_j^n\right)^4 \tag{14}$$

which we can rewrite to

$$\mathbf{B}T^{n+1} = \mathbf{A}T^n + F \tag{15}$$

where we define $\mathbf{A}$ and $\mathbf{B}$ the same as in equation 12.

Studying the boundary conditions of our problem which are the same as earlier, we have

$$-\frac{s}{2}T_1^{n+1} + (1+s)T_0^{n+1} - \frac{s}{2}T_0^{n+1} = sT_1^n + (1-s)T_0^n - sT_0^n \tag{16}$$

$$\longrightarrow -\frac{s}{2}T_1^{n+1} + (1+\frac{s}{2})T_0^{n+1} = sT_1^n + (1-\frac{s}{2})T_0^n \tag{17}$$

$$\vdots \tag{18}$$

$$-\frac{s}{2}T_{j+1}^{n+1} + (1+s)T_j^{n+1} - \frac{s}{2}T_{j-1}^{n+1} = sT_{j+1}^n + (1-s)T_j^n - sT_{j-1}^n \tag{19}$$

$$\longrightarrow (1+\frac{s}{2})T_j^{n+1} - \frac{s}{2}T_{j-1}^{n+1} = (1-\frac{s}{2})T_j^n + sT_{j-1}^n \tag{20}$$

To implement the CN scheme, we can reuse almost all parts of our code in a), except we would need to define matrix **B** and redefine matrix **A** and find the matrix $B^{-1}A$ to solve $T^{n+1} = \mathbf{B}^{-1}\mathbf{A}T^n + F$.

The changes needed are then a new setup for the CN scheme. Below are the two added functions and we run the program for s = 4 and s = 0.4. As usual plotting is left out.

```python
def setupMatrixCN(n, s):
    # Setup the matrix A
    A = np.zeros([n-1, n-1])
    A[0, 0] = 1 - s/2
    A[0, 1] = s/2
    A[n-2, n-2] = 1 - s/2
    A[n-2, n-3] = s/2

    B = np.zeros([n-1, n-1])
    B[0, 0] = s/2 + 1
    B[0, 1] = -s/2
    B[n-2, n-2] = s/2 + 1
    B[n-2, n-3] = -s/2

    for i in range(1, n-2):
        A[i, i] = 1 - s
        A[i, i - 1] = s/2
        A[i, i + 1] = s/2
        B[i, i] = 1 + s
        B[i, i - 1] = - s/2
        B[i, i + 1] = - s/2
    return np.matmul(np.linalg.inv(B), A)
```

```python
def solverCN(s, n, rho, e, tEnd, Sj, aj, dt):
    A = setupMatrixCN(n, s)
    # Forcing constant term
    F = q * Sj * (1-aj)*dt

    t = 0
    T = np.zeros(n-1)
    while t < tEnd:
        t += dt
        T = np.dot(A, T) + F - Q*(T**4) * dt
    return T
```

And the solver for FTCS and CN are called as follow

```python
T_ftcs = solverFTCS(s, n, rho, epsilon, t_end, Sj, aj, dt)

T_cn = solverCN(s, n, rho, epsilon, t_end, Sj, aj, dt)
```

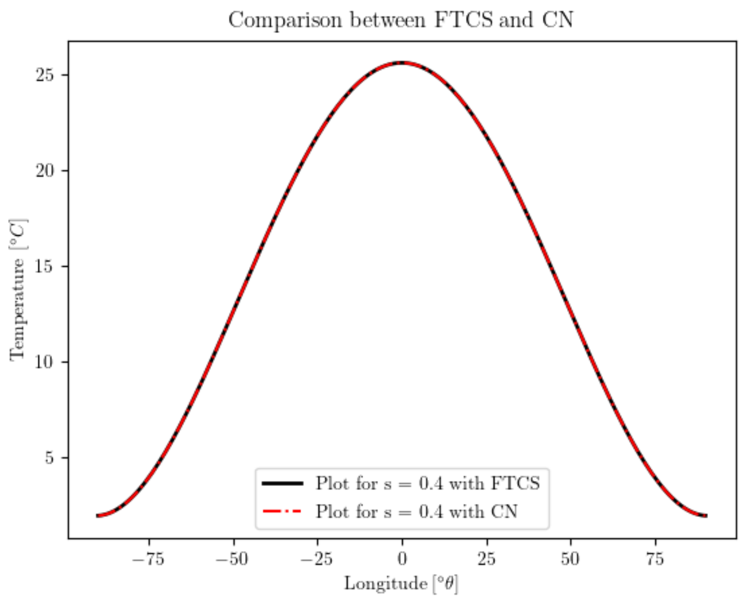

Figure 5. The FTCS scheme compared to CN scheme, plotting for s = 0.4. We see that the two scheme produces solutions which are visibly no different from each other. Both scheme seem stable, and have little to no variance.

Studying the results of this, we seen in figure 5 that there is no visible difference between the CN and FTCS schemes for s = 0.4. Meaning that for certain choices of s, for the FTCS this would be an $s < 0.5$, we can choose either scheme, and they would produce almost equal solutions. So as long as the FTCS scheme is stable, it is the preferred method as it is simpler to setup.

When we look at s = 4, we can see from figure 6 that, for either value of s, the CN scheme, produces solution with no visible difference. That means that we can choose a large value for s, for which the FTCS scheme is not stable, and in turn have to do fewer iterations, to achieve the same result. Now iterations does not speak for total run time as the number of floating point operations are also needed to be taken into account. Mainly for constructing the matrices here, however this is beneficial either way. So for long term simulations, using the CN scheme, outperforms the FTCS scheme, as fewer iterations are required for the same results.

Also to keep in mind, when trying our specific case for s = 4, using the FTCS scheme, we end up with a run time warning due to an overflow. Meaning that the FTCS scheme is blowing up and causing overflows for numerical values. This is not desirable, and we need to swap scheme, as we saw, the CN scheme performs perfectly here. This can also be seen from the Neumann stability analysis[2], where $s < 0.5$, for FTCS to be stable.

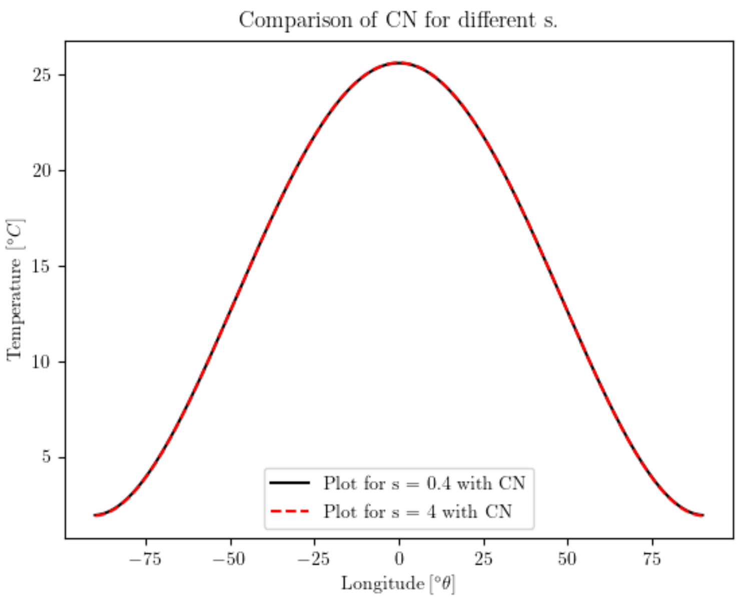

FIGURE 6. The CN scheme compared to itself for s = 4 and s = 0.4. We have no visible difference between the two choices of s.

1.5. **e).**

2. Problem 2: Woman Overboard, Without and With Eddies.

We want to study how a body moves after falling into a river with and without eddies. We will look at this as a probability problem where we look at the probability of finding the person at position x at time t after initially falling into the river.

2.1. **a).** What we know initially is that the woman, falls into the river about 100 m downstream from a dam. We are not sure exactly where she falls into the water, but know it is an area with a radius of 10 m. We assume the river flows at a constant velocity of 50 cm/s.

To find the probability of where she fell into the river, we will describe the probability density function(PDF) as a Gaussian function. Thus it takes on the form

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{21}$$

where σ is the standard deviation of 10 m and μ is the mean value of 100 m at $t = 0$. Inserting these values we get the equation of the probability to find the woman at $t = 0$, given any position x. This equation reads

$$p(x) = \frac{1}{10\sqrt{2\pi}} \exp\left(-\frac{(x-100)^2}{200}\right). \tag{22}$$

2.2. **b).** As we are currently considering a case where we do not have any eddies in the river, the motion of the woman is only described by the advection due to the water flow.

The general advection equation reads

$$\frac{\partial}{\partial t}\phi + u\frac{\partial}{\partial x}\phi = 0 \tag{23}$$

where ϕ describes the property we are studying, in this case the probability of finding a woman in the river, and u is the velocity of the river as we are only considering a one dimensional case.

Using the method of characteristics, we know a given solution to equation 23 can be expressed at $f(x-ut)$, where x is the position, u is the velocity and t is the time step. Applying this to equation 22, we find that the analytical expression for the womans positions in time is given by

$$p(x-ut) = \frac{1}{10\sqrt{2\pi}}\exp\left(-\frac{(x-u*t-100)^2}{200}\right), \tag{24}$$

inserted for $u = 0.5\text{m/s}$ we find

$$p(x-ut) = \frac{1}{10\sqrt{2\pi}}\exp\left(-\frac{(x-0.5*t-100)^2}{200}\right). \tag{25}$$

which describes the probability to find the woman at position x at time t from the initial state.

2.3. **c).** We are interested in studying this problem numerically through expanding the advection equation, from a continuous problem to a discretized problem which we can evaluate numerically.

Starting with the simplest numerical scheme, we consider the forward in time, centered in space(FTCS) scheme. Writing our the advection equation, using p instead of ϕ, we have

$$\frac{\partial}{\partial t}p + u\frac{\partial}{\partial x}p = 0 \tag{26}$$

$$\frac{p_i^{n+1}-p_i^n}{\Delta t} + u\frac{p_{i+1}^n - p_{i-1}^n}{2\Delta x} = 0 \tag{27}$$

$$\frac{p_i^{n+1}-p_i^n}{\Delta t} = -u\frac{p_{i+1}^n - p_{i-1}^n}{2\Delta x} \tag{28}$$

$$p_i^{n+1} = -u\Delta t\frac{p_{i+1}^n - p_{i-1}^n}{2\Delta x} + p_i^n \tag{29}$$

$$p_i^{n+1} = -\frac{C_0}{2}\left(p_{i+1}^n - p_{i-1}^n\right) + p_i^n \tag{30}$$

p_i^n, references the $i-$th index and $n-$th time step and $C_0 = (u\Delta t)/\Delta x$.

From equation 30 we see that our problem takes on the form of

$$p^{n+1} = \mathbf{P}p^n. \tag{31}$$

Where $\mathbf{P}$ describes the matrix on the form

$$\mathbf{P} = \begin{bmatrix} 1 & -C_0/2 & 0 & \dots & \dots \\ C_0/2 & 1 & -C_0/2 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & 0 & C_0/2 & 1 & -C_0/2 \\ \dots & \dots & 0 & C_0/2 & 1 \end{bmatrix} \tag{32}$$

using Dirichlet boundary conditions $p_0 = 0 \wedge p_n = 0$. This is a safe assumption to make as the probability of finding the woman at the dam would be zero[1], and can consider that at the end of the considered domain, there is something that causes the chance of finding the woman to be zero, such as a waterfall, pipe or similar.

[1]She would essentially be found already as she would be on the dam.

A Python script that creates the matrix **P** and plots the numerical against the analytical solution with the following parameters

- $x \in [0, 1000]$m
- dx $= 5$
- $\mu = 100$m
- $C = 0.1$
- u $= 0.5$m/s
- $\sigma = 100$m
- dt $=$ (cdx)/u
- $t = [0, 500, 1000, 1500]$s

can look as follows

```python
def pdf(x, sigma, mu):
    leading_term = 1/(sigma * np.sqrt(2*np.pi))
    divisor = (2*sigma*sigma)
    return leading_term * np.exp(-(x-mu)*(x-mu) / divisor)


def analytical(x, t, u, sigma, mu):
    leading_term = 1/(sigma * np.sqrt(2*np.pi))
    divisor = (2*sigma*sigma)
    return leading_term * np.exp(-(x-u*t-mu)*(x-u*t-mu) / divisor)


def FTCS_matrix(n, c):
    A = np.zeros([n, n])
    upper = - c / 2
    lower = c / 2
    for i in range(1, n-1):
        A[i, i] = 1
        A[i, i-1] = lower
        A[i, i+1] = upper
    A[0, 0] = 1
    A[-1, -1] = 1
    A[0, 1] = upper
    A[-1, -2] = lower
    return A


if __name__ == "__main__":
    end_times = [0, 500, 1000, 1500]  # [s]
    C = 0.1  # courant number
    mu = 100  # [m] Relative mean
    sigma = 10  # [m] Standard deviation

    dx = 5
    n = ceil(1000/dx)
    x = np.linspace(0, 1000, n+1)  # meters
    u = 0.5  # m/s

    dt = C/u * dx

    A = FTCS_matrix(n+1, C)

    stability = C * dt / dx
    print(f"Stability r: {stability:.3f}")

    for t_end in end_times:
        t = 0
        p = pdf(x, sigma, mu)
        while t < t_end:
```

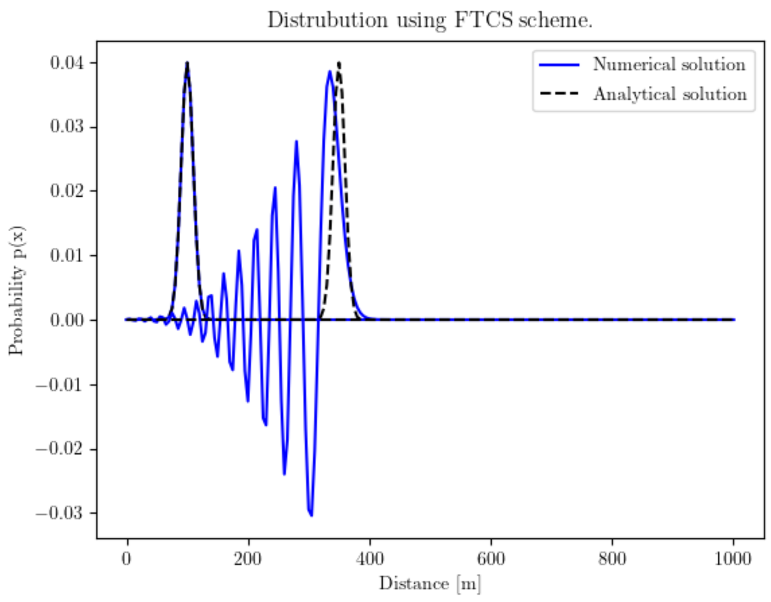

FIGURE 7. Plot of the FTCS scheme using t = 0 and t = 500. Already for t = 500, the numerical values are highly unstable and produces no usable results.

```
            t += dt
            p = np.dot(A, p)
        plt.plot(x, p, "b")
        plt.plot(x, analytical(x, t_end, u, sigma, mu), "k--")
```

excluded from the code is imports and extra plotting details for labels ect.

From figure 7 we see the plot for the value of t = 500s, which already produces an unstable solution. We can not use this data for anything, and this persists when we increase the final time as seen in figure 8. Doing a stability analysis of the problem, we find that measure of stability, mainly $r = max\{u\} \cdot dt/dx$, for the FTCS scheme, is given by

```
"Stability r: 0.020"
```

which for the FTCS scheme is should be stable. However applying the FTCS scheme to a one dimensional advection equation can be shown to be unconditionally unstable, thus we are presented with results which are unstable. This originates from the Von Neumann stability analysis of finite difference schemes[2].

As we now know, we are unable to produce usable results with the FTCS scheme, instead let us consider the implicit forward in time, centered in space(iFTCS) scheme.

2.4. **d).** For the iFTCS scheme, we look at where we want to go, to decide where we are going, instead of looking at where we are to decide where we go. That means instead of solving equation 31, we are interested in solving

$$\mathbf{P}p^{n+1} = p^n \longrightarrow p^{n+1} = \mathbf{P}^{-1}p^n \tag{33}$$

where $\mathbf{P}$ is defined to be the matrix originating from the following equation

$$\frac{C_0}{2}\left(p_{i+1}^{n+1} - p_{i-1}^{n+1}\right) + p_i^{n+1} = p^n \tag{34}$$

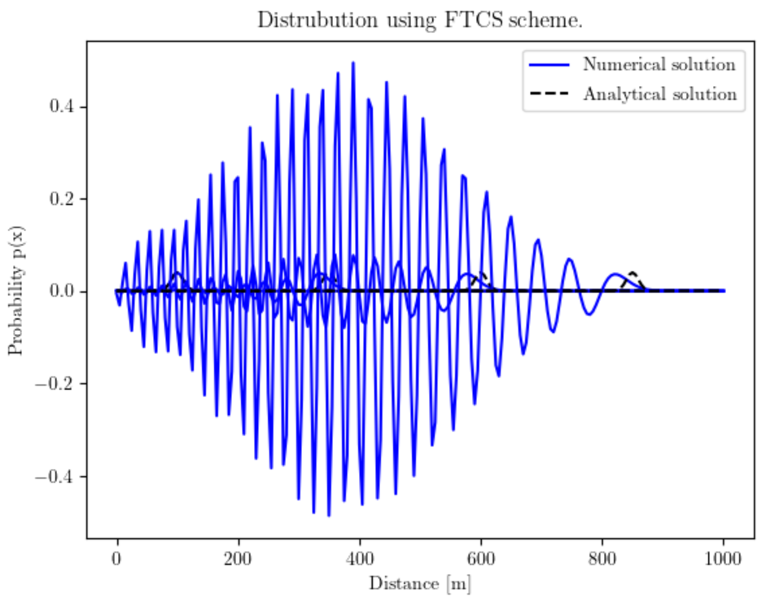

FIGURE 8. Plot using FTCS scheme, of the probability against position for different time steps. We see that the solution is extremely unstable, and produces no usable result. This goes for all values of t.

created in the same way as 32.

The only changed needed to our code is how we define our matrix **P**, the following function implements matrix **P** for the iFTCS scheme.

```python
def iFTCS_matrix(n, c):
    A = np.zeros([n, n])
    upper = c / 2
    lower = - c / 2
    for i in range(1, n-1):
        A[i, i] = 1
        A[i, i-1] = lower
        A[i, i+1] = upper
    A[0, 0] = 1
    A[-1, -1] = 1
    A[0, 1] = upper
    A[-1, -2] = lower
    return np.linalg.inv(A)
```

Whilst the iFTCS scheme is unconditionally stable[3], it does not mean that it would produce the best results. As we can see from figure 9, we are far off the analytical results. However from the iFTCS scheme, we can at least draw some conclusion, whilst rather poor, we can still have a better idea of where the woman will be at each time step. And knowing one thing or two about numerical methods, we can rule out the numerical artifacts and are left with a solution which have three peaks, at almost the same mean x-values as for the analytical solutions. Thus we are able to, with a fairly high accuracy, guesstimate where the woman will be.

2.5. **e).** To simulate a more realistic case, we will consider a river with eddies. The eddies will cause the woman to remain stationary for some time, before probably

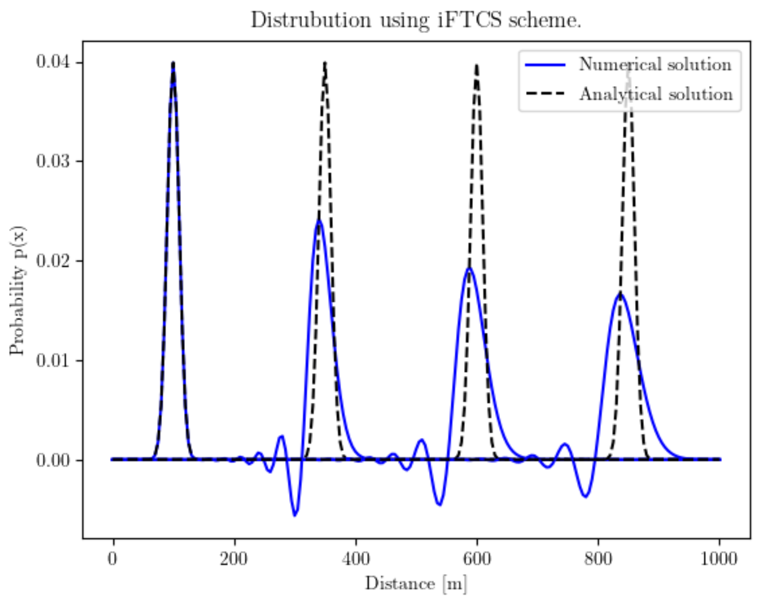

FIGURE 9. Plot using the iFTCS scheme for the same t values as for figure 8. We can see that whilst the iFTCS is unconditionally stable, our results are far off that of the analytical solution. We also so artifacts of the numerical methods, tailing the our curves.

keep moving with the river flow. This can be modeled as diffusion, as the probability of finding the woman at time t, would increase laterally, widening the range where we can find the woman. We will be using a diffusivity D $= 0.5m^2/s$.

We will start off by looking at this using the FTCS scheme, which is stable when we add diffusivity to the problem[2]. The base equation for our problem now becomes

$$\frac{\partial}{\partial t}p + u\frac{\partial}{\partial x}p = D\frac{\partial^2}{\partial x^2}p \tag{35}$$

where D is the diffusivity of the probability which we assume to be constant. For the FTCS scheme from equation 30 we get an additional term and our expression we now look as follows

$$\frac{p_i^{n+1} - p_i^n}{\Delta t} + u\frac{p_{i+1}^n - p_{i-1}^n}{2\Delta x} = D\frac{p_{i+1}^n - 2p_i^n + p_{i-1}^n}{\Delta x^2} \tag{36}$$

$$p_i^{n+1} = -\frac{C_0}{2}\left(p_{i+1}^n - p_{i-1}^n\right) + s\left(p_{i+1}^n - 2p_i^n + p_{i-1}^n\right) + p_i^n \tag{37}$$

where C_0 is defined as for the FTCS scheme 30, and $s = (D\Delta t)/\Delta x^2$. Collecting terms

$$p_i^{n+1} = \left(s - \frac{C_0}{2}\right)p_{i+1}^n + (1-2s)\,p_i^n + \left(s + \frac{C_0}{2}\right)p_{i-1}^n. \tag{38}$$

From equation 38 we can create our matrix $\mathbf{P}$ in the same way as for 32, thus being able to solve the following equation

$$p^{n+1} = \mathbf{P}p^n. \tag{39}$$

2.6. **f).** Adapting this problem to the same initial conditions as in c), we only need to make one adaption again, that is how we define the matrix $\mathbf{P}$. This can be done with the following function

```python
def FTCS_matrix(n, c, s):
    A = np.zeros([n, n])
    diagonal = 1 - 2*s
    upper = s - c/2
    lower = s + c/2
    for i in range(1, n-1):
        A[i, i] = diagonal
        A[i, i-1] = lower
        A[i, i+1] = upper
    A[0, 0] = diagonal
    A[-1, -1] = diagonal
    A[0, 1] = upper
    A[-1, -2] = lower
    return A
```

From studying the relation between the advection transport rate and diffusive transport rate, we get what is called a Péclet number, which in this case is 10. As this number is greater than 1, we know that the advection dominates our equation. From this we should expect that we would see a greater transport down stream, than lateral spread of the probability function. The results of this we can see in figure 10, with numerical curves closely resemble those from d).

Compared to what we found in c), using the FTCS scheme, we know have a stable solution, however this comes to no shock. As stated earlier, when we add a diffusive term to our advection equation the scheme becomes stable. What we can also note of importance, is that the analytical and numerical solution are very close in value, and we have no numerical artifacts in the plots that are visible.

The analytical solution is given by

$$P(x,t) = \frac{1}{2\sqrt{\pi D t}} \exp\left(-\frac{(x-ut)^2}{4Dt}\right). \tag{40}$$

This stems from the same solution method as for the initial equation 22, where we use method of characteristics, and we assume that we have an initial point source, with a standard deviation. So our initial point source would be given by the Gaussian function 22.

2.7. **g).** Studying a case where we have non-constant diffusivity at a point x downstream from where the woman fell into the water. At this point, the diffusivity follows the following equation $D(x) = D_0 + D_1 x$, where $D_0 = 0.5m^2/s$. D_1 is chosen with the following criteria: *Chose D_1 such that the advective effect due to the non-constant diffusivity exactly cancels the advection by the river velocity..*

We have that $D'(x) = D_1$. Starting of with the advection-diffusion equation we get the following

$$\frac{\partial}{\partial t}p + u\frac{\partial}{\partial x}p = \frac{\partial}{\partial x}\left(D\frac{\partial}{\partial x}p\right) \tag{41}$$

$$= \frac{\partial}{\partial x}D\frac{\partial}{\partial x}p + D\frac{\partial^2}{\partial x^2}p \tag{42}$$

$$\frac{\partial}{\partial t}p + \frac{\partial}{\partial x}\left[u - D_1\right] = D\frac{\partial^2}{\partial x^2} \tag{43}$$

From this we have that $u = D_1$, which gives $D_1 = 0.5$.

Now having a new expression for the diffusivity, we want to see over what distance the diffusivity would double. From table 1, we see different values of x which gives

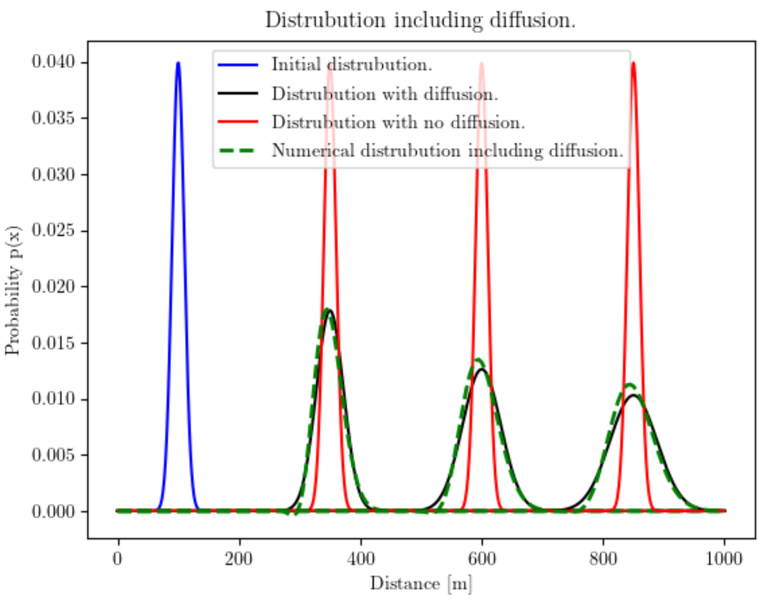

FIGURE 10. Plot of analytical and numerical values found with diffusion. Included also is the analytical results from c). We have not included the numerical results from c), but we can see that the shape of our analytical curves closely resembles those from figure 9. So we have numerical diffusion in d).

TABLE 1. n is the index of the values that results in a double of D(x), with their respective x and D(x) values.

n	x_k	$D(x_k)$
0	0	0.5
1	1	1
2	3	2
3	7	4
4	15	8
5	31	16

a doubling in diffusivity. From this we have that

$$2 * D(x_{k+1}) = D(x_k) \tag{44}$$

$$\frac{1}{2} + \frac{1}{2}x_{k+1} = 1 + x_k \tag{45}$$

$$\frac{1}{2}x_{k+1} = \frac{1}{2} + x_k \tag{46}$$

$$x_{k+1} = 2x_k + 1 \tag{47}$$

Which gives us that the diffusivity doubles, following the recurrence relation given by equation 47.

Writing a short python script as below

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

```

```python
def x(p):
    return 2*p + 1


n = 10*100*1000*10

X = []
Y = []
for i in range(1, n):
    Y.append(x(i))
    X.append(i)

x = np.array(X)
y = np.array(Y)

log_x = np.log(x)
log_y = np.log(y)
np.insert(log_x, 0, 0)
np.insert(log_y, 0, 0)
curve_fit = np.polyfit(log_x, log_y, 1)
print(curve_fit)
"""
[0.99999411 0.69323699]
>>> 0.99999411*exp(0.69323699*x)
"""
```

We can use polynomial fit to find a governing equation for our recurrence relation can be given by

$$0.99999411 \exp(0.69323699x). \tag{48}$$

This equation does not produce exact values, however it gives us the ability to find different values of x, for which the diffusivity would double from the previous value.

In short, we see the first doubling at x = 1m, and then the next at x = 3m, which we see from table 1.w

References

[1] Joe LaCasce. Physical processes in the geosciences. *Department of Geosciences, University of Oslo, Norway.*

[2] Randall J. LeVeque. *Finite-Volume Methods for Hyperbolic Problems.* Cambridge University Press, 2002.

[3] A. MITCHELL and David Griffiths. *The Finite Difference Method in Partial Differential Equations.* 01 1980.