PROBLEM SET 2: ADVECTION AND DIFFUSION
DUE: 12 OCT. 2020

GEO2300: FYSISKE PROSESSER I GEOFAG

SIGURD SANDVOLL SUNDBERG

1. Problem 1: Tsunami

1.1. a). Given a wave with initial disturbance assumed to be Gaussian and propagating to the left, which obeys the following equation in 1D:

$$\frac{\partial}{\partial t}h - c_0\frac{\partial}{\partial x}h = 0 \tag{1}$$

We want to write the expression of the moving wave, when the following is given

- $c_0 = 700km/hr$
- $x_0 = 0$
- $A = 1.0m$
- Standard deviation $= 50km$

Which gives us that at $t = 0$:

$$h = 1.0\exp\left(-\frac{x^2}{50^2}\right) \tag{2}$$

The solution to equation 1 is given by

$$h = F(x + c_0 t) \tag{3}$$

in our case. Given the initial condition 2 when $t = 0$, being initially a Gaussian function, on the form

$$\phi(x, 0) = \phi_0 \exp\left[-(x - x_0)^2/L^2\right] \tag{4}$$

then our solution becomes

$$h(x, t) = 1.0\exp\left(-\frac{(x + c_0 t)^2}{50^2}\right) \tag{5}$$

1.2. b). We want to find the finite difference form, using the FTCS scheme for equation 1. This is given by Taylor expansion around the first degree term, and reads as follows

$$\frac{h_j^{n+1} - h_j^n}{\Delta t} - c_0\frac{h_{j+1}^n - h_{j-1}^n}{2\Delta x} = 0 \qquad \text{Rearrange for } h_j^{n+1} \tag{6}$$

$$h_j^{n+1} = \frac{C}{2}h_{j+1}^n + h_j^n - \frac{C}{2}h_{j-1}^n \qquad \text{Where } C = \frac{c_0\Delta t}{2\Delta x} \tag{7}$$

We have here defined h_j^n as the following:

- $h(t, x) \simeq h(t_n, x_j) = h_j^n$
- t_n is the time partitioned into n equally sized time steps.
- x_j is the distance partitioned into j equal steps.
- $\Delta t = t_{n+1} - t_n$
- $\Delta x = x_{j+1} - x_j$

Date: October 10, 2020.

1.3. **c).** Given the following Neumann boundary conditions

$$\frac{\partial}{\partial x} h = 0 \tag{8}$$

This gives that

$$h_{-1} = h_0 \tag{9}$$

$$h_{n+1} = h_n \tag{10}$$

for n grid points. Where we have ghost points that follows the above equations.

Let $n = 6$, i.e. $j = 0, 1, 2, 3, 4, 5, 6$, be the number of grid points. Our set of equations then become

$$h_0^{n+1} = \frac{C}{2} h_1^n + h_0^n - \frac{C}{2} h_0^n \qquad \text{Here } h_0 = h_{-1} \tag{11}$$

$$h_1^{n+1} = \frac{C}{2} h_2^n + h_1^n - \frac{C}{2} h_0^n \tag{12}$$

$$h_2^{n+1} = \frac{C}{2} h_3^n + h_2^n - \frac{C}{2} h_1^n \tag{13}$$

$$h_3^{n+1} = \frac{C}{2} h_4^n + h_3^n - \frac{C}{2} h_2^n \tag{14}$$

$$h_4^{n+1} = \frac{C}{2} h_5^n + h_4^n - \frac{C}{2} h_3^n \tag{15}$$

$$h_5^{n+1} = \frac{C}{2} h_6^n + h_5^n - \frac{C}{2} h_4^n \tag{16}$$

$$h_6^{n+1} = \frac{C}{2} h_6^n + h_6^n - \frac{C}{2} h_5^n \qquad \text{Here } h_6 = h_7 \tag{17}$$

This gives us a tridiagonal matrix on the following form, for a given n.

$$\begin{bmatrix} 1-\frac{C}{2} & \frac{C}{2} & 0 & 0 & 0 & 0 & 0 \\ -\frac{C}{2} & 1 & \frac{C}{2} & 0 & 0 & 0 & 0 \\ 0 & -\frac{C}{2} & 1 & \frac{C}{2} & 0 & 0 & 0 \\ 0 & 0 & -\frac{C}{2} & 1 & \frac{C}{2} & 0 & 0 \\ 0 & 0 & 0 & -\frac{C}{2} & 1 & \frac{C}{2} & 0 \\ 0 & 0 & 0 & 0 & -\frac{C}{2} & 1 & \frac{C}{2} \\ 0 & 0 & 0 & 0 & 0 & -\frac{C}{2} & 1+\frac{C}{2} \end{bmatrix} \tag{18}$$

This can easily be extended to a arbitrary $n \times n$ matrix.

1.4. **d).** The code in 1.7 implements the FTCS scheme found above for the following conditions

- $x \in [-1400, 300]$, with unit km.
- $n = 400$ grid points.
- $c_0 = 700$ kmh.
- $C = c_0 dt/dx = 0.1$
- $dt = C dx/c_0$

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA


def H(x):
    return 1.0 * np.exp(-(x*x)/(50*50))


def analytical_solution(t):
    A = 1.0
    std = 50
    c = 700
    x = np.linspace(-1400, 300, 400)
    X = x + c * t
    y = A * np.exp(-(X*X)/(std*std))
    return x, y


def solver(t_end):
    c_0 = 700

    N = 400
    C = 0.1
    n = N - 2

    d = 1
    nd = C/2
    X = np.linspace(-1400, 300, N)[1:-1]

    dx = X[1]-X[0]
    dt = C*dx/c_0
    A = np.zeros([n, n])
    for i in range(1, n-1):
        A[i, i] = d
        A[i, i-1] = -nd
        A[i, i+1] = nd

    A[0, 0] = d - nd
    A[n-1, n-1] = d + nd
    A[0, 1] = nd
    A[n-1, n-2] = - nd

    t = 0
    x, y = analytical_solution(t_end)
    h = H(X)
    while t <= t_end:
        t += dt
        h = np.dot(A, h)
    return [(X, h), (x, y)]


if __name__ == "__main__":
    color = ["c", "r", "g", "b", "m"]
    i = 0
    for t_end in [0, 0.5, 1, 1.5, 2]:
        [X, h], [x, y] = solver(t_end)
        plt.plot(
            X, h, color[i], label=fr"Numerical solution for t = {t_end
    :.1f}", linewidth=1)
        plt.plot(
            x, y, color[i] + '--', label=fr"Analytical solution for t
    = {t_end:.1f}", linewidth=1)
        i += 1

    plt.rcParams.update({
        "text.usetex": True,
        "font.sans-serif": ["Helvetica"],
        "font.family": "DejaVu Sans"})
    plt.title(r"Wave equation using FTCS")
    plt.xlabel(r"Position $\left[ m\right]$")
    plt.ylabel(r"Amplitude $[-]$")
    # plt.legend()
    plt.savefig("FTCS.pdf")
    plt.show()
```

1.5. **e).** Seen in figure 1 we see the numerical solutions plotted against the analytical solutions for time values $t = 0, 0.5, 1, 1.5, 2$. We see that the FTCS scheme is unstable when plotting the wave equation, as seen in the out bursts seen above and below the black curve. However is still reproduces the results to a fairly okay, and gives an easy way to get an idea of the solutions to the problem. Before choosing a more appropriate scheme for the problem at hand.

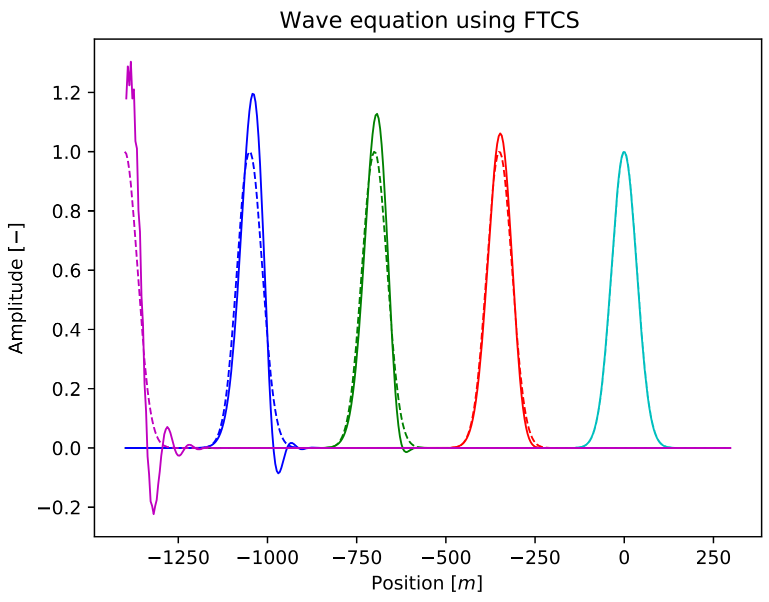

FIGURE 1. Plot of the analytical solution to the wave equation against the FTCS scheme for the numerical solution. The time values are shown from left to right, so the far most left top is for t = 0, and the right most top is for t = 2. Numerical is shown in full line, and the analytical is shown as dashed line.

1.6. **f).** The Crank-Nicholson(CN) scheme is a combination of the FTCS scheme shown in 6 and the iFTCS scheme shown 19.

$$-\frac{C}{2}h_{j+1}^{n+1} + h_j^{n+1} + \frac{C}{2}h_{j-1}^{n+1} = h_j^n \tag{19}$$

Combining FTCS and iFTCS schemes, in equal parts the CN scheme reads as follows

$$-\frac{C}{4}h_{j+1}^{n+1} + h_j^{n+1} + \frac{C}{4}h_{j-1}^{n+1} = \frac{C}{4}h_{j+1}^n + h_j^n - \frac{C}{4}h_{j-1}^n \tag{20}$$

Writing this on the same form as we did for FTCS scheme we get two matrices for solving the following equation

$$Bh^{n+1} = Ah^n \tag{21}$$

where A and B are our two matrices. Matrix A reads as for FTCS scheme, with all the elements $\frac{C}{2} \rightarrow \frac{C}{4}$. Then matrix B read as follows

$$\begin{bmatrix} 1+\frac{C}{4} & -\frac{C}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{C}{4} & 1 & -\frac{C}{4} & 0 & 0 & 0 & 0 \\ 0 & \frac{C}{4} & 1 & -\frac{C}{4} & 0 & 0 & 0 \\ 0 & 0 & \frac{C}{4} & 1 & -\frac{C}{4} & 0 & 0 \\ 0 & 0 & 0 & \frac{C}{4} & 1 & -\frac{C}{4} & 0 \\ 0 & 0 & 0 & 0 & \frac{C}{4} & 1 & -\frac{C}{4} \\ 0 & 0 & 0 & 0 & 0 & \frac{C}{4} & 1-\frac{C}{4} \end{bmatrix} \tag{22}$$

This gives us the ability to solve 21 the following way

$$h^{n+1} = B^{-1}Ah^n \tag{23}$$

where B^{-1} is the inverse matrix of B.

1.7. **g).** Using the same conditions as for problem e). We see from figure 2 that we have a better numerical solution compared to the FTCS scheme. Our solution is fairly steady above the analytical solution, without increasing as it did for FTCS. However we still see artifacts of the numerical solution, as seem especially for the last time, $t = 2$. This is from the scheme not being that well suited to deal with wave equation when number of iterations grows large.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA


def H(x):
    return 1.0 * np.exp(-(x*x)/(50*50))


def initialize_tridiag(diagonal, offDiagonalL, offDiagonalU, n):
    A = np.zeros([n, n])

    for i in range(1, n-1):
        A[i, i] = diagonal
        A[i, i-1] = offDiagonalL
        A[i, i+1] = offDiagonalU
    return A


def analytical_solution(t):
    A = 1.0
    std = 50
    c = 700
    x = np.linspace(-1400, 300, 400)
    X = x + c * t
    y = A * np.exp(-(X*X)/(std*std))
    return x, y


def solver(t_end):
    c_0 = 700

    N = 400
    C = 0.01
    n = N - 2

    d = 1
    nd = C/4
```

```python
    A = initialize_tridiag(d, -nd, nd, n)
    B = initialize_tridiag(d, nd, -nd, n)
    X = np.linspace(-1400, 300, N)[1:-1]

    dx = X[1]-X[0]
    dt = C*dx/c_0

    # Apply boundaries
    A[0, 0] = (d-nd)
    A[n-1, n-1] = d+nd
    A[0, 1] = nd
    A[n-1, n-2] = -nd
    B[0, 0] = d + nd
    B[n-1, n-1] = d - nd
    B[0, 1] = - nd
    B[n-1, n-2] = nd
    Binv = LA.inv(B)

    t = 0
    x, y = analytical_solution(t_end)
    h = H(X)
    while t <= t_end:
        t += dt
        h = np.dot(Binv, np.dot(A, h))
    return [(X, h), (x, y)]


if __name__ == "__main__":
    plt.rcParams.update({
        "text.usetex": True,
        "font.sans-serif": ["Helvetica"],
        "font.family": "DejaVu Sans"})
    plt.title(r"Wave equation using CN")
    plt.xlabel(r"Position $\left[ m\right]$")
    plt.ylabel(r"Amplitude $[-]$")
    color = ["c", "r", "g", "b", "m"]
    i = 0
    for t_end in [0, 0.5, 1, 1.5, 2]:
        [X, h], [x, y] = solver(t_end)
        plt.plot(
            X, h, color[i], label=fr"Numerical solution for t = {t_end
    :.1f}", linewidth=1)
        plt.plot(
            x, y, color[i] + '--', label=fr"Analytical solution for t
    = {t_end:.1f}", linewidth=1)
        i += 1

    # plt.legend()
    plt.savefig("CN.pdf")
    plt.show()
```

2. Problem 2: Boundary Diffusion

We will now be looking at a one dimensional aquifer with oil seeping into one end. The oil concentration obeys the 1D diffusion equation

$$\frac{\partial}{\partial t}C = D\frac{\partial^2}{\partial x^2}C \tag{24}$$

2.1. **a).** We want to find the analytical solution for $C(x,t)$ assuming, $C = C_0$ at $x = 0$ and that $C(x,0) = 0$.

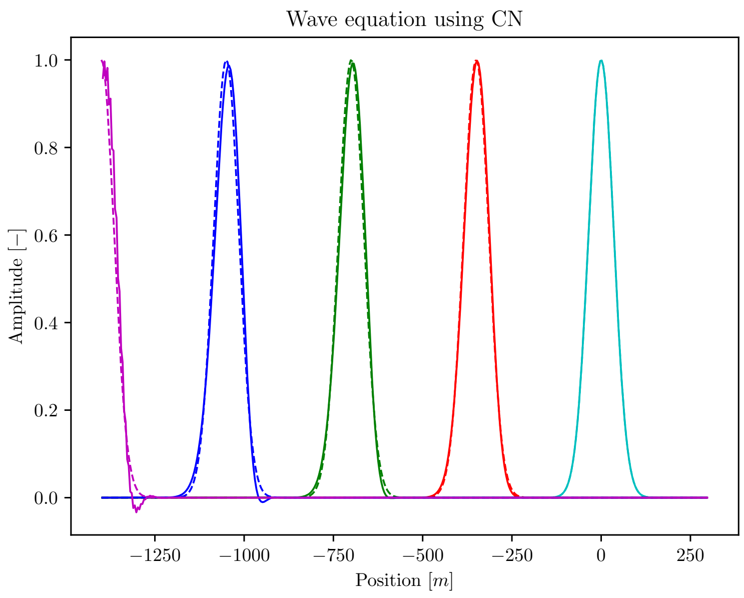

FIGURE 2. Plot of the analytical solution to the wave equation against the CN scheme for the numerical solution. The time values are shown from left to right, so the far most left top is for t = 0, and the right most top is for t = 2. Numerical is shown in full line, and the analytical is shown as dashed line.

The analytical solution follows as from the compendium[1] is

$$C = C_0 \mathrm{erfc}\left(\frac{x}{2\sqrt{Dt}}\right) \tag{25}$$

2.2. **b).** The following program is used to find the answers in both problem b) and c). It finds the oil concentration at times $t = 1, 10, 100$ days, with $C_0 = 0.5$.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import special

if __name__ == "__main__":
    cms_to_ms = 1/(100*100)  # cm^2/s to m^2/s
    day_to_s = 24 * 60 * 60  # days to seconds

    def diffusion(t, x, c):
        D = 1 * cms_to_ms  # cm^2/s
        return c*special.erfc(x/(2*np.sqrt(D*t)))

    c0 = 0.5
    t_list = np.array([1, 10, 100]) * day_to_s
    x = np.linspace(0, 100, 2000)

    x_list = np.array([100, 500, 1000, 1500, 2000])
    dt = 1
    c = 0.5
    c1 = 0
    t_old = []
    for x_value in x_list:
        c1 = 0
        x_array = np.linspace(0, x_value, 5000)
        t_start = 0
        while c1 < 0.25:
            t_start += dt
            c1 = diffusion(t_start*day_to_s, x_array, c0)
            c1 = c1[-1]
        print(
            f"Number of days needed for the concentration to reach 25%
     is: {t_start:d} days, with x = {x_value:d}")
        t_old.append(t_start)
    for t in t_list:
        plt.plot(x, diffusion(t, x, c0),
                 label=f"Diffusion for t = {t/day_to_s:.0f}")
    plt.savefig("FTCS2.pdf")
    plt.rcParams.update({
        "text.usetex": True,
        "font.sans-serif": ["Helvetica"],
        "font.family": "DejaVu Sans"})
    plt.title(r"Analytical solution for Diffusion equation")
    plt.xlabel(r"Distance x $\left[ m\right]$")
    plt.ylabel(r"Concentration $[-]$")
    plt.legend()
    plt.savefig("FTCS2.pdf")
    plt.show()

    q = np.log10(x_list)
    p = np.log10(t_old)
    m, b = np.polyfit(q, p, 1)
    plt.plot(q, p, "r")
    plt.plot(q, p, "ko", label=r"datapoints")
    plt.annotate(fr"The incline of the line goes as m = {m:.2f}", [2,
    5])
    plt.title(r"Log/Log plot of distance vs time")
    plt.xlabel(r"$\log_{10} (x)$")
    plt.ylabel(r"$\log_{10} (t)$")
    plt.legend()
    plt.savefig("Relation.pdf")
    plt.show()
"""
Number of days needed for the concentration to reach 25% is: 1273 days
Number of days needed for the concentration to reach 25% is: 31802
    days
Number of days needed for the concentration to reach 25% is: 127206
    days
Number of days needed for the concentration to reach 25% is: 286213
    days
Number of days needed for the concentration to reach 25% is: 508822
    days
"""
```

The concentration as a function of time is seen in figure 3, where we see what is expected with a large part of the aquifer being filled with oil after a longer period of time.

2.3. **c).** Using the same code as in b), we want to find when the concentration reaches 25% at distance $x = 100$m and $x = 1000$m from the source. Also how much longer do you have to wait if the distance is increased by a factor of α.

Running our program we get the following output

```
  """
```

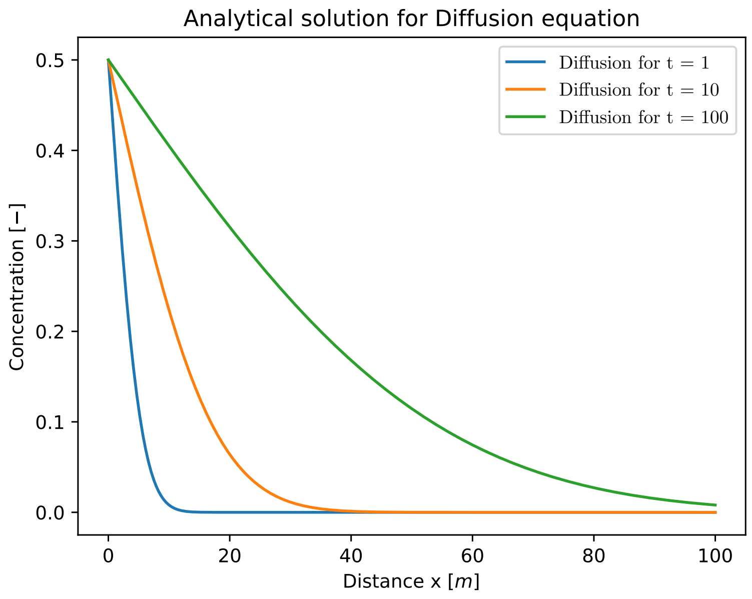

FIGURE 3. Plot of the analytical solution to the 1D diffusion equation for a aquifer which is 100m long..

```
Number of days needed for the concentration to reach 25% is: 1273
  days, with x = 100
Number of days needed for the concentration to reach 25% is: 127206
  days, with x = 1000
"""
```

The relation between these two is roughly a factor of 100, with an increase in distance of a factor of 10. However this does not tell us really tell us the relation, however running more iterations for different values of x, we find the results which can be seen in 4 From here we can see that the relation follows a logarithmic scale, of $\log_{10}$. Such that our relation goes as

$$\log_{10}(x) = 2\log_{10}(t) \tag{26}$$

or

$$x = t^2 \tag{27}$$

So for any a given α_x we have the following relation $\alpha_x \propto \alpha_t^2$. Where α_x and α_t is the increase in position and time, respectively by a factor α

2.4. **d).** Writing up the FTCS scheme we have the following base expansion

$$\frac{C_j^{n+1} - C_j^n}{dt} = D\frac{C_{j+1}^n - 2C_j^n + C_{j+1^n}}{dx^2} \tag{28}$$

$$C_j^{n+1} = sC_j{+}1^n + (1-2s)C_j^n + sC_{j-1}^n \qquad \text{With } s = \frac{D\,dt}{dx^2} \tag{29}$$

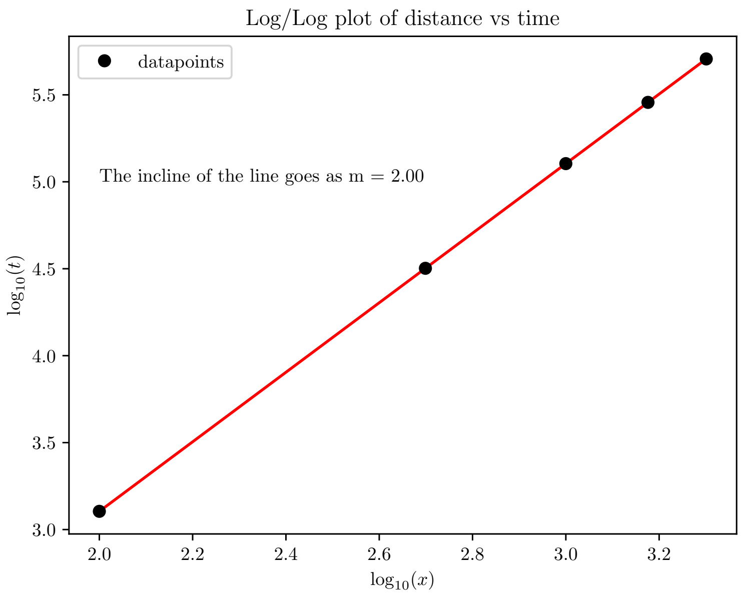

FIGURE 4. Plot for different values of x against the respective number of days.

Expanding this with 7 grid points, and using $C(x = 0) = 0.5 \wedge C(x = 100) = 0$ we have

$$C_0^{n+1} = 0.5 \tag{30}$$
$$C_1^{n+1} = sC_0^n + (1 - 2s)C_1^n + sC_2^n \tag{31}$$
$$C_2^{n+1} = sC_1^n + (1 - 2s)C_2^n + sC_3^n \tag{32}$$
$$C_3^{n+1} = sC_2^n + (1 - 2s)C_3^n + sC_4^n \tag{33}$$
$$C_4^{n+1} = sC_3^n + (1 - 2s)C_4^n + sC_5^n \tag{34}$$
$$C_5^{n+1} = sC_4^n + (1 - 2s)C_5^n + sC_6^n \tag{35}$$
$$C_6^{n+1} = 0 \tag{36}$$

From this we can construct a matrix A such that we can solve the following equation

$$C^{n+1} = AC^n \tag{37}$$

2.5. **e).** The following code implements the FTCS scheme with N grid points.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import special

if __name__ == "__main__":
    cms_to_ms = (60*60*24)/(100*100)  # cm^2/s to m^2/s
    day_to_s = 1  # 24 * 60 * 60  # days to seconds

    def diffusion(t, x, c):
        D = 1 * cms_to_ms  # cm^2/s
        return c*special.erfc(x/(2*np.sqrt(D*t)))

    N = 100
    n = N
    xmin = 0
    xmax = 100
    X = np.linspace(xmin, xmax, N*5)
    dx = (X[1]-X[0])*5
    x = np.linspace(xmin, xmax, n)
    c0 = 0.5
    D = 1 * cms_to_ms
    s = 0.4
    dt = s/D * dx*dx
    A = np.zeros([n, n])
    m0 = s
    mD = (1-2*s)
    C = np.zeros(n)
    cU = 0
    cL = 0.5
    C[0] = cL

    # Setup A
    A[0, 0] = 1
    A[-1, -1] = 1
    for i in range(1, n-1):
        A[i, i] = mD
        A[i, i-1] = m0
        A[i, i+1] = m0

    t_list = np.array([1, 10, 100])
    plt.savefig("FTCS2.pdf")
    plt.rcParams.update({
        "text.usetex": True,
        "font.sans-serif": ["Helvetica"],
        "font.family": "DejaVu Sans"})
    plt.title(r"Analytical solution for Diffusion equation")
    plt.xlabel(r"Distance x $\left[ m\right]$")
    plt.ylabel(r"Concentration $[-]$")
    for t in t_list:
        ts = 0
        while ts <= t:
            ts += dt
            C = np.dot(A, C)
        plt.plot(x, C, "--", label=f"Numerical solution for t = {t:.0f
    }")
        plt.plot(X, diffusion(t*day_to_s, X, c0), 'k-.',
                 label=f"Analytical solution for t = {t:.0f}")
    plt.legend()
    # plt.savefig("exNUMFTCS.pdf")
    plt.show()
```

We can see from figure 5 that for t = 1 that our solution agrees extremely well with the analytical solution, is there is no visible difference between the two graphs. For higher values of t, namely t = 10 and t = 100, we see that our numerical solution strays further from the analytical one. What is notable is that for t = 100, our numerical solution reproduces the exponential function poorly as our difference scheme is based around a second order polynomial. Importantly this means that for higher values of t>100, we would no longer get solutions which represents the analytical one. This means that for low values of t, we can reproduce the analytical results, but for higher values of t, we would need a more appropriate scheme, to predict the analytical results.

2.6. **f).** We see that when we choose s = 0.6 that our solution blows up as is expected from the stability of the FTCS scheme. We can also see that for higher values of t that the solution blows up more. This is because how much a solution

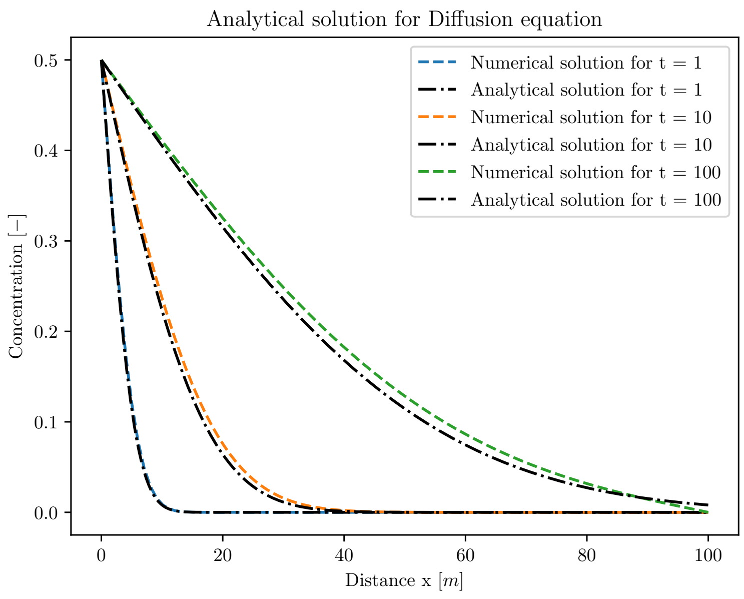

FIGURE 5. Plot of the analytical solution and numerical solution to the 1D diffusion equation, with s = 0.4.

blows up is dependent on the number of iterations which is performed on the data set. All solutions blow up, but we can only see the effects of the higher time step when plotting them in the same plot. This results can be seen in 6.

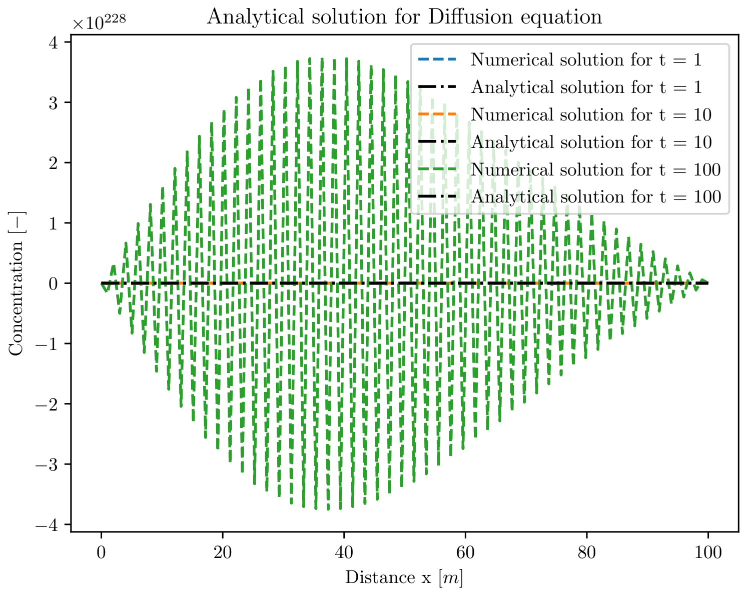

FIGURE 6. Plot of the analytical solution and numerical solution to the 1D diffusion equation, with s = 0.6.

References

[1] Joe LaCasce. *Physical Processes in the Geosciences.* Department of Geosciences, University of Oslo, Norway. Downloaded: 25.09.2020