Exercise 2

In the first exercise, we solved a simple "steady-state" problem (i.e. there was no time derivative) in the problem. We will now look at one of the simplest transient problems that you should be familiar with from `Heat and Mass Transfer 1`.

This exercise is broken into three parts as we want to explore different aspects of the same equation system. In the first part, we focus more on how transient problems are tackled and we consider the case where Dirichlet boundary conditions are applied. In the second part, we consider the solution of the diffusion equation with Neumann boundary conditions and in the third part, we consider how a reaction in the system can be modelled.

Learning objectives

By the end of this exercise, we hope to achieve the following goals:

1. Appreciate the difference between numerically solving transient and steady-state problems
2. Appreciate how different time-stepping methods can impact the solution
3. Visualize how diffusion takes place in different conditions
4. Perform basic post-processing to generate relevant plots
5. Appreciate how reactions and similar processes are treating in the context of numerical solutions

Part 1

The equation system of for the transient problem can be written as follows:

$$\frac{\partial c}{\partial t} = D\nabla^2 c,$$

where c is the concentration / mole fraction, t is the time and D is the diffusion coefficient. The treatment of the diffusion term is not too dissimilar from the previous exercise, but we need to think about how we handle the time derivative.

Time stepping

For this exercise, we will consider two simple ways of handling the time derivative:

1. Fully explicit forward Euler time-stepping
2. Fully implicit backward Euler time-stepping

This part may get a bit technical, so I will attempt to flesh out every step and keep it as simple as possible. We are able to write an approximation for the full equation as follows for x at a point i as follows:

$$\frac{c_i^{new} - c_i^{old}}{\Delta t} = \alpha D \frac{c_{i+1}^{new} - 2c_i^{new} + c_{i-1}^{new}}{\Delta x^2} + (1-\alpha) D \frac{c_{i+1}^{old} - 2c_i^{old} + c_{i-1}^{old}}{\Delta x^2}$$

We introduce a new parameter α which readily enables us to pick a specific numerical scheme. The three values of α that are of particular interest are (We are only considering the first 2):

1. $\alpha = 1$: Fully implicit backward Euler time-stepping.

2. $\alpha = 0$: Fully explicit forward Euler time-stepping
3. $\alpha = 0.5$: Crank-Nicolson time-stepping.

When $\alpha = 1$:

$$\frac{c_i^{new} - c_i^{old}}{\Delta t} = D\frac{c_{i+1}^{new} - 2c_i^{new} + c_{i-1}^{new}}{\Delta x^2}$$

When $\alpha = 0$:

$$\frac{c_i^{new} - c_i^{old}}{\Delta t} = D\frac{c_{i+1}^{old} - 2c_i^{old} + c_{i-1}^{old}}{\Delta x^2}$$

When a fully explicit scheme such as the forward Euler is used, the new value (moving forward in time) can be evaluated explicitly from values from the old values from the previous timestep which makes it much more easy to implement than implicit methods. However, the trade off is that often, explicit time stepping requires excessively small timesteps for numerical stability while the implicit backwards Euler time-stepping is far more stable.

Stability for forward Euler time stepping

For diffusion problems (We can treat heat and mass transfer as diffusion problems collectively), we can consider the following metric:

We can write down the dimensionless mesh Fourier number Fo_{Mesh}:

$$Fo_{Mesh} = D\frac{\Delta t}{\Delta x^2}$$

For fullness, a not entirely rigorous derivation of the stability criteria for the explicit Forward Euler method is presented. Feel free to gloss over this section.

So we start with the discretization scheme of the explicit forward Euler:

$$\frac{c_i^{new} - c_i^{old}}{\Delta t} = D\frac{c_{i+1}^{old} - 2c_i^{old} + c_{i-1}^{old}}{\Delta x^2}$$

Rejigging things:

$$c_i^{new} = c_i^{old} + D\frac{\Delta t}{\Delta x^2}\left(c_{i+1}^{old} - 2c_i^{old} + c_{i-1}^{old}\right)$$

Introducing Fo_{Mesh}:

$$c_i^{new} = c_i^{old} + Fo_{Mesh}\left(c_{i+1}^{old} - 2c_i^{old} + c_{i-1}^{old}\right)$$

We can further rejig this to group all the c_i^{old} terms together:

$$c_i^{new} = (1 - 2Fo_{Mesh})c_i^{old} + 2Fo_{Mesh}\left(\frac{c_{i+1}^{old} + c_{i-1}^{old}}{2}\right)$$

The insight to be gained from rejigging the equation to the above form is that x_i^{new} is the weighted average of the old value x at that point i and the average of the neighboring points' $i+1$ and $i-1$ old value of c. If $Fo_{Mesh} = 0$, then its easy to see that we are not advancing forward in time. When $Fo_{Mesh} = 0.5$:

$$c_i^{new} = \frac{1}{2}(c_{i+1}^{old} + c_{i-1}^{old})$$

We see that the new value is the average of the old value of the two adjacent points. When we have $Fo_{Mesh} > 0.5$, we run into the problem where updated value unphysically overshoots the average of the neighboring values. So this gives us the following stability criteria which we can use:

$$\Delta t \leq \frac{\Delta x^2}{2D}$$

Another useful insight that can be gleaned from this relationship is that when we increase the mesh resolution (i.e. make Δx smaller by a 2X), we have to decrease the timestep Δt 4X

Problem statement

We first intend to model a simple 1D slab of size 1 where one end is held at a fixed concentration and the other end is held is zero concentration. There is no reaction and the concentration throughout the block is initially zero. The model equation of interest can be written as follows:

$$\frac{\partial c}{\partial t} = D\nabla^2 c$$

With the following boundary conditions:

$$c(x=0,t) = 1$$
$$c(x=1,t) = 0$$

And the initial conditions:

$$c(x,t=0) = 0$$

Analytical solution

Let's work through the problem. We quite quickly encounter the fact that due to the non-homogenous boundary conditions, a naïve application of separation of variables will not work. The boundary conditions require modification.

Fortunately, the physical nature of the problems lends to the solution. By fixing the concentrations at the ends of the slab and not considering a flux condition, the system will eventually tend to a steady state solution. So we can employ this idea to perform a variable transformation that enables us to reformulate the boundary conditions as homogenous BCs.

The steady state solution can be obtained as follows (We cancel the time derivative which simplifies the problem to a 1D ODE) :

$$\frac{d^2 c}{dx^2} = 0$$

The solution of this is given by:

$$c_{S.S.}(x) = c_1 x + c_2$$

Substituting the boundary conditions, we get:

$$c_{S.S.}(x) = 1 - x$$

We now introduce the following variable transform:

$$\phi(x,t) = c(x,t) - c_{S.S.}(x)$$

Evaluating the relevant derivatives:

$$\frac{\partial \phi}{\partial t} = \frac{\partial c}{\partial t}$$

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\partial^2 c}{\partial x^2}$$

This is a nice result! The equation does not change shape in anyway with the variable transform!! Let's consider how the BCs evolve:

$$\phi(0,t) = c(0,t) - c_{S.S.}(0) = 1 - 1 = 0$$

$$\phi(1,t) = c(1,t) - c_{S.S.}(1) = 0 - 0 = 0$$

We get nice homogenous BCs which enable us to solve for $\phi(x,t)$ using separation of variables. We obtain the following result:

$$\phi(x,t) = \sum_{n=1}^{\infty} a_n \sin{(n\pi x)} e^{-Dn^2\pi^2 t}$$

To evaluate the value of the coefficients a_n: we need to consider the initial condition $\phi(x,0)$:

$$\phi(x,0) = c(x,0) - c_{S.S.}(x) = 0 - (1-x) = x - 1$$

a_n is given as follows:

$$a_n = 2\int_0^1 (x-1)\sin{(n\pi x)}\, dx, \quad n = 1,2,3\ldots$$

This is a chore to evaluate. We employ the power of `WolframAlpha`!

$$a_n = \frac{2}{\pi^2 n^2}(\sin{(\pi n)} - \pi n) = \frac{-2}{n\pi}, \quad n = 1,2,3\ldots$$

Now that we have a nice clean expression for a_n, let us finally write down our full analytical solution to this problem.

$$c(x,t) = \phi(x,t) + c_{S.S.}(x) = 1 - x - \sum_{n=1}^{\infty} \frac{2}{n\pi} \sin{(n\pi x)} e^{-Dn^2\pi^2 t}$$

Numerical solution

Setting up the problem for the most part is similar, the only difference being how the equation is defined and how time-stepping is performed. I will detail those two sections here for clarity.

Defining the equation

We make use of the full time discretization scheme involving α as outlined above as this gives us the convenience of toggling between explicit and implicit time-stepping very easily

```
eq = (TransientTerm() == DiffusionTerm(coeff= alpha* D) +
ExplicitDiffusionTerm(coeff = (1.0 - alpha)*D))
```

We can now set α as a simulation parameter easily and not worry about rejigging the equation.

Time-stepping

First we need to pick an α value. Suppose we pick $\alpha = 0$ for now, we can use our stability criteria as follows:

$$\Delta t = \frac{\Delta x^2}{2D} \times (\%Margin) = \frac{4 \times 10^{-4}}{2.0} \times 0.9 = 0.00018$$

Now that we have a time-step, the next question we need to figure out is how long to run the simulation for! Now that may seem like a straightforward question in that this should normally be pre-defined. But in this case, we don't immediately have a fixed value for the time taken for the system to reach steady state. So let's employ a bit of street-fighting for this:

- Our simulation domain is $1m$ long
- We set our diffusion coefficient to be $1m^2/s$
- So an approximation for the time taken for a molecule to diffuse across the entire domain is $t \sim \frac{D}{L^2}$ by considering simple dimensional analysis.
- Therefore, the time would be approximately $1s$

We can set up the time-stepping! First we define a couple of parameters:

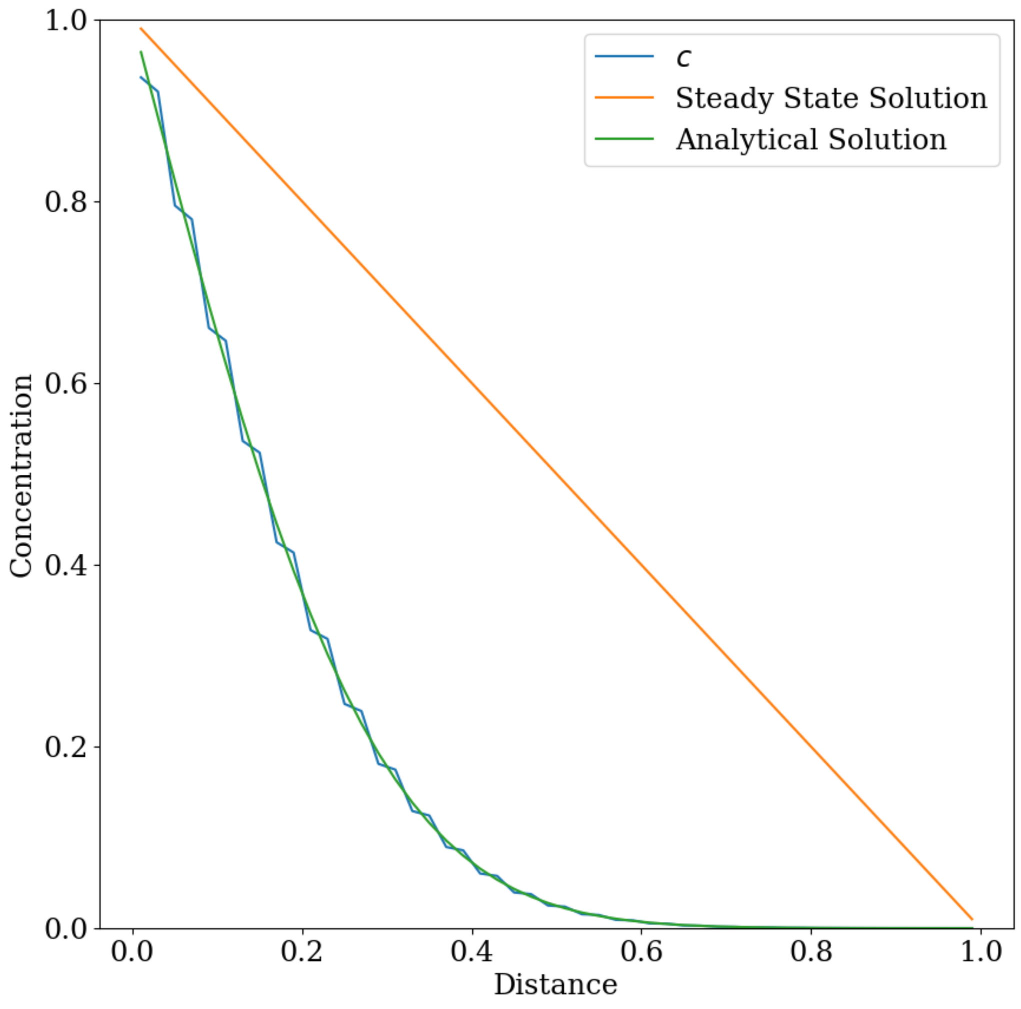
```
time_stride = 50
timestep = 0
run_time = 1
```

We introduce a time stride as we do not want to see the output from every single timestep. We then introduce a timestep counter and lastly define the total simulation run time. We when initialize the viewer and use a while loop to advance in time:

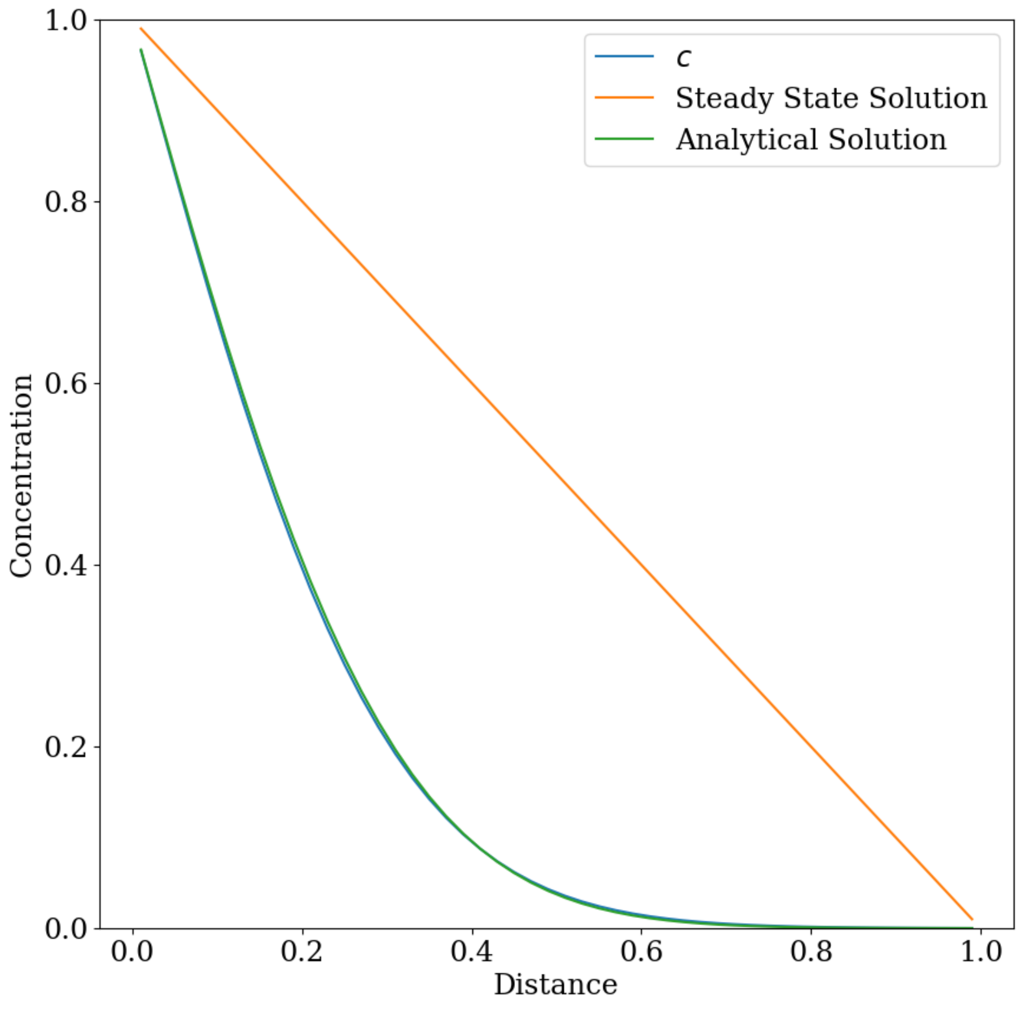
```
if __name__ == "__main__":
    viewer = Viewer(vars=(c, analytical_steady, analytical_solution_transient),
datamin=0., datamax=1.)

while t < run_time:
    t += dt
    timestep += 1
    eq.solve(var=c, dt=dt)
    if (timestep % time_stride ==0):
        if __name__ == '__main__':
            viewer.plot()
if __name__ == '__main__':
    input("Press <return> to proceed...")
```

Results

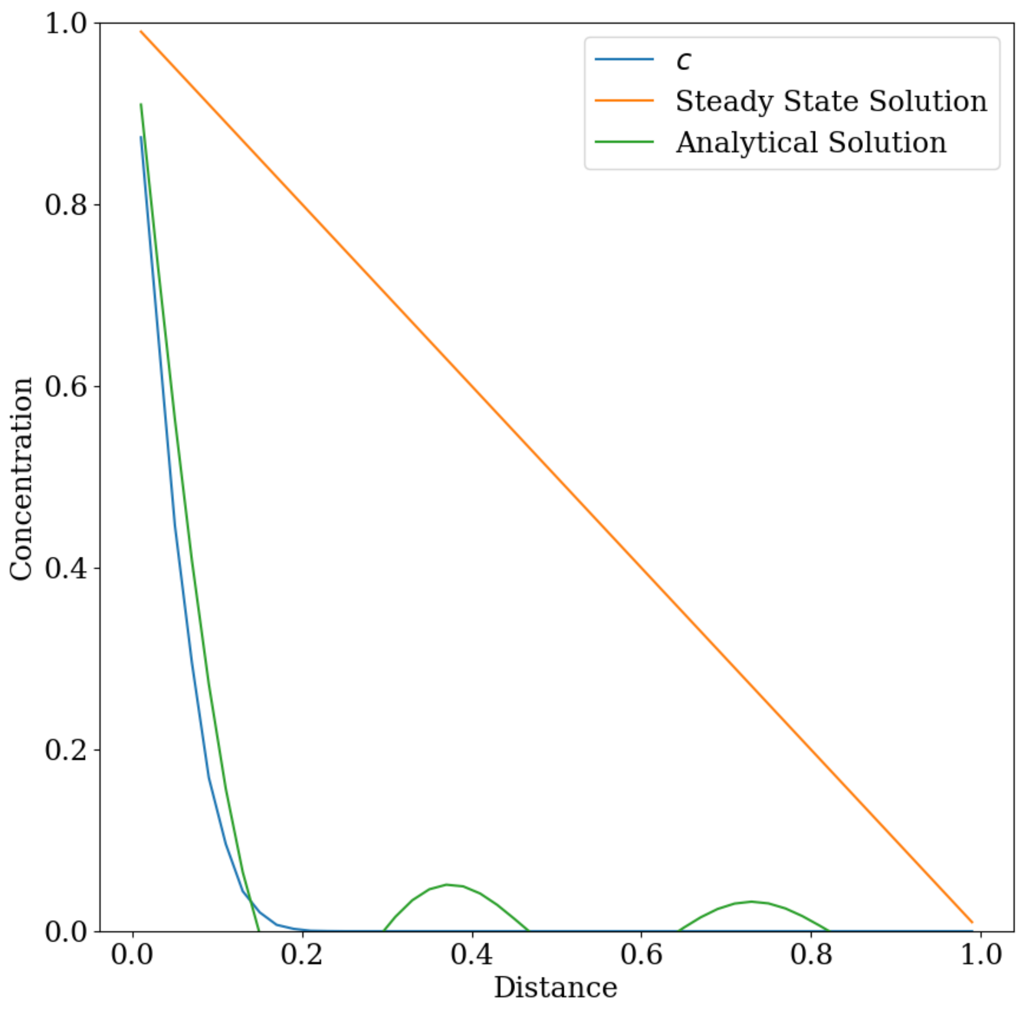
When using the explicit time-stepping method with a timestep ~ 1.11×0.00018, you can see "wiggles" in the solution which is a numerical artifact. If you further increase the timestep, the solution will oscillate wildly.



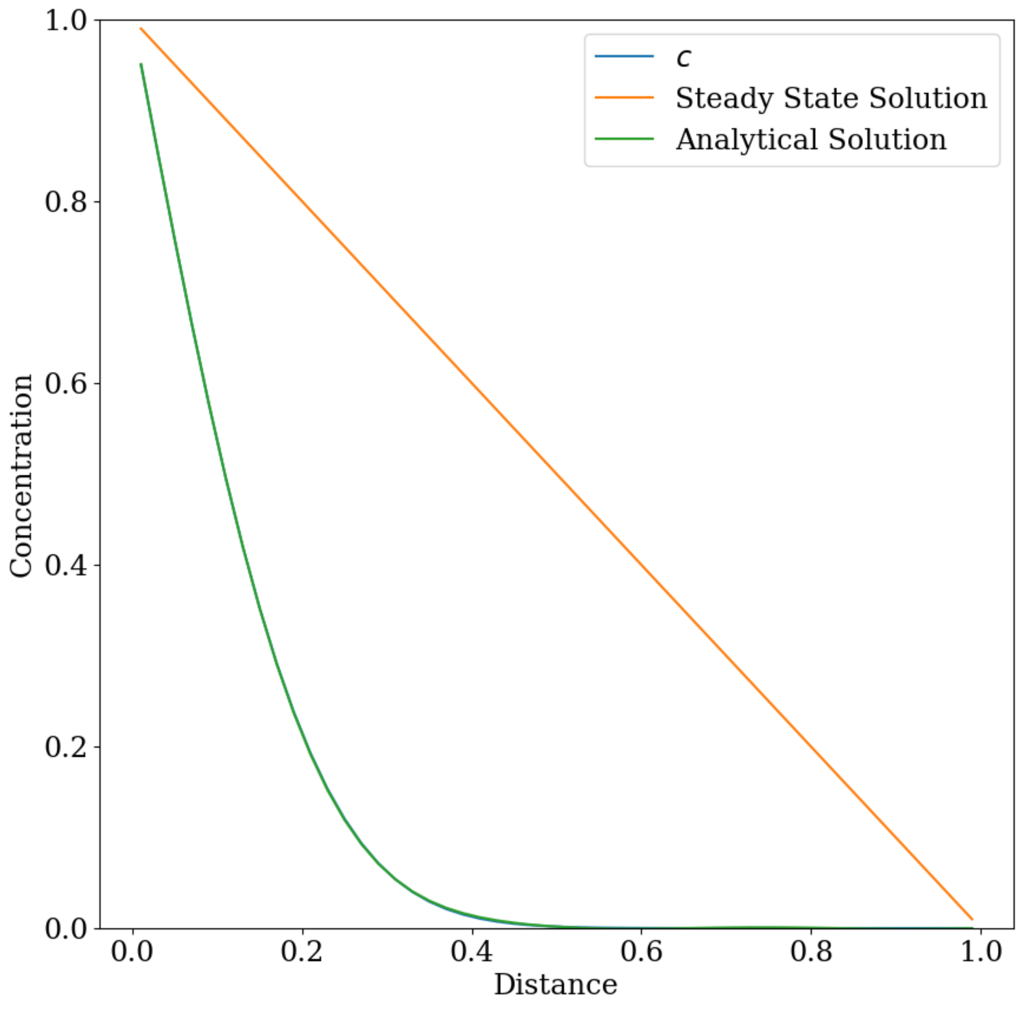
When we shift over to an implicit method, we end up with a much more stable solution even when the timestep is an order of magnitude larger:



Another dynamic worth exploring is the behavior of the truncated analytical solution. We use the timestep for the explicit method and use a time-stride of 1 to see how things appear in the first few timesteps of the simulation. In the first few timesteps, we see high frequency oscillations in the analytical solution:



However, these high frequency modes decay quickly and the truncated analytical solution behaves as expected:



Part 2

The second part of this exercise starts to move on to generating results in a manner that might be usable to you instead of the images we saw previously. At this point, you also have reasonable familiarity with how the diffusion problem can be benchmarked to analytical solutions, so we shall omit that.

Problem statement

We consider the same equation system:

$$\frac{\partial c}{\partial t} = D\nabla^2 c$$

However, we apply the no-flux boundary condition at both ends. This is a Neumann boundary condition.

$$\frac{\partial c}{\partial x}(x=0,t) = 0$$
$$\frac{\partial c}{\partial x}(x=1,t) = 0$$

We then supply an initial condition:

$$c(x, t = 0) = 0.5 + 0.3 \sin(2\pi x)$$

The steady state solution of this problem is as follows:

$$\lim_{t \to \infty} c(x, t) = 0.5$$

Numerical implementation

Most of the code is the same as first problem with a couple of modifications to the boundary conditions and the initial conditions.

Initial conditions:

```
x = mesh.cellCenters[0]
c.value=0.0
c.setValue((0.3*numerix.sin(2.0*x*numerix.pi)) + 0.5)
```

Boundary conditions:

```
c.faceGrad.constrain(0.0, where=mesh.facesLeft)
c.faceGrad.constrain(0.0, where=mesh.facesRight)
```

Output

So far, we have been using the default `Viewer` from `FiPy` which is based on `matplotlib`. This makes it adequate for quickly visualizing the simulation and taking occasional snapshots. But this is not very useful since we don't have access to the actual data. This is where we implement the `TSVViewer` which enables us to output the simulation data as a `.tsv` file which we then convert into a more usable format `.csv`:

```
if __name__ == "__main__":
    vw = TSVViewer(vars=(c))

while t < run_time:
    t += dt
    timestep += 1
    eq.solve(var=c, dt=dt)
    if (timestep % time_stride ==0):
        print ("Beep")
        if __name__ == '__main__':
            vw.plot(filename="%s.tsv"%(t))
if __name__ == '__main__':
    input("Press <return> to proceed...")

# The next section of code is to convert the tsv files into csv which is more
useable:

# We first generate a list of the tsv files in the current folder
tsv_list = glob.glob("./*.tsv")

# Next we use a for loop to go through the list
for tsv in tsv_list:
    csv_filename = tsv.replace("tsv", "csv")
```

```python
    # This is a hack to remove the first line of the tsv file
    with open(tsv, "r") as fin:
        data = fin.read().splitlines(True)
    with open(tsv, "w") as fout:
        fout.writelines(data[1:])

    # We read the tsv file into a dataframe
    csv_table = pd.read_table(tsv, sep="\t")
    csv_table.to_csv(csv_filename)

    # Remove the tsv files
    os.remove(tsv)
```

Once we have the `.csv` files, it is easy to then use the data to generate custom plates based on your needs.

Part 3

The last part of this exercise introduces an additional complexity to the diffusion equation. We consider how a reaction might affect the transient profile of the concentration. We use the case outlined in part 2. The default viewer is used and you should be able to swap things out if you want to export the simulation data as a `.csv` file in the future.

Problem statement

The model equation takes the form:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + \kappa c$$

κ is introduced as the rate constant. Essentially we have introduced a simple first order reaction. `FiPy` and other PDE solvers would typically require us to treat the reaction term as a source term. `FiPy` makes this rather easy.

We impose the same boundary conditions as before and start with the same initial conditions:

$$\frac{\partial c}{\partial x}(x=0,t) = 0$$
$$\frac{\partial c}{\partial x}(x=1,t) = 0$$
$$c(x,t=0) = 0.5 + 0.3\sin(2\pi x)$$

The choice of the rate constant is typically constrained due to the physical nature of the parameter, but here we have the flexibility to experiment with different values of κ.

Numerical implementation

We first add a value for the rate constant:

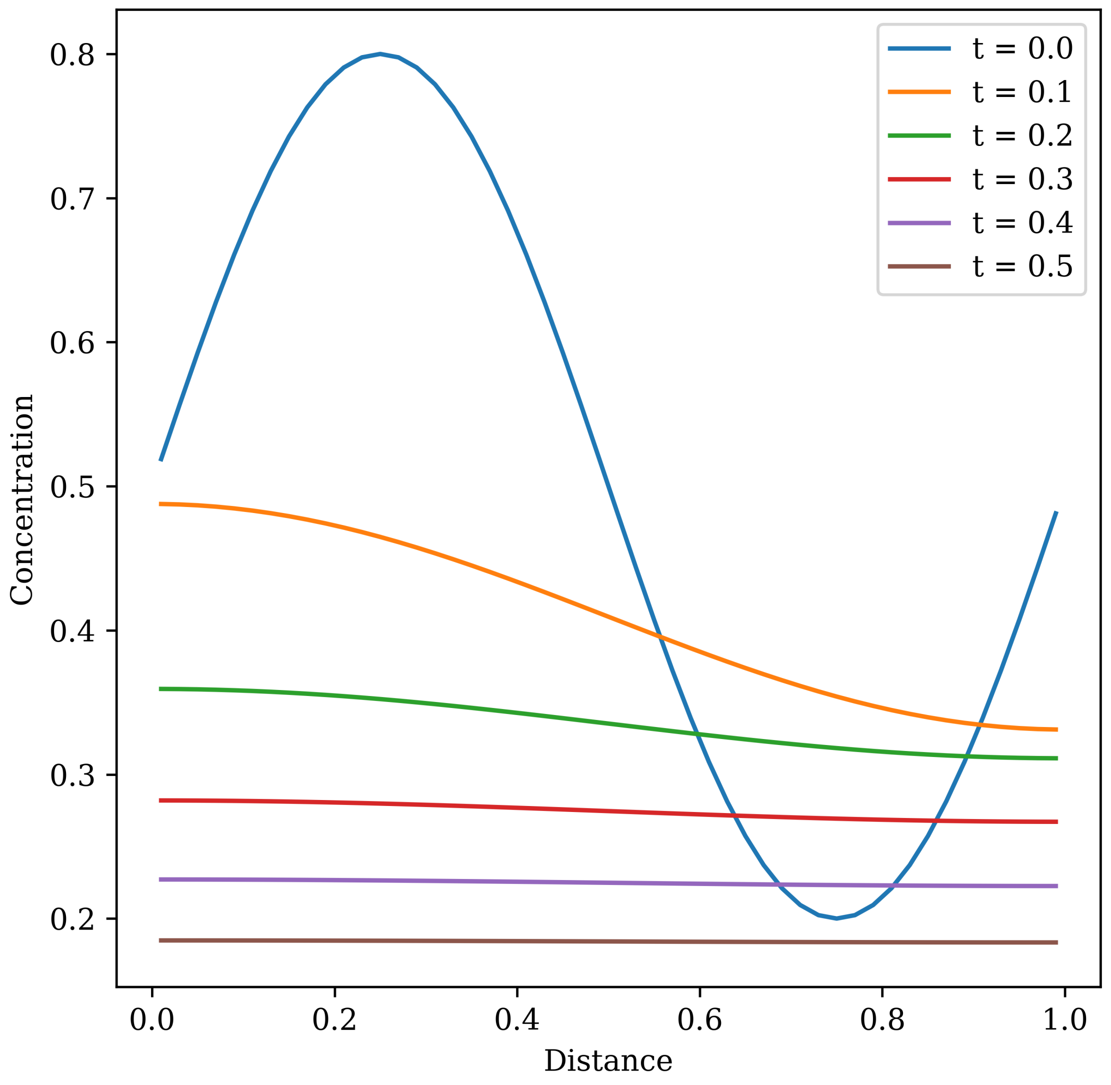
```python
# Provide value for diffusion and reaction coefficient
D = 1.0
k = -2.0
```

We then modify the equation to include the reaction term:

```
eq = TransientTerm() == DiffusionTerm(coeff=  D)  + ImplicitSourceTerm(coeff=k)
```

Results

We combine the results from parts 2 and 3 cause they are quite similar. If you plot the results from multiple timesteps together, you would get something like this:


t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
Concentration
Distance