Transport in Porous Media - From Laboratory Scale to Field Scale

MOD510: Mandatory project #2

Deadline: 11 October 2020 (23:59)

Sep 25, 2020

Learning objectives. By completing this project, the student will:

- Learn to use matrix equation to do linear regression, and estimate model parameters from experimental data
- Solve the diffusivity equation using an implicit solution method
- Compare sparse matrix solvers with standard dense solvers
- Estimate reservoir properties, such as reservoir permeability, and reservoir volume using a model for the well pressure and data from a well test

Abstract

In this project we want to show you how we extract information from both lab and field data. As many of you do not have a proper background in the study of transport in porous media, we have included thorough introduction and derivation of the necessary equations. If you want to get started quickly with the coding, without having to read and understand all the theory first, we suggest that you jump straight to equations (19), (20), and (21). Show how you can derive these equations from the earlier equations involving the r-coordinate, equations (11), (12), and (13). Next, implement a solver that solves the system *iteratively*. Start from a known initial pressure p_i, and then compute the pressure distribution at a sequence of times Δt, $2\Delta t$, $3\Delta t$ etc. After you have finished implementing the solver, you can go back and do exercise 1, and then 3 and 4 (or 3, 4 and then 1).

A final tip.

If you want to copy and paste some of the provided example code, it is best to use the html version of the project. If you copy from the pdf, some of the symbols might not be pasted correctly.

1 Introduction

Reservoirs are large pieces of porous rocks (usually sediments) that are buried underground. The size of a reservoir may vary a lot: typically by as much as 100-1000 m horizontally, and 10-100 m vertically. A sedimentary rock consists of many small grains, like sand on a beach. During geological time the sand grains have become cemented together, but still there is plenty of room for oil, water and gas to occupy the pores between the grains. Usually 20-30 % of a rock is void space. The absolute *permeability* is a measure of how easily fluids flow through the rock. It plays the same role as does the inverse of resistance in electromagnetism. For example, a high permeability means that fluids flow easily through the rock, just as a low resistance conductor transmits electric current easily. What actually drives the fluid flow are *pressure differences*, and if we continue the analogy with electricity, these pressure differences play the same role as voltage gradients inside a conductor.

When a reservoir is opened for production, the pores close to the producer might clog due to the mobilization of particles, and because of chemical reactions taking place etc. [2]. If this happens, the permeability will be reduced. To monitor changes in reservoir permeability, a key tool is *well pressure testing* [3], which is the topic of this project. It turns out that by studying the time-dependence of well pressures during production, one can actually learn a lot about the properties of a reservoir. This information is clearly extremely relevant for an oil company, but it of equal importance in geothermal applications, as well as during groundwater monitoring. Reservoirs are huge ($\sim$ km size), and the well is only about half a foot, but still the information coming from a single producing well can provide valuable information about, e.g., the size of the reservoir, or about formation damage close to well. If one finds that the reservoir has become clogged, one has to implement some kind of intervention, after which one can measure the effect of the intervention by repeating the well test.

Another factor that may be important to consider is reservoir compaction. As fluids are produced, the pressure in the formation drops, and this could lead to large increases in the effective stress in the reservoir, and to seabed subsidence. An example of this is Ekofisk, Norway's largest oil field, where platforms were observed to sink by several meters [4]; even if the reservoir itself is located more than 3 km below the seabed! A similar process has been taking place in Venice, which is currently sinking. Part of the recorded subsidence of the city has been attributed to groundwater pumping operations [5].

2 Exercise 1

Darcy's law [1] is frequently used to calculate the relationship between flow rate and pressure when a fluid phase is moving inside a permeable medium. If we neglect gravity and if we use a consistent set of units (e.g., SI-units), Darcy's law can be expressed in coordinate-free form as

$$\mathbf{u} = -\frac{\mathbf{k}}{\mu}\nabla p\,, \tag{1}$$

where $\mathbf{u}$[L/T] is the Darcy velocity, $\mathbf{k}$[L^2] is the absolute permeability tensor, μ[ML/T] is the fluid viscosity, and p[$\mathrm{M/LT}^2$] is fluid pressure.

Part 1. Consider a homogeneous cylindrical core sample of length $L = 7$ cm, diameter $d = 3.8$ cm. Imagine that we have first saturated the sample with water, and that we subsequently perform a sequence of injection tests in which water is pumped through the core at a constant volumetric flow rate, Q. For each flow rate, we record the (steady-state) pressure difference across the core, Δp. Darcy's law then implies that:

$$Q = \frac{kA}{\mu} \cdot \frac{\Delta P}{L}\,, \tag{2}$$

where $A = \pi r^2$ is the cross-sectional area of the core.

- Read the data stored in the file `core_flood.dat`. Make a scatter plot of ΔP versus Q.

Part 2. At room temperature, the viscosity of water is approximately 1 cP (or 10^{-3} Pa $\cdot$ s in SI-units).

- Use linear regression to estimate the slope of the data, and estimate the permeability from equation (2). If you use the given units (atm, ml/s, cm, cP) the permeability, k, will have unit of Darcy (1 D $\simeq 10^{-12}$ m^2). When doing the linear regression formulate the problem as a linear problem (see example in the lecture notes), and use a matrix solver do find the regression coefficients.

3 Theoretical background for the rest of the project

The time-dependent pressure development in a reservoir can be studied by setting up a *fluid mass balance*:

$$\frac{\partial}{\partial t}(\phi\rho) = -\nabla \cdot (\rho\mathbf{u})\,, \tag{3}$$

where ρ[$\mathrm{M/L}^3$] is the fluid density, and ϕ is the porosity; the void fraction of the reservoir that permits fluid flow. We are going to assume that

- Darcy's law accurately describes fluid motion,
- gravity and capillary forces can be neglected,
- viscosity is independent of pressure,
- the permeability tensor is isotropic and homogeneous, so that it can be represented by a single constant value, and
- fluid compressibility is low (i.e., we do not consider gases)

Inserting Darcy's law, equation (1), and then expanding both sides of (3), yields:

$$\phi \frac{\partial \rho}{\partial t} + \rho \frac{\partial \phi}{\partial t} = \frac{k}{\mu} \left(\nabla \rho \cdot \nabla p + \rho \nabla^2 p \right) . \tag{4}$$

The left side contains the *fluid density*, ρ. We want to replace the density with pressure. This can be done by using the fluid compressibility:

$$c_f = -\frac{1}{V} \cdot \frac{\partial V}{\partial p} = \frac{1}{\rho} \cdot \frac{\partial \rho}{\partial p} . \tag{5}$$

Similarly, we want to replace the derivative of porosity by using the rock compressibility:

$$c_r = \frac{1}{\phi} \cdot \frac{\partial \phi}{\partial p} . \tag{6}$$

Both c_f and c_r are measured in units of inverse pressure (LT^2/M). Equation (4) can therefore be written as

$$\rho \phi c_t \cdot \frac{\partial p}{\partial t} = \rho \frac{k}{\mu} \left(c_f \nabla p \cdot \nabla p + \nabla p^2 \right) . \tag{7}$$

where $c_t = c_f + c_r$ is the total compressibility. Since we have assumed our fluid to be relatively incompressible (this is the case for water), it is often a very good approximation to drop the term proportional to c_f on the right-hand side:

$$\rho \frac{k}{\mu} c_f \nabla p \cdot \nabla p \approx 0 . \tag{8}$$

If we use this approximation, and if we further define

$$\eta = \frac{k}{\mu \phi c_t} , \tag{9}$$

we end up with the *diffusivity equation*.

The diffusivity equation.

The diffusivity equation describes how pressure waves travel in a porous rock.

$$\frac{\partial p}{\partial t} = \eta \nabla^2 p \,. \tag{10}$$

The factor η can be consider a diffusion constant, in analogy with molecular diffusion. Whenever you have a diffusion equation, you can use the relation

$$\text{diffusion constant} \approx \frac{\text{length}^2}{4 \cdot \text{time}} \,.$$

to give an order of magnitude estimate the speed of the "diffusion". For example, if $\eta = 2.5$ m^2/s and the reservoir radius is 1 km, it will take approximately

$$\frac{(1000 \text{ m})^2}{4 \cdot 2.5 \text{ m}^2/\text{s}} \simeq 1.2 \text{ days}$$

for the pressure wave to reach the outer boundaries of the reservoir.

3.1 Diffusivity equation in radial flow

If we use cylindrical coordinates, and if we assume perfectly radially symmetric flow, the diffusivity equation reduces to

$$\frac{\partial p}{\partial t} = \eta \frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial p}{\partial r} \right) . \tag{11}$$

To solve this equation we need some boundary conditions. Later when doing numerical calculations, we shall assume that there is a vertical well placed in the middle of the reservoir, with well radius $r = r_w$. The well is set to produce fluids at a *constant flow rate*, $Q > 0$, in other words:

$$\frac{2\pi h k r}{\mu} \frac{\partial p}{\partial r}|_{r=r_w} = Q \,. \tag{12}$$

Far away from the well, at a distance $r = r_e$, we shall assume that there is *no* flow of fluids:

$$\frac{\partial p}{\partial r}|_{r=r_e} = 0 \,. \tag{13}$$

Note that equation (12) is a special case of Darcy's law, equation (1); this can be seen by inserting $\nabla = \partial/\partial r$, and $Q = -uA = -u \cdot 2\pi r h$ [1].

3.2 Analytical solutions to the radial diffusivity equation?

The diffusivity equation is extremely hard to solve exactly, and explicit formulas can be found only in idealized cases. One such solution is called the *line source*

[1] We use the sign convention that $Q > 0$ for production from the well.

solution. It employs slightly different boundary conditions than formulated above:

- It considers the well to be infinitely small (i.e. a line), thus neglecting the influence of a finite well radius, and
- The reservoir is assumed to be infinite. Moreover, rather than assuming no flow far away from the well, the line source solution assumes a constant pressure p_i, the initial reservoir pressure, at infinity.

In mathematical terms, the boundary conditions for the line source solution are:

$$\lim_{r\to 0}\left(\frac{2\pi hk}{\mu}r\frac{\partial p}{\partial r}\right) = Q\,, \tag{14}$$

and

$$\lim_{r\to\infty} p(r,t) = p_i\,. \tag{15}$$

The exact solution to equations (11), (14), and (15) is[2]

$$p(r,t) = p_i + \frac{Q\mu}{4\pi kh}\cdot\mathcal{W}\left(-\frac{r^2}{4\eta t}\right)\,, \tag{16}$$

where $\mathcal{W}$ is the *exponential function*, defined in terms of an integral:

$$\mathcal{W}(x) = \int_{-\infty}^{x}\frac{e^u}{u}\,du\,. \tag{17}$$

In Python this function can easily be computed by using the `sp.special.expi` function.

4 A Crash Course on Classes

You are free to code the project as you see fit, but regardless of how you do it you should strive to a) write clear, concise code and b) reduce unnecessary code repetition. One way to achieve that is to use classes, because classes provide a mechanism for wrapping parts of your code into objects, which you can afterwards reuse in several places.

If you want to use classes for this project, there are really only a couple of things you need to know. First, all of your classes should include a special function called `__init__`, in which you declare the variables (attributes) you wish an instance / object of the class to keep track of. Second, all of these attributes should start with the keyword `self`, followed by a dot and then the actual attribute name. Third, functions inside a class should have `self` as the first function argument (when defining the function). All of this is best understood via an example:

[2] You do not have to show this.

```
class Person:
"""
A class representing custom objects of the type Person.
"""
    def __init__(self, name, age):

        self.name = name
        self.age = age
        self.species = 'Human'

    def greeting(self):
        print('Hello, my name is {}.'.format(self.name))
```

This first few lines tells Python that in order to create a new *object* of the type `Person`, you have to provide the name and age of the person as input arguments, for example:

```
p1 = Person('John', 36)
```

When executing the above code, Python automatically invokes the `__init__` method, and sets the object attributes `self.name` and `self.age` based on the user input. In addition, we see that the object `p1` has a third attribute, `species`, which is hard-coded inside `__init__`, and is therefore not needed as input (this is a good way to set constants that you expect to re-use across parts of your program).

Once we have created a particular Person object, we can call the other function we have defined, namely `greeting`:

```
p1.greeting()
```

Note that we do not include `self` as an input argument when *calling* the function; only when we define it [3] If you want to access variables belonging to an instance of the class, you can also access it via "dot-notation", e.g.:

```
print(p1.name)
print(p1.age)
```

Why we suggest using classes.

The main advantage of using classes in this project is that you can define all the numerical model parameters once and for all within `__init__`, and then simply re-use them inside all your other functions.

However, you are not required to use classes to complete this project. If it seems too advanced at the moment, just code using ordinary functions. You should still try to make your functions re-usable, though!

[3] This is to let Python know that the function should be called on a particular object.

5 Exercise 2

You are going to implement a numerical solver for the radial diffusivity equation. To avoid unnecessary complications with units, you may use the following class definition as a starting point:

```
class RadialDiffusivityEquationSolver:
    """
    A finite difference solver for the radial diffusivity equation.
    We use the coordinate transformation y = ln(r/rw) to set up and
    solve the pressure equation.

    The solver uses SI units internally, while "practical field units"
    are required as input.

    Except for the number of grid blocks / points to use, all class
    instance attributes are provided with reasonable default values.

    Input arguments:

        name                                  symbol      unit
        ----------------------------------------------------------------
        Number of grid points                  N          dimensionless
        well radius                            rw         ft
        extent of reservoir                    re         ft
        height of reservoir                    h          ft
        permeability                           k          mD
        porosity                               phi        dimensionless
        fluid viscosity                        mu         mPas (cP)
        total compressibility                  ct         1 / psi
        constant flow rate at well             Q          bbl / day
        initial reservoir pressure             pi         psi
        constant time step                     dt         days
        maximal simulation time           max_time        days
        ----------------------------------------------------------------
    """

    def __init__(self, N, rw=0.328, re = 100000., h=8.0,
                 k=500, phi=0.1, mu=1.0, ct=17.7e-6,
                 Q=1000, pi=2000, dt=0.1, max_time=10):

        # Unit conversion factors (input units --> SI)
        self.ft_to_m_ = 0.3048
        self.psi_to_pa_ = 6894.75729
        self.day_to_sec_ = 24*60*60
        self.bbl_to_m3_ = 0.1589873

        # Grid
        self.N_ = N
        self.rw_ = rw*self.ft_to_m_
        self.re_ = re*self.ft_to_m_
        self.h_ = h*self.ft_to_m_

        # Rock and fluid properties
        self.k_ = k*1e-15 / 1.01325
        self.phi_ = phi
        self.mu_ = mu*1e-3
        self.ct_ = ct / self.psi_to_pa_

        # Initial and boundary conditions
        self.Q_ = Q*self.bbl_to_m3_ / self.day_to_sec_
        self.pi_ = pi*self.psi_to_pa_
```

```
        # Time control for simulation
        self.dt_ = dt*self.day_to_sec_
        self.max_time_ = max_time*self.day_to_sec_
        self.current_time_ = 0.
```

READ THIS FIRST: A note on default model parameters.

Whether you code with classes or not, we shall assume that the default model input parameters are those shown in the above class definition. For example, the default value for viscosity is 1 cP, the default well radius is 0.328 ft, and so on.

Part 1.

- Implement a Python function called `line_source_solution` that returns the analytical line source solution, equation (16), as a function of r and t. The function should work both in the case when r and t are floating-point numbers, and when one of them is a float and the other is a NumPy array.
- Implement another function called `plot_line_source_solution` that makes a plot of the reservoir pressure versus radial distance at a specific point in time. This second function should take the time in question as an input argument, and it should call `line_source_solution` in order to compute the pressures.

For those of you who choose to code using classes, the functions should be member functions of the solver class; otherwise, use standalone functions with appropriate input arguments and default values.

The following code snippet shows how the above functions might work in practice (if following the class approach):

```
# Create a solver object. Use only default arguments:
solver = RadialDiffusivityEquationSolver(N=100)

# We can calculate the analytical solution at a specific point
# in space and time:
analytical_sol = solver.line_source_solution(10.0, 42.0)

# ...or at a specific position, but at several times:
time_array = np.linspace(0.0, 50, 51)
analytical_sol = solver.line_source_solution(10.0, time_array)

# ...or at a specific time, but at multiple locations:
position_array = np.linspace(0.1, 10, 11)
analytical_sol = solver.line_source_solution(position_array, 42.0)
```

Part 2.

- Plot the line source solution as a function of r assuming default model input parameters (i.e., at $t = 10$ days).

- What do you see? Does the curve approach the expected limits?

Part 3. Next, we are going to use a clever coordinate transformation. Define

$$y(r) = \ln \frac{r}{r_w}, \tag{18}$$

where, as before, r_w is the radius of the well.

- Show that in the new y-coordinate, equations (11), (12), and (13) can be rewritten as:

$$\frac{\partial p}{\partial t} = \eta \frac{e^{-2y}}{r_w^2} \frac{\partial^2 p}{\partial y^2}\,, \tag{19}$$

$$\frac{\partial p}{\partial y}|_{y=y_w} = \frac{Q\mu}{2\pi hk}\,, \tag{20}$$

$$\frac{\partial p}{\partial y}|_{y=y_e} = 0\,. \tag{21}$$

Part 4. We are now going to discretize equation (19). We start by dividing the total flow domain into N equally sized grid blocks in the y-coordinates, see figure 1 for an illustration. The numerical pressure solution will be calculated at the *midpoints* (centers) of the y-blocks; note that this does *not* correspond to the center of the r-blocks!

- Make a NumPy array of size N where the elements are the midpoint coordinates of each y-interval. You should give the array a meaningful name, for example: `y_centres_` or `y_midpoints_`.
- Create another array holding the r-coordinates of the grid points. Again, this will be a NumPy array of size N (hint: invert equation (18).)
- Plot the line solution using your new r-coordinates. Use a logarithmic scale on the x-axis. Comment on the plot. Are the r-points spaced out as they should?

If you code your solver as a class, the arrays should be attributes that are initialized within the `__init__` function. However, the plotting should *not* be included inside `__init__`.

Part 5. Let i denote an arbitrary grid point inside our simulation domain. To discretize the left-hand side of (19) for point i, we use the following approximation:

$$\frac{\partial p}{\partial t} = \frac{p(y_i, t+\Delta t) - p(y_i, t)}{\Delta t} + \mathcal{O}(\Delta t) \equiv \frac{p_i^{n+1} - p_i^n}{\Delta t} + \mathcal{O}(\Delta t). \tag{22}$$

We have introduced the following short-hand notation: $p_i^{n+1} \equiv p(y_i, t+\Delta t)$, and $p_i^n \equiv p(y_i, t)$. Next, we are going do discretize the spatial (right-hand side) term

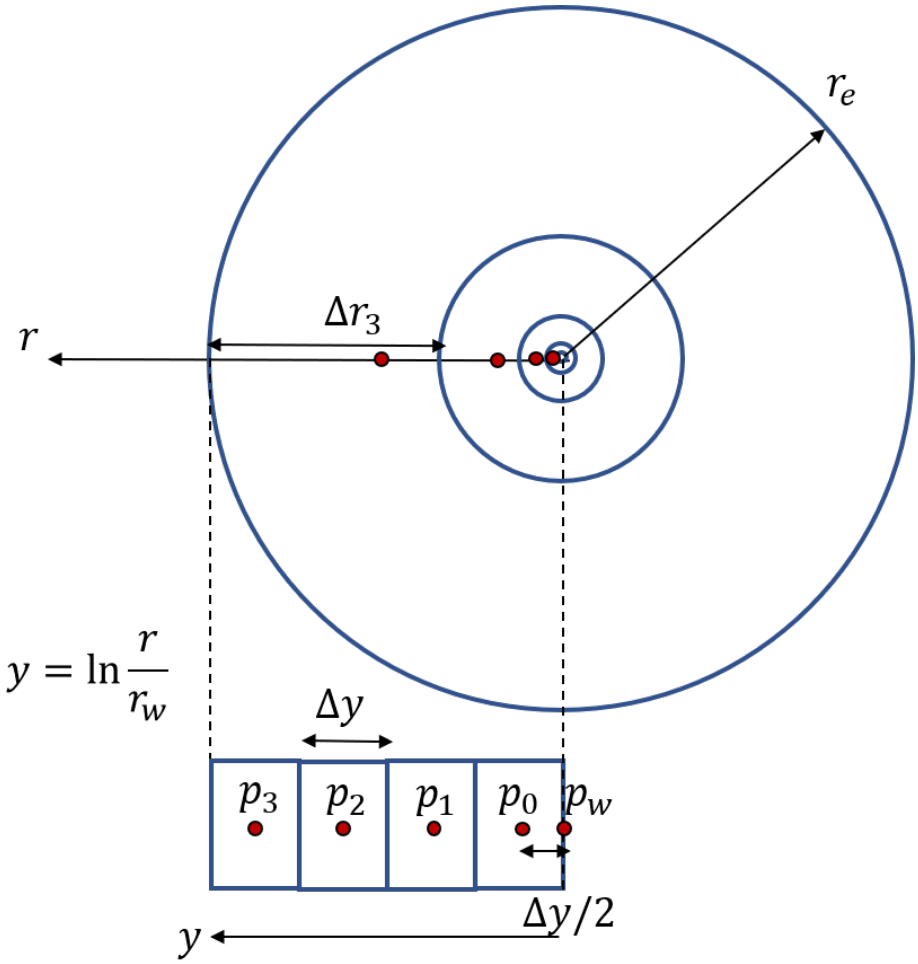

Figure 1: Sketch of the coordinate transformation $r \to y$. Note that the pressure is always evaluated at the center of the y-blocks.

of Eq. (19). However, before we can approximate partial derivatives with respect to y, we have to decide what value to use for the time; after all, the pressure is a function of both space and time, $p = p(r, t)$.

If we select the time t, corresponding to the pressure solutions p_i^n, we say that we use an *explicit scheme*. On the other hand, if we use $t + \Delta t$, we say that we have an *implicit* scheme. It turns out that the implicit scheme is much more numerically stable, hence we choose that option here:

$$\frac{\partial^2 p}{\partial y^2} = \frac{p_{i+1}^{n+1} + p_{i-1}^{n+1} - 2p_i^{n+1}}{\Delta y^2} + \mathcal{O}(\Delta y^2) \tag{23}$$

Numerical scheme for interior grid points.

By neglecting higher order terms, the finite difference discretization of equation (19) becomes

$$\frac{p_i^{n+1} - p_i^n}{\Delta t} = \eta \cdot \frac{e^{-2y_i}}{r_w^2} \cdot \frac{p_{i+1}^{n+1} + p_{i-1}^{n+1} - 2p_i^{n+1}}{\Delta y^2} \tag{24}$$

- Suppose that we use $N = 4$ grid points. Write down equation (24) for all four points, $i = 0, 1, 2, 3$.

Note that some of the equations contains p_{-1} and p_4, both of which are outside of the physical simulation domain!

Part 6.

- Again, let $N = 4$. Replace the derivatives in the boundary conditions equation (20) and (21) by the (second order) central finite difference approximation to eliminate p_{-1} and p_4. Show that we now can formulate the equations as the following matrix equation

$$\underbrace{\begin{pmatrix} 1+\xi_0 & -\xi_0 & 0 & 0 \\ -\xi_1 & 2+\xi_1 & -\xi_1 & 0 \\ 0 & -\xi_2 & 2+\xi_2 & -\xi_2 \\ 0 & 0 & -\xi_3 & 1+\xi_3 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} p_0^{n+1} \\ p_1^{n+1} \\ p_2^{n+1} \\ p_3^{n+1} \end{pmatrix}}_{\mathbf{p}^{n+1}} = \underbrace{\begin{pmatrix} p_0^n - \xi_0\beta \\ p_1^n \\ p_2^n \\ p_3^n \end{pmatrix}}_{\mathbf{p}^n - \mathbf{d}}, \tag{25}$$

where we have defined

$$\xi_i \equiv \frac{\eta e^{-2y_i} \Delta t}{r_w^2 \Delta y^2},$$

and

$$\beta \equiv \frac{Q\mu\Delta y}{2\pi kh}.$$

If we know the initial pressure, $\mathbf{p}^0$, in the reservoir at $t = 0$, the pressure distribution at $t = \Delta t$, $\mathbf{p}^1$, is found by solving equation (25). Schematically:

$$\begin{aligned} \mathbf{p}^1 &= \mathbf{A}^{-1}\left(\mathbf{p}^0 - \mathbf{d}\right), \\ \mathbf{p}^2 &= \mathbf{A}^{-1}\left(\mathbf{p}^1 - \mathbf{d}\right), \\ \mathbf{p}^3 &= \mathbf{A}^{-1}\left(\mathbf{p}^2 - \mathbf{d}\right), \\ &\text{etc.} \end{aligned}$$

6 Exercise 3 Implement the full numerical solution

Once you have understood what goes on in special case $N = 4$, it is time to tackle the general situation (*any* choice of N). We assume the pressure is the same everywhere at time zero:

$$p_0^0 = p_1^0 = p_2^0 \ldots = p_i \,.$$

After setting the initial condition, we need to implement a time loop and solve the discretized equations (25) repeatedly for each time step: Δt, $2\Delta t$, $3\Delta t$, etc. As before, we need to take special care when coding the equations for the first and last grid points, $i = 0$ and $i = N - 1$.

Part 1. Start by making three different Python functions:

1. A function `setup_matrix`, in which you store the coefficient matrix as a regular NumPy matrix[4].
2. Another function `setup_sparse_matrix`, where only the non-zero elements of the matrix are stored. For ideas on how to do this, see the official SciPy documentation, e.g.: `scipy.sparse.diags`.
3. A function `rhs`, which calculates the right-hand side of the linear equation system.

Part 2. Note that as long as we use a constant Δt, we only need to set up the matrices once; when initializing the model. On the other hand, the right-hand side vector must be updated each time step.

Finish implementing the numerical solver for the radial diffusivity equation:

- Write a "top-level" function that executes all the necessary steps. The simulator should be able to select at run-time whether the dense and sparse matrix solver is going to be used.
- Solve the diffusivity equation numerically assuming default model input parameters and $N = 100$.
- Compare your numerical solutions with the analytical line source solution. Check that you get the same result regardless of whether you use a dense or sparse matrix solver.

Part 3.

- Increase the number of grid points to $N = 1000$. Compare the efficiency of the two linear solvers.

[4] An $N \times N$ NumPy array.

7 Exercise 4 Compare with well data

Although we have measured properties of a rock sample in the lab, the reality in the reservoir might be very different. In this exercise we are going to study data from a well test. During a well test, the production engineer starts to produce from the reservoir, while at the same time monitoring how the well pressure changes as a function of time.

Part 1.

- Read the data from `well_test.dat` into a Pandas DataFrame.
- Plot the experimental well pressures versus time. Describe what you see.

Part 2. So far we have only calculated the pressure distribution *inside the reservoir*; the actual observable well pressure is missing! However, we can estimate the well pressure by discretizing equation (21):

- Use a first-order finite difference approximation to determine the well pressure

(Hint: Use Taylor's formula with step-size $\Delta y/2$.)

Part 3.

- Further extend your program by calculating, and storing, the estimated well pressure at each time step in your simulation.

Part 4.

- Try to manually match your numerical model to the well test data (Hint: use log scale on the x-axis)
- To be able to do so, you might have to modify one or both of the following input parameters: k, p_i.
- Make a plot in which you show 1) the experimental well test data, 2) your numerical well pressure solution, and 3) the corresponding line source solution (i.e., at $r = r_w$)

Part 5. Notice that towards the end of the well test, the pressure starts to rapidly decline. This is actually a sign that the pressure wave has reached the end of the reservoir.

- Use the permeability, k, and initial pressure, p_i, you found in the previous task, but change r_e to to match the start of the late decline period.
- Use the value you found for r_e to calculate the volume of fluids in the reservoir.

Part 6. (OPTIONAL) Use a non linear solver in Python to estimate p_i, k, r_e as a best fit to the data from the well pressure test.

8 Guidelines for project submission

The project has to be handed in as a notebook, though you can submit an additional PDF if you want. You should bear the following points in mind when working on the project:

- Start your notebook by providing a short introduction in which you outline the nature of the problem(s) to be investigated.
- End your notebook with a brief summary of what you feel you learned from the project (if anything). Also, if you have any general comments or suggestions for what could be improved in future assignments, this is the place to do it.
- All code that you make use of should be present in the notebook, and it should ideally execute without any errors (especially run-time errors). If you are not able to fix everything before the deadline, you should give your best understanding of what is not working, and how you might go about fixing it.
- Functions that you code *should be reusable, never copy and paste code.* Copy and paste is a strong indication that you instead should define function.
- If you use an algorithm that is not fully described in the assignment text, you should try to explain it in your own words. This also applies if the method is described elsewhere in the course material.
- In some cases it may suffice to explain your work via comments in the code itself, but other times you might want to include a more elaborate explanation in terms of, e.g., mathematics and/or pseudocode.
- In general, it is a good habit to comment your code (though it can be overdone).
- When working with approximate solutions to equations, it is always useful to check your results against known exact (analytical) solutions, should they be available.
- It is also a good test of a model implementation to study what happens at known 'edge cases'.
- Any figures you include should be easily understandable. You should label axes appropriately, and depending on the problem, include other legends etc. Also, you should discuss your figures in the main text.

- It is always good if you can reflect a little bit around *why* you see what you see.

References

[1] M. King Hubbert. Darcy's law and the field equations of the flow of underground fluids. *Transactions of the AIME*, 207(01):222–239, 1956.

[2] Roland F. Krueger. An overview of formation damage and well productivity in oilfield operations: an update. In *SPE California Regional Meeting*. Society of Petroleum Engineers, 1988.

[3] John Lee, John B. Rollins, and John Paul Spivey. *Pressure Transient Testing*. Richardson, Tex.: Henry L. Doherty Memorial Fund of AIME, Society of Petroleum Engineers, 2003.

[4] AM Sulak and J. Danielsen. Reservoir aspects of Ekofisk subsidence. In *Offshore Technology Conference*. Offshore Technology Conference, 1988.

[5] Luigi Tosi, Pietro Teatini, and Tazio Strozzi. Natural versus anthropogenic subsidence of venice. *Scientific reports*, 3(1):1–9, 2013.