1 MOD510 project feedback

Project 2:

Group number: 5

Abstract (4.5 / 5.0)

- Informative, nice overview of the project.

Introduction (4.0 / 5.0)

- Introduction should set the context of the project and transition into what the project is generally about, without being too technical, next time when you use figures from other sources - cite the source.

Reflections (10.0 / 10.0)

- Nice to see that you enjoyed the project.

Conclusion (9.0 / 10.0)

- Nice work. Include your numerical findings for full marks.

Figures,tables (10.0 / 10.0)

- Good use of figures and tables.

References (3.0 / 5.0)

- References are copied from project document, no new references.

CodingQuality (10.0 / 10.0)

- Very good coding, easy to read and due to very good function name, it is ok to skip doc string on most functions.

Analysis (20.0 / 20.0)

- E1,E2:All good. E4: Impressive report

Implementation (25.0 / 25.0)

- All good.

Bonus (5.0 / 5.0)

- Well done.

Overall points: 100.5 / 100.0

2 Solutions to selected exercises

2.1 Finite difference scheme with truncation error

We perform two Taylor expansions about the point y_i:

$$p(y_i + \Delta y) = p(y_i) + \Delta y \cdot p'(y_i) + \frac{\Delta y^2}{2} \cdot p''(y_i) + \frac{\Delta y^3}{6} \cdot p^{(3)}(y_i) + \frac{\Delta y^4}{24} \cdot p^{(4)}(\xi_{i,1}) \,, \tag{1}$$

$$p(y_i - \Delta y) = p(y_i) - \Delta y \cdot p'(y_i) + \frac{\Delta y^2}{2} \cdot p''(y_i) - \frac{\Delta y^3}{6} \cdot p^{(3)}(y_i) + \frac{\Delta y^4}{24} \cdot p^{(4)}(\xi_{i,2}) \,. \tag{2}$$

Assuming the pressure solution is sufficiently smooth, Taylor's theorem with remainder says that the above relations are *exact*, but unfortunately the constants $\xi_{i,1} \in (y_i, y_i + \Delta y)$ and $\xi_{i,2} \in (y_i - \Delta y, y_i)$ are unknown. By the intermediate value theorem, there exists a third, unknown constant η_i in-between $\xi_{i,1}$ and $\xi_{i,2}$ such that:

$$p^{(4)}(\eta_i) = \frac{p^{(4)}(\xi_{i,1}) + p^{(4)}(\xi_{i,2})}{2} \,. \tag{3}$$

Define $p_i \equiv p(y_i)$, $p_{i+1} \equiv p(y_i + \Delta y)$, and $p_{i-1} \equiv p(y_i - \Delta y)$. By adding equations (**??**) and (**??**) and rearranging, we get:

$$\frac{p_{i+1} + p_{i-1} - 2p_i}{\Delta y^2} - \frac{\Delta y^2}{12} \cdot p^{(4)}(\eta_i) = 0 \,. \tag{4}$$

For $N = 4$:

$$\frac{p_1 + p_{-1} - 2p_0}{\Delta y^2} - \frac{\Delta y^2}{12} \cdot p^{(4)}(\eta_0) = 0,$$
$$\frac{p_2 + p_0 - 2p_1}{\Delta y^2} - \frac{\Delta y^2}{12} \cdot p^{(4)}(\eta_1) = 0,$$
$$\frac{p_3 + p_1 - 2p_2}{\Delta y^2} - \frac{\Delta y^2}{12} \cdot p^{(4)}(\eta_2) = 0,$$
$$\frac{p_4 + p_2 - 2p_3}{\Delta y^2} - \frac{\Delta y^2}{12} \cdot p^{(4)}(\eta_3) = 0\,. \tag{5}$$

In general, we cannot assume the same η works for all grid points. However, is worth noting that in this case, the analytical solution is a first order polynomial, which implies that all derivatives of order ≥ 2 disappear. Therefore, the finite difference scheme given above has no truncation error at interior points:

$$p^{(4)}(\eta) = 0 \text{ for all } \eta\,. \tag{6}$$

2.2 Errors introduced by the boundary conditions

Errors may still be introduced by the boundary conditions. For the boundary condition at the well, a central difference approximation yields

$$\alpha = \frac{dp}{dy}(y = y_w) = \frac{p_0 - p_{-1}}{\Delta y} + \mathcal{O}(\Delta y^2)\,. \tag{7}$$

A similar analysis as conducted above shows that, since the third derivative of pressure vanishes, the pre-factor in the truncation error term is zero.

As for the outer boundary, note that the pressure at the fictive ghost point outside of the edge of the reservoir domain is

$$p_N = p(y_e + \frac{\Delta y}{2}) = p(y_e) + p'(y_e)\frac{\Delta y}{2} + \frac{1}{2}\frac{d^2 p}{dy^2}\left(\frac{\Delta y}{2}\right)^2 + \text{ higher order terms}\,, \tag{8}$$

$$= p_e + p'(y_e)\frac{\Delta y}{2}\,. \tag{9}$$

where we again have used that we are assuming steady-state flow. Thus, the truncation error due to the "lazy" approximation is precisely

$$\text{error in approximation of } p_e = p_N - p_e = p'(y_e)\frac{\Delta y}{2} = \alpha\frac{\Delta y}{2}\,. \tag{10}$$

By applying Gaussian elimination one can easily verify that the truncation error of the lazy scheme is the same at each point:

$$\epsilon_k = |p_k^{num} - p_k^{ana}| = \alpha\frac{\Delta y}{2}\,, \tag{11}$$

for $k = 0, 1, \ldots, N-1$. Theoretically, increasing N by a factor of 10 should therefore decrease the error by exactly the same amount, regardless of which point we choose for the comparison.

Of course, there will also be round-off errors. For the non-lazy scheme, this is the only source of error.

2.3 Evaluation of error at a single point

To compare the numerical pressure solver to the analytical solution, it is most accurate to do so in a *single physical point*. One method to always be able to select the same physical point is to increase N by a factor of three each time; then, whenever the grid is refined, the previous grid points are then kept.

Luckily, for this particular model the truncation error is the same for each grid point (see above), hence even if we compare different points it should not matter for the analysis, but this is not the case in general!

2.4 Brief note on the line-source solution

It is important to note that the line-source solution does not solve the same problem as does the numerical pressure solver:

- It neglects the finite size (radius) of the well.
- It assumes an infinite reservoir.

Once the effects of the boundary are felt in the well pressure response, the second assumption will no longer provide a good approximation: while the numerical well pressure will stabilize at a constant value, the line source solution continues to decrease.

2.5 Fitting the model manually to the data

To fit the model to the data, you should first make sure that you simulate at least as long there are data points. Next, it pays off to plot the pressures using a logarithmic time axis. Then, we can obtain a good fit by manually tuning the model parameters k, p_i, and r_e:

- First, adjusting k changes the slope, m, of the $p_w(t)$ versus $\log_{10}(t)$ curve.
- Changing the exterior radius is needed in order to see the impact of the reservoir boundary in the simulation; that is, the deflection from a straight-line in the $p_w(t)$ versus $\log_{10}(t)$ plot.
- Finally, modifying the initial reservoir pressure is needed to obtain the correct pressure level.

For example, choosing $k = 250$ mD, $p_i = 4000$ psi, and $r_e = 8000$ ft produces an almost identical match to the "data" (actually, produced by a commercial reservoir simulator).

2.6 Why the permeability determines the early-time pressure slope

By performing a Taylor expansion, it is possible to show that the exponential integral in the line source solution can be approximated with a logarithmic term:

$$\mathcal{W}(-x) \approx \gamma - \ln\left(\frac{1}{x}\right) \tag{12}$$

$$= \gamma - \ln(10) \cdot \log_{10}\left(\frac{1}{x}\right) , \tag{13}$$

where $\gamma = 0.57721\ldots$ is known as the *Euler-Mascheroni constant*. The approximation is accurate to within one per cent if $x < 0.01$ (figure **??**).

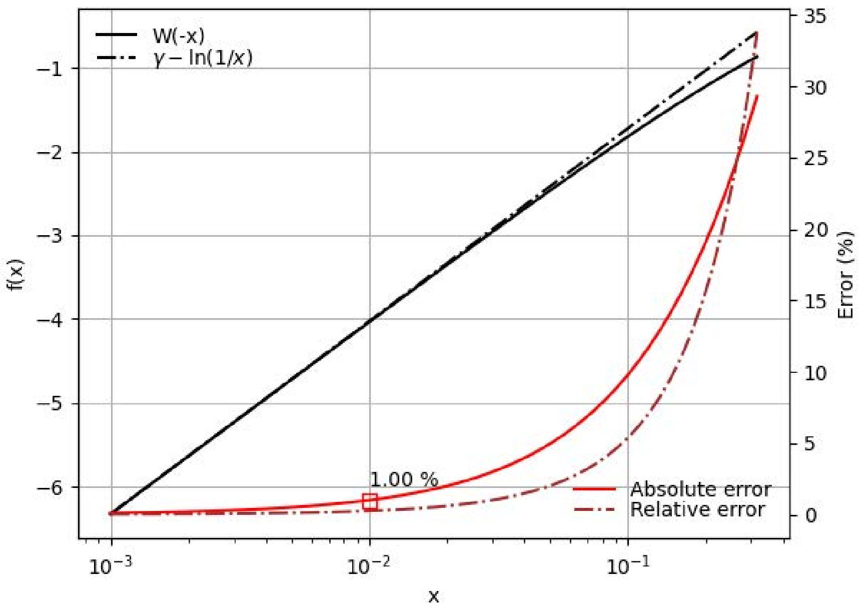

Figure 1: Logarithmic approximation to the exponential integral.

Thus, if $t > 25r^2/\eta$, the line source solution is approximately equal to

$$p(r,t) \approx p_i + \frac{Q\mu}{4\pi kh} \cdot \left(\gamma - \ln(10) \cdot \log_{10}\left(\frac{4\eta t}{r^2}\right)\right) , \tag{14}$$

For example, with the default input parameters used in the project, $\eta \approx 1.7448$ and this criterion applied to the well pressure ($r = r_w$) requires that $t > 0.13$ seconds, which is hardly no time. It follows, by setting $r = r_w$, that we can estimate the permeability from:

$$k \approx -\frac{Q\mu \ln(10)}{4\pi h m}\,. \tag{15}$$

Note that this formula presumes SI-units. In terms of the field units used as input (k in units of mD, h in ft, μ in cP, and Q in barrels per day), the formula becomes

$$k \approx -162.57 \cdot \frac{Q\mu}{hm}\,. \tag{16}$$

2.7 Using automated curve-fitting

To be able to use `scipy.optimize.curve_fit`, you should start by reading the documentation. Then, you will find that you need to pass in at least three arguments:

1. A function which, given any valid input, produces output to match (`f`).
2. An arrae-like object of input values to match (`xdata`); typically, a list or a NumPy array.
3. A corresponding set of output values (`ydata`).

Optionally, you can also provide an initial guess for the parameters you wish to fit (`p0`).

For the problem in this project, the `xdata` and `ydata` parameters are the time and well pressure values found in the text file `well_bhp.dat`. Thus, all that remains is to create a Python function which takes "the independent variable as the first argument and the parameters to fit as separate remaining arguments" (direct quotation from the documentation). One way to do that could be to use the following function signature:

```
def forecast_well_pressure(t, k, re, pi):
        """
        :param t: Array of time values (unit: hours).
        :param k: Absolute permeability (mD).
        :param re: Exterior reservoiradius (feet).
        :param pi: Initial reservoir pressure (psi).
        :return: An array of simulated well pressures (psi), one value
                 for each report time in the input time array.
        """
```

Of course, the ordering of the second, third, and fourth input argument could be different; the important thing is to be consistent throughout your notebook. The main difficulty is to be able to compute well pressure values at the times given in the input set of values, but at no other times. Some possible solution strategies are:

- "Hacking it": Notice that for this particular problem, all data points are listed as "full hours", thus use a time step that always divides 1 hour, e.g., 1 hour or 0.5 hours. Now, you just need to filter out results from all times other than those in the input. Of course, this solution will not work for arbitrary data sets.
- Implement additional time-step logic: This is more difficult, but you could add extra checks in your time loop to always record the solution at "special times" given in the input to the solver. For the general case, this will require you to take variable time step sizes, which means you have to update the matrix...(it is no longer a constant throughout the entire simulation)
- Interpolate: Pick the solutions you get from the solver, and do e.g. linear interpolation in-between to get the required values, e.g., by using scipy.interpolate.interp1d.
- Use an approximate solution: For this particular problem, we know the line-source solution provides a good approximation at least initially. We could try to estimate, e.g., the permeability from the line-source solution, or from the logarithmic approximation discussed above. However, we would still have to do some extra (manual?) tuning, because we would not capture the late time period.

3 General coding tips

Importing and plotting:

- Import all external dependencies (libraries, modules, etc.) at the top of your notebook, and do it only once.
- By typing `%matplotlib inline` at the top of your Jupyter notebook, you never have to invoke `plt.show()` to produce figures.

How can you write clearer code?:

- Reduce the amount of code repetition by defining smart functions or classes.
- Any single function should not do too many things, because that makes it harder to re-use the function.
  - Example: It is almost never a good idea to have a single function do both model calculations *and* plotting.
- It is very important to document any code that is not 'self-explanatory'.
- On the other hand, you should not comment things that are completely obvious from reading the code.

- Focus on improving code clarity rather than cluttering your code with comments (which quickly become obsolete as the code changes).
- Whenever you can, choose descriptive names for variables/functions/classes etc.
- Explain how your code can be used by writing docstrings (see below).
- The PEP8 style guide contains many useful tips for how to structure your code.

Docstrings:

- When defining your own functions, classes etc., it is considered good practice to include a docstring.
- The docstring is a special string literal placed right after, e.g. a function definition, to explain the purpose of the function and how to use it.
- It is recommended to use triple double quotes `"""` around docstrings.
- Add 'r' in front of the docstring if it contains a backslash (the 'r' stands for *raw string*)
- When calling Python's `help` function, the docstring will be printed.