

# Lab 2: Gradient Descent Calibration

(Note, while this lab is not directly assessed, the final exam will make explicit reference to your experience with this lab and the tasks completed.)

## Problem introduction

Generally, numerical methods of model calibration require an algorithm for choosing a parameter estimate,  $\theta_{i+1}$ , based on the existing estimate,  $\theta_i$ , and information about how well the model fits the observations,  $S(\theta)$ . As discussed in lectures, *gradient descent* updates  $\theta$  by incrementing in the direction that  $S(\theta)$  is decreasing most rapidly. The key steps in implementing the algorithm are:

1. Choose an initial parameter vector,  $\theta_0$ .
2. Compute the descent direction,  $-\mathbf{s}'$ , at  $\theta_0$
3. Increment in the descent direction to obtain  $\theta_1$ .
4. Repeat steps 3 and 4 until some criteria for stopping is met

This will require three functions:

- **obj** () – for *input* parameter vector,  $\theta$ , *returns* the objective function,  $S(\theta)$ .
- **obj\_dir** () – for *inputs* parameter vector,  $\theta$ , and objective function,  $r$ , *returns* the gradient,  $\mathbf{s}'(\theta)$ .
- **step** () – for *inputs* initial parameter vector,  $\theta_i$ , gradient,  $\mathbf{s}'(\theta_i)$ , and step size,  $\alpha$ , *returns* the updated parameter vector,  $\theta_{i+1}$ .

In this lab, you will be completing the partially written functions **obj\_dir** () and **step** () so as to implement your own gradient descent algorithm. The function, **obj** () is provided for you; however, [for assignment 2](#), you will need to formulate your own objective function and associated function **obj** () .

You will initially develop your gradient descent algorithm to operate on a simple 2D Gaussian objective function. When it is working properly, we will apply the algorithm to calibrate a model for a discharging geothermal well.

This lab makes use of the Python programming language. Ensure that you use Python 3.X to complete the lab (should be the default in Visual Studio Code).

## Tasks

As you complete the tasks below, you will encounter several *italicised* questions. While there is nothing to submit for this lab, you may nevertheless wish to write down answers to these questions as an aid for your end of year exam revision.

[Download lab2\\_files.zip from Canvas and extract the contents](#). We will be working with both the *main.py* and *gradient\_descent.py* files. [Open these files in Visual Studio Code](#). The other files are related to execution of the discharging well model and plotting. [You can run python scripts by hitting Ctrl+F5 or just F5 to enter debugging](#).

***“HELP! Visual Studio Code won’t run my Python File!” See end of document for troubleshooting.***

## 1. Compute initial descent direction

Complete the function `obj_dir()` in the file `gradient_descent.py`. This function computes components of the objective function sensitivity,  $\mathbf{s}'(\boldsymbol{\theta})$ . You may wish to refer to Section 2.3.2 of `calibration.ipynb`.

When you have completed `obj_dir()`, run `main.py`. A figure window will appear showing contours of the objective function (a 2D Gaussian) and an arrow indicating the initial descent direction for  $\boldsymbol{\theta}_0 = [0.5, -0.5]$ . If `obj_dir()` is properly implemented, your plot should match Figure 1.

*Where is the minimum of  $S$ ?*

*How does the variable `dtheta` in `obj_dir()` affect the calculation of  $\mathbf{s}'$ ?*

*In one call to `obj_dir()` how many calls are made to `obj()`? How does this relate to the dimensionality of parameter space?*

## 2. Take one gradient descent step

Complete `step()` in `gradient_descent.py`. This function computes the updated parameter vector,  $\boldsymbol{\theta}_{i+1}$  by advancing in the descent direction along a line starting at  $\boldsymbol{\theta}_i$ . You may wish to refer Section 2.3.3 of `calibration.ipynb`, in particular step 3.

When you have completed `step()`, run `main.py` again. Ensure that you have commented the `return` function to advance to the next part of `main.py`.

If `obj_dir()` is properly implemented, your plot should approximately match Figure 2, which shows the first step taken along the descent direction calculated in Task 1.

## 3. Compute a new descent direction

Complete the code for Task 3. The objective here is to calculate the new descent direction for the next step. When implemented correctly, you should be able to replicate the plot in Figure 3.

*Why is the new descent direction different from the previous one?*

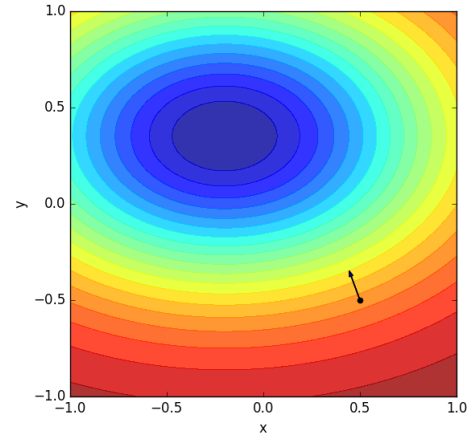


Figure 1: Objective function contours and initial descent direction for  $\boldsymbol{\theta}_0 = [0.5, -0.5]$ .

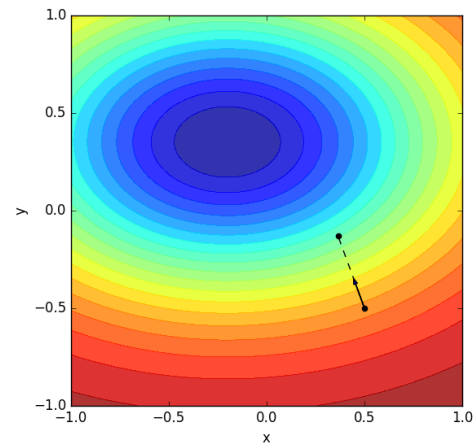


Figure 2: Initial descent direction and first step in the gradient descent.

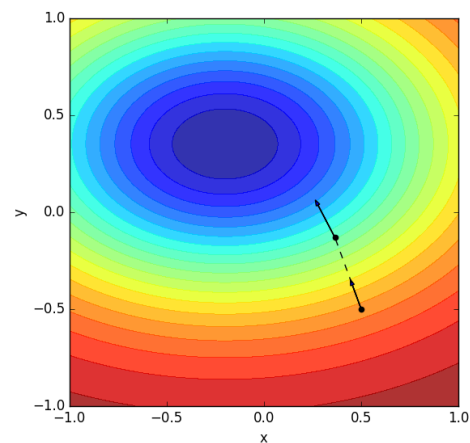


Figure 3: New descent direction after the first gradient descent step.

#### 4. Find minimum of objective function

Complete the code for Task 4 in `main.py`. Here, you are generalizing Tasks 1-3 above inside a loop so that the descent algorithm takes multiple steps. Run the completed code and check the plot matches Figure 4.

What is the stopping criteria for the `while` loop in Task 4? Can you think of an alternative stopping criteria?

Experiment with different values for `theta0` (line 34). How is the gradient descent path different? How is it the same?

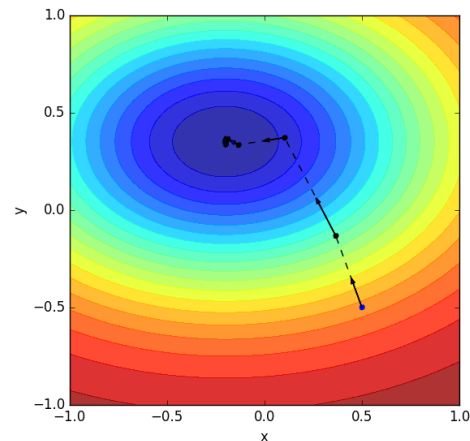


Figure 4: Path to objective function minimum using gradient descent.

#### 5. Gradient descent stepping

So far we have used a fixed step size for the gradient descent algorithm. You can modify the step size that is used by changing the value of `alpha` (line 51).

Describe the effect of increasing or decreasing step size in terms of computational effort and algorithm stability.

Uncomment line 97 to implement the line search (first command in the `while` loop for Task 4). The line search finds the minimum value of the objective function along the search direction, setting `alpha` accordingly.

What is the advantage of using a line search compared to a fixed step size?

#### 6. Calibrating a model of a discharging geothermal well

In this final task, we will use the gradient descent algorithm that you completed above to demonstrate calibration of a model for a discharging geothermal well.

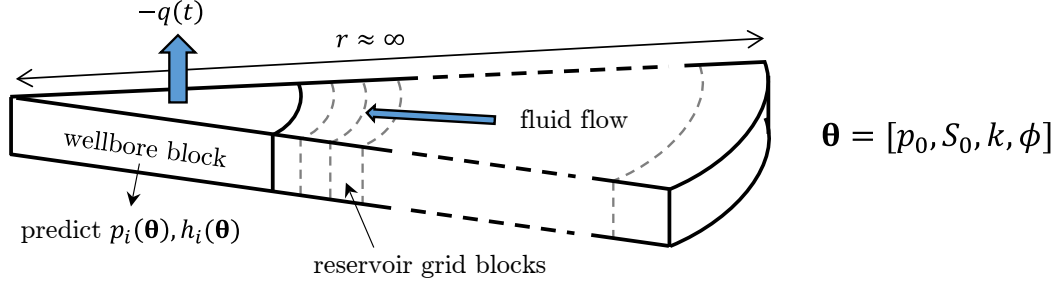
##### Model description

What happens when you inflate a balloon, pinch one end and then let it go? The mass inside the balloon is expelled due to the pressure difference between air inside (high pressure) and outside (lower). A similar phenomenon occurs when a geothermal well is discharged: hot water, which is under high pressure inside the reservoir, moves rapidly up the well and is expelled to the atmosphere (usually in the form of steam; of course, it is much better to stick a turbine on the end that just let the well vent). Discharging a well causes the pressure in the reservoir to drop and, if this results in a sufficient reduction of the boiling point, flash to steam. Water in the form of steam has much higher enthalpy than water in liquid form.

We will look at the specific case of the discharging well SKG9D at Fushime, Japan. A radial model has been constructed for the section of the reservoir adjacent to the discharging well (Fig. 5). The four key parameters to be calibrated are:

- **Initial reservoir pressure,  $p_0$  (free):**
- **Initial reservoir steam fraction,  $S_0$  (free):** the proportion of the water in the gas phase.
- Porosity,  $\phi$  (fixed, 0.08): assumed to be homogeneous (the same everywhere).
- Permeability,  $k$  (fixed,  $2.8 \times 10^{-15} \text{ m}^2$ ): assumed to be homogeneous and isotropic (the same in all directions).

The primary boundary condition driving changes in the system is the extraction of mass from the reservoir at rate,  $q(t)$ , from the wellbore block over a 140 day period. The data available for calibration are (1) the enthalpy,  $\tilde{h}_i$ , and (2) pressure,  $\tilde{p}_i$ , of fluid discharged from the well. The corresponding model prediction of these quantities is the fluid state (pressure and enthalpy) in the wellbore block (Fig. 5).

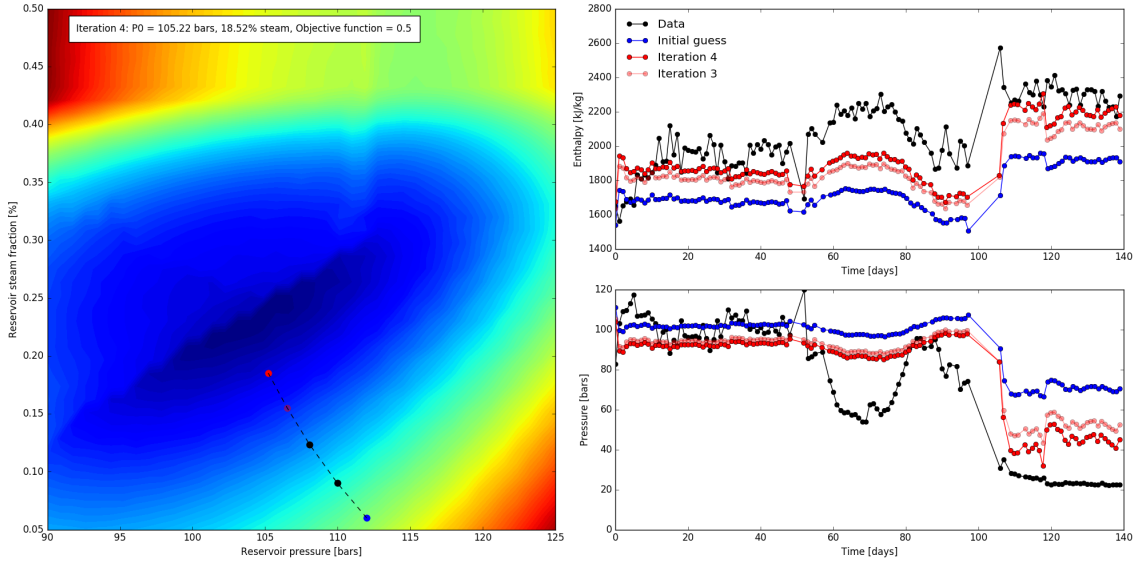


**Figure 5: Discharging well, model schematic.** The interior block has the same radius as the well. Mass is extracted at a variable rate,  $q(t)$ , from the wellbore block, which modifies the pressure,  $p_i$ , and enthalpy,  $h_i$ , in it. These predictions are compared to data during calibration.

### Inverse modelling tasks

The forward model and calibration algorithm have been implemented for you (inspect *wellbore\_model.py* if you are interested in the details). [Execute the code for Task 6 in \*main.py\* to automatically calibrate the discharge model.](#) You should see some information about iterations being printed to the screen – each of these is a step taken by the gradient descent algorithm.

Inspect the plots that are generated (one for each step, see Fig. 6 for an example).



**Figure 6: Calibration of an AUTOUGH2 discharging well model.** Left: path of the gradient descent algorithm through parameter space, superimposed on contours of the objective function. Right: output of the forward model (prediction of enthalpy, top, and pressure histories, bottom) for parameter estimate,  $\theta_4$ . Black profiles are field data, blue profiles are the initial model prediction, i.e., for  $\theta_0$ , and pink and red profiles compare the model predictions at the beginning and end of the gradient descent step.

What is plotted on the horizontal axis of the righthand plots? What is plotted on the vertical axes?

Characterise the initial model fit to the data, compared to the calibrated model.

What quantity is plotted as coloured contours on lefthand plot?

From an inverse modelling perspective, explain (i) what new information has been gained, and (ii) what we first needed to know to gain this new information.

For the initial parameter estimate, misfit between observation and prediction of enthalpy are of the order 300 kJ/kg while for pressure the misfit is of order 30 bar. If we combine these observations together inside a single least-squares residual

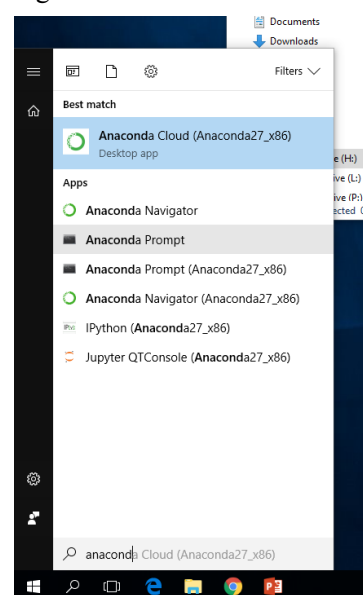
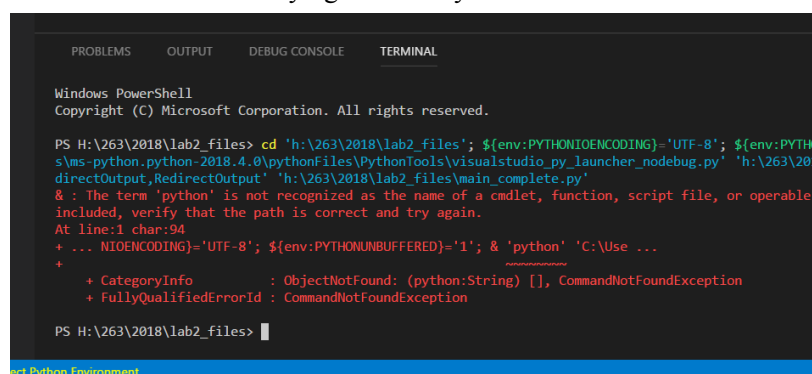
$$S(\boldsymbol{\theta}) = \sum_{i=1}^n (\tilde{h}_i - h_i(\boldsymbol{\theta}))^2 + \sum_{i=1}^n (\tilde{p}_i - p_i(\boldsymbol{\theta}))^2,$$

which observation (enthalpy or pressure) will dominate  $S$  and therefore be “most” calibrated against? What could we do to ensure the calibration against each observation is more evenly balanced?

Suppose we had twice as many measurements of enthalpy as pressure, which would naturally weight the calibration in favour of enthalpy (there would be more  $\tilde{h}_i - h_i(\boldsymbol{\theta})$  terms in the equation above). What could be done to balance the calibration to account for different sampling rates?

## Troubleshooting Visual Studio Code (VSC)

A common error when trying to run a Python file within VSC looks something like the below

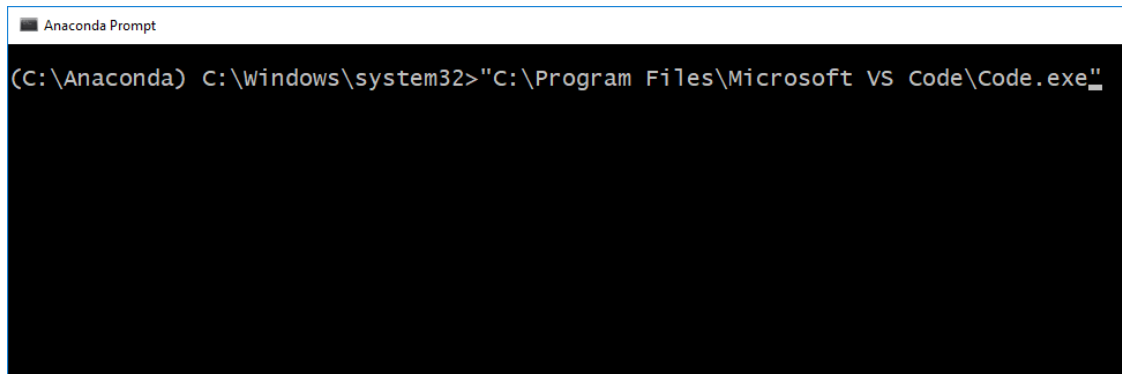


Basically, VSC is telling you it cannot find a Python executable. We'll need to tell it where to look.

1. **Close** Visual Studio Code.
2. **Open** an Anaconda Command Prompt by typing “Anaconda” in the Start Menu. This will open a terminal with a cursor beginning (C:\Anaconda) C:\Windows\system32> Environment variables have been defined in this terminal that tell Windows where to find Python.
3. **Open** a new instance of Visual Studio Code from the terminal by typing “C:\Program Files\Microsoft VS Code\Code.exe”

Note the quote marks around the entire command. This is because the people who originally designed Windows put a ‘space’ between the words “Program” and “Files”. Operating systems are remarkably fragile.

4. You should now be able to open and run Python files from within VSC. If it's still not working, call me over to share in the misery.



```
Anaconda Prompt
(C:\Anaconda) C:\Windows\system32>"C:\Program Files\Microsoft VS Code\Code.exe"
```

