Lab 4: Lumped Parameter Models

(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

Prior to the development of sophisticated numerical flow simulators in the 1980s, analysis of a geothermal reservoir and its response to production was conducted using a *lumped parameter model.* As discussed in class, these models imagine the reservoir as a single control volume, and then consider how flows of mass into and out of the volume affect its pressure. Fradkin et al. (1980) developed one such lumped parameter model, which is governed by the first-order ODE below:

$$\frac{dP}{dt} = -aq - bP - c\frac{dq}{dt},$$

where P is the **change** in reservoir pressure from its initial value, $q(t)$ is an arbitrary time series of the mass extraction rate from the reservoir (via wells), and a, b and c are the model parameters governing, respectively: extraction, recharge, and slow drainage.

These models were developed, in part, to describe the Wairākei geothermal field, a large geothermal system that sits on the banks of the Waikato River near Taupō township. In the early years of development, pressure in the Wairākei reservoir dropped rapidly in response to the new production (Fig. 1). However, over time, a new equilibrium pressure state was established.

In the previous lab (Lab 3: Forecasting Geothermal Impacts), we saw how a distribution of parameters could be used to construct an ensemble of models and a corresponding distribution of predictions: *a forecast*. In this lab, we will use the objective function to formally define a **posterior distribution** over the parameters, and sample from this to obtain a forecast. We shall also investigate one source of structural error.

This lab should be used with Python 3.X.

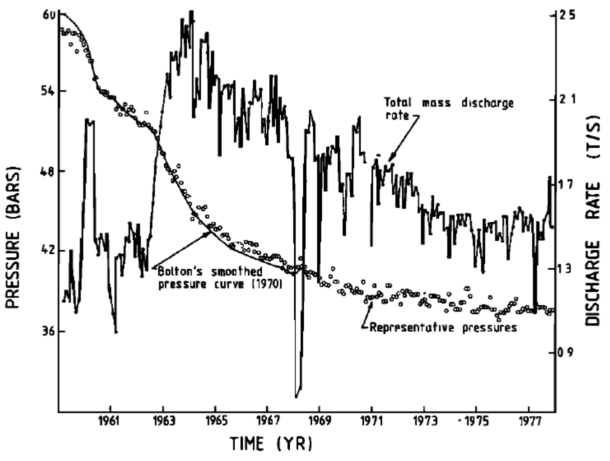


Figure 1: Pressure decline in the Wairākei reservoir in response to mass extraction, $q(t)$, for geothermal electricity generation. Data from 1959 to 1978. Figure from Fradkin et al. (1980).

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 *lab4_files.zip* from Canvas and extract the contents. You will be making modifications to the files *main.py*, which in turn calls functions from *plotting.py* and *lumped_parameter_model.py*. Open these files in Visual Studio Code. 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. Model familiarization

As in the previous lab, our first task is to get to know the model. Open *main.py* and execute the code block for Task 1. This section of code implements the lumped parameter model using q shown in Fig. 1, while ***explicitly neglecting*** the effects of slow drainage. We do this by setting $c = 0$. The ODE is solved using an improved Euler step. The plot that is generated compares the predicted pressure decline in the reservoir (blue line) to the measured values (red dots with error bars), up to 1980.

Select values for the two parameters, a, and, b, that give you a reasonable match to the data (use your judgement, if you get stuck, consult Section 3.4.2 in the Uncertainty notebook).

By neglecting the physics of slow drainage, what kind of error are we introducing into our model?

For constant mass extraction, what would the steady-state pressure of the reservoir be?

Modify the line beginning `S =` to compute the sum of squares objective function between the variables `po` and `pm`. This will be displayed in the title of your plot.

How does the variance, `v`*, enter into the objective function calculation?*

2. Construct the posterior

In Lab 3, we constructed a *prior parameter distribution,* and then sampled from this to create an ensemble of models and a forecast of the future. In this task, we will instead compute a *posterior parameter distribution* using the measured pressure data up to 1980.

As detailed in Section 3.2 of the Uncertainty notebook, the posterior is obtained by (1) computing the objective function for each parameter combination, $\boldsymbol{\theta}$, (2) taking the exponential, and then (3) normalising the distribution.

Complete the grid search code under Task 2 to compute the posterior. You should obtain a plot that looks similar to Fig. 2.

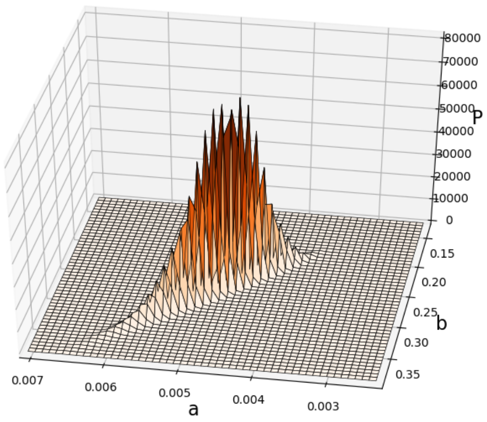


Figure 2: The posterior parameter distribution for $c = 0$.

Inspecting the plot that is generated, what can you say about parameter correlation? (i.e., do the calibrated values of a *and* b *appear independent?)*

In computing the posterior this way, what assumptions have we made about (1) the prior, and (2) the measurement errors?

What is the effect of modifying the parameter `N=51`*? What are the trade offs?*

3. Fun with multivariate normal distributions

What is a **multivariate normal distribution**? Not as scary as it sounds.

In ONE DIMENSION, it is just the usual normal distribution you are familiar with. It has the equation

$$P(\theta) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(\theta-\mu)^2}{2\sigma^2}\right)$$

where θ is the parameter, μ is its mean, and σ^2 is its variance. What does this equation look like? Open the file *fun_with_multivariate_normals.py* and execute Part I.

What about in TWO DIMENSIONS? One way to write the multivariate normal distribution is

$$P(\theta_1, \theta_2) = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{1}{2}\left[\frac{(\theta_1-\mu_1)^2}{\sigma_1^2} + \frac{(\theta_2-\mu_2)^2}{2\sigma_2^2}\right]\right)$$

where now we have two parameters ($[\theta_1, \theta_2]$), two means ($[\mu_1, \mu_2]$) and two variances ($[\sigma_1^2, \sigma_2^2]$). What does this equation look like? Execute Part II in *fun_with_multivariate_normals.py*.

The expression above is actually not the most general form of a 2D multivariate normal distribution, as it does not allow for correlated parameters:

$$P(\theta_1, \theta_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)}\left[\frac{(\theta_1-\mu_1)^2}{\sigma_1^2} + \frac{(\theta_2-\mu_2)^2}{2\sigma_2^2} - \frac{2\rho(\theta_1-\mu_1)(\theta_2-\mu_2)}{\sigma_1\sigma_2}\right]\right)$$

where ρ is a measure of how strong the parameters θ_1 and θ_2 are correlated. At first glance, that equation looks a bit intimidating, but we can immediately see that setting $\rho = 0$ (uncorrelated parameters) returns the simpler 2D normal distribution introduced earlier. Execute Part III in *fun_with_multivariate_normals.py*.

Once you have answered the questions, continue with Task 4 in *main.py*.

If we fix ONE of the parameters in the 2D multivariate normal distribution to a fixed value, e.g., $\theta_1 = \text{const.}$, *what form does the resulting distribution over* θ_2 *take?*

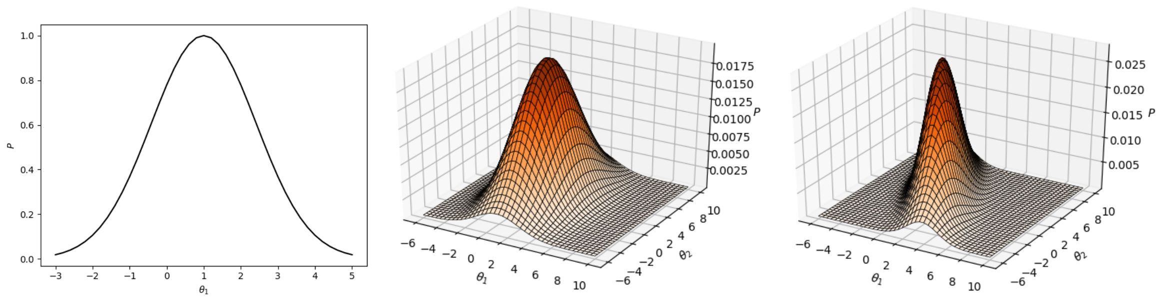


Figure 3: Multivariate normal distributions in various dimensions and with varying degree of correlation (from left to right, Parts I to III in *fun_with_multivariate_normals.py*).

4. Sampling from the posterior

In our case, the shape of the posterior distribution can be approximated by a *multivariate normal distribution*. The rotation of the distribution (evident in Fig. 2) reflects parameter correlation, which can be encoded in a *covariance matrix*. This is just a multi-dimensional matrix-version of the variance, σ_i^2, that describes a 1D normal distribution.

Fitting of a multivariate normal distribution to the posterior generated in Task 2 is beyond the scope of this lab. However, if you are interested, I encourage you to inspect the code in `fit_mvn()`.

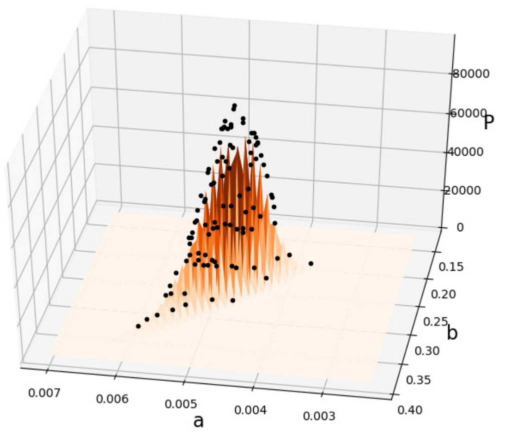


Figure 4: Plot of samples (black dots) overlaid on the posterior distribution.

Complete the code for Task 4, by researching (use Google, look for StackOverflow links) how to implement the `np.random.multivariate_normal()` function. This will generate a set of parameter samples that are taken from a multivariate normal distribution with mean and covariance matrix input by you. When it is working correctly, you should get a plot like Fig. 4.

5. Constructing the forecast

The samples generated in Task 4 are passed out from the function *construct_samples()* and into the function *model_ensemble()*. The job now is to run the model for each of these samples, not just up to the end of the calibration period (1980), but also more than 30 years into the "future" (up to 2012).

Complete the code for Task 5. Much more of this function has been left blank for you to complete, although it should largely replicate the code in Task 1. Once completed, you should have a set of model predictions (the forecast) of how pressure in the Wairākei reservoir changes between 1980 and 2012. You can compare these to how the pressure actually changed.

Assess critically the quality of this forecast?

How does modifying the error variance parameter, v*, affect the forecast?*

6. Addressing structural error

The problem with the model developed above is that it does not account for "slow drainage": we made sure of that by setting $c = 0$. For this final task, you will extend your analysis in Tasks 1-5 (computing and sampling from the posterior) for a model with three parameters. The approach is no different, although given the increased dimensionality of parameter space, the computational burden is greater.

Make a copy of *main.py* and rename it *main_3D.py*. Modify the code to estimate a posterior over a, b and c, sample it, and construct a forecast. You will need to extend your grid search to three dimensions and then fit and sample a 3D normal distribution. Inspect the plots that are generated, which should look similar to Fig. 5.

Inspecting the posterior plots, are some pairs of parameters more or less correlated than others?

Each of the posterior distribution plots is an example of a marginal distribution. *How are these generated from the full posterior?*

Comparing the outputs of Tasks 5 and 6, describe the impact of structural error on the forecast.

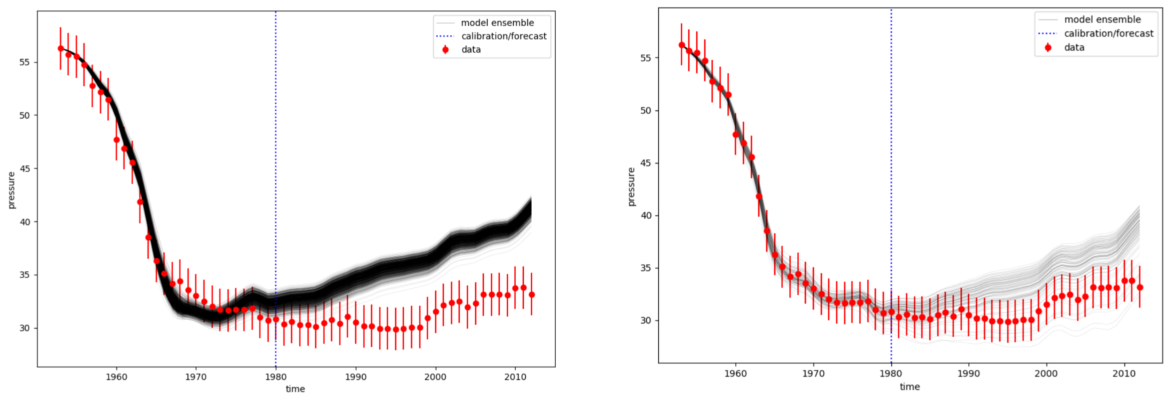
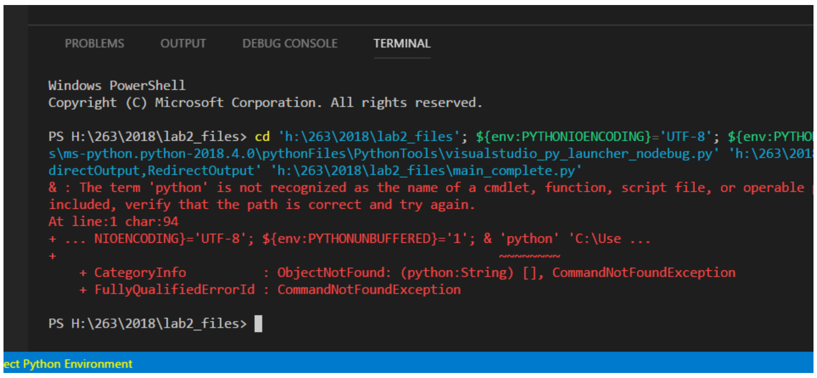


Figure 5: Ensemble of model predictions overlaid by the true outcome for models with (right) and without (left) slow drainage.

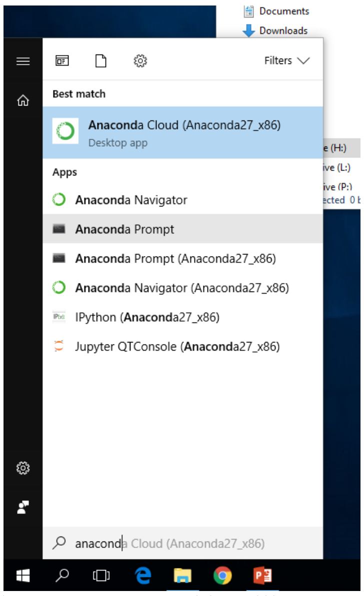
Troubleshooting Visual Studio Code (VSC)

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



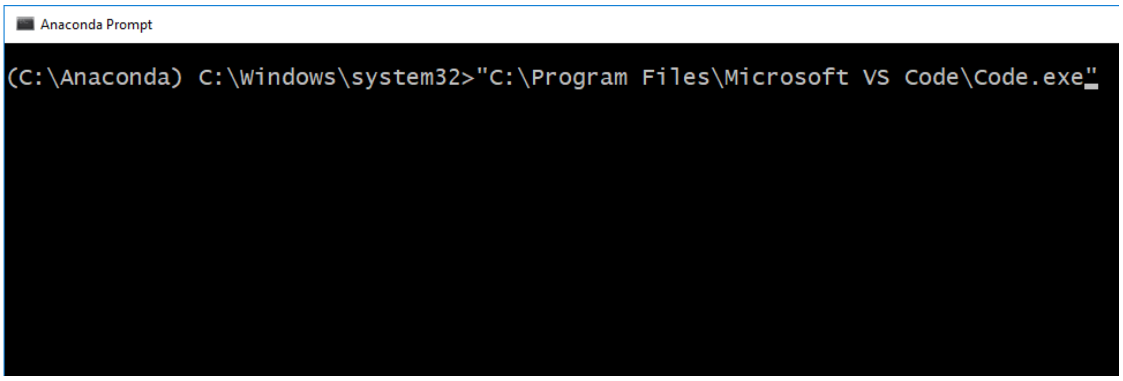
```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS H:\263\2018\lab2_files> cd 'h:\263\2018\lab2_files'; ${env:PYTHONIOENCODING}='UTF-8'; ${env:PYTHO
s\ms-python.python-2018.4.0\pythonFiles\PythonTools\visualstudio_py_launcher_nodebug.py' 'h:\263\201
directOutput,RedirectOutput' 'h:\263\2018\lab2_files\main_complete.py'
& : The term 'python' is not recognized as the name of a cmdlet, function, script file, or operable
included, verify that the path is correct and try again.
At line:1 char:94
+ ... NIOENCODING}='UTF-8'; ${env:PYTHONUNBUFFERED}='1'; & 'python' 'C:\Use ...
+                                                          ~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (python:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException

PS H:\263\2018\lab2_files> 
```

ect Python Environment


Documents
Downloads
Filters
Best match
Anaconda Cloud (Anaconda27_x86)
Desktop app
Apps
Anaconda Navigator
Anaconda Prompt
Anaconda Prompt (Anaconda27_x86)
Anaconda Navigator (Anaconda27_x86)
IPython (Anaconda27_x86)
Jupyter QTConsole (Anaconda27_x86)
anaconda Cloud (Anaconda27_x86)

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"
```