EVR-5086 Assignments

Adyan Rios

2025-09-10

Table of contents

Introduction

Although the EVR-5086 class is being taught using Python, my prior experience is with R. I am also fond of sharing my work on GitHub. I have learned how GitHub pages combined with Quarto and R Studio are an extraordinary resource for developing and maintaining lab notebooks. To get better at using these tools (and the reproducibility and accessibility of my future research) I have created a html quarto book and pdf to show my work associated with the course assignments.

Set Up

I started by creating a GitHub account (username: arios101-fiu). Then, I created a GitHub repository with a gitignore and readme.md. Initially, the repository was called EVR-5086-Assignement1, but I updated it to EVR-5086-Assignements (EVR-5086-Assignements). I cloned the repository into R Studio, thereby creating a R project. I copied in a _quarto.yml and index.qmd files from another project. I updated the files, rendered, committed, and pushed. Next, I turned on GitHub pages and updated the URLs in the yml and repository. Before moving on the first assignment in the following chapter, The following R code needs to be run to set up R Studio to run Python.

```
# Install and activate {reticulate} in R
# install.packages("reticulate") # Run once
library(reticulate)

# Install matplotlib
# py_install("matplotlib") # Run once
```

1 Assignment 1 – Calculus Review

1.1 Polynomial Plot

Below are the steps I took to complete the first part of EVR-5086 Assignment 1.

In doing this exercise in R, I started by loading the R libraries I will use in this chapter. I used {ggplot2} for plotting, and {tidyr} and {dplyr} for data wrangling.

```
# load libraries
library(ggplot2)
library(tidyr)
library(dplyr)
```

Next, I defined the variables and created the vectors I will need for the plot.

```
# Define variables
a <- 1
n <- 1
b <- 1
p <- 2
c <- 1
q <- 3

# Create x vectors from -1 to 1
x <- seq(from = -1, to = 1, by = 0.1)

# Calculate the value of y for each value of x
y <- (a * (x^n)) + (b * (x^p)) + (c * (x^q))

# Calculate the analytical derivatives for each value of x
dy_dx <- (a * n * (x^(n - 1))) + (b * p * (x^(p - 1))) + (c * q * (x^(q-1)))

# Calculate the numerical derivatives
deltay <- diff(y)
deltax <- diff(x)
```

```r
deltay_deltax <- deltay / deltax

# For plotting purposes, derive the midpoint
deltax_vec <- x[-length(x)] + deltax/2
```

My next goal was to unite all of the vectors into a long data format. I did this by creating a data frame, then pivoting the data to only have the values that will be plotted on the x and y axis, as well as a label. Later, I will use my "linetype" label to define line types as well as the colors and shapes in my plot.

```r
# Build data frames for plot
plot_prep <- data.frame(x, y, dy_dx) |>
  dplyr::rename(Polynomial = y,
                "Analytical derivative" = dy_dx)

# Wrangle for ggplot
plot_tidy <- plot_prep |>
  tidyr::pivot_longer(!x, names_to = "linetype", values_to = "y") |>
  dplyr::bind_rows(
    data.frame(
      x = deltax_vec,
      y = deltay_deltax,
      linetype = "Numerical derivative"
    )
  )
```

Lastly, I create the plot and reflect on the observations and limitations of the numerical derivative.

Figure 1.1 shows that the numerical derivative, shown as red open circles, is very similar to the analytical derivative, shown as a blue solid line. The good match we see relates to the scale over which we calculated the numerical derivative compared to the scale of the rate of change in the polynomial. When calculating the numerical derivative, we can get the average rate of change between two points.

Note that for the analytical derivative we are only providing the plot with information associated with x values ranging -1 to 1, in steps of 0.1. Meanwhile, the numerical derivative is plotted at the midpoints of our original segments, with x values ranging from -0.95 to 0.95. Including the numerical derivatives in the appropriate position relative to the curved lines plotted between our analytical derivatives results in an overlay of the points and the line.

If the numerical derivative had a significantly lower resolution (e.g. just -1 and 1), it would not match well, and would be just one point above the "U" shape of the analytical derivative at

x = 0. Although that course spacing is an extreme, it helps to emphasize that "grid spacing and position of the computed derivative need to be considered."

```
# Plot the analytically derivative as a solid line
# and the numerical derivative as open symbols
polynomial_plot <- ggplot(data = plot_tidy,
                          aes(x = x, y = y, color = linetype)) +
  geom_point(data = dplyr::filter(plot_tidy, linetype == "Numerical derivative"),
             shape = 21, stroke = 1.25) +
  geom_line(data = dplyr::filter(plot_tidy, linetype != "Numerical derivative")) +
  theme(legend.title = element_blank()) +
  scale_color_manual(values = c(4, 2, 1)) +
  theme_minimal()

polynomial_plot
```

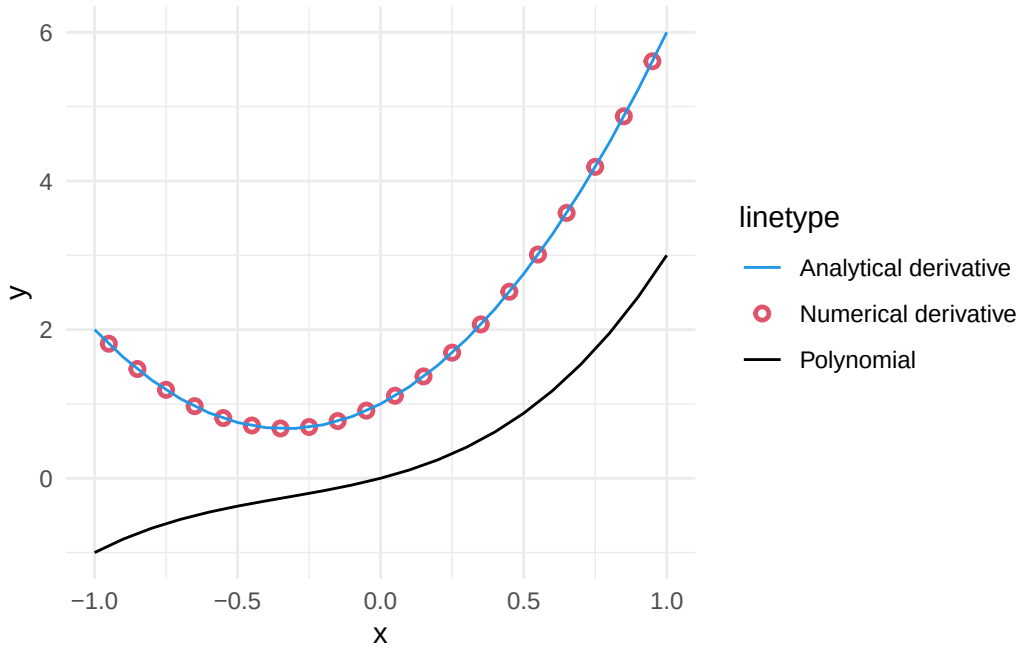

Figure 1.1: Plot of the polynomial defined by the values provided in EVR-5086 Assignment 1 (black line), along with analytical (blue line) and numerical derivatives (red open circle).

1.2 Solve the 2-D Laplace in Excel

I created a 28 by 28 grid of the 2-D Laplace Equation. I included three internal "boundary values"; one high value of 4 and two low values of -2 and -3. The two low values were near each other compared to their respective distances to the high value. I allowed excel to iteratively calculate for 10,000 iterations with a minimum change of 0.0001. I saved the file as a CSV file after including explicit zeros surrounding the formulas. The dimensions of my data were 30 by 30. I rounded to four significant digits to see if it would mean that the stagnation areas would be more pronounced.

1.3 Read in and plot contours

I tried working on this part of the assignment in R, but I ran short on time. Instead, I pivoted to running python inside of my R Studio Project. This first chunk of code is mostly commented out. I may consider returning to this. Since much of my work is in R, I think it might still be useful for me to bring the python code and examples into my Tidyverse and R for Data Science knowledge foundation.

```
# Load libraries
library(plot3D)
library(ggplot2)

# Read in csv file
h = as.matrix(read.csv("tripole.csv", header = FALSE, row.names = NULL))

# Create vectors
x_vec <- seq(-1.5, 1.4, by = 0.1)
y_vec <- seq(-1.5, 1.4, by = 0.1)

# Create "np.meshgrid"
X <- matrix(rep(x_vec, each = length(y_vec)), nrow = length(y_vec))
Y <- matrix(rep(y_vec, times = length(x_vec)), nrow = length(y_vec))

# # Plot the surface
# surf3D(
#   x = X,
#   y = Y,
#   z = h,
#   bty = "b2",
#   ticktype = "detailed",
#   phi = 20, theta = 30
```

```
# )
#
# # Plot the contour
# library(ggplot2)
#
# # Example: compute gradients with finite differences
# dhdx <- apply(h, 1, diff)
# dhdx <- cbind(dhdx, NA)
#
# dhdy <- apply(h, 2, diff)
# dhdy <- rbind(dhdy, NA)
#
# # Build data frame
# dat <- expand.grid(x = x_vec, y = y_vec)
# dat$h <- c(h)
# dat$dhdx <- c(dhdx)
# dat$dhdy <- c(dhdy)
#
# dat <- expand.grid(x = x_vec, y = y_vec)
# dat$h <- c(h)
# dat$dhdx <- c(dhdx)
# dat$dhdy <- c(dhdy)
#
# ggplot(dat, aes(x, y)) +
#   geom_contour_filled(aes(z = h)) +
#   theme_minimal()
```

Switch to using python to create Figure 1.2 and Figure 1.3. I did not identify any stagnation points. My low values were close together, so a saddle effect didn't appear as it would if they were spaced farther apart.

```
# Import packages
import numpy as np
import matplotlib.pyplot as plt

# Load
h=np.loadtxt('tripole.csv',delimiter=',')

x_vec=np.linspace(-1.5, 1.4, 30)
y_vec=np.linspace(-1.5, 1.4, 30)
X,Y = np.meshgrid(x_vec,y_vec)
[dhdy,dhdx]=np.gradient(h,x_vec,y_vec)
```

```
fig=plt.figure(figsize=[4,4],dpi=300)
plt.contourf(X,Y,h)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar object at 0x000001F3262D9290>
```

```
plt.axis('equal')
```

```
(np.float64(-1.5), np.float64(1.4), np.float64(-1.5), np.float64(1.4))
```

```
plt.quiver(X, Y, -dhdx, -dhdy)
plt.show()
```

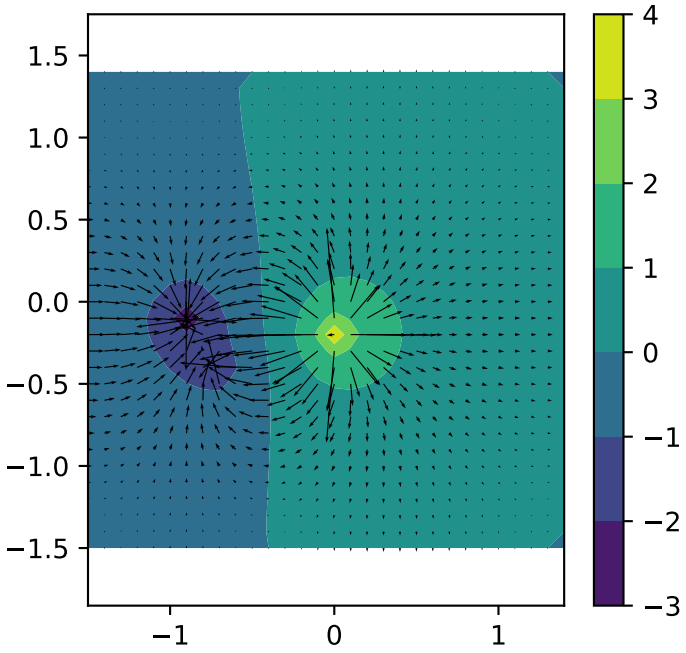

Figure 1.2: Plot of vector arrows using Python. The vectors indicate strength and direction of the negative gradient. The vectors are displayed over the contours of a tri-pole solution with a high value of 4 and lows of -2 and -3. The two low values were near each other compared to their respective distances to the high value.

1.4 Plot streamlines

```
# Plot streamlines
plt.contourf(X,Y,h)
plt.streamplot(X, Y, -dhdx, -dhdy)
```

```
<matplotlib.streamplot.StreamplotSet object at 0x000001F328A144D0>
```

```
plt.axis('equal')
```

```
(np.float64(-1.5), np.float64(1.4), np.float64(-1.5), np.float64(1.4))
```

```
plt.show()
```

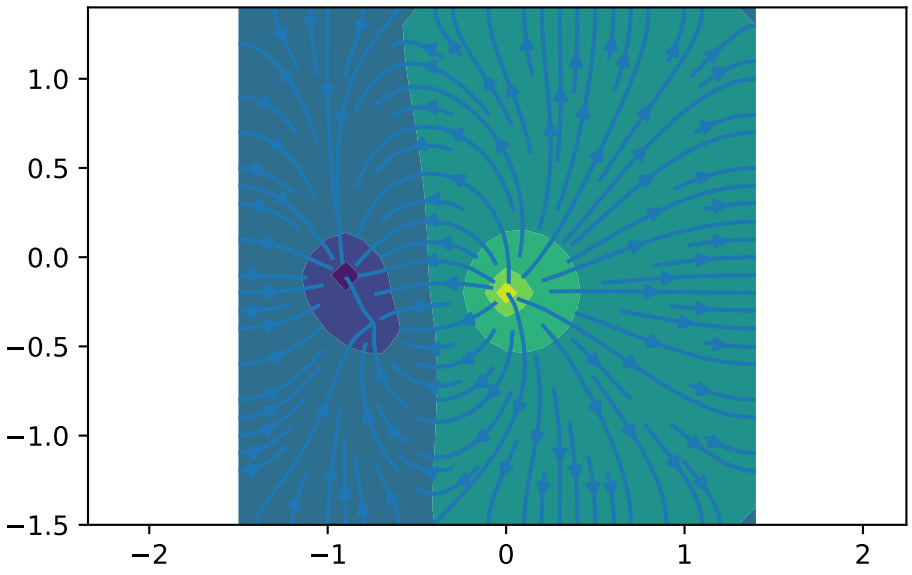

Figure 1.3: Plot of streamlines using Python. The streamlines are displayed over the contours of a tri-pole solution with a high value of 4 and lows of -2 and -3. . The two low values were near each other compared to their respective distances to the high value.