EVR-5086 Assignment 1 - Calculus Review

Adyan Rios

2025-09-09

Table of contents

Introduction

I learned that there are various ways that I can create and execute Python chunks in R Studio (Velásquez 2021). Although EVR-5086 class is being taught using python, I have years of experience using R that I hope will be complementary to some of the python I learn to use course. 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. In order 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 (arios101-fiu) and a GitHub repository with a gitignore and readme.md (EVR-5086-Assignment1). I cloned the repository into R Studio, thereby creating a R project. I copied in a _quarto.yml and index file from another project. I simplified the index file and inserted a reference to create a new reference.bib. I updated the yml, rendered, committed and pushed. Next, I turned on GitHub pages and updated the URLs in the yml and repository. On to the assignment…

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 Wickham et al. (2019).

```
# 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)
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 identifying that I will use to define colors, shapes, and line types.

```
# 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 (x values goes 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 much courser 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 a 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 symbo
polynomial_plot <- ggplot(data = plot_tidy, aes(x = x, y = y, color = linetype)) +
  geom_point(data = filter(plot_tidy, linetype == "Numerical derivative"), shape = 21, stroke
  geom_line(data = filter(plot_tidy, linetype != "Numerical derivative")) +
```

```
  theme(legend.title = element_blank()) +
  scale_color_manual(values = c(4, 2, 1))

polynomial_plot
```

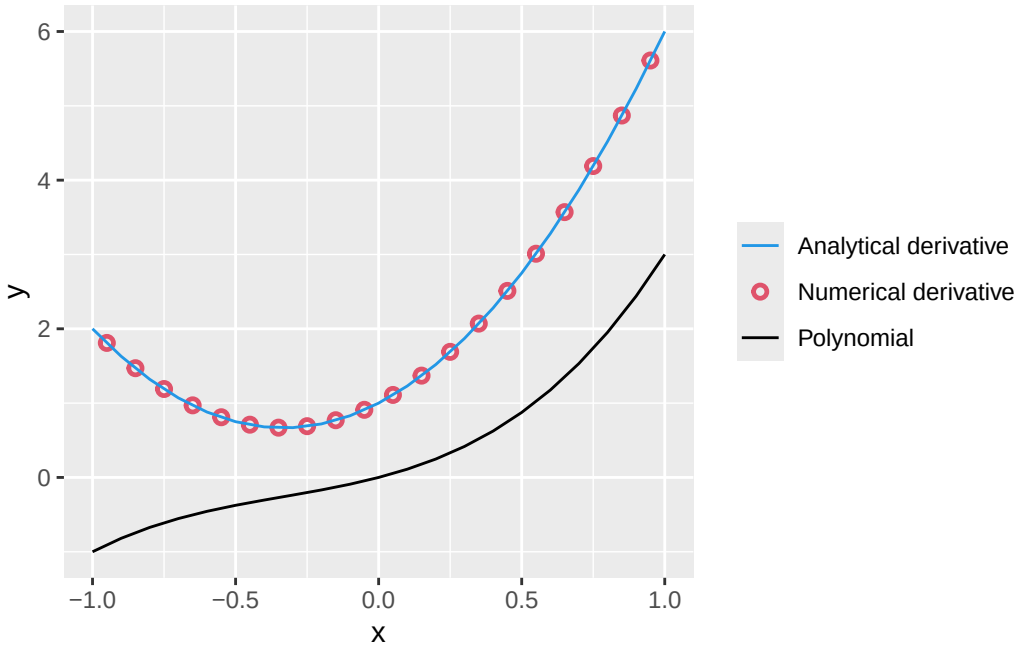

Figure 1.1: Plot of the polynomial defined by the values provided in EVR-5086 Assignement 1.

2 Solve the 2-D Laplace in Excel

I created a 29 by 29 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. I allowed excel to iteratively calculate with 10,000 iterations and a minimum change of 0.0001. Saving as a CSV file, surrounded by explicit zeros, the dimensions of my data were 30 by 30. Additionally, I rounded to four significant digits to see if it would mean that the stagnation areas would be more pronounced.

3 Read in and plot contours

4 Plot streamlines

5 References

Velásquez, Isabella. 2021. “Posit.” https://www.posit.co/.

Wickham, Hadley. 2016. “Ggplot2: Elegant Graphics for Data Analysis.” https://ggplot2.tidyverse.org.

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D’Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. “Welcome to the {Tidyverse}” 4: 1686. https://doi.org/10.21105/joss.01686.