Assignment 1 – Calculus Review

EVR-5086 Fall 2025

Adyan Rios

2025-10-10

Table of contents

Assignment 1 - Calculus Review

1 Plot the polynomial

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.

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

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

```r
# 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 between each value of x
deltay <- diff(y)
deltax <- diff(x)
deltay_deltax <- deltay / deltax

# For plotting purposes, derive the midpoint across the original values of x
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 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() +
  theme(legend.title=element_blank()) +
  labs(title = "EVR-5086 Assignment 1.1", subtitle =
         "Plot of polynomial with analytical and numerical derivatives")

polynomial_plot
```

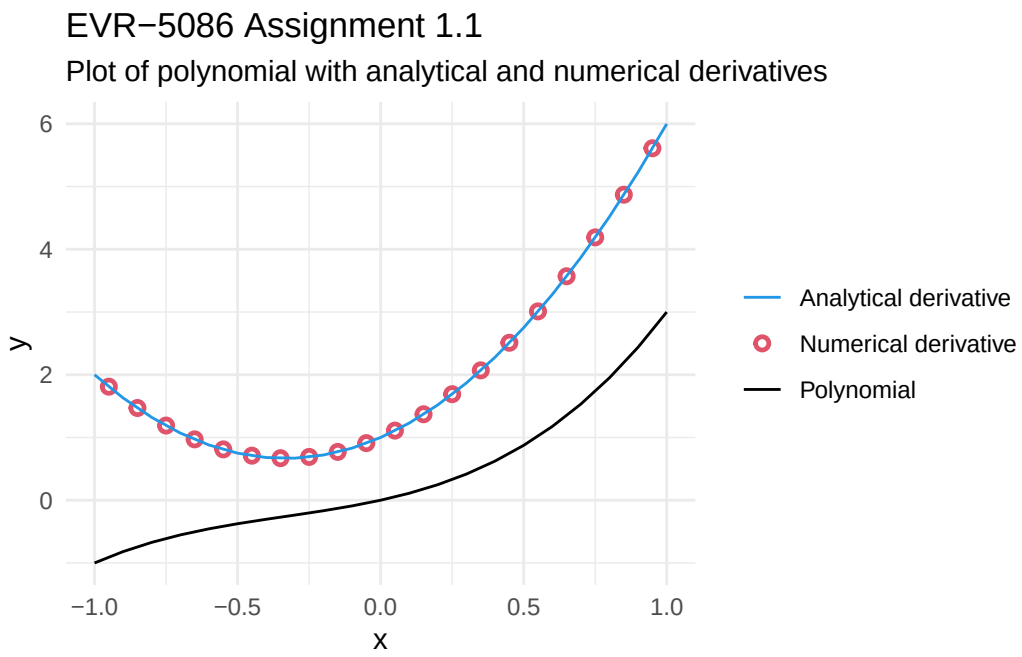

Figure 1: Plot of the polynomial defined by the values provided in EVR-5086 Assignment 1 (black line), along with analytical (blue line) and numerical derivatives (red open circle).

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.

2.1 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.

```r
# Load libraries
library(plot3D)
library(ggplot2)
library(reticulate)
py_require(c("matplotlib"))

# 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()
```

2.2 Switch to Python

Switch to using python to create Figure 2, Figure 3 and Figure 4. 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 csv file from excel
h=np.loadtxt('tripole.csv',delimiter=',')

# Create a grid of x and y coordinates
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)

# Calculate gradient/partial derivatives
[dhdy,dhdx]=np.gradient(h,x_vec,y_vec)
```

2.3 Surface plot

```
fig = plt.figure(figsize=[4,4],dpi=300)
ax = plt.axes(projection = '3d')
ax.set_title(" " * 20 + 'Surface plot using Python'+ " " * 20)
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('H-axis', rotation=90)
```

```
ax.zaxis.labelpad=-0.7
ax.tick_params(axis='x', labelsize=8)
ax.tick_params(axis='y', labelsize=8)
ax.tick_params(axis='z', labelsize=8)
plt.tight_layout()
surf = ax.plot_surface(X,Y,h)
plt.show()
```

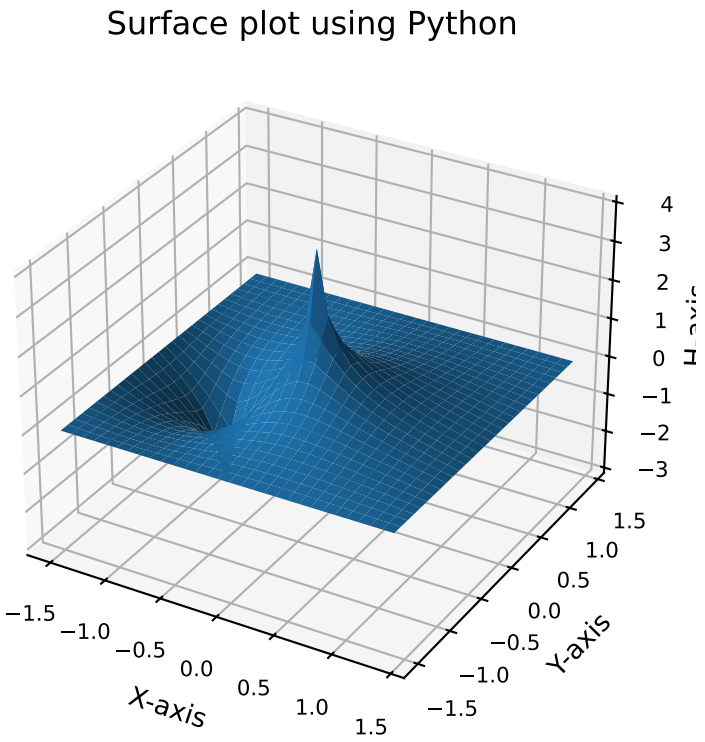

Figure 2: Plot of vector arrows using Python. The vectors indicate strength and direction of the negative gradient. The vectors are displayed over the contours of a tri-pole solution with a high value of 4 and lows of -2 and -3. The two low values were near each other compared to their respective distances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

```
plt.close('all')
```

2.4 Plot contour map and flow vectors

```
plt.contourf(X,Y,h)
cbar=plt.colorbar()
cbar.set_label('Ground water potential surface (h)')
plt.axis('equal');
qplt=plt.quiver(X,Y,-dhdx,-dhdy, scale=360)
plt.title('Contour map and flow vectors')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```

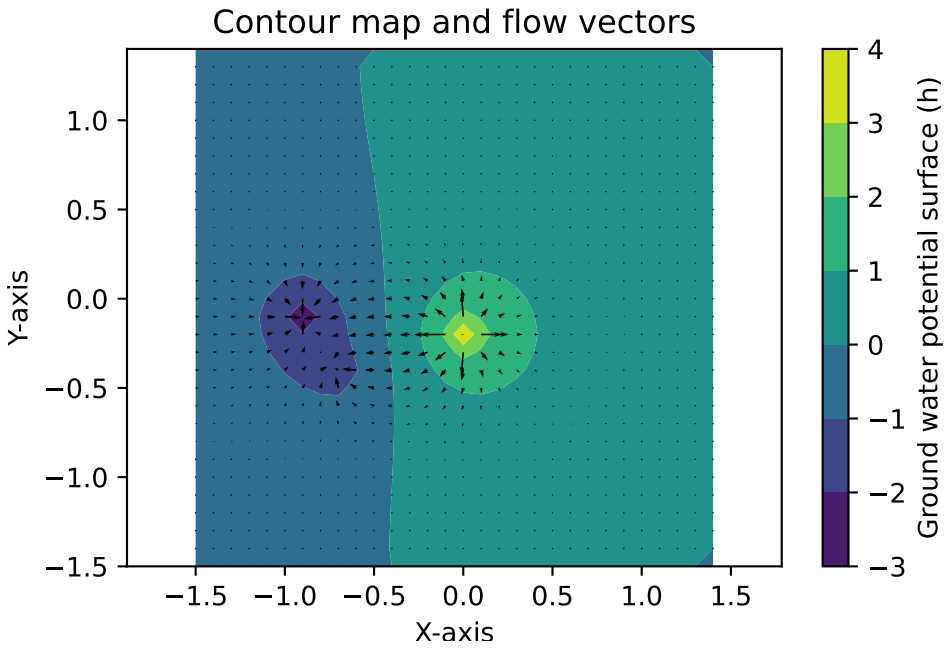

Figure 3: Plot of vector arrows using Python. The vectors indicate strength and direction of the negative gradient. The vectors are displayed over the contours of a tri-pole solution with a high value of 4 and lows of -2 and -3. The two low values were near each other compared to their respective distances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

```
plt.close('all')
```

3 Plot streamlines instead of arrows in Section 2.4

```
# Plot streamlines
plt.contourf(X,Y,h)
cbar=plt.colorbar()
cbar.set_label('Ground water potential surface (h)')
plt.axis('equal')
```

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

```
plt.title('Contour map and streamlines')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.streamplot(X,Y,-dhdx,-dhdy)
```

```
<matplotlib.streamplot.StreamplotSet object at 0x000001AC3E679410>
```

```
plt.show()
```

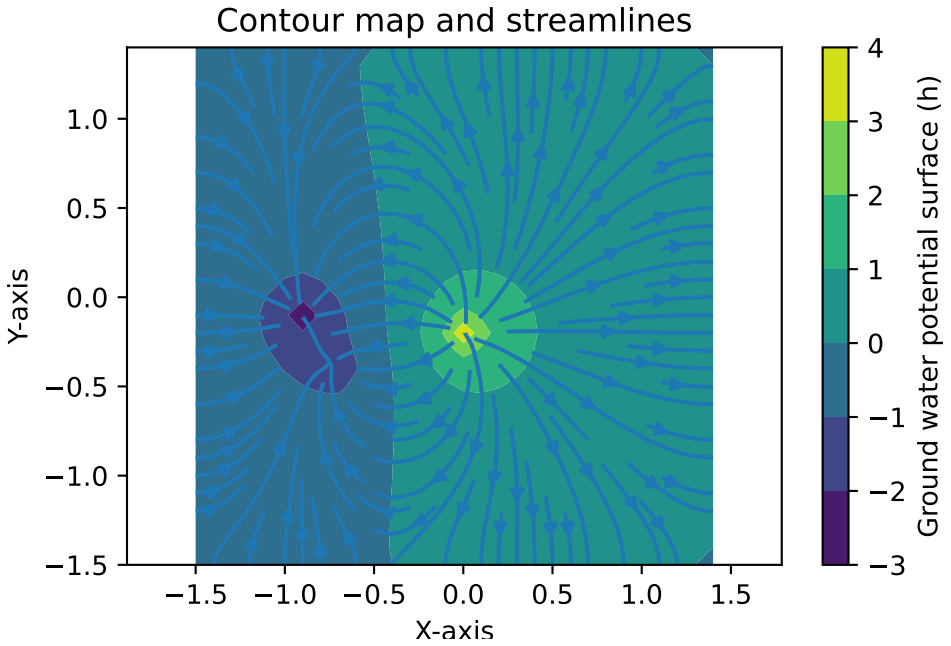

Figure 4: Plot of streamlines using Python. The streamlines are displayed over the contours of a tri-pole solution with a high value of 4 and lows of -2 and -3. The two low values were near each other compared to their respective distances to the high value.

```
plt.close('all')
```

4 Links to Assignment 1 Colab and GitHub

Draft code on Colab

Quarto book chapter on GitHub