Assigment II

Pablo Agustín Ortega-Kral

September 27, 2024

Problem 1. Divided difference interpolation

Subproblem 1.a. *Implement a procedure that interpolates f(x) based on a divided difference approach.*

```python
def divided_diff(X: np.array, Y: np.array) -> np.array:
    # Helper array for storing the divided difference table
    N = len(X)
    divided_diff = np.zeros((N, N))
    divided_diff[:, 0] = Y
    # Build the divided difference table, each column is the kth divided difference. We will
     have N-1 columns.
    for i in range(1, N):
        for j in range(N-i):
            divided_diff[j,i] = (divided_diff[j, i-1] - divided_diff[j+1, i-1])/(X[j] - X[i+
    j])
    return divided_diff

def get_interpolated_value(value: float, X: np.array, divided_diff: np.array) -> float:
    N = len(X)
    interpolated_value = 0
    for n in range(N):
        coeff = divided_diff[0, n]
        term = 1
        # Build newton's polynomial coeff_k(x-x_0)(x-x_1)...(x-x_k-1)
        for i in range(n):
            term *= (value - X[i])
        interpolated_value += coeff * term
    return interpolated_value
```

Listing 1: Divided Difference Implementation

In the implemented approach, I first construct the divided difference table for the provided data points. Effectively, this computes $N-1$ divided differences, where each column will be the kth difference. Once the table has been constructed the `get_interpolated_value` uses the precomputed table to build the interpolating polynomial in Newton's form. The coefficients are in indexed from the table, and we construct the difference by taking the datapoint at the step with the desired x value $(\bar{x}-x_0)\dots(\bar{x}-x_{k-1})$.

Subproblem 1.b. *Use your procedure to interpolate* $\cos \pi x$ *at* $x = \frac{3}{10}$*, based on known values of* $(x, \cos \pi x)$ *at the following* x *locations:* $0, \frac{1}{8}, \frac{1}{4}, \frac{3}{8}, \frac{1}{2}$

Using the above code we obtain that $f(\frac{3}{10}) = 0.5878567543147465$. By reading the coefficients in the divided difference table, we know that the interpolating polynomial found was

$$\begin{aligned}
1.00 &- 0.61(x-0.00)\\
&- 4.50(x-0.00)(x-0.12)\\
&+ 2.82(x-0.00)(x-0.12)(x-0.25)\\
&+ 2.80(x-0.00)(x-0.12)(x-0.25)(x-0.38)
\end{aligned}$$

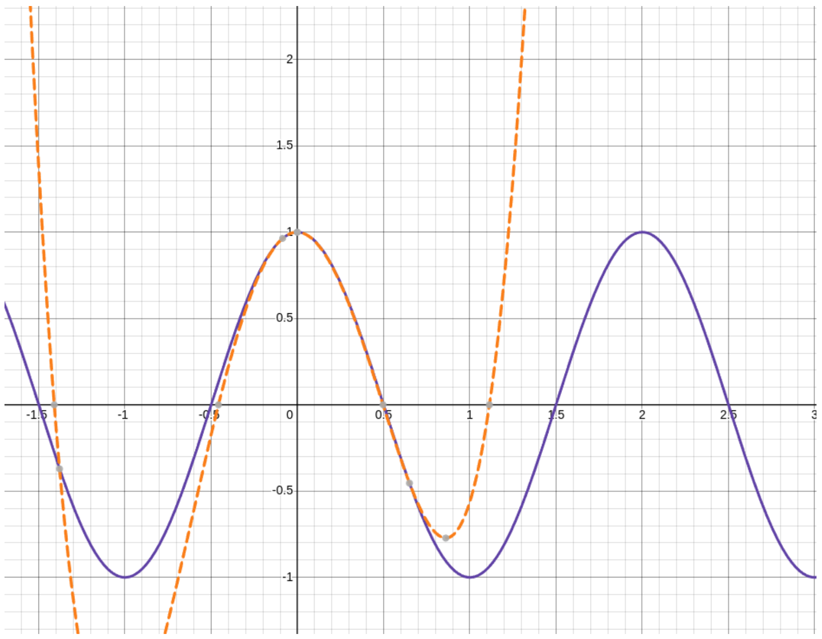


Figure 1: Interpolated polynomial obtained by divided differences. The orange dotted line is the polynomial obtained using the points specified in Q1, while the purple is the function being interpolated $\cos \pi x$

Subproblem 1.c. *Consider the function* $f(x) = \frac{2}{1+9x^2}$. *Use divided differences with the points* $x_i = I\frac{2}{n} - 1$ *and* $i = 0, \ldots, n$. *Interpolate for* 0.07 *with* $n = 2, n = 4, n = 40$. *What is the real value?*

We run the interpolation code using the specified points and report the results in Table 1. Note that we seemingly get close to the real value when increasing the degree; however, as explored in the following section, this does not necessarily indicate a better estimate of the over all polynomial; quite the opposite, it could be a sign of overfitting to the range with the specified points.

Table 1: Estimated values of $f(0.07)$ for different number of points n used in interpolation. Note that the real value $f(0.07) = 1.915525$

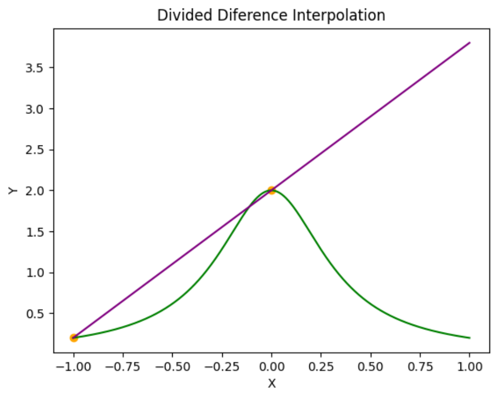
n	Estimated value
2	2.126000
4	2.053825
40	1.915525

Subproblem 1.d. *Estimate the maximum interpolation error. Estimate En for n = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, and 40, for the function*

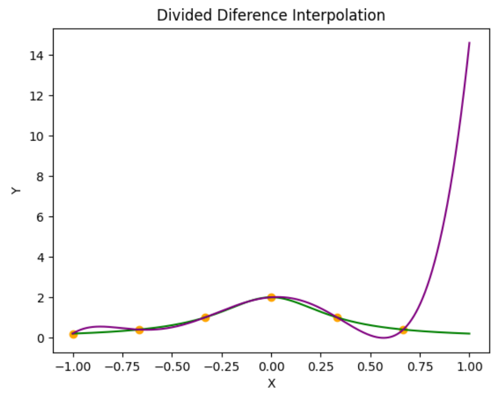
As we increase the degree so too does the maximum error over the specified interval. This is because even though we increase the coverage of the function, we are also increasing the degree of the polynomial, allowing for *overfitting* to the provided datapoints; notice that the function used to generate datapoints means there will not be a point at $x = 1$, so there will always be an error value for an unknown point in the range. Figure 2 illustrates the overfitting problems, notice that even though the polynomial does a better job at passing through the provided datapoints, it is not a better approximation for the underlying function.

Table 2: Interpolation error for different degrees of the interpolating polynomial

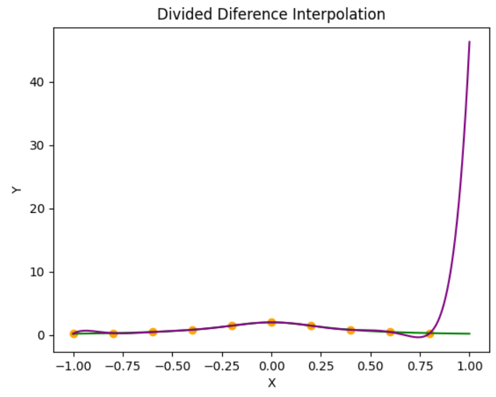
N	Maximum interpolation error
2	3.6
4	7.476923076923078
6	14.40000000000002
8	26.223187946074543
10	46.1385822265148
12	79.41007957559721
14	134.62721401274
16	225.75352859266758
18	375.43119508737647
20	620.2948711349311
40	78006.68436625767



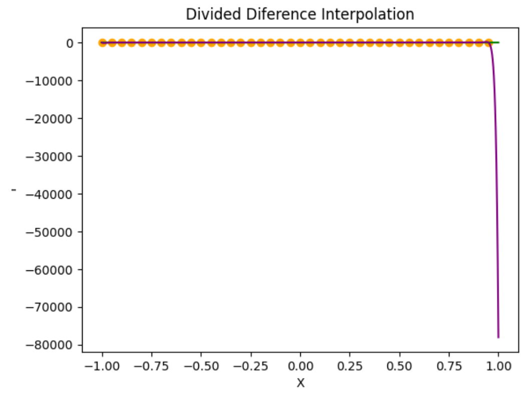
(a) $n = 2$



(b) $n = 6$



(c) $n = 10$



(d) $n = 40$

Figure 2: Interpolating polynomials for $f(x)$ and different number of points.

Problem 2. Suppose you wish to build a "sliding-window interpolation table" (as discussed in class) with entries of the form $(x, f(x))$ for the function $f(x) = sinx$ over the interval $[0, 2\pi]$. Please use uniform spacing between points. How fine must the table spacing be to ensure 6 decimal digit accuracy, assuming that you will use linear interpolation between adjacent points in the table? How fine must the table spacing be if you will use quadratic interpolation? In each case, how many entries do you need in the table?

To solve this problem we can solve the inequality posed by the equation that described the error for an interpolating polynomial

$$e_n(\bar{x}) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{n}^{i=0} (\bar{x} - x_i)$$

For a linear interpolation (ie. a polynomial of first degree $n = 1$), we will slide the window along the interval taking two points. If we want an accuracy of 6 decimal points, then the resulting expression must be greater than $5 \cdot 10^{-7}$

$$5 \cdot 10^{-7} < \left| \frac{f''(\xi)}{2!} (\bar{x} - x_{i-1})(\bar{x} - x_i) \right|$$

Given that, we know the function to interpolate, we can approximate $\left| f''(\xi) \right|$ by looking at the second derivative of the function and analyzing the maximum value in the chosen interval. In this case,

$$\left| f''(\xi) \right| = \max_{x \in [0, 2\pi]} -\sin x$$

$$\left| f''(\xi) \right| = 1$$

For estimating $\left| (\bar{x} - x_{x-1})(\bar{x} - x_i) \right|$ we can formulate this using a proxy function that describes the intervals in terms of a step h, such that $g(y) = (y - h)y$, for convenience, we express as $g(y) = y^2 - hy$ where y is the current position the interval is centered around. Because of this, $\left| \frac{f''(\xi)}{2!} (\bar{x} - x_{i-1})(\bar{x} - x_i) \right| = \max g(y)$. We find this maximum by solving for y when the first derivative equals 0.

$$g'(y) = 2y - h$$

$$0 = 2y - h$$

When at the maximum, $y = \frac{h}{2}$. We can substitute this value in $g(y)$ to find its maximum

$$\max g(y) = g(\frac{h}{2})$$

$$g(\frac{h}{2}) = \frac{h^2}{4} - \frac{h^2}{2}$$

$$g(\frac{h}{2}) = -\frac{h^2}{4}$$

$$\max g(y) = \left| g(\frac{h}{2}) \right|$$

$$\max g(y) = \frac{h^2}{4}$$

Substituting these values in our original inequality,

$$5 \cdot 10^{-7} < \left| \frac{1}{2!} (\frac{h^2}{4}) \right|$$

Solving for h

$$h > \sqrt{5 \cdot 10^{-7} \cdot 8}$$

$$h = 0.002$$

With a step $h = 0.002$ for the interval $[0, 2\pi]$ we have approximately $3,142$ points. That is, for 6 decimal places of accuracy the interpolation table needs $3,142$ points.

For the quadratic, we conduct a similar analysis. Given that the degree of our desired polynomial is now 2 ($n = 2$) we need 3 points in the sliding window. Thus the error equation is

$$5 \cdot 10^{-7} < \left| \frac{f^{'''}(\xi)}{3!}(\bar{x} - x_{i-1})(\bar{x} - x_i)(\bar{x} - x_{i+i}) \right|$$

We estimate $f^{'''}(\xi)$ as before. In this case $\max_{x \in [0,2\pi] - cosx} = 1$. As we now are interpolating using 3 points, $g(y) = (y - h)y(y + h)$, thus $g(y) = y^3 - yh^2$. The maximum can be found at

$$g'(y) = 3y^2 - h^2$$
$$0 = 3y^2 - h^2$$
$$y = \sqrt{\frac{h^2}{3}}$$
$$y = \pm\frac{h}{\sqrt{3}}$$

$$\max g(y) = g(\frac{h}{\sqrt{3}}) =$$
$$\max g(y) = \frac{2}{3}\frac{1}{\sqrt{3}}h^3$$

Substituting in the error inequality

$$5 \cdot 10^{-7} < \left| \frac{1}{3!}(\frac{2}{3}\frac{1}{\sqrt{3}}h^3) \right|$$
$$5 \cdot 10^{-7} < \left| \frac{h^3}{9\sqrt{3}} \right|$$

Solving for h

$$h > \sqrt[3]{5 \cdot 10^{-7} \cdot 9\sqrt{3}}$$

$$h = 0.01982$$

With a step $h = 0.01982$ for the interval $[0, 2\pi]$ we have approximately 317 points. That is, for 6 decimal places of accuracy the interpolation table needs $3,142$ points using quadratic interpolation.

Problem 3. Implement Newton's Method. Consider the following equation:

$$f(x) = \tan x - x$$

There are an infinite number of solutions x to this equation. Use Newton's method to find the two solutions on either side of 15. In other words, find two solutions xlow ¡ 15 ¡ xhigh such that the interval [xlow, xhigh] contains no other solutions. — Use any techniques you need to start Newton in regions of convergence. Those regions may be very small.

```python

def initial_root_estimate(f: callable, pivot: float, search_range: int = 3, step: float =
    0.001) -> float:
    lower_bound = pivot
    upper_bound = pivot

    prev_val =lower_bound
    while lower_bound > pivot - search_range:
        # Check if there is a sign change
        if f(lower_bound) * f(prev_val) < 0:
            break
        prev_val =lower_bound
        lower_bound -= step

    prev_val = upper_bound
    while upper_bound < pivot + search_range:
        # Check if there is a sign change
        if f(upper_bound) * f(prev_val) < 0:
            break
        prev_val = upper_bound
        upper_bound += step

    return lower_bound, upper_bound

def get_newton_root(fuction: callable, derivative: callable, x0: float, tol: float, max_iter
    : int) -> tuple:
    x = x0
    for i in range(max_iter):
        if derivative(x) == 0:
            return -1, i, x
            print("Derivative is zero, can't continue")

        if abs(fuction(x)) < tol:
            print(f"Converged in {i} iterations")
            return 0, i, x

        x = x - fuction(x) / derivative(x) # Use newton's update rule

    print(f"Did not converge in {max_iter} iterations")
    return -1, i, x
```

Listing 2: Newton Root Finding

Newton's method assumes that the function is known, as well as a continuous derivative for it. We see that we iteratively follow the derivative and update x by

$$x_m = x_{m-1} - \frac{f(x)}{f'(x)}$$

This method relies on an initial guess, and if this is too far it might fail to converge. In this case, we can employ certain tricks to make an educated initial guess. We know that roots are points where the functions cross the x-axis and therefore change sign. Therefore, we can find good initial guesses around our search point by finding points where the function changes signs. I implement a function that searches within a certain window of a point of interest and increments (or decrements) in discrete steps, stopping if a sign change is detected.

By doing this with the point of interest $x = 15$, I obtain $x_{low} = 14.137$ and $x_{high} = 17.221$. Running Newton's method with these initial guess yields,

$$x_1 = 14.066$$

$$x_2 = 17.2208$$

For x_{low} this takes 13 iterations; while for the high end, this takes 2.

Problem 4. Suppose ξ is a root of order 2 of $f(x)$, meaning $f(\xi) = 0$, $f'(\xi) = 0$ and $f''(\xi) \neq 0$

Note, this problem was solved using [1] as a reference

Subproblem 4.a. *Show that in this case Newton's method no longer converges quadratically. Do so by showing that the method now converges linearly.*

Applying the Taylor series expansion formula $\sum_{n=0}^{\inf} \frac{f^{(n)(a)}}{n!}(x-a)^n$, we can expand $f(\xi)$ and $f'(\xi)$, and considering $n = 2$

$$f(x) \approx \frac{(x-\xi)^2}{2!} f''(\xi)$$

$$f'(x) \approx \frac{(x-\xi)}{1!} f''(\xi)$$

Plugin in to the update function $u(x_n) = x_n + \frac{f(x_n)}{f'(x_n)}$, we have

$$u(x_n) = x_n + \frac{\frac{(x-\xi)^2}{2!} f''(\xi)}{\frac{(x-\xi)}{1!} f''(\xi)}$$

Simplifying by eliminating terms,

$$u(x_n) = x_n + \frac{(x_n - \xi)}{2}$$

Rewriting $u(x) - \xi$ as the convergence error e_{n+1},

$$e_{n+1} = -\frac{(x_n - \xi)}{2}$$

We can see that the error reduces constantly by $\frac{1}{2}$ each iteration.

Subproblem 4.b. *We can modify Newton's method in this case: Show that the iteration*

$$x_{n+1} = x_n - 2\frac{f(x_n)}{f'(x_n)}$$

We can solve this by substituting the expansions in the updated formula. From last step we know that $\frac{f(x)}{f'(x)} = \frac{(x_n - \xi)}{2}$, therefore

$$u(x) = x_n - 2\frac{(x_n - \xi)}{2} = x_n - (x_n - \xi)$$

$$u(x) = \xi$$

We see that the update function is constant, therefore it will converge in one step.

Problem 5. Implement Muller's method. Use Muller's method to find good numerical approximations for all the real and complex roots of the polynomial

$$p(x) = x^3 + x + 1$$

```python
import numpy as np

def deflated_poly(coeffs: list, root: float) -> list:
    deflated_poly, _ = np.polynomial.polynomial.polydiv(coeffs, [-root, 1])
    return deflated_poly

def poly_from_coeffs(coeffs: list) -> callable:
    def poly(x: float) -> float:
        return np.polynomial.polynomial.polyval(x, coeffs)
    return poly

def mueller_root_finder(f: callable, guesses: float, max_iter: int = 10000, tol: float = 1e
    -8) -> float:
    iter = 0
    x0, x1, x2 = guesses
    while iter < max_iter:
        f2 = f(x2)
        f1 = f(x1)
        f0 = f(x0)
        # Calculate the step and function differences
        dx1, dx2 = x1 - x0, x2 - x1
        df1 = (f1 - f0) / dx1
        df2 = (f2 - f1) / dx2

        # Calculate the coefficients of the quadratic interpolation
        c = f2
        a = (df2 - df1) / (dx2 + dx1)
        b = a * dx2 + df2

        # Calculate the denominator and pick value the maximize the denominator
        D = np.lib.scimath.sqrt(b**2 - 4 * a * c)
        if abs(b + D) > abs(b - D):
            denominator = b + D
        else:
            denominator = b - D
        # Check if denominator is zero, to avoid division by zero
        if denominator == 0:
            print("denominatorinator is zero")
            return None, iter

        root = 2 * c / denominator
        x3 = x2 - root

        if abs(root) < tol:
            return x3, iter

        x0, x1, x2 = x1, x2, x3
        iter += 1

    return x3, iter

def get_roots(coefficients, x0 = 1.0, x1 = 1.5, x2 = 2.0):
    # Perform muller root finding and deflation N times
    N = len(coefficients) - 1
    roots = []
    for n in range(N):
        root, _ = mueller_root_finder(poly_from_coeffs(coefficients), guesses= (x0, x1, x2))
        roots.append(root)
        print(f"Root {n}: {root}")
        coefficients = deflated_poly(coefficients, root)
    return roots
```

Listing 3: Muller's Root Finding

Muller's root-finding algorithm can be viewed as a generalization of the secant method. Here, we start with 3 initial guesses for the polynomial; we use the divided differences method to find a quadratic interpolating polynomial; the next proposal for the roots will be the roots of the interpolating polynomial; we do this iteratively until convergence. Note that for numerical stability, we choose the value of the discriminator such that is maximizes the value of the denominator in the formula for finding the roots of a quadratic. Given that we are solving for quadratic roots, this method allows us to find complex numbers.

For finding all roots of a polynomial using this method, we can iteratively deflate using the discovered roots. That is for each root x_n we obtain a lower degree polynomial by dividing by this root. $\frac{p(x)}{x-r_1}$.

Using this method for $p(x) = x^3 + x + 1$, with initial guesses $x_0 = -0.683; x_1 = 0; x_2 = 3.001$, we obtain

$$r_1 = -0.6823278$$
$$r_2 = 0.34116 + 1.16154i$$
$$r_3 = 0.34116 - 1.16154i$$

It is worth noting that due to numerical error, the real root is estimated to have a small imaginary part, concretely $1.0521 \cdot 10^{-22}$

Problem 6. Consider the two univariate polynomials

$$p(x) = x^3 - 3x^2 + x - 3$$
$$q(x) = x^2 + x - 12$$

Subproblem 6.a. *Using the method of resultants, decide whether $p(x)$ and $q(x)$ share a common root.*

We first begin by multiplying $p(x)$ by x and $q(x)$ by x, such that,

$$p_s(x) = x^4 - 3x^2 + x^2 - 3x$$
$$q_s(x) = x^3 + x^2 - 12x$$

Note that we must again multiply $q_s(x)$ to match the degree of $p_s(x)$.

$$q_{s2} = x^4 + x^3 - 12x^2$$

We can write these equations as a linear system $Qx = 0$ such that,

$$\begin{pmatrix} 1 & -3 & 1 & -3 & 0 \\ 0 & 1 & -3 & 1 & -3 \\ 1 & 1 & -12 & 0 & 0 \\ 0 & 1 & 1 & -12 & 0 \\ 0 & 0 & 1 & 1 & -12 \end{pmatrix} \begin{pmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = 0$$

We can then analyze the determinant to determine if both equations share a common root. Using *WolframAlpha*, we get that the $\texttt{det}(Q) = 0$, this indicates that indeed, **the given polynomials share a common root.**

Subproblem 6.b. *If the two polynomials share a common root, use the ratio method discussed in class to find that root.*

Using the ratio method, we can leverage the equations built in the left step to define a quotient, simplify, and solve for the given root. In particular, we can work directly over Q and remove the necessary columns. The subscripts indicate the removed columns

$$x = -1\frac{\texttt{det}(Q_1)}{\texttt{det}(Q_2)}$$

We can evaluate $\texttt{det}(Q_1) = \det \begin{pmatrix} -3 & 1 & -3 & 0 \\ 1 & -3 & 1 & -3 \\ 1 & -12 & 0 & 0 \\ 1 & 1 & -12 & 0 \end{pmatrix} = 1377$ and $\texttt{det}(Q_2) = \det \begin{pmatrix} 1 & 1 & -3 & 0 \\ 0 & -3 & 1 & -3 \\ 1 & -12 & 0 & 0 \\ 0 & 1 & -12 & 0 \end{pmatrix} = -459$

There for the root is $x = -\frac{1377}{-459} = 3$

Problem 7. Consider the two bivariate polynomials

$$p(x, y) = 2x^2 + 2y^2 - 4x - 4y + 3$$
$$q(x, y) = x^2 + y^2 + 2xy - 5x - 3y + 4$$

Subproblem 7.a. *Draw the zero contour $p(x, y) = 0$ and the zero contour $q(x, y) = 0$, with x and y real*

The following figure shows the contours for the function. The code generated to draw them is provided in Appendix A.

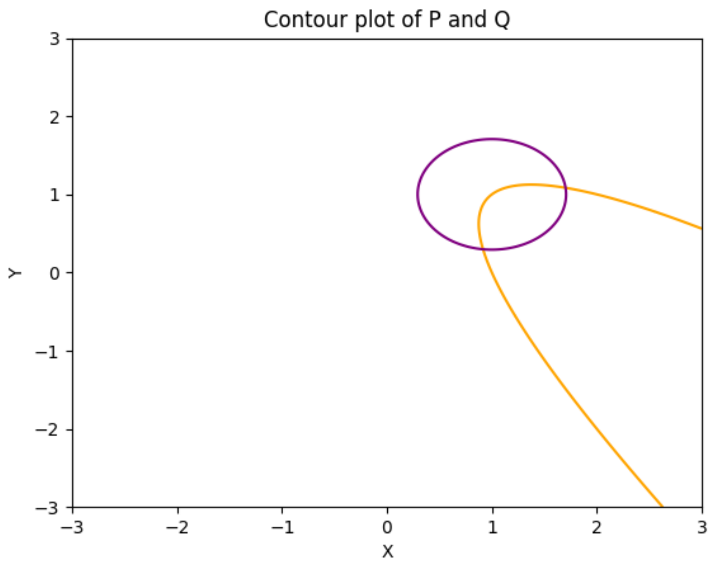


Figure 3: Counter plot for bivariate functions $p(x, y)$ and $q(x, y)$

Subproblem 7.b. *Using the method of resultants, solve for the intersection points of these contours, i.e., find all (x, y) for which p(x, y)=0= q(x, y), with x and y real. Do so by treating the polynomials p and q as functions of y, temporarily viewing x as a constant. Obtain a 4 × 4 matrix whose determinant, now a function of x, is the desired resultant. Find the x-roots of that determinant, thereby projecting the intersection points onto the x-axis. Use those x-roots to find the y-coordinates of the intersection points. Verify your intersection points.*

Similar to question 6, we will construct a surrogate function to then solve the system $Qx = 0$, in this case, we achieve this for bivariate functions by expressing the functions concerning one variable. Analyzing the functions taking y as variable and x as constant we get

$$p(y) = 2y^2 - 4y + (2x^2 - 4x + 3)$$
$$q(y) = y^2 + (2x - 3)y + (x^2 - 5x + 4)$$

Multiplying by y as done in 6, we can build the following matrix

$$\begin{pmatrix} 2 & -4 & 2x^2 - 4x + 3 & 0 \\ 0 & 2 & -4 & 2x^2 - 4x + 3 \\ 1 & 2x - 3 & x^2 - 5x + 4 & 0 \\ 0 & 1 & 2x - 3 & x^2 - 5x + 4 \end{pmatrix} x = 0$$

The intersections will be those points where x makes the determinant 0, therefore, we must solve the equation posed by the determinant of this matrix.

$$\det Q = 16x^4 - 48x^3 + 48x^2 - 28x + 11$$

For the intersections, we look at the real roots of the determinant

$$r = \frac{1}{4}(3 + \sqrt{5} \pm \sqrt{2(\sqrt{5} - 1)})$$

$$x_1 \approx 0.91594130$$

$$x_2 \approx 1.70209$$

Finally, we can substitute these root into $p(y)$ and solve for the corresponding y,

$$0 = 2y^2 - 4y + (2x^2 - 4x + 3)$$

For x_1,

$$2y^2 - 4y + 1.01412 = 0$$

$$y_1 = 0.29790$$

For x_2,

$$2y^2 - 4y + 1.98586 = 0$$

$$y_2 = 1.0841$$

Thus the intersections are $(0.91594130, 0.29790)$ and $(1.70209, 1.0841)$. We can verify this by plotting on the original contour graph

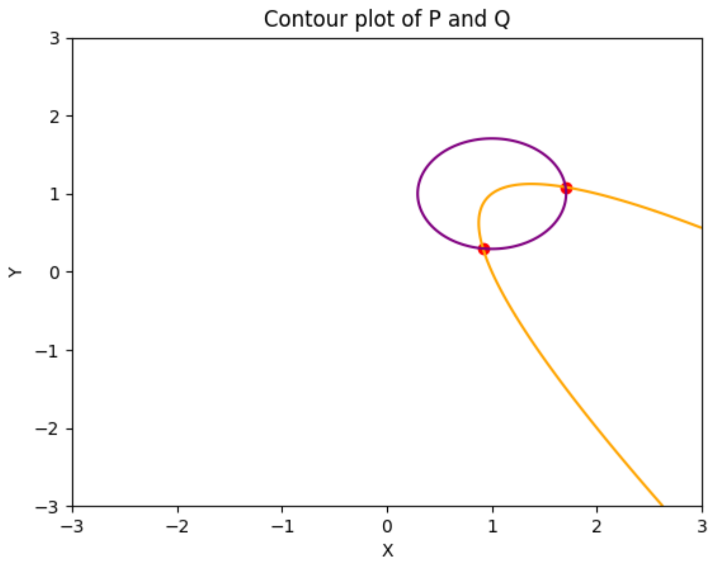


Figure 4: Countour intersections

Problem 8. You are preparing a robotic unicycle to take part in a circus act. For most of the show, a talented acrobat rides the unicycle. But at one point, the acrobat jumps off the unicycle onto a trapeze. After a short trapeze act, the acrobat leaps through the ring of fire in the center of the stage to land on the waiting unicycle, which has moved autonomously to the other side. Your job is to plan a path for the unicycle to take around the ring of fire to the acrobat's landing point.

You are given a precomputed set of paths which all begin at different points, avoid the ring of fire, and end at the destination (shown below). You must mimic these paths as closely as possible since they are precisely choreographed for the circus act. However, the acrobat is only human and does not position the unicycle precisely at any of the paths' starting points. You will need to interpolate a new path from other paths with nearby starting points

Subproblem 8.a. *Write a system of linear equations (in the form $Av = b$, with v representing variables of some sort, appropriately chosen) and constraints that will help you determine whether a 2D point (x, y) falls within the triangle formed by three 2D points*

We can make use of the properties of barycentric coordinates. A point inside a triangle with vertices A,B,C and be expressed as

$$P = \alpha A + \beta b + \gamma C$$

Where (α, β, γ) represent barycentric coordinates with the constraint that they must be greater than or equal to 0, and their sum must be the unit, ie. $\alpha + \beta + \gamma = 1$. Applying this to the given points we can construct a linear system such that

$$\begin{pmatrix} x_i & x_j & x_k \\ y_i & y_j & y_k \\ 1 & 1 & 1 \end{pmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Subproblem 8.b.

References

[1] Amzoti. (2013, May 13). Convergence rate of Newton's method. In Math Stack Exchange. Retrieved from https://math.stackexchange.com/questions/389368/convergence-rate-of-newtons-method

[2] Barycentric coordinates (no date) from Wolfram MathWorld. Available at: https://mathworld.wolfram.com/BarycentricCoordinates.html (Accessed: 26 September 2024).

Appendices

A Contour Drawing Code

```python
import matplotlib.pyplot as plt

x = np.linspace(x_min, x_max, n_points)
y = np.linspace(x_min, x_max, n_points)

X, Y = np.meshgrid(x, y)

P = 2*X**2 + 2*Y**2 - 4*X -4*Y + 3
Q = X**2 + Y**2 + 2*X*Y -5*X -3*Y +4

plt.contour(X, Y, Q, levels=[0], colors='orange')
plt.contour(X, Y, P, levels=[0], colors='purple')
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Contour plot of P and Q")
plt.savefig("results/contour_plot.png")
```

Listing 4: Countour drawing