STAT 31050: Homework 4

Caleb Derrickson

May 8, 2024

Collaborators: Karl Tayeb, and Matt Frazier.

Contents

Exercise 22

Consider the integral operator $T : L^2[-1,1] \to L_2[-1,1]$ defined by

$$T[f](x) = \int_{-1}^{1} e^{-(x-y)^2} f(y)\, dy,\ x \in [-1,1]$$

Exercise 22, part 1

Prove that $T[f]$ is continuous if $f \in L_2[-1,1]$.

Solution:

Here, I will take a page from functional analysis, showing that the operator T is bounded, linear, and maps continuous functions to continuous functions. Note that it is sufficient to show T being a bounded linear operator, since it is continuous by virtue of this fact. To show linearity, let $f, g \in L^2[-1,1]$, and $\alpha, \beta \in \mathbb{R}$. Then,

$$T[\alpha f + \beta g] = \int_{-1}^{1} e^{-(x-y)^2} [\alpha f + \beta g]\,(y)\, dy = \alpha \int_{-1}^{1} e^{-(x-y)^2} f(y)\, dy + \beta \int_{-1}^{1} e^{-(x-y)^2} g(y)\, dy$$

$$= \alpha T[f](x) + \beta T[g](x).$$

Next, we show boundedness of T. First, since suppose $f \in L^2[-1,1]$, its norm is well defined in the L^2 sense. We will show that $\|T[f]\|_{L^2} \leq M\|f\|_{L^2}$. Then,

$$\begin{aligned}
\|T[f]\|_{L^2} &= \int_{-1}^{1} \left| \int_{-1}^{1} e^{-(x-y)^2} f(y)\, dy \right|^2 dx \\
&\leq \int_{-1}^{1} \int_{-1}^{1} \left| e^{-(x-y)^2} f(y) \right|^2 dy\, dx \\
&\leq \int_{-1}^{1} \int_{-1}^{1} \left(\sup_{y \in [-1,1]} \left| e^{-(x-y)^2} \right|^2 \right) |f(y)|^2\, dy\, dx \\
&\leq \left(\sup_{x,y \in [-1,1]} \left| e^{-(x-y)^2} \right|^2 \right) \int_{-1}^{1} \int_{-1}^{1} |f(y)|^2\, dy\, dx \\
&= 2 \left(\sup_{x,y \in [-1,1]} \left| e^{-(x-y)^2} \right|^2 \right) \int_{-1}^{1} |f(y)|^2\, dy \\
&= M\|f\|_{L^2}
\end{aligned}$$

Where M is the sup squared of the given exponential. Note that the sup of the exponential is easy to find : since the squared exponential cannot be larger than one, M = 1. Therefore, the mapping is bounded and hence continuous.

Exercise 22, part 2

Using a Gauss-Legendre quadrature rule, write a code that discretizes the integral using a 40 point Gauss-Legendre quadrature and evaluates it at arbitrary points $x \in [-1, 1]$. You may assume f is continuous. Plot the output for $f \equiv 1$ and $f \equiv x$. Then, increase the number of nodes to 80 and compare the output with the 40 point rule.

Solution:

I have implemented the Gauss-Legendre rule in Python, and have used it to integrate the given functions. My results are given below. Due the fast convergence of the integration, I have also provided the errors for the two below. After these plots, I will provide my code.

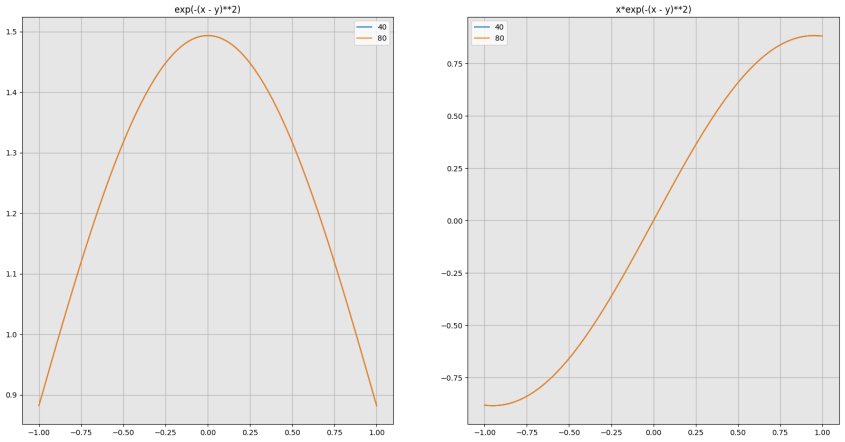


Figure 1: Gauss - Quadrature with $n = 40, 80$ for the two functions.

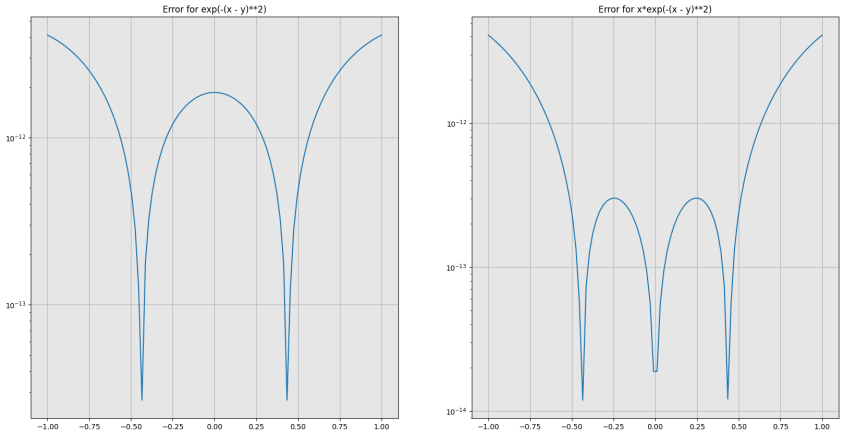


Figure 2: Relative difference between the two integrations.

In [ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import sympy as sm
```

In [ ]:
```
def gauleg(x1, x2, x, w, n):
    EPS = 3.0e-11
    m = (n + 1) // 2  # Find only half the roots because of symmetry
    xm = 0.5 * (x2 + x1)
    xl = 0.5 * (x2 - x1)
    for i in range(1, m + 1):
        z = math.cos(math.pi * (i - 0.25) / (n + 0.5))
        while True:
            p1 = 1.0
            p2 = 0.0
            for j in range(1, n + 1):
                # Recurrence relation
                p3 = p2
                p2 = p1
                p1 = ((2.0 * j - 1.0) * z * p2 - (j - 1.0) * p3) / j
            # Derivative
            pp = n * (z * p1 - p2) / (z * z - 1.0)
            z1 = z
            # Newton's method
            z = z1 - p1 / pp
            if abs(z - z1) <= EPS:
                break
        x[i] = xm - xl * z
        x[n + 1 - i] = xm + xl * z
        # Weights
        w[i] = 2.0 * xl / ((1.0 - z * z) * pp * pp)
        w[n + 1 - i] = w[i]
```

In [ ]:
```
def Gauss_Legendre_Quad(fs, weights: np.ndarray, zeros: np.ndarray) :
    sum = 0
    n = len(weights)
    y = sm.symbols('y')
    for i in range(n):
        sum += weights[i] * fs.subs(y, zeros[i])
    return sum

y = sm.symbols('y')
x = sm.symbols('x')
funcs = [sm.exp(-(x - y)**2), x*sm.exp(-(x - y)**2)]
```

In [ ]:
```
# Input the number of quadrature points
ns = [40, 80]
f_res = np.zeros((2*len(ns), 100))
xspan = np.linspace(-1, 1, 100)
# Allocate arrays x and w

for n, N in enumerate(ns):
    xs = [0.0] * (N + 1)
    ws = [0.0] * (N + 1)

    # Call the gauleg function
    gauleg(-1.0, 1.0, xs, ws, N)

    for i, fs in enumerate(funcs):
        # Calculate the integration
        res = Gauss_Legendre_Quad(fs, ws, xs)

        for k in range(len(xspan)):
            f_res[i*len(ns) + n, k] = res.subs(x, xspan[k])
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize = (20, 10))

for i in range(len(axs)):
    for j in range(len(ns)):
        axs[i].plot(xspan, f_res[j + i*len(ns)], label = f"{ns[j]}")
        axs[i].legend()

    axs[i].grid(True)
    axs[i].set_facecolor("#E6E6E6")
    axs[i].set_title(f"{funcs[i]}")
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize = (20, 10))

for i in range(len(axs)):
    axs[i].plot(xspan, np.abs(f_res[0 + i*len(ns)] - f_res[1 + i*len(ns)]))
    axs[i].grid(True)
    axs[i].set_yscale('log')
    axs[i].set_facecolor("#E6E6E6")
    axs[i].set_title(f"Error for {funcs[i]}")
```

Exercise 22, part 3

Construct the matrix that takes f sampled at an n point Gauss-Legendre rule and maps it to $T[f]$ evaluated at the same points, i.e., K maps $(f(x_1), ..., f(x_n))$ (approximately) to $(T[f](x_1), ..., T[f](x_n))$. Diagonalize K with $n = 40$ and plot all eigenvalues and the first 4 eigenfunctions. Now set $n = 80$ and repeat. Compare the eigenvalues for $n = 40$ and $n = 80$. What does this say about the original integral operator T? You don't need to be very precise here.

Solution:

At any given x, Gauss-Legendre integration will look like (for these functions)

$$T[f](x) = \int_{-1}^{1} e^{-(x-y)^2} f(y)\, dy \approx \sum_{i=1}^{n} e^{-(x-y_i)^2} f(y_i) w_i$$

When sampling $T[f](x)$ at the n nodes obtained from the Gauss-Legendre rule, we will get

$$T[f](x_j) = \sum_{i=1}^{n} e^{-(x_j-y_i)^2} f(y_i) w_i$$

The matrix K will then have entries $K_{i,j} = e^{-(x_j-y_i)^2} w_i$. After constructing the matrices, extracting the eigenvalues, we get the two plots below. It really seems as though there is no true difference between the two methods, since the spectrum bottoms out after $n = 15$. Also, it seems as though the two functions have the same spectrum (For the spectrum, I am taking the absolute value of the eigenvalues, since we have to worry about complex numbers). My code to generate these two plots will be given afterwards.

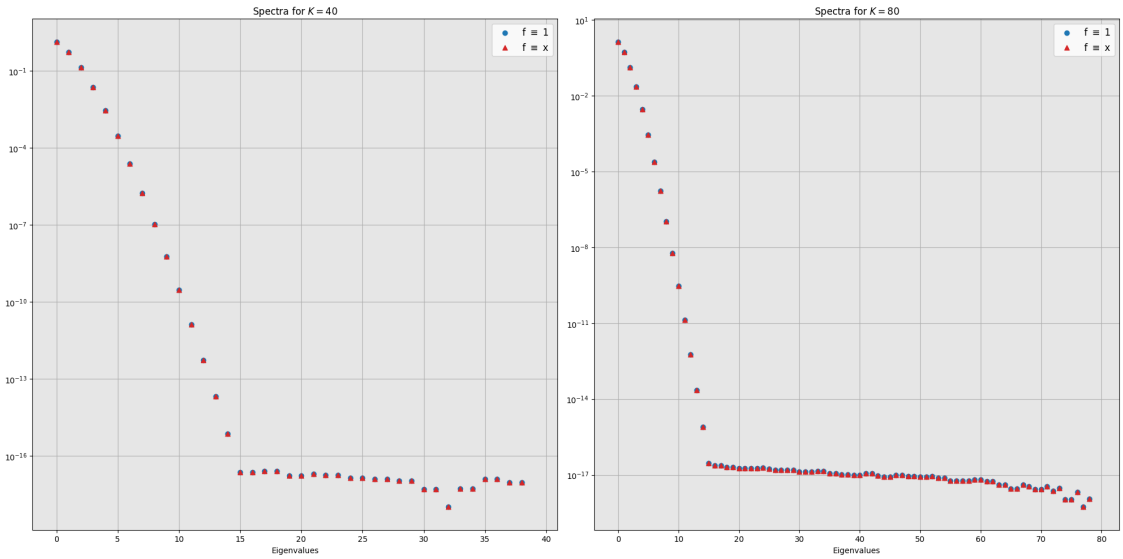


Figure 3: Spectrum for the Operator for $N = 40, 80$.

In [ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import sympy as sm
import scipy as scp
```

In [ ]:
```
def gauleg(x1, x2, x, w, n):
    EPS = 3.0e-11
    m = (n + 1) // 2  # Find only half the roots because of symmetry
    xm = 0.5 * (x2 + x1)
    xl = 0.5 * (x2 - x1)
    for i in range(1, m + 1):
        z = math.cos(math.pi * (i - 0.25) / (n + 0.5))
        while True:
            p1 = 1.0
            p2 = 0.0
            for j in range(1, n + 1):
                # Recurrence relation
                p3 = p2
                p2 = p1
                p1 = ((2.0 * j - 1.0) * z * p2 - (j - 1.0) * p3) / j
            # Derivative
            pp = n * (z * p1 - p2) / (z * z - 1.0)
            z1 = z
            # Newton’s method
            z = z1 - p1 / pp
            if abs(z - z1) <= EPS:
                break
        x[i] = xm - xl * z
        x[n + 1 - i] = xm + xl * z
        # Weights
        w[i] = 2.0 * xl / ((1.0 - z * z) * pp * pp)
        w[n + 1 - i] = w[i]
```

In [ ]:
```
def Gauss_Legendre_Quad(fs, weights: np.ndarray, zeros: np.ndarray) :
    sum = 0
    n = len(weights)
    y = sm.symbols('y')
    for i in range(n):
        sum += weights[i] * fs.subs(y, zeros[i])
    return sum

y = sm.symbols('y')
x = sm.symbols('x')
funcs = [1, x]
Tfuncs = [sm.exp(-(x - y)**2), x*sm.exp(-(x - y)**2)]
```

In [ ]:
```
# Input the number of quadrature points
ns = [40, 80]
xspan = np.linspace(-1, 1, 100)
eigvals_40 = np.zeros((2, ns[0]))
eigvals_80 = np.zeros((2, ns[1]))
# Allocate arrays x and w

for n, N in enumerate(ns):
    xs = [0.0] * (N + 1)  # Zeros of Gauss-Legendre
    ws = [0.0] * (N + 1)

    # Call the gauleg function
    gauleg(-1.0, 1.0, xs, ws, N)

    for i, fs in enumerate(Tfuncs):
        res = Gauss_Legendre_Quad(fs, ws, xs)
        K = np.zeros((N, N))
        for k in range(N):
            for l in range(N):
                K[k][l] = np.exp(-(xs[l] - xs[k])**2) * ws[k]
        eigvals = scp.linalg.eigvals(K)

        if N == 40:
            for j in range(ns[0]):
                eigvals_40[i, j] = abs(eigvals[j])
        if N == 80:
            for j in range(ns[1]):
                eigvals_80[i, j] = abs(eigvals[j])
```

In [ ]:
```
colors = ['tab:blue', 'tab:red']
eigs = [eigvals_40, eigvals_80]
fig, axs = plt.subplots(1, 2, figsize=(20, 10))
labels = ["f $\equiv$ 1", "f $\equiv$ x"]

for i in range(len(eigs[0])):
    # Plot the scatter plots and specify labels and markers
    axs[i].scatter(range(len(eigs[i][0])), eigs[i][0], c=colors[0], label=labels[0])
    axs[i].scatter(range(len(eigs[i][1])), eigs[i][1], c=colors[1], label=labels[1], marker='^')
    axs[i].set_yscale('log')
    axs[i].set_facecolor("#E6E6E6")
    axs[i].set_xlabel("Eigenvalues")
    axs[i].set_title(f"Spectra for $K = {ns[i]}$")
    axs[i].grid(True)
    axs[i].set_axisbelow(True)
    # Add legend for each plot
    axs[i].legend(loc='upper right', fontsize='large')

plt.tight_layout()
plt.show()
```

Exercise 23

In this problem we will play around with polynomial root finding. In the following suppose that q_n, $n = 0, 1, 2, ...$, is a sequence of polynomials which satisfies a three-term recurrence

$$q_{n+1}(x) = (\alpha_n x + \beta_n)q_n(x) + \gamma_n q_{n-1}(x),\ n > 0,$$

where α_n, β_n, and γ_n are real numbers and the α_n are bounded away from zero. Also, suppose $q_1 = (a_0 x + b_0)q_0(x)$. Suppose p is a polynomial with

$$p(x) = \sum_{j=0}^{n} c_j q_j(x),$$

for some coefficients $c_0, ..., c_n$ with $c_n \neq 0$. Hint: if you get stuck, look at Trefethen's book Approximation Theory and Approximation Practice, Chapter 18, but try not to!

Exercise 23, part 1

Find a linear map which sends the vector

$$v = [q_0(x), q_1(x), ..., q_{n-1}(x)]^T$$

to

$$[xq_0(x), xq_1(x), ..., xq_{n-2}(x), xq_{n-1} - q_n/\alpha_{n-1}]^T$$

for all $x \in [-1, 1]$. Your matrix should not depend on x. Is this map unique (assuming the independence of x)?

Solution:

Define T as our linear map. Let us examine the first element. If we impose the condition above, then we have $Tq_0 = xq_0$. We also have the initial recurrence relation, given above. If we isolate the xq_0 term in the recurrence, we obtain

$$Tq_0 = xq_0 = \frac{1}{\alpha_0}q_1 - \frac{\beta_0}{\alpha_0}q_0$$

Since this equation neatly maps linear combinations of the components of v, we can define the first row of T as such. For the general recurrence, note that we can shuffle it around via the following:

$$q_{n+1} = \alpha_n x q_n + \beta_n q_n + \gamma_n q_{n-1}$$

$$\alpha_n x q_n = -\gamma_n q_{n-1} - \beta_n q_n + q_{n+1}$$

$$xq_n = -\frac{\gamma_n}{\alpha_n}q_{n-1} - \frac{\beta_n}{\alpha_n}q_n + \frac{1}{\alpha_n}q_{n+1}$$

Then for intermediate n, we can write our mapping T as

$$Tq_n = -\frac{\gamma_n}{\alpha_n}q_{n-1} - \frac{\beta_n}{\alpha_n}q_n + \frac{1}{\alpha_n}q_{n+1}$$

This is again a linear combination of the components of v, so we can define the intermediate rows of T as such. Note that we cannot do this for the last coordinate of v, since there is clearly no final q_{n+1}. Getting the last row, we can examine the recurrence formula for q_n.

$$q_n = \alpha_{n-1}xq_{n-1} + \beta_{n-1}q_{n-1} + \gamma_{n-1}q_{n-2}$$

$$q_n = \alpha_{n-1}xq_{n-1} + \beta_{n-1}q_{n-1} + \gamma_{n-1}q_{n-2}$$

$$-\gamma_{n-1}q_{n-2} - \beta_n q_{n-1} = -q_n + \alpha_{n-1}xq_{n-1}$$

$$-\frac{\gamma_{n-1}}{\alpha_{n-1}}q_{n-2} - \frac{\beta_{n-1}}{\alpha_{n-1}}q_{n-1} = xq_{n-1} - q_n/\alpha_{n-1}$$

Therefore, solely by the recurrence formula, we have found a linear combination of the components of v to form the desired output. The matrix T is therefore

$$T = \begin{bmatrix} \frac{\beta_0}{\alpha_0} & \frac{1}{\alpha_0} & 0 & ... & ... & 0 \\ \frac{-\gamma_1}{\alpha_1} & -\frac{\beta_1}{\alpha_1} & \frac{1}{\alpha_1} & 0 & ... & 0 \\ \dots & & & & & \\ 0 & ... & 0 & -\frac{\gamma_{n-1}}{\alpha_{n-1}} & -\frac{\beta_{n-1}}{\alpha_{n-1}} & \frac{1}{\alpha_{n-1}} \\ 0 & ... & 0 & 0 & -\frac{\gamma_{n-1}}{\alpha_{n-1}} & -\frac{\beta_{n-1}}{\alpha_{n-1}} \end{bmatrix}.$$

Exercise 23, part 2

Show that x^* is a root of p if and only if $q_n(x^*) = -\frac{1}{c_n}\sum_{j=0}^{n-1} c_j q_j(x^*)$.

Solution:

I will start with the backward implication, as that is almost immediate.

$$q_n(x^*) = -\frac{1}{c_n}\sum_{j=0}^{n-1} c_j q_j(x^*) \iff c_n q_n(x^*) + \sum_{j=0}^{n-1} c_j q_j(x^*) = 0.$$

Since $p(x^*)$ is a linear combination of the q_j's, then $p(x^*) = 0$. Therefore, x^* is a root of p. This shows the backward implication.

Next, suppose x^* is a root of p. Then we have that

$$\sum_{j=0}^{n} c_j q_j(x^*) = 0.$$

This implies that there is at least one linearly dependent q_j's. It is not immediate from here that q_n is the linearly dependent vector, but we can show this is the case. Note that each q_j is evaluated at x^*, which fixes its value. I will suppress each function being evaluated at x^* for the sake of simplicity. First, we will show that q_0 cannot be the linearly dependent component. If it was, then we have the two facts below:

$$q_0 = -\frac{1}{c_0}\sum_{j=1}^{n} c_j q_j, \quad q_1 = (\alpha_0 x + \beta_0) q_0.$$

Substituting the first into the second, we have that

$$q_1 = (\alpha_0 x + \beta_0)\left(-\frac{1}{c_0}\sum_{j=1}^{n} c_j q_j\right) \implies q_1 = -\frac{(\alpha_0 x + \beta_0)c_1}{c_0} q_1 - \frac{(\alpha_0 x + \beta_0)c_2}{c_0} q_2 + \ldots$$

Note the only two q_j's which depend on q_0 is q_1 and q_2. By the recurrence relation, we cannot possibly have that q_1 depends on any other q_j, thus all c_j's where $j \neq 0, 1, 2$ are zero, which is a contradiction. Therefore, $q_0(x^*)$ cannot be linearly dependent.

We will next show that q_i for $i \neq n$ cannot be linearly dependent. Again, suppose false, so that the two equations below are true:

$$q_i = -\frac{1}{c_i}\sum_{j\neq i} c_j q_j, \quad q_{i+1} = (\alpha_i x + \beta_i) q_i + \gamma_i q_{i-1}$$

We can again substitute in q_i into the recurrence relation to get that all c_j's for $j > i + 1$ are zero. This is then a contradiction, since we suppose the all c_j's are nonzero. Therefore, $q_i(x^*)$ cannot possibly be linearly dependent for $i < n$. Since at least one q has to be linearly dependent, this leaves q_n to be linearly dependent, which is what we wanted to show.

Exercise 23, part 3

If M is the matrix you found in part (a), consider the matrix C defined by $C = M + L$, where

$$L = -\frac{1}{\alpha_{n-1}c_n}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \dots & & & \\ c_0 & c_1 & \dots & c_{n-1} \end{bmatrix}.$$

Show that if p has distinct roots, then λ is a eigenvalue of C if and only if $p(\lambda) = 0$. Thus, diagonalizing C is the same as finding the roots of p.

Solution:

I will first show the forward implication. Suppose that v is the corresponding eigenfunction to eigenvalue λ, where v is of the same form as in part 1. Then $\lambda v = Cv = Mv + Lv$. We have the form of Mv, which is the output of part 1. The form of Lv will only be nonzero in the last coordinate, which is

$$Lv = -\frac{1}{\alpha_{n-1}c_n}\begin{bmatrix} 0 & 0 & \dots & c_0q_0 + c_1q_1 + \dots + c_{n-1}q_{n-1} \end{bmatrix}^\intercal.$$

This then gives us

$$Cv = \begin{bmatrix} xq_0 & xq_1 & \dots & xq_{n-2} & xq_{n-1} - \frac{1}{\alpha_{n-1}}q_n - \frac{1}{\alpha_{n-1}c_n}(c_0q_0 + c_1q_1 + \dots + c_{n-1}q_{n-1}) \end{bmatrix} = \lambda v.$$

The fist $n-1$ equations gives us that $(x-\lambda)q_i(x) = 0$ which is true only when $q_i(x) = 0$ for all $i < n$, or that $x = \lambda$. The latter is almost certainly true. The last coordinate needs to be massaged a bit, which will be done below.

$$\begin{aligned} & xq_{n-1} - \frac{1}{\alpha_{n-1}}q_n - \frac{1}{\alpha_{n-1}c_n}(c_0q_0 + c_1q_1 + \dots + c_{n-1}q_{n-1}) \\ &= xq_{n-1} - \frac{1}{\alpha_{n-1}c_n}(c_nq_n) - \frac{1}{\alpha_{n-1}c_n}(c_0q_0 + c_1q_1 + \dots + c_{n-1}q_{n-1}) \\ &= xq_{n-1} - \frac{1}{\alpha_{n-1}c_n}p(x) \end{aligned}$$

By the equation $Cv = \lambda v$, we have that

$$xq_{n-1} - \frac{1}{\alpha_{n-1}c_n}p(x) = xq_{n-1}$$

From the first $n-1$ equations, $x = \lambda$. Furthermore, the last coordinate gives us the equation above, which by simplification gives us $p(\lambda) = 0$. Therefore, the forward implication has been shown.

For the backward implication, suppose that $p(\lambda) = 0$. This implies that λ is a root of the function p, then by part 2, $q_n(\lambda) = -\frac{1}{c_n}\sum_{j=0}^{n-1} c_j q_j(\lambda)$. Define the vector v as in part 1. Then

$$\begin{aligned} Mv(\lambda) &= \Big[\lambda q_0(\lambda), \lambda q_1(\lambda), ..., \lambda q_{n-2}(\lambda), \lambda q_{n-1} - q_n/\alpha_{n-1}\Big]^T \\ &= \Big[\lambda q_0(\lambda), \lambda q_1(\lambda), ..., \lambda q_{n-2}(\lambda), \lambda q_{n-1}(\lambda) + \tfrac{1}{\alpha_{n-1}c_n}\sum_{j=0}^{n-1} c_j q_j(\lambda)\Big]^T \\ &= \lambda v - (Lv)(\lambda) \\ \implies (M+L)(v) &= \lambda v \end{aligned}$$

Therefore, $Cv(\lambda) = \lambda v$, so λ is an eigenvalue of the matrix C.

Exercise 23, part 4

Specialize your construction to the case where:

a) the q_n are monomials,

b) the q_n are Chebyshev polynomials

Use the former to compute the roots of $p = x^2 - 5x + 2$, and the latter to compute the roots of $p = T_0 + 2T_1 + 4T_2$.

Solution:

For the monomial case, we take $q_0 = 1, q_1 = x, q_2 = x^2$ and $c_0 = 2, c_1 = -5, c_2 = 1$. From the recursion relation, we find $\alpha_0 = 1, \beta_0 = 0, \alpha_1 = 1, \beta_1 = 0$. The matrices C, M and L are then

$$M + L = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ -2 & 5 \end{bmatrix} \implies M = \begin{bmatrix} 0 & 1 \\ -2 & 5 \end{bmatrix}.$$

The sanest way to find the roots of the polynomial is by the quadratic formula, which will give us that $\lambda_{1,2} = \frac{5 \pm \sqrt{17}}{2}$. This is the same as the eigenvalues of the matrix above. Note that by the construction of monomials, $\alpha_i = 1, \beta_i = 0, \gamma_i = 0$ for $i = 0, 1, ..., n$.

To specialize this to the Chebyshev polynomials, we note their recurrence relation,

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

This implies that for any generic n, $\alpha_n = 2, \beta_n = 0, \gamma_n = -1$. This does not work for $n = 0$ obviously, since there is no T_{-1}. Analyzing this independently, we get $\alpha_0 = 1, \beta_0 = 0$. Then, we have the following matrix

$$C = \begin{bmatrix} 0 & 1 \\ \frac{1}{2} & 0 \end{bmatrix} - \frac{1}{8}\begin{bmatrix} 0 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{3}{8} & -\frac{1}{4} \end{bmatrix}.$$

The matrix above gives us the eigenvalues $-3/4, 1/2$. Expanding the Chebyshev polynomials into the basis of monomials, we have that

$$p(x) = T_0 + 2T_1 + 4T_2 = 8x^2 + 2x - 3$$

The roots of this polynomial are $x_{1,2} = -3/4, 1/2$. These are the same values as the matrix above. Therefore, the specialization works.

Exercise 25

In this exercise we will explore piece-wise Gauss-Legendre approximations both analytically and numerically. In the following, let $x_0, ..., x_n$ and $w_0, ..., w_n$ be the standard $n+1$ point Gauss-Legendre quadrature. Given an interval $[\alpha, \beta]$ let $x_i^{\alpha,\beta}$ and $w_i^{\alpha,\beta}$ be the nodes and weights of the Gauss-Legendre quadrature mapped to the interval $[\alpha, \beta]$, i.e.,

$$x_i^{\alpha,\beta} = (\beta - \alpha)\frac{(x_i + 1)}{2} + \alpha, \quad \text{and} \quad w_i^{\alpha,\beta} = \frac{\beta - \alpha}{2} w_i.$$

Exercise 25, part 1

Consider the function $f(z; z_0, p) = 1/(z - z_0)^p$. Suppose that for any $p > 0$, the L^2 error of approximating $f(z; -3, p)$ on $[-1, 1]$ by $P_0, ..., P_n$ (the Gauss-Legendre polynomials) is denoted by $q_{p,n}$. Find an expression for the error of approximating $f(\cdot; 2\alpha - \beta, p)$ on $[\alpha, \beta]$ using the appropriately shifted and scaled Gauss-Legendre polynomials.

Solution:

As given, we have the following definition for $q_{p,n}$, for some coefficients $c_0, ..., c_n$:

$$q_{p,n} = \int_{-1}^{1} \left| f(z; -3, p) - \sum_{k=0}^{n} c_k P_k(z) \right|^2 dz.$$

Just explicitly writing out the L^2 error in approximating $f(x; 2\alpha - \beta, p)$ using shifted Gauss-Legendre polynomials, we have

$$\int_{\alpha}^{\beta} \left| f(x; 2\alpha - \beta, p) - \sum_{k=0}^{n} c_k P_k(x) \right|^2 dx.$$

Here, I am writing $x \in [\alpha, \beta]$, $z \in [-1, 1]$. Using Gauss-Legendre approximation to the integral above, and supposing there are $n+1$ nodes, we have

$$\int_{\alpha}^{\beta} \left| f(x; 2\alpha - \beta, p) - \sum_{k=0}^{n} c_k P_k(x) \right|^2 dx. \approx \sum_{k=0}^{n} \left| f(x_k^{\alpha,\beta}; 2\alpha - \beta, p) - \sum_{j=0}^{n} c_j P_j(x_k^{\alpha,\beta}) \right|^2 w_k^{\alpha,\beta}$$

Using the given transformation of the nodes to $[\alpha, \beta]$ from $[-1, 1]$, we can write the following:

$$f(x_k^{\alpha,\beta}; 2\alpha - \beta, p) = \left(x_k^{\alpha,\beta} - 2\alpha + \beta\right)^{-p} = \left(\left(\frac{\beta - \alpha}{2}\right)(x_k + 1) + \alpha - 2\alpha + \beta\right)^{-p}$$

$$= \left(\left(\frac{\beta - \alpha}{2}\right)(x_k + 1) - \alpha + \beta\right)^{-p} = \left(\left(\frac{\beta - \alpha}{2}\right)(x_k + 3)\right)^{-p}$$

$$= \left(\frac{2}{\beta - \alpha}\right)^{p} f(z; -3, p)$$

Furthermore, the weights are shifted by

$$w_k^{\alpha,\beta} = \left(\frac{2}{\beta-\alpha}\right)^{-1} w_k$$

Therefore, the summation above can be rewritten to

$$\sum_{k=0}^{n} \left| \left(\frac{2}{\beta-\alpha}\right)^{p} f(z;-3,p) - \sum_{j=0}^{n} c_j P_j(z(x_k^{\alpha,\beta})) \right|^2 \left(\frac{2}{\alpha-\beta}\right)^{-1} w_k$$

If we allow the coefficients c_j to "absorb" the constants tacked to $f(z;-3,p)$, we can pull out that constant, and collect terms together to get

$$\left(\frac{2}{\beta-\alpha}\right)^{2p-1} \sum_{k=0}^{n} \left| f(z;-3,p) - \sum_{j=0}^{n} c_j P_j(z(x_k)) \right|^2 w_k$$

If we look at the Gauss-Legendre approximation to the L^2 error of $f(z;-3,p)$, we see the summation is the same. Therefore, the error E_L in approximating $f(x;2\alpha-\beta,p)$ on the interval $[\alpha,\beta]$ using Legendre polynomials, we have that

$$E_L(f(x;2\alpha-\beta,p)) = \left(\frac{2}{\beta-\alpha}\right)^{2p-1} E_L(f(z;-3,p)) = q_{p,n}\left(\frac{2}{\beta-\alpha}\right)^{2p-1}.$$

There are some qualifiers to what I've written, which are believe are justified. Since $\beta-\alpha \geq 0$, we can just drop the absolute value on the ratio that comes from inside the absolute value. Also, the polynomials $P_j(z(x_k))$ which have been shifted to the interval $[-1,1]$ are assumed to be the same as just the Legendre polynomials at the evaluation points themselves.

Exercise 25, part 2

Consider the discretization which seeks to represent $f(z;0,p)$ on $[0,1]$ by the distinct shifted and scaled n-th order Gauss-Legendre expansion on the intervals $[1/2,1],[1/4,1/2],...,[1/2^j,1/2^{j+1}]$ and 0 on $[0,1/2^j]$. That is to say that in each interval, the function approximated by a different n-th order polynomial. What is the L^2 error in this approximation in terms of p, $q_{p,n}$, and j. For what values (if any) does it converge?

Solution:

For each subinterval, we are approximating the function by different combinations of Legendre polynomials, so their errors should be independent. Therefore, we can simply sum over each subinterval. Note that on the interval $[1/2^{k+1},1/2^k]$, the error goes as

$$E_L = q_{p,n}\left(\frac{2}{1/2^k - 1/2^{k+1}}\right)^{2p-1} = q_{p,n}\left(\frac{1}{2^{1-2p}}\right)^k$$

Therefore, over the j intervals, we have the total error to be

$$E_L = q_{p,n}\sum_{k=0}^{j}\left(\frac{1}{2^{1-2p}}\right)^k$$

This is then a geometric sum. We also need to take care of the interval $[0,1/2^j]$, which we are approximating with zero. If we allow an ε buffer from the singularity $z=0$, we can add the final error as

$$\int_{\varepsilon}^{1/2^j}\frac{1}{z}\,dz = \ln\left(1/2^j\right) - \ln(\varepsilon) = \ln\left(\frac{1}{2^j\varepsilon}\right)$$

Note that $\varepsilon < 1/2^j$, so that $1/(2^j\varepsilon) > 1$. The final error is

$$E_L = \ln\left(\frac{1}{2^j\varepsilon}\right) + q_{p,n}\sum_{k=0}^{j}\left(\frac{1}{2^{1-2p}}\right)^k.$$

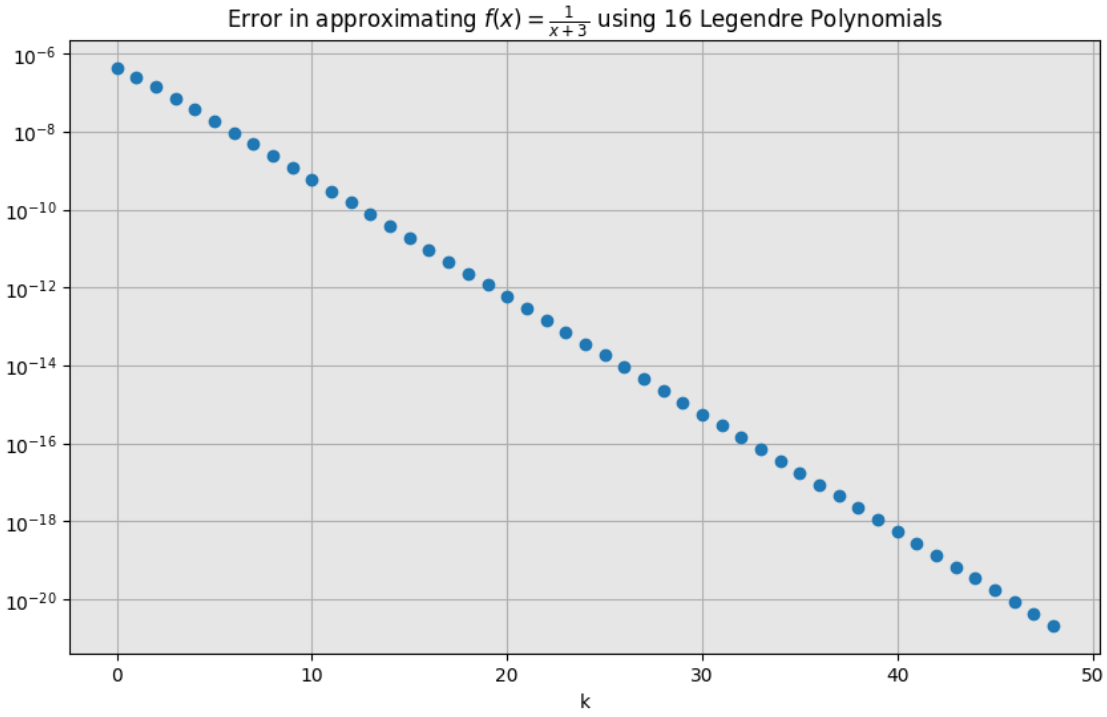
Provided we allow $j\to\infty$, for some ε which approaches zero faster than 2^{-j}, the summation will converge provided the exponent of our ratio is positive - that is, $1-2p>0$. Therefore, the sum will converge for $p\in(0,1/2)$.

Exercise 25, part 3

Numerically determine $q_{1,16}$ and confirm your scalings in parts 1 and 2.

Solution:

If I am reading this correctly, my results are below. What the plot describes is the L^2 error in the difference between the best approximation using Legendre polynomials and the given function. Since convergence is exponential (I am plotting on a log axis), it seems that the convergence is faster that 2^j. My code will be given after my results.



In [ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import sympy as sm
import scipy as scp
```

In [ ]:
```
def gauleg(x1, x2, x, w, n):
    EPS = 3.0e-11
    m = (n + 1) // 2  # Find only half the roots because of symmetry
    xm = 0.5 * (x2 + x1)
    xl = 0.5 * (x2 - x1)
    for i in range(1, m + 1):
        z = math.cos(math.pi * (i - 0.25) / (n + 0.5))
        while True:
            p1 = 1.0
            p2 = 0.0
            for j in range(1, n + 1):
                # Recurrence relation
                p3 = p2
                p2 = p1
                p1 = ((2.0 * j - 1.0) * z * p2 - (j - 1.0) * p3) / j
            # Derivative
            pp = n * (z * p1 - p2) / (z * z - 1.0)
            z1 = z
            # Newton's method
            z = z1 - p1 / pp
            if abs(z - z1) <= EPS:
                break
        x[i] = xm - xl * z
        x[n + 1 - i] = xm + xl * z
        # Weights
        w[i] = 2.0 * xl / ((1.0 - z * z) * pp * pp)
        w[n + 1 - i] = w[i]
```

In [ ]:
```
def Gauss_Legendre_Quad(fs, weights: np.ndarray, zeros: np.ndarray, alpha: float = -1.0, beta: float = 1.0) :
    if beta <= alpha:
        print("Alpha is greater than (or equal to) beta: alpha: ", alpha, " beta: ", beta)
        return
    sum = 0
    scaled_zeros = np.zeros_like(zeros)
    for i in range(len(zeros)):
        scaled_zeros[i] = 0.5*(beta - alpha) * zeros[i] + 0.5*(beta + alpha)
    n = len(weights)
    x = sm.symbols('x')
    for i in range(n):
        sum += 0.5*(beta - alpha) * weights[i] * fs.subs(x, scaled_zeros[i])
    return sum
```

In [ ]:
```
# For Legendre polynomials
def calculate_coeffs(func, leg_funcs):
    x = sm.symbols('x')
    coeffs = np.zeros(len(leg_funcs))
    for l in range(len(coeffs)):
        inte = sm.integrate(func*leg_funcs[l], (x, -1, 1))
        coeffs[l] = 0.5*(2*l + 1) * inte
    return coeffs

def calculate_leg_funcs(n: int):
    x = sm.symbols('x')
    leg_funcs = []
    for deg in range(n):
        leg = 0
        for k in range(n):
            leg += scp.special.binom(deg, k)**2.0 *(x - 1)**(deg - k) * (x+1)**k
        leg *= 2**(-deg)
        leg_funcs.append(leg)

    return leg_funcs

def calculate_L2_norm(func, coeffs, leg_funcs):
    approx = 0
    xspan = np.linspace(-1, 1, 1000)
    x = sm.symbols('x')
    for k in range(len(coeffs)):
        if not math.isnan(coeffs[k]):
            approx += coeffs[k] * leg_funcs[k]
    integrand = sm.lambdify(x, abs(approx - func)**2)
    inte = scp.integrate.trapz(integrand(xspan), xspan)

    inte = inte**(0.5)
    return inte
```

In [ ]:
```
x = sm.symbols('x')
func = 1.0/(x+3)
# Input the number of quadrature points
ns = [16]
j = 50
errors = np.zeros(j)

leg_funcs = calculate_leg_funcs(16)
coeffs = calculate_coeffs(func, leg_funcs)
inte = calculate_L2_norm(func, coeffs, leg_funcs)

for n, N in enumerate(ns):
    xs = [0.0] * (N + 1)
    ws = [0.0] * (N + 1)

    # Call the gauleg function
```

```
    gauleg(-1.0, 1.0, xs, ws, N)

    for k in range(len(errors)):
        errors[k] = inte*Gauss_Legendre_Quad(func, ws, xs, 0.5**(k+1), 0.5**(k))
    for j in range(len(errors)-1):
        errors[j] = errors[j] - errors[j+1]
```

In [ ]:

```
plt.figure(figsize=(10, 6))

plt.scatter(range(len(errors)-1), errors[:-1])
plt.yscale('log')
plt.gca().set_facecolor((0.9, 0.9, 0.9))
plt.grid(True)
plt.gca().set_axisbelow(True)
plt.xlabel("k")
plt.title(r"Error in approximating $f(x) = \frac{1}{x+3}$ using 16 Legendre Polynomials")
plt.show()
```

Exercise 26

Let $x_0 < ... < x_N$ be equispaced points on $[-1, 1]$ with $x_0 = -1$ and $x_N = 1$. Suppose we are given data $\{(x_j, y_j) : 0 \leq j \leq N\}$, where $y_j = x_j^3$.

Exercise 26, part 1

What is the natural interpolating cubic spline, s_N, for the data when $N = 3$?

Solution:

I will follow the notation used on the Wikipedia page. From this, we seek to find, for each interval $[x_i, x_{i+1}]$, the polynomial

$$q_i(x) = (1 - t_i)y_{i-1} + t_i y_i + t_i(1 - t_i)((1 - t_i)a_i + t_i b_i)$$

where we define the following terms as

$$a_i = k_{i-1}(x_i - x_{i-1}) - (y_i - y_{i-1}), \quad b_i = -k_i(x_i - x_{i-1}) + (y_i - y_{i-1}), \quad t_i = \frac{x - x_{i-1}}{x_i - x_{i-1}}.$$

For our given data points, and given N, our data points are $\{(-1, -1), (0, 0), (1, 1)\}$. The k_i terms are given by solving the system

$$\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Note that this matrix is symmetric, and the terms in the matrix are different than the ones defined outside the matrix. Since there are quite a few terms to explicitly write out, please look at the Wikipedia page on spline interpolation for explicit terms. I will write out the concrete system we will use to find the k_i's.

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 3 \end{bmatrix} \implies \begin{bmatrix} k_0 \\ k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

The a_i's and b_i's outside the system can then be found

$$a_1 = (x_1 - x_0) - (y_1 - y_0) = 0, \quad b_1 = -(x_1 - x_0) + (y_1 - y_0) = 0, \quad a_2 = 0 \quad b_2 = 0$$

Since the a_i's and b_i's are zero, we can safely ignore the third term. Note that by the data points given $t_1 = 1 + x, t_2 = x$. The interpolating polynomials are then

$$q_1 = (-x)(-1) + 0 = x, \quad q_2 = (1 - x) \cdot 0 + x = x.$$

The polynomial is then just $q(x) = x$. This is a little surprising, but with the data points given the unknown

function is indistinguishable from the function $q(x) = x$.

Exercise 26, part 2

Write a code for computing s_N for arbitrary N. Plot s_N for $N = 3, 5, 7$.

Solution:

I have implemented code for the natural cubic spline. My results, as well as the code which I wrote, will be provided after the next part. You can see my results from the first part are confirmed.

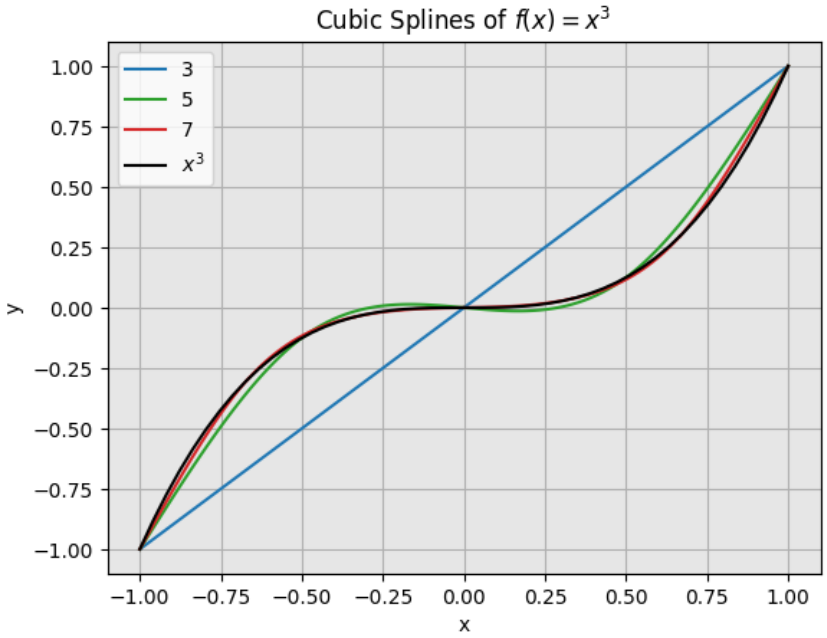


Figure 4: Natural Cubic splines for the function $f(x) = x^3$ for given N.

Exercise 26, part 3

Compare s_N with the function $f(x) = x^3$. For which $x \in [-1, 1]$ is $|s_N - f(x)|$ maximized? why?

Solution:

The plot for the error of each cubic spline is given below. I have highlighted the maximum points for each spline. It is clear that for each maximum, there is an equal maximum mirrored across the x-axis. Furthermore, it seems as though the minimum distance between a max and its closest node is mirrored.

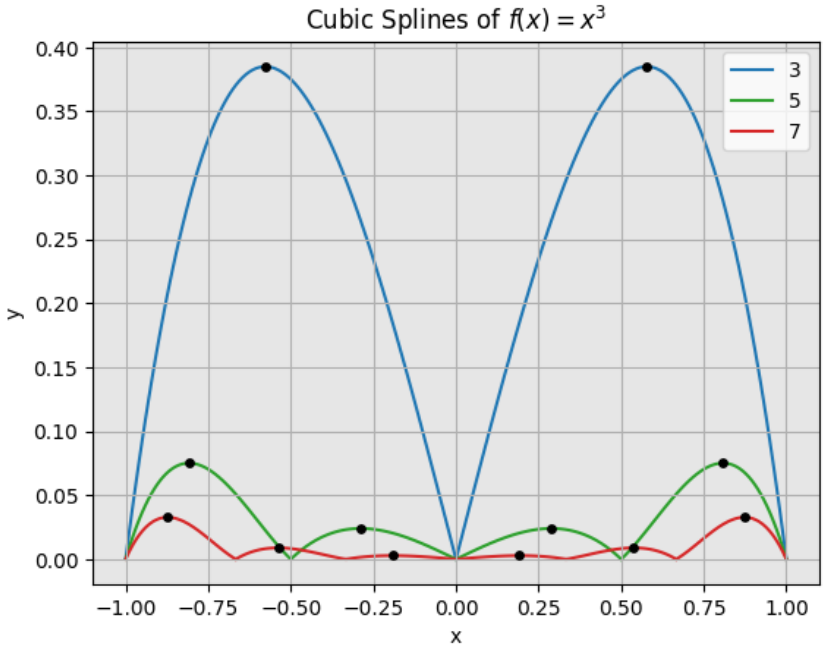


Figure 5: Error for each cubic spline for given orders. Maxes have been highlighted.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
```

In [ ]:
```
def make_system(xs: np.ndarray, ys: np.ndarray) :
    # Constructs the matrix A and the vector b for spline interpolation
    # Might not work
    N = len(xs)
    A = np.zeros((N, N))
    b = np.zeros(N)
    # Make A
    for i in range(N):
        if i == 0:
            temp = 1.0 / (xs[i+1] - xs[i])
            A[i][i] = 2.0 * temp
            A[i][i+1] = temp
        elif i == N-1:
            temp = 1.0 / (xs[i] - xs[i-1])
            A[i][i] = 2.0 * temp
            A[i][i-1] = temp
        else:
            A[i][i-1] = 1.0 / (xs[i] - xs[i-1])
            A[i][i+1] = 1.0 / (xs[i+1] - xs[i])
            A[i][i] = 2.0* (A[i][i-1] + A[i][i+1])

    # Make b
    for i in range(N):
        if i == 0:
            temp = (ys[i+1] - ys[i]) / (xs[i+1] - xs[i])**2
            b[i] = 3 * temp
        elif i == N-1:
            temp = (ys[i] - ys[i-1]) / (xs[i] - xs[i-1])**2
            b[i] = 3 * temp
        else:
            templ = (ys[i] - ys[i-1]) / (xs[i] - xs[i-1])**2
            tempr = (ys[i+1] - ys[i]) / (xs[i+1] - xs[i])**2
            b[i] = 3* (templ + tempr)
    return A, b

def make_terms(xs: np.ndarray, ys: np.ndarray, ks: np.ndarray):
    N = len(xs)
    ays = np.zeros(N-1)
    bs = np.zeros(N-1)
    for i in range(1, N):
        ays[i-1] = ks[i-1] * (xs[i] - xs[i-1]) - (ys[i] - ys[i-1])
        bs[i-1] = -ks[i] * (xs[i] - xs[i-1]) + (ys[i] - ys[i-1])
    return ays, bs

def make_spline(x_left: float, x_right: float, y_left: float, y_right: float, k: float, a: float, b: float):
    # Makes the spline bewteen the two endpoints
    # 100 is a placeholder
    t = np.linspace(0, 1, 100)  # t should range from 0 to 1

    # Construct the spline between the two endpoints
    q_i = (1.0 - t) * y_left + t * y_right + t * (1.0 - t) * ((1.0 - t) * a + t * b)

    return q_i

def cubic_interpolation(xs: np.ndarray, ys: np.ndarray):
    x_interp = np.zeros((len(xs)-1, 100))
    y_interp = np.zeros((len(ys)-1, 100))
    # Make system
    A, b = make_system(xs, ys)
    # solve system
    ks = sp.linalg.solve(A, b, assume_a='sym')
    ays, bs = make_terms(xs, ys, ks)
    for i in range(len(xs)-1):
        x_interp[i, :] = np.linspace(xs[i], xs[i+1], 100)
        y_interp[i, :] = make_spline(xs[i], xs[i+1], ys[i], ys[i+1], ks[i], ays[i], bs[i])
    return x_interp, y_interp
```

In [ ]:
```
colors = ['tab:blue', 'tab:green', 'tab:red', 'k']
ns = [3, 5, 7]

# Lists to store labels and handles for legend
labels = []
handles = []

for i, n in enumerate(ns):
    xs = np.linspace(-1, 1, n)
    ys = xs**3
    x_cubic, y_cubic = cubic_interpolation(xs, ys)

    for j in range(len(x_cubic)):
        plt.plot(x_cubic[j], y_cubic[j], color=colors[i])  # No need for labels here

    # Collect labels and handles for legend
    labels.append(f"{n}")
    handles.append(plt.Line2D([], [], color=colors[i]))  # Creating handles manually

xs = np.linspace(-1, 1)
ys = xs**3

# Plot the original function (outside the loop)
plt.plot(xs, ys, label='Original Function', color = colors[-1])
labels.append(r"$x^3$")
handles.append(plt.Line2D([], [], color=colors[-1]))
```

```
# Add legend using collected labels and handles
plt.legend(handles, labels)
plt.grid(True)
plt.gca().set_facecolor((0.9, 0.9, 0.9))
plt.xlabel("x")
plt.ylabel("y")
plt.title(r"Cubic Splines of $f(x) = x^3$")

# Show the plot
plt.show()
```

In [ ]:

```
colors = ['tab:blue', 'tab:green', 'tab:red', 'k']
ns = [3, 5, 7]
indices = {}
# Lists to store labels and handles for legend
labels = []
handles = []


for i, n in enumerate(ns):
    xs = np.linspace(-1, 1, n)
    ys = xs**3

    x_cubic, y_cubic = cubic_interpolation(xs, ys)
    y_true = np.zeros_like(x_cubic)
    y_err = np.zeros_like(x_cubic)
    for k in range(len(x_cubic)):
        y_true[k, :] = x_cubic[k]**3
        y_err[k, :] = np.abs(y_cubic[k] - y_true[k])


    y_err_line = y_err.reshape(-1)
    peak_indices, _ = sp.signal.find_peaks(y_err_line)

    index_i = []
    for j in range(len(x_cubic)):
        index_i.append(x_cubic[j][peak_indices[j]%100])
        plt.plot(x_cubic[j], y_err[j], color=colors[i], zorder=0)
        plt.scatter(x_cubic[j][peak_indices[j] % 100], y_err_line[peak_indices[j]], c='k', zorder=1 , s = 15)
    indices[ns[i]] = index_i

    # Collect labels and handles for legend
    labels.append(f"{n}")
    handles.append(plt.Line2D([], [], color=colors[i]))  # Creating handles manually



# Add legend using collected labels and handles
plt.legend(handles, labels)
plt.grid(True)
plt.gca().set_facecolor((0.9, 0.9, 0.9))
plt.xlabel("x")
plt.ylabel("y")
plt.title(r"Cubic Splines of $f(x) = x^3$")
plt.gca().set_axisbelow(True)
# Show the plot
plt.show()
```

Exercise 26, part 4

(Open ended) Can you come up with an extension of 'natural' splines to higher-order? Do they satisfy an orthogonality condition like natural cubic splines? Can you write them in a 'radial basis function' type format?

Solution:

I am not entirely sure what would be the best extension to natural splines. My first guess would be to use higher order polynomials (say quintic polynomials) and to transform the boundary conditions to have the fourth derivatives to be zero. I am not sure how I would prove this.