APPM 4600 Lab 10

31 October 2024

The code for this lab can be seen at the end of this document, or on github here.

1 Prelab

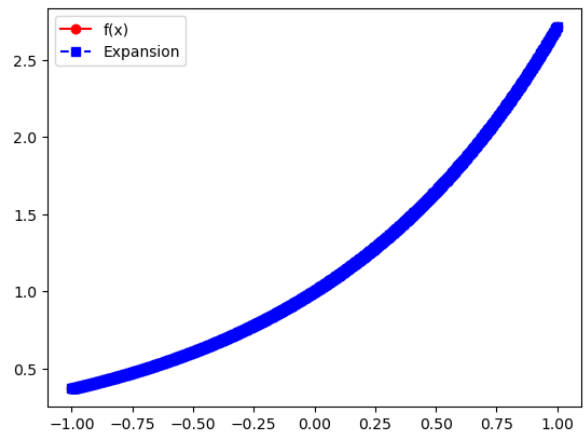
1. The code to evaluate the legendre polynomials is included at the end of the document, in the function `eval_legendre`.

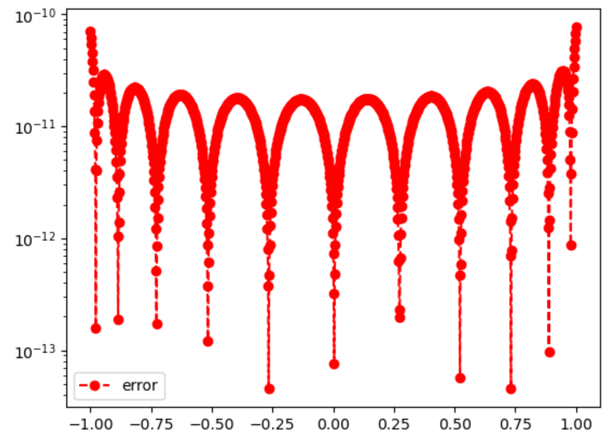
2 Legendre Approximation

The code to evaluate the Legendre approximations is included at the end of this document. We first approximate the function

$$f(x) = e^x,$$

with $n = 10$ with results shown in the figures below.

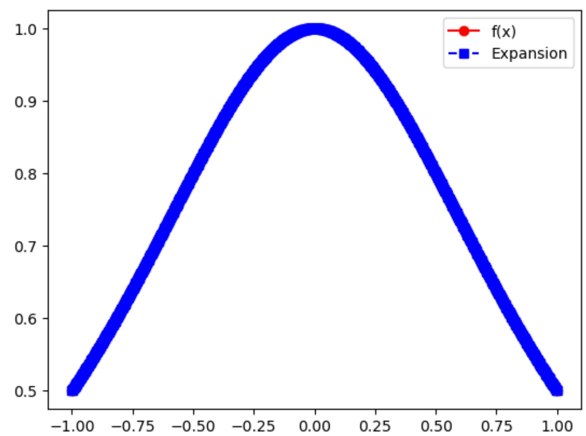


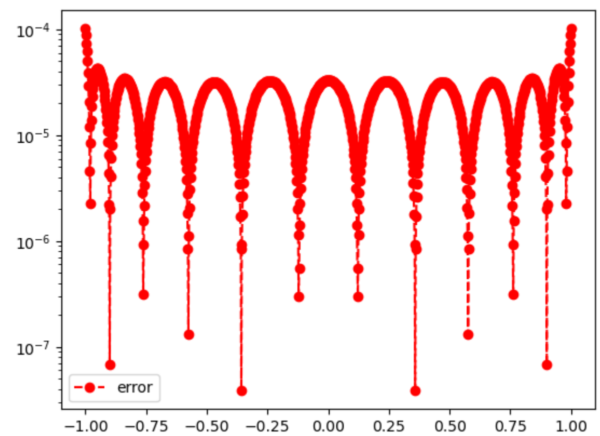


We now approximate the function

$$f(x) = \frac{1}{1+x^2},$$

with results for $n = 10$ shown below.





```
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as la
import math
from scipy.integrate import quad

def eval_legendre(n, x):
    polys = [np.ones(np.array(x).size), x]
    for i in range(1, n):
        p_next = 1/(i+1) * ((2*i+1)*x*polys[i] - i*polys[i-1])
        polys.append(p_next)

    return polys

def driver():

#  function you want to approximate
    f = lambda x: 1 / (1 + x**2)

# Interval of interest
    a = -1
    b = 1
# weight function
    w = lambda x: 1.

# order of approximation
    n = 3

#  Number of points you want to sample in [a,b]
    N = 1000
    xeval = np.linspace(a,b,N+1)
    pval = np.zeros(N+1)

    for kk in range(N+1):
      pval[kk] = eval_legendre_expansion(f,a,b,w,n,xeval[kk])

    ''' create vector with exact values'''
    fex = np.zeros(N+1)
    for kk in range(N+1):
        fex[kk] = f(xeval[kk])

    plt.figure()
    plt.plot(xeval,fex,'ro-', label= 'f(x)')
    plt.plot(xeval,pval,'bs--',label= 'Expansion')
    plt.legend()
    plt.show()

    err = abs(pval-fex)
    plt.semilogy(xeval,err,'ro--',label='error')
    plt.legend()
    plt.show()



def eval_legendre_expansion(f,a,b,w,n,x):

#   This subroutine evaluates the Legendre expansion
```

```
#  Evaluate all the Legendre polynomials at x that are needed
# by calling your code from prelab
  p = lambda x: eval_legendre(n, x)
  # initialize the sum to 0
  pval = 0.0
  for j in range(0,n+1):
      # make a function handle for evaluating phi_j(x)
      phi_j = lambda x: p(x)[j]
      # make a function handle for evaluating phi_j^2(x)*w(x)
      phi_j_sq = lambda x: phi_j(x)**2 * w(x)
      # use the quad function from scipy to evaluate normalizations
      norm_fac, err = quad(phi_j_sq, a, b)
      # make a function handle for phi_j(x)*f(x)*w(x)/norm_fac
      func_j = lambda x: phi_j(x) * f(x) * w(x) / norm_fac
      # use the quad function from scipy to evaluate coeffs
      aj, err = quad(func_j, a, b)
      # accumulate into pval
      pval = pval+aj*p(x)[j]

  return pval

if __name__ == '__main__':
  # run the drivers only if this is called from the command line
  driver()
```