APPM 4600 Lab 12

14 November 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 composite trapezoidal and simpsons is included in the attached code, in the functions `eval_composite_trap` and `eval_composite_simpsons`.

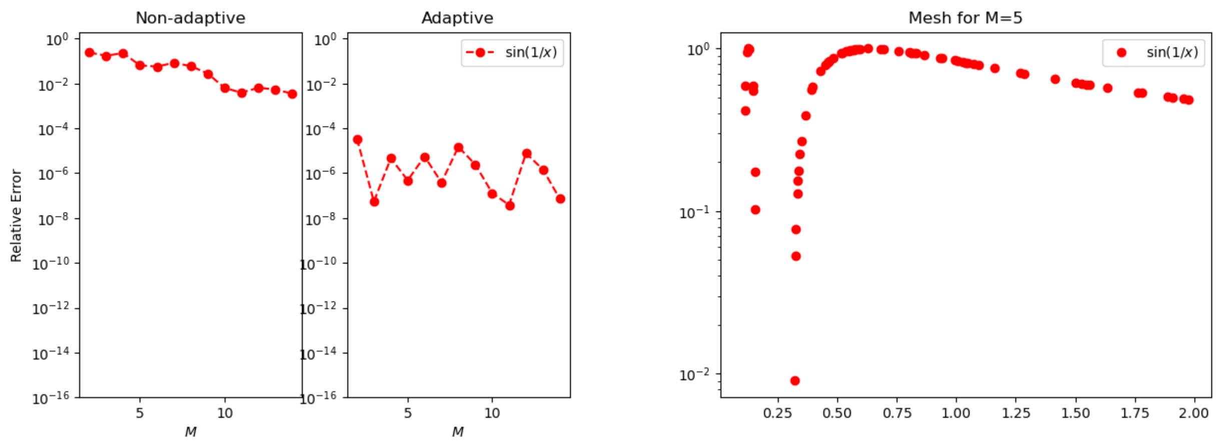
2 Adaptive Quadrature

We use adaptive quadrature to evaluate the integral

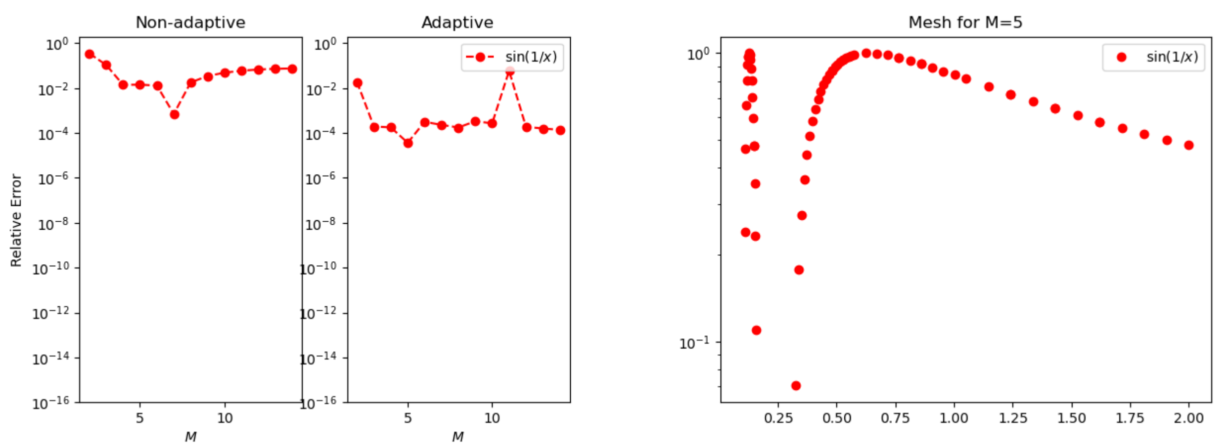
$$I = \int_{0.1}^{2} \sin\left(\frac{1}{x}\right) \mathrm{d}x,$$

with $n = 5$ nodes on each interval.

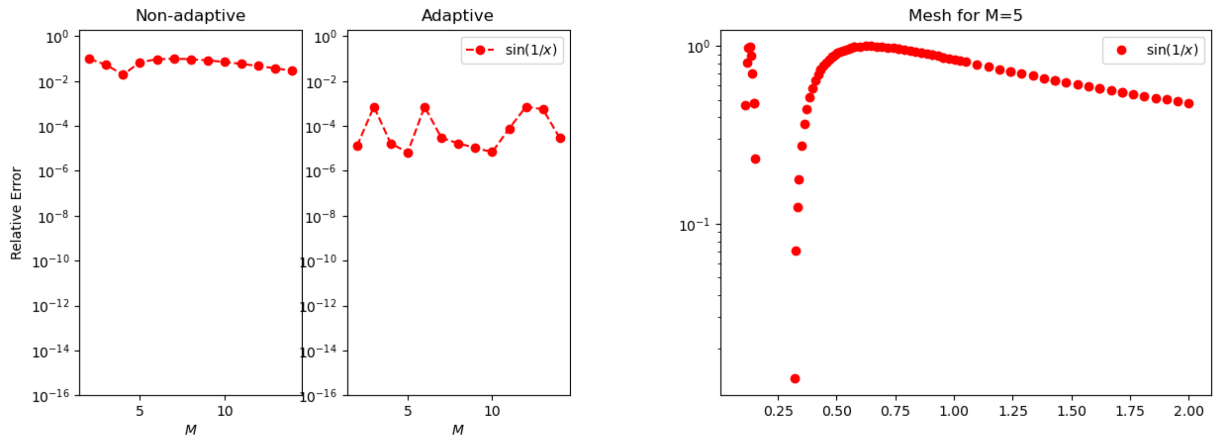
Gauss quadrature requires 6 intervals (5 splits) to get to the desired accuracy. The absolute accuracy versus n and final evaluation points for $n = 5$ are plotted below.



Composite trapezoidal requires 9 intervals (8 splits) to get to the desired accuracy. The absolute accuracy versus n and final evaluation points for $n = 5$ are plotted below.



Composite Simpsons requires 6 intervals (5 splits) to get to the desired accuracy. The absolute accuracy versus n and final evaluation points for $n = 5$ are plotted below.




```
# get lgwts routine and numpy
from gauss_legendre import *

# adaptive quad subroutines
# the following three can be passed
# as the method parameter to the main adaptive_quad() function

def eval_composite_trap(M,a,b,f):
    """
    put code from prelab with same returns as gauss_quad
    you can return None for the weights
    """
    h = (b-a)/M
    x = np.arange(0, M+1)*h + a
    # trapezoidal weights
    w = h*np.ones(M+1)
    w[0] = h*0.5
    w[-1] = h*0.5

    I = np.sum(f(x) * w)

    return I, x, w

def eval_composite_simpsons(M,a,b,f):
    """
    put code from prelab with same returns as gauss_quad
    you can return None for the weights
    """
    h = (b-a)/(2*M)
    x = np.arange(0, (2*M)+1)*h + a
    w = np.ones((2*M)+1)
    for i in range(M):
        w[2*i+1] = 4
        if i < M-1:
            w[2*i+2] = 2

    w = w * h/3

    I = np.sum(f(x) * w)
    return I, x, w


def eval_gauss_quad(M,a,b,f):
```

```
    """
    Non-adaptive numerical integrator for \int_a^b f(x)w(x)dx
    Input:
      M - number of quadrature nodes
      a,b - interval [a,b]
      f - function to integrate

    Output:
      I_hat - approx integral
      x - quadrature nodes
      w - quadrature weights

    Currently uses Gauss-Legendre rule
    """
    x,w = lgwt(M,a,b)
    I_hat = np.sum(f(x)*w)
    return I_hat,x,w

def adaptive_quad(a,b,f,tol,M,method):
    """
    Adaptive numerical integrator for \int_a^b f(x)dx

    Input:
    a,b - interval [a,b]
    f - function to integrate
    tol - absolute accuracy goal
    M - number of quadrature nodes per bisected interval
    method - function handle for integrating on subinterval
           - eg) eval_gauss_quad, eval_composite_simpsons etc.

    Output: I - the approximate integral
            X - final adapted grid nodes
            nsplit - number of interval splits
    """
    # 1/2^50 ~ 1e-15
    maxit = 50
    left_p = np.zeros((maxit,))
    right_p = np.zeros((maxit,))
    s = np.zeros((maxit,1))
    left_p[0] = a; right_p[0] = b;
    # initial approx and grid
    s[0],x,_ = method(M,a,b,f);
    # save grid
    X = []
    X.append(x)
    j = 1;
    I = 0;
    nsplit = 1;
    while j < maxit:
      # get midpoint to split interval into left and right
      c = 0.5*(left_p[j-1]+right_p[j-1]);
      # compute integral on left and right spilt intervals
      s1,x,_ = method(M,left_p[j-1],c,f); X.append(x)
      s2,x,_ = method(M,c,right_p[j-1],f); X.append(x)
      if np.max(np.abs(s1+s2-s[j-1])) > tol:
        left_p[j] = left_p[j-1]
        right_p[j] = 0.5*(left_p[j-1]+right_p[j-1])
        s[j] = s1
        left_p[j-1] = 0.5*(left_p[j-1]+right_p[j-1])
```

```
            s[j-1] = s2
            j = j+1
            nsplit = nsplit+1
        else:
            I = I+s1+s2
            j = j-1
            if j == 0:
                j = maxit
    return I,np.unique(X),nsplit

# This script tests the convergence of adaptive quad
# and compares to a non adaptive routine

# get adaptive_quad routine and numpy from adaptive_quad.py
from adaptive_quad import *
# get plot routines
import matplotlib.pyplot as plt

# specify the quadrature method
# (eval_gauss_quad, eval_composite_trap, eval_composite_simpsons)
method = eval_composite_simpsons

# interval of integration [a,b]
a = 0.; b = 1.
# function to integrate and true values
# TRYME: uncomment and comment to try different funcs
#        make sure to adjust I_true values if using different interval!
#f = lambda x: np.log(x)**2; I_true = 2; labl = '$\log^2(x)$'
#f = lambda x: 1./(np.power(x,(1./5.))); I_true = 5./4.; labl = '$\\frac{1}{x^{1/5}}$'
# f = lambda x: np.exp(np.cos(x)); I_true = 2.3415748417130531; labl = '$\exp(\cos(x))$'
# f = lambda x: x**20; I_true = 1./21.; labl = '$x^{20}$'
# below is for a=0.1, b = 2
a=0.1;b=2;f = lambda x: np.sin(1./x); I_true = 1.1455808341; labl = '$\sin(1/x)$'

# absolute tolerance for adaptive quad
tol = 1e-3
# machine eps in numpy
eps = np.finfo(float).eps

# number of nodes and weights, per subinterval
Ms = np.arange(2,15); nM = len(Ms)
# storage for error
err_old = np.zeros((nM,))
err_new = np.zeros((nM,))


# loop over quadrature orders
# compute integral with non adaptive and adaptive
# compute errors for both
for iM in range(nM):
    M = Ms[iM];
    # non adaptive routine
    # Note: the _,_ are dummy vars/Python convention
    # to store uneeded returns from the routine
    I_old,_,_ = method(M,a,b,f)
    # adaptive routine
    I_new,X,nsplit = adaptive_quad(a,b,f,tol,M,method)
    if M == 5:
        print(nsplit)
    err_old[iM] = np.abs(I_old-I_true)/I_true
```

```
    err_new[iM] = np.abs(I_new-I_true)/I_true
    # clean the error for nice plots
    if err_old[iM] < eps:
        err_old[iM] = eps
    if err_new[iM] < eps:
        err_new[iM] = eps
    # save grids for M = 5
    if M == 5:
        mesh = X

# plot the old and new error for each f and M
fig,ax = plt.subplots(1,2)
ax[0].semilogy(Ms,err_old,'ro--')
ax[0].set_ylim([1e-16,2]);
ax[0].set_xlabel('$M$')
ax[0].set_title('Non-adaptive')
ax[0].set_ylabel('Relative Error');
ax[1].semilogy(Ms,err_new,'ro--',label=labl)
ax[1].set_ylim([1e-16,2]);
ax[1].set_xlabel('$M$')
ax[1].set_title('Adaptive')
ax[1].legend()
plt.savefig("simp.png")
#plt.show()

# plot the adaptive mesh for M=2
fig,ax = plt.subplots(1,1)
ax.semilogy(mesh,f(mesh),'ro',label=labl)
ax.legend()
ax.set_title("Mesh for M=5")
#plt.show()
plt.savefig("simp_mesh.png")
```