HW 1 Solutions

Text, letter

Description automatically generated

1. Using Taylor series Theorem, we know f(x)=P\_n(x)+R\_n(x):

P\_2(0.5)=f(0)+f’(0)\*(0.5-0)^1/1!+f’’(0)\*(0.5-0)^2/2!

R\_2(0.5)= f’’’(0)\*(0.5-0)^3/3!

By hand or using Matlab (Using symbolic commands in example DerivApprox.m and with a quick google search to differentiate higher derivatives)

syms f(x0)

f(x0)=cos(x0)\*exp(x0);

%1st derivative of f with respect to x0

deriv1=diff(f,x0,1);

%2nd derivative of f with respect to x0

deriv2=diff(f,x0,2);

%3rd derivative of f with respect to x0

deriv3=diff(f,x0,3);

%initialize point to evaluate taylor series at

x=0.5;

%initialize point to expand taylor series about

x0=0;

diff=x-x0;

%evaluate 0th deriv at x0 (function)

deriv0Atx0=eval(f);

%evaluate 1st deriv at x0

deriv1Atx0=eval(deriv1);

%evaluate 2nd deriv at x0

deriv2Atx0=eval(deriv2);

diff=x-x0

TaylorPoly=deriv0Atx0+deriv1Atx0\*diff+deriv2Atx0\*diff^2/2

exact= cos(x)\*exp(x)

Error=(exact-TaylorPoly)

~~~~~~~~~~~~~~~~~

Matlab output:

TaylorPoly = 1.500000000000000

exact = 1.446889036584169

Error = -0.053110963415831

The actual error we observe from the calculation is Error=-0.05311 and absolute error is 0.053110963415831.

We note that often we do not know what f(x) is, so we utilize the upper bound of the Taylor series as our worst case. Also, error can be positive or negative, representing an over or under estimation. Some series alternate (e.g. Taylor Series for ln(x) from class) while others converge strictly from above or strictly from below to the point f(x).

We can calculate R\_2 as:

Remainder=deriv3\*diff^3/factorial(3)

Note that for the remainder, diff=(x-x0)=0.5, but we want to evaluate the third derivative at some point that is on the interval between 0 and 0.5. Similar to example m files, we want to create a plot on an interval 0 to 0.5, so we can create an evenly spaced vector:

xval=[0:.05:.5];

With a quick google search, we can determine that evaluating a symbolic function Remainder at a vector of values can be done by using subs command:

PlotRemainder=subs(Remainder,xval);

Since the only functional dependence is in the deriv3 term, we are trying to find the worst case value to evaluate at. Similar to example m files, we want to create a plot of the x-values on 0 to 5 and the Remainder evaluated at those values:

plot(xval,PlotRemainder)

Chart, line chart

Description automatically generated

The question asks for the “upper bound for absolute error”. We want to maximize |R\_2(x)|, so we want the largest value, which is maximized at the endpoint 0.5. We could also determine this by using derivative test on R\_2(x) to determine if it is increasing or decreasing on this interval. We know it is strictly decreasing, so again, should have smallest value (largest magnitude when taking absolute value) at the endpoint 0.5.

To determine this value of remainder at x=0.5, we can use subs command again but it just gives us cos(1/2) etc., not in decimal form. To get decimal form evaluation, we also use vpa command, which is a matlab command to get symbolic variable-precision floating-point arithmetic (VPA) to evaluate each element of the symbolic input :

RemainderMax=vpa(subs(Remainder,x))

Matlab output:

0.093222004991574336245618576795828

What we observe is that are worst case absolute upper bound on error from theory is ~0.093. From direct calculation, we observe that the actual absolute error is ~0.053. This shows that actual error is sometimes better than the theoretical upper bound. The error is also still pretty large, so it might make sense to increase the number of terms if we desired more accuracy than within 0.05.]

1. We will follow the same process. Now we are just integrating f(x) vs P\_2(x). We need to define things slightly differently. Instead of directly evaluating x-x0=0.5-0=0.5, we will keep x as a variable since we are integrating over x. However, we are still directly plugging in.

syms f(x)

f(x)=cos(x)\*exp(x);

%1st derivative of f with respect to x

deriv1=diff(f,x,1);

%2nd derivative of f with respect to x

deriv2=diff(f,x,2);

%3rd derivative of f with respect to x

deriv3=diff(f,x,3);

%initialize point to expand taylor series about

x0=0

diff=x-x0;

fAtx0=vpa(subs(f,x0));

deriv1Atx0=vpa(subs(deriv1,x0));

deriv2Atx0=vpa(subs(deriv2,x0));

%1 makes this deriv term largest for R\_n(x)on [0,1]

deriv3At1=vpa(subs(deriv3,1));

TaylorPolyF=fAtx0+deriv1Atx0\*diff+deriv2Atx0\*diff^2/2

RemainderF=deriv3At1\*diff^3/factorial(3);

IntF=vpa(int(f,0,1))

IntP=vpa(int(TaylorPolyF,0,1))

IntError=IntF-IntP

IntRemainder=vpa(int(abs(RemainderF),0,1))

~~~~~~~~~~~~~~~~~~~

Matlab Output:

IntF =1.3780246135473637741735697520136

IntP =1.5

IntError =-0.12197538645263622582643024798645

IntRemainder =0.313004…

So, the observed absolute error in approximating the integral of f(x) on 0 to 1 with the integral of P\_2(x) on 0 to 1 is ~0.12. The absolute value of R\_2 uses larger positive values of R\_2 and when integrating, adds up as additional area under the curve, hence this is our worst case scenario. Using the R\_2 with derivative evaluated at 0.5 and integrating with respect to x gives us an upper bound of error of 0.313. This is slightly larger than the observed error. We know that the accuracy of P\_2 is actually much better locally—close to 0, and worse further away from 0, so error will be large and accumulate on the 0.5 to 1 region. Hence, the total absolute error over the region is 0.12 and an order of magnitude larger than the error at just a single point as in (a).

Note: Error accumulates and if we want to approximate the integrand with a function, we want it to be a good approximation on the entire interval. OR maybe we will want to break up the interval and use good approximations of the integrand on those intervals.

Text

Description automatically generated

1. Matlab Code and Output:

%initialize a value for x

x=1

%run loop until Matlab thinks x=1

while(x+1>1)

%progressively make x smaller

x=x/2;

end

%output matlab machine epsilon

eps

x =

1.110223024625157e-16

ans =

2.220446049250313e-16

1. Comparing: It looks like Matlab Machine epsilon is ~2 times greater than the one calculated in the code. However, we stopped the while loop at the point that Matlab thinks x is equal to 0. Hence, one step before, at 2\*1.11e-16 is approximately the smallest number Matlab still recognizes, so there is agreement.

Text, letter

Description automatically generated

1. Plugging in: 1\*1 +1\*(1-alpha)=2-alpha

(1+alpha)\*1+1\*1=2+alpha

Simplifying the left hand side of each of the equations, we can see equality.

1. Matlab code:

clear all

alpha=2e-3

A=[1 (1-alpha); (1+alpha) 1];

b=[2-alpha; 2+alpha];

%Direct matlab calculation of matrix inverse

Ainv=inv(A);

%Inverse of 2x2 matrix [a b; c d] is (1/(ad-bc))[d -b; -c a]

%Using known formula

AinvDef=(1/(A(1,1)\*A(2,2)-A(1,2)\*A(2,1)))\*[A(2,2) -A(1,2); -A(2,1) A(1,1)];

%Calculating x

x=Ainv\*b

xdef=Ainv\*b

clear all

alpha=2e-3

alpha=single(alpha)

A=[1 (1-alpha); (1+alpha) 1];

b=[2-alpha; 2+alpha];

%Direct matlab calculation of matrix inverse

Ainv=inv(A);

%Inverse of 2x2 matrix [a b; c d] is (1/(ad-bc))[d -b; -c a]

%Using known formula

AinvDef=(1/(A(1,1)\*A(2,2)-A(1,2)\*A(2,1)))\*[A(2,2) -A(1,2); -A(2,1) A(1,1)];

%Calculating x

x=Ainv\*b

xdef=Ainv\*b

Matlab output:

x =

1.000000000000000

0.999999999941792

x =

2×1 single column vector

0.9687500

1.0312500

1. We can see that the exact solution is [1 1]T. However, we see that even in double precision, we do not get exactly 1. We get close to 1, but within 10^-10 in the second component. This is certainly less than machine epsilon. In contrast, with single precision accuracy of alpha, we only get within 10^-2. Due to this matrix having a determinant that is 1/10^5, this is possibly causing a loss of accuracy. What we see here in this problem is that if we do a few calculations, the accuracy of our solution can be greatly changed by the decimal accuracy of the inputs.

Note: In general, one would never want to do calculations in single precision when we want accurate solutions to mathematical calculations! However, double precision has more memory and computational time. Single precision calculations might be faster with less memory, so when accuracy is not needed, these might be better. Single precision is often done on gpus and in gaming applications where the exact mathematical accuracy is not necessary.

Text, letter

Description automatically generated

1. If we multiply g(x) by (sqrt(x+1)-sqrt(x)), we would get:

x(sqrt(x+1)-sqrt(x)) in the numerator

denominator would give us: x+1-x=1

Therefore, f(x)=g(x)

1. Matlab code / Or by hand:

p=500;

%Matlab command Y = round( X , N ) rounds X to N digits

f=round(p\*(round(sqrt(p+1),6)-round(sqrt(p),6)),6)

g=round(round(p,6)/(round(round(sqrt(p+1),6)+round(sqrt(p+1),6))),6)

Matlab output:

f = 11.174500000000000

g = 11.111110999999999