18.330 Pset 1

Kathleen Brandes

February 19, 2019

1 Problem 1

1.1 Part A

We do not know the exact value of $S = \sum_{n=1}^{\infty} \frac{1}{n^4}$. But since the function $f(x) = \frac{1}{x^4}$ is positive and decaying, then we know that $S_N < S$ for any partial sum and we can use this to upper bound the accuracy of our error (where error is $\epsilon_N = |S_N - S|$) as follows: $E_N = \frac{|S_N - S|}{S} = \frac{\epsilon_N}{S} < \frac{\epsilon_N}{S_N}$.

We can then solve for the error ϵ_N:

$|S_N - S| = |\sum_{n=1}^{N} \frac{1}{n^4} - \sum_{n=1}^{\infty} \frac{1}{n^4}| =$

$|\sum_{n=N+1}^{\infty} \frac{1}{n^4}| < \int_N^{\infty} \frac{1}{x^4} dx =$

$\frac{-1}{3x^3}|_N^{\infty} = 0 + \frac{1}{3N^3}$.

Additionally, we can compute an exact answer for the partial sum $S_N = \sum_{n=1}^{N} \frac{1}{n^4} = \int_1^N \frac{1}{x^4} dx = \frac{-1}{3N^3} + \frac{1}{3} = \frac{N^3-1}{3N^3}$.

Now, to estimate S with 9 digit precision, we need $E_N < \frac{\epsilon_N}{S_N} < 10^{-9}$. This happens when $\frac{\frac{1}{3N^3}}{\frac{N^3-1}{3N^3}} = \frac{3N^3}{3N^3(N^3-1)} = \frac{1}{N^3-1} < 10^{-9}$, which occurs when $N > 1000$.

1.2 Part B

This Julia program evaluates the partial sum for $f(x) = \frac{1}{x^4}$ and the error from the exact solution.

```
function partial_sum(f, N)
        sum = 0
        for i in 1:N
                sum+=f(i)
        end
        return sum
```

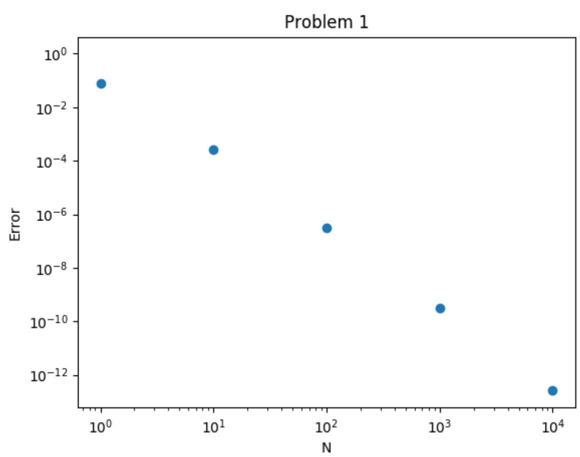
```
end
function f(input)
        return 1/(input^4)
end
function error(psum)
        exact_soln = pi^4/90.
        num = abs(psum-exact_soln)
        return num/exact_soln
end

result = partial_sum(f, 1000)
println(result)
println(error(result))

x = [0,1,10,100,1000,10000,100000,1000000,10000000]
y = [error(partial_sum(f, i)) for i in x]

using PyPlot
figure()
p = loglog(x,y, "o")
xlabel("N")
ylabel("Error")
title("Problem 1")
savefig("problem1.png")
```

And produces the following graph:



The prediction from part A seems fairly accurate since the outputted error for $N = 1000$ by Julia was $E_N = 3.05 * 10^{-10}$, which has the desired precision.

2 Problem 2

2.1 Part A

This sum will add N multiples of the fraction $\frac{\pi}{N}$, which should add up to exactly π. Therefore, it should have no error because each partial sum P_N, for any value of N, should evaluate to exactly π, which would make the error evaluate to zero since the numerator is $|P_N - \pi|$. So, the error should behave independently from the value of N increasing.

2.2 Part B

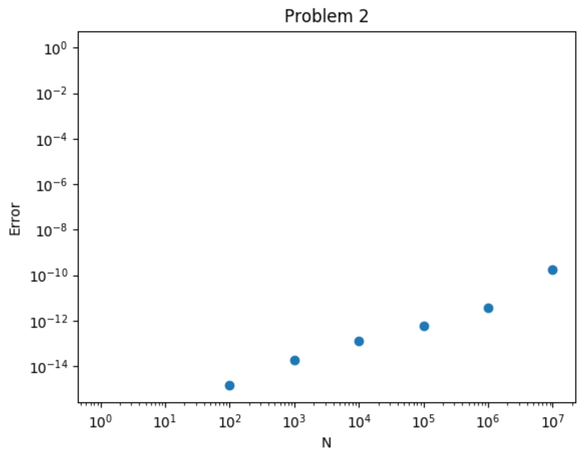
Here is my program in Julia to evaluate how the error changes as N increases:

```
function P_N(N)
        summand = pi/N
        S = 0.0
        for i=1:N
                S += summand
        end
        return S
end
function error(psum)
        exact_soln = pi
        num = abs(psum-exact_soln)
        return num/exact_soln
end

x = [0,1,10,100,1000,10000,100000,1000000,10000000]
y = [error(P_N(i)) for i in x]

using PyPlot
figure()
p = loglog(x,y, "o")
xlabel("N")
ylabel("Error")
title("Problem 2")
savefig("problem2.png")
```

The error is slightly different from what I expected as in theory it should have 0 error every time, however I believe the increase in error is likely due to floating point mistakes in the division being propagated more and more when the sum is for larger values of N. The code comparing N vs. error produces the following graph:



3 Problem 3

3.1 Part A

For the new generalized interval of $[a, b]$, the new steps will be $x'_n = a + \frac{b-a}{1-(-1)}(x_n - (-1)) = a + \frac{b-a}{2}(x_n + 1)$, and it follows similarly that the new weights will be $w'_n = \frac{b-a}{2}w_n$.

3.2 Part B

Let $P(x) = a + bx + cx^2$; we can interpolate this function at three points, $(-1, f(-1)), (0, f(0)), (1, f(1))$, to get the following:

$P(x) = f(-1)\frac{(x-0)(x-1)}{(-1)(-2)} + f(0)\frac{(x+1)(x-1)}{(1)(-1)} + f(1)\frac{(x+1)(x-0)}{(2)(1)} =$

$f(-1)\frac{x^2-x}{2} + f(0)\frac{x^2-1}{-1} + f(1)\frac{x^2+x}{2} =$

$\frac{f(-1)}{2}x^2 - \frac{f(-1)}{2}x - f(0)x^2 + f(0) + \frac{f(1)}{2}x^2 + \frac{f(1)}{2}x =$

$\frac{1}{2}(f(-1) + 2f(0) + f(1))x^2 + \frac{1}{2}(f(1) - f(-1))x + f(0)$

Therefore, we can see that $c = \frac{1}{2}(f(-1) + 2f(0) + f(1))$, $b = \frac{1}{2}(f(1) - f(-1))$, and $a = f(0)$ for the polynomial $P(x)$.
Now, if we integrate this polynomial on the range [-1,1], we get this result:

$\int_{-1}^{1}(a + bx + cx^2)dx =$

$\frac{c}{3}x^3 + \frac{b}{2}x + ax + k|_{-1}^{1} =$

$\frac{c}{3} + \frac{b}{2} + a + k + \frac{c}{3} - \frac{b}{2} + a - k =$

$\frac{2}{3}c + 2a =$

$\frac{2}{3}(\frac{1}{2}(f(-1) + 2f(0) + f(1)) + 2f(0) =$

$\frac{1}{3}f(-1) - \frac{2}{3}f(0) + \frac{1}{3}f(1) + 2f(0) =$

$\frac{1}{3}(f(-1) + 4f(0) + f(1))$

3.3 Part C

We can now express this as a general Simpson's Rule over an interval $[a, b]$ by letting $a = -1$, $b = 1$. This gives us the result $\int_a^b f(x)dx = \frac{1}{3}(f(a) + 4f(a + \frac{b-a}{2}) + f(b))$ as a generalized form for a single step.

3.4 Part D

To create the composite Simpson's Rule, considering the interval $[u, v]$ for N steps, we get a step size of $h = \frac{v-u}{N}$. Additionally, this will create $N - 1$ sub-intervals of the form $[u + hi, u + h(i + 1)]$ for $i = 0...N - 1$. If we apply the single step Simpson's rule to each sub-interval, we will get 3 different function evaluations for $u + hi$, $u + \frac{h}{2}i$, and $u + h(i + 1)$. However, each sub-interval will repeat the function evaluation at endpoint of the previous sub-interval, excluding the evaluations at u and v.This is reflected by the resulting composite rule:

$\int_u^v f(x)dx = \frac{h}{3}[f(x_0) + 4f(x_0 + \frac{h}{2}) + 2f(x_1) + 4f(x_1 + \frac{h}{2}) + ... + 2f(x_{N-1}) + 4f(x_{N-1} + \frac{h}{2}) + f(x_N)]$

To count the total number of function evaluations, we know there are $N - 1$ function evaluations with a weight of 4, there are N function evaluations with a weight of 2, and then 2 function evaluations for the endpoints u, v, which leads to a $N - 1 + N + 2 = 2N + 1$ total function evaluations.

4 Problem 4

Here are my functions for implementing the composite 0th, 1st and 2nd order Newton-Cotes quadrature rules for arbitrary functions and error:

```
function newton_zero(f,a,b, N)
        delta = (b-a)/(1.0*N)
        sum=0.0
        for i in 1:N-1
                sum+= delta*f(a+i*delta)
        end
```

```
        return sum
end

function newton_one(f,a,b, N)
        delta = (b-a)/(1.0*N)
        sum=.5*delta*(f(a)+f(b))
        for i in 1:N-1
                sum+= delta*f(a+i*delta)
        end
        return sum
end

function newton_two(f, a, b, N)
        delta = (b-a)/(1.0*N)
        h = delta/2.0
        sum = h*f(a)/3+h*f(b)/3
        for n in 1:N-1
                sum+=(h/3)*(2*f(a+2*n*h)+4*f(a+(2*n-1)*h))
        end
        sum+= 4*h*f(a+(2*N-1)*h)/3
        return sum
end

function error(psum,exact_soln)
        num = abs(psum-exact_soln)
        return num/exact_soln
end
```

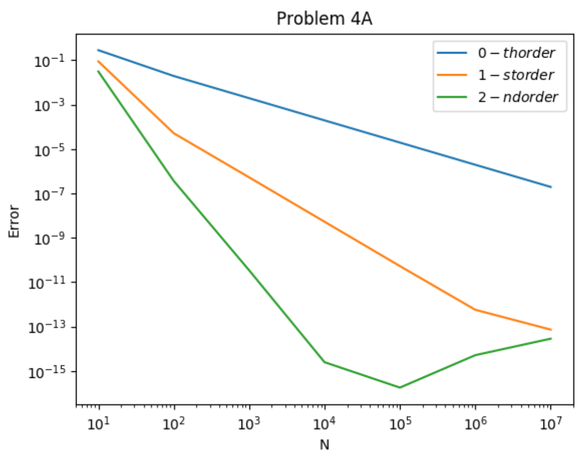
4.1 Equation A

Here is my code to compute and evaluate the different methods on equation A:

```
I_a_exact=2.519307982030761255
function I_a(x)
        return exp(cos((x+1)^2+2*sin(4*x+1)))
end
x = [10,100,1000,10000,100000,1000000,10000000]
y_0 = [error(newton_zero(I_a,0,pi,n), I_a_exact) for n in x]
y_1 = [error(newton_one(I_a,0,pi,n), I_a_exact) for n in x]
y_2 = [error(newton_two(I_a,0,pi,n), I_a_exact) for n in x]
using PyPlot
figure()
p = loglog(x,y_0,label=L"0-th order")
loglog(x,y_1,label= L"1-st order")
loglog(x,y_2, label = L"2-nd order")
legend()
```

```
xlabel("N")
ylabel("Error")
title("Problem 4A")
savefig("problem4A.png")
```

The resulting plot of error from the exact solution and the estimates of each quadrature rule:



The resulting error analysis of this function looks as I expected, with the first order error falling at a slope of -1 and second order falling off with a slope of roughly -2. Looking at a graph of the function, it further makes sense that quadratic polynomial estimates for the integral would provide the least error as it is smooth.

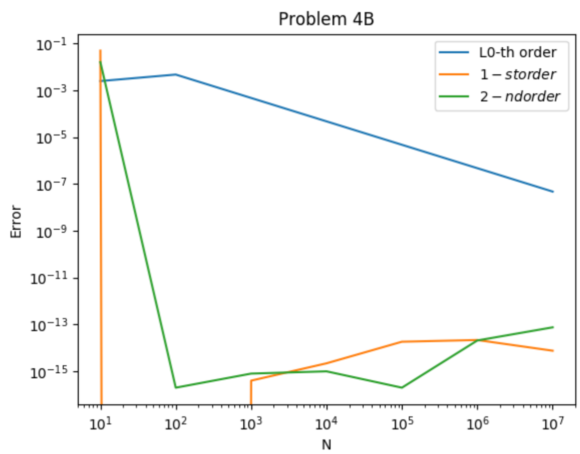
4.2 Equation B

Here is my code to compute and evaluate the different methods on equation B:

```
I_b_exact = 4.48895606126995688 30
function I_b(x)
        return exp(cos((cos(x+1))^2+2*sin(4*x+1)))
end
x = [10,100,1000,10000,100000,1000000,10000000]
y_0 = [error(newton_zero(I_b,0,pi,n), I_b_exact) for n in x]
y_1 = [error(newton_one(I_b,0,pi,n), I_b_exact) for n in x]
y_2 = [error(newton_two(I_b,0,pi,n), I_b_exact) for n in x]
using PyPlot
figure()
p = loglog(x,y_0,label="L0-th order")
loglog(x,y_1,label= L"1-st order")
loglog(x,y_2,label=L"2-nd order")
```

```
legend()
xlabel("N")
ylabel("Error")
title("Problem 4B")
savefig("problem4B.png")
```

The resulting plot of error from the exact solution and the estimates of each quadrature rule:



Since this function is periodic and smooth, it makes sense that the errors are so small for the estimates of the integral because when the length of the interval is equal to the period, then the error rapidly will shrink to zero. However, I am surprised that the trapezoidal Newton Cotes estimate is better than the second order Simpson's Newton Cotes estimate.

4.3 Equation C

The function $f(x) = \frac{tanh(x)}{(|x-\pi|)^{1/2}}$ has a singularity at $x = \pi$. To address this before trying to estimate this function using the various quadrature rules, I did a u-substitution to remove the singularity:

$\int_0^{2\pi} f(x)dx = \int_0^{\pi} \frac{tanh(x)}{(\pi-x)^{1/2}}dx + \int_{\pi}^{2\pi} \frac{tanh(x)}{(x-\pi)^{1/2}}$

Then substitute $u = (\pi - x)^{1/2}$ and $du = \frac{-1}{2(\pi-x)^{1/2}}$ for the first integral and $u = (x-\pi)^{1/2}$ and $du = \frac{1}{2(x-\pi)^{1/2}}$. From this, you get the integral to be:

$\int_{\sqrt{\pi}}^{0} -2tanh(\pi - u^2)du + \int_0^{\sqrt{\pi}} 2tanh(u^2 + \pi)du =$

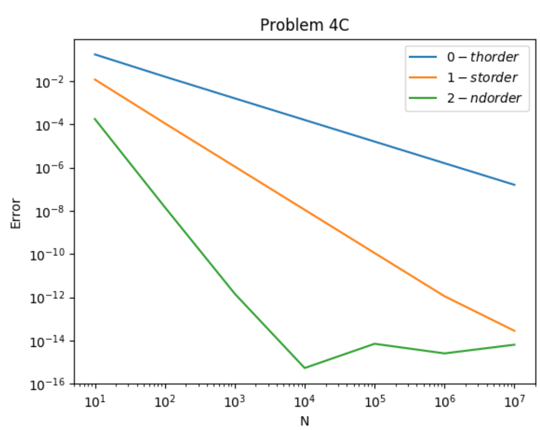
$\int_0^{\sqrt{\pi}} 2tanh(\pi - u^2)du + \int_0^{\sqrt{\pi}} 2tanh(u^2 + \pi)du =$

And now, we can do Newton Cotes separately on these two functions over new intervals to get a more accurate estimate of the exact value. Here is my code to compute and evaluate the different methods on equation C:

```
I_c_exact = 6.6388149923287733132
#u-substitution
function I_c_left(u)
        return 2*tanh(pi-u^2)
end
function I_c_right(u)
        return 2*tanh(u^2+pi)
end

y_0 = [error(newton_zero(I_c_left,0,sqrt(pi),n/2.)+newton_zero(I_c_right,0,sqrt(
y_1 = [error(newton_one(I_c_left,0,sqrt(pi),n/2.)+newton_one(I_c_right,0,sqrt(pi
y_2 = [error(newton_two(I_c_left,0,sqrt(pi),n/2.)+newton_two(I_c_right,0,sqrt(pi
test_zero = [newton_zero(I_c_left,0,sqrt(pi),n/2.)+newton_zero(I_c_right,0,sqrt(
using PyPlot
figure()
p = loglog(x,y_0,label=L"0-th order")
loglog(x,y_1,label= L"1-st order")
loglog(x,y_2, label =L"2-nd order")
legend()
xlabel("N")
ylabel("Error")
title("Problem 4C")
savefig("problem4C.png")
```

The resulting plot of error from the exact solution and the estimates of each quadrature rule:



This function is discontinuous for $x = \pi$, so I would expect estimates of the exact value for the integral to be incorrect, especially for large step sizes where it may not catch the discontinuity. I am surprised that the 3 quadrature methods were able to successfully estimate the exact value with very little error; I believe it is so successful because the u-substitution makes it so that there is no discontinuity in the function calculation, so therefore the estimates can be more accurate.

4.4 Equation D

Since the function for equation d was an improper integral with a singularity at zero, I did a substitution of $\frac{arctan(kx)}{k}|_1^\pi = \int_1^\pi \frac{1}{1+k^2x^2}dk$, which created a double integral:

$\int_0^\infty \frac{arctan(\pi x)-arctan(x)}{x}dx = \int_0^\infty \int_1^\pi \frac{1}{1+k^2x^2}dkdx$

$= \int_1^\pi \int_0^\infty \frac{1}{1+k^2x^2}dxdk = \int_1^\pi \int_0^\infty \frac{1}{k}\frac{k}{1+k^2x^2}dxdk$

$= \int_1^\pi arctan(kx)|_0^\infty \frac{1}{k}dk = \int_1^\pi \frac{1}{k}(\frac{\pi}{2}-0)dk$

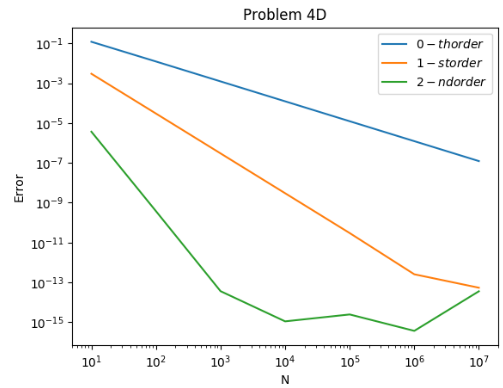
$= \int_1^\pi \frac{\pi}{2k}dk$

Using this new integral and function, which has no singularities and is not improper, I have written code to compute and evaluate it with the different methods:

```
I_d_exact = 1.7981374998645790990
I_d_exact = 1.7981374998645790990
function I_d(x)
        return pi/(2*x)
end

y_0 = [error(newton_zero(I_d,1,pi,n), I_d_exact) for n in x]
y_1 = [error(newton_one(I_d,1, pi,n), I_d_exact) for n in x]
y_2 = [error(newton_two(I_d,1,pi,n), I_d_exact) for n in x]
using PyPlot
figure()
p = loglog(x,y_0,label=L"0-th order")
loglog(x,y_1,label= L"1-st order")
loglog(x,y_2, label =L"2-nd order")
legend()
xlabel("N")
ylabel("Error")
title("Problem 4D")
savefig("problem4D.png")
```

The resulting plot of error from the exact solution and the estimates of each quadrature rule is below. Since this function can be simplified to a smooth, continuous function on a closed interval using different substitutions and meth-



ods to remove/evaluate improper integrals, it makes sense that the simplified expression for equation D can be estimated with minimal error for a relatively small N.

5 Problem 5

Show that composite Simpson's Rule will exactly solve all cubic polynomials, $Q(x) = c_0 + c_1x + c_2x^2 + c_3x^3$. The exact solution can be computed as follows:

$\int_a^b Q(x)dx =$

$\int_a^b (c_0 + c_1x + c_2x^2 + c_3x^3)dx =$

$\frac{c_3}{4}x^4 + \frac{c_2}{3}x^3 + \frac{c_1}{2}x^2 + c_0x|_a^b =$

$\frac{c_3}{4}b^4 + \frac{c_2}{3}b^3 + \frac{c_1}{2}b^2 + c_0b - \frac{c_3}{4}a^4 - \frac{c_2}{3}a^3 - \frac{c_1}{2}a^2 - c_0a =$

$\frac{c_3}{4}(b^4 - a^4) + \frac{c_2}{3}(b^3 - a^3) + \frac{c_1}{2}(b^2 - a^2) + c_0(b - a)$

Now, if we apply the composite simpson's rule to estimate the result of this integral:

$\int_a^b Q(x)dx =$

$\frac{b-a}{6}(Q(a) + 4Q(\frac{a+b}{2}) + Q(b)) =$

$\frac{b-a}{6}(c_3a^3 + c_2a^2 + c_1a + c_0 + 4[c_3(\frac{a+b}{2})^3 + c_2(\frac{a+b}{2})^2 + c_1(\frac{a+b}{2}) + c_0] + c_3b^3 + c_2b^2 + c_1b + c_0) =$

$\frac{b-a}{6}(c_3(a^3 + b^3) + c_2(a^2 + b^2) + c_1(a + b) + 2c_0 + \frac{c_3(a+b)^3}{2} + c_2(a + b)^2 + 2c_1(a +$

$b) + 4c_0) =$

$\frac{b-a}{6}(\frac{3c_3}{2}(a^3 + a^2b + ab^2 + b^3) + 2c_2(a^2 + ab + b^2) + 3c_1(a + b) + 6c_0) =$

$\frac{c_3}{4}(ba^3 + a^2b^2 + ab^3 + b^4 - a^4 - a^3b - a^2b^2 - ab^3) + \frac{c_2}{3}(ba^2 + b^2a + b^3 - a^3 - b^2a - ab^2) + \frac{c_1}{2}(b^2 - a^2) + c_0(b - a) =$

$\frac{c_3}{4}(b^4 - a^4) + \frac{c_2}{3}(b^3 - a^3) + \frac{c_1}{2}(b^2 - a^2) + c_0(b - a)$

By computing both the exact solution to the integral of a cubic polynomial on the range $[a, b]$ and the estimated solution using the composite Simpson's Rule, we can see that they are identical, therefore showing that the composite Simpson's Rule can be used to exactly integrate all cubic polynomials.