Group: Daniel McBride, Sean O'Reilly, Sarah Moon

Analysis of Composite Simpson's, Romberg, and Quadrature Methods

By: Adam Headley December 12, 2014

The program was created using C++ files that can be found in the appendix at the end.

A The analytical evaluation of the following integral is demonstrated below.

$$I = \int_0^{\pi} \frac{dx}{1.1+\cos(x)}.$$

For the integrand $\frac{1}{\frac{11}{10}+\cos(x)}$, substitute $u = \tan\left(\frac{x}{2}\right)$ and $du = \frac{1}{2}\sec^2\left(\frac{x}{2}\right)dx$.
Then transform the integrand using the substitutions

$\sin(x) = \frac{2u}{u^2+1}$, $\cos(x) = \frac{1-u^2}{u^2+1}$, and $dx = \frac{2du}{u^2+1}$:

$$\int \frac{dx}{\frac{11}{10}+\cos(x)} = \int \frac{2\,du}{(u^2+1)\left(\frac{1-u^2}{u^2+1}+\frac{11}{10}\right)} = \int \frac{20\,du}{u^2+21} = \frac{20}{21}\int \frac{du}{\frac{u^2}{21}+1}.$$

For the integrand $\frac{1}{\frac{u^2}{21}+1}$, substitute $s = \frac{u}{\sqrt{21}}$ and $ds = \frac{du}{\sqrt{21}}$:

$$\frac{20}{21}\int \frac{du}{\frac{u^2}{21}+1} = \frac{20}{\sqrt{21}}\int \frac{ds}{s^2+1} = \frac{20\tan^{-1}(s)}{\sqrt{21}} = \frac{20\tan^{-1}\left(\frac{u}{\sqrt{21}}\right)}{\sqrt{21}} = \frac{20\tan^{-1}\left(\frac{\tan\left(\frac{x}{2}\right)}{\sqrt{21}}\right)}{\sqrt{21}}.$$

$$I = \frac{10\,\pi}{\sqrt{21}} \approx 85551720847258.$$

B Using the error formula for the Composite Simpson's Method an estimate for subinterval h can be found, however the fourth derivative needs to be taken. WolframAlpha shows:

$$\frac{d^4}{dx^4}\left(\frac{1}{1.1+\cos(x)}\right) =$$
$$\frac{6\cos^2(x)}{(\cos(x)+1.1)^3} - \frac{\cos(x)}{(\cos(x)+1.1)^2} + \frac{24\sin^4(x)}{(\cos(x)+1.1)^5} - \frac{8\sin^2(x)}{(\cos(x)+1.1)^3} + \frac{36\sin^2(x)\cos(x)}{(\cos(x)+1.1)^4}$$

This fourth derivative has a maximum of 6100 with $x \in [0,\pi]$. Using Composite Simpson's Rule's error term, $\frac{b-a}{180}\,h^4 f^{(4)}(\xi)$, h can be found small enough to keep the absolute error less than 0.0001 :

$$\frac{\pi}{180}\, h^4 \times (6100) < 0.0001 \quad \Rightarrow \quad h < \sqrt[4]{\frac{0.018}{6100\pi}} \quad \Rightarrow \quad h < 0.03113137.$$

Implementing the program in the appendix under "CompositeSimpsons.cpp" with evenly space nodes an approximation for the integral above can be obtained under the given precision. Taking $n = 32$, this code returns a value of 6.5126620012. Comparing the actual value of $I = 6.8555$ to this approximation returns an absolute error of 0.3428379987. When $n = 32$, $h = 0.0981747 \not< 0.03113137$. Setting $n = 158$, yields $h = 0.0198834978$ and the approximation of the integral 6.7860563119. When $n = 158$, the absolute error is $0.0694612084 \not< 0.0001$, this is most likely due to round off error.

C

By implementing the Romberg procedure, which is repeated Richardson's Extrapolation on the Trapezoid Rule, to produce a sequence of values found recursively that minimizes the trapezoidal error along the curve ending at the most accurate approximation of the integral. The "Romberg.cpp" in the appendix was written using the routine described in Section 4.5 of the text. It takes in the points to integrate between and runs n iterations of the Romberg procedure.

Taking $n = 100$ under the same interval, $[0, \pi]$, the approximation yields 6.855437, and an absolute error of $0.0000800798087659496 < 0.0001$.

To empirically analyze the rate of convergence of I_n to I, the ratios $R_n = \frac{I_{2n} - I_n}{I_{4n} - I_{2n}}$ need to be calculated where I_n is the approximate integral using step size $h = \frac{b-a}{n}$

Nodes n	Ratio R_n
1	1.52434
2	2.52087
3	9.39154
4	16.9101

Notice that R_n seems to grow at a rate of about n^2.

D

To implement Gaussian Quadrature to this integral, the interval of integration needs to be converted from $[0, \pi]$ to $[-1, 1]$. A change of variable is needed:

$$t = \frac{2}{\pi}x - 1 \implies x = \frac{\pi}{2}(t+1) \implies dx = \frac{\pi}{2}dt$$

Producing the integral

$$\frac{\pi}{2}\int_{-1}^{1} \frac{1}{1.1 + \cos\left(\frac{\pi}{2}(t+1)\right)}\,dt$$

The values of the coefficients and the associated roots for $n = 5$ for the Legendre polynomials on the interval $[-1,1]$ are given in table 4.12 page 232 of the text. The approximation returns 6.893937, with an absolute error 0.038419981733564. This is not that great of an approximation however given the amount of computation that went into this evaluation compared to the other programs, it is quite impressive.

E

The adaptive quadrature method described in algorithm 4.3, page 224 of the text is in the Appendix under "AdaptiveQuad.cpp". This program takes in points to integrate between, a given tolerance for error, and a maximum amount of iterations N. Then it computes Composite Simpson's Rule approximations on the interval. Finally it focuses on intervals with higher error and splits them into smaller subintervals to execute more Composite Simpson's Rule.

Using the adaptive quadrature method with a tolerance of 0.0001 the integral is estimated to 6.8555244126654928394 with an absolute error of $7.2041929128394 \times 10^{-6}$, the best approximation yet.

The subintervals are smaller near $x = \pi$. This region contained the most subintervals because of how volatile the function behaves here with precipitous oscillations. In other programs with evenly spaced points, the error is greater near $x = \pi$. This adaptive quadrature program notices the region of large error and improves the error by splitting this region of the interval into subintervals to observe the behavior more closely.

F

This programming project compared four different integration approximation formulas including Composite Simpson's Rule, Romberg Integration, Gaussian Quadrature, and Adaptive Quadrature. The integral analytically evaluated in the beginning operated as the actual value, $I = 6.85551720847258$. The following table shows the relationship between absolute errors of each method's result compared with the actual value:

Method	Nodes n	Absolute Error
Composite Simpson's Method	32	3.42855×10^{-1}
Composite Simpson's Method	158	$6.946120847258 \times 10^{-2}$
Romberg Integration	100	$8.00798087659496 \times 10^{-5}$
Gaussian Quadrature	5	$3.8419981733564 \times 10^{-2}$
Adaptive Quadrature	?	$7.2041929128394 \times 10^{-6}$

By comparing the absolute errors in the table above, Romberg Integration and Adaptive Quadrature yield more accurate approximations than the other methods. Ideally, the Composite Simpson's Rule and Gaussian Quadrature have the potential of yielding similarly accurate errors, if they were assessed with higher computing power and smaller step sizes. Gaussian Quadrature having been done manually with five nodes effects the accuracy of the approximation. Adaptive Quadrature involves Composite Simpson's Rule, reducing the errors by splitting the intervals into subintervals where the errors accumulate from high frequency oscillations. This yields an approximation within a given tolerance, that does not abuse computing power by decreasing insignificant errors.

In the end, Adaptive Quadrature worked best for this integral, however these different methods still serve a purpose. There is no "one method to rule them all". Various methods are used for different purposes, like the Trapezoid Rule is less accurate in some cases they may be easier to compute. Some may converge slower and/or require more computation than others in certain situations. By making qualitative observations about the operation it is possible to choose an appropriate method that optimizes accuracy, rate of convergence, and/or computation cost. Being able to choose a method based on these factors is fundamental to being Numerical Analyst.

Appendix

CompositeSimpsons.cpp

```
#include <iostream>
#include <tgmath.h>
#include <iomanip>

//CompositeSimpsons.cpp

using namespace std;

int main()
{
	double a = 0; double b = 3.14159; int n = 32; double h = (b+a)/n;
	double XI[n]; double X;

	XI[0] = 1/(1.1 + cos(a)) + 1/(1.1 + cos(b));
	XI[1] = 0;
	XI[2] = 0;

	for(int i = 1; i < n; i++)
	{
		X = a + i*h;
		if(i % 2 != 0)
		{
			XI[2] = XI[2] + 1/(1.1 + cos(X));
		 }
		 else
		{
			XI[1] = XI[1] + 1/(1.1 + cos(X));
		}
	}

	X = h*(XI[0]+2*XI[2]+4*XI[1])/3;
	cout << std::setprecision(40) << X << endl;
}
```

Romberg.cpp

```
#include <iostream>
#include <tgmath.h>
#include <iomanip>
#include <math.h>

using namespace std;

int main()
{
	double a = 0; double b = 3.14159; double h = (b-a); int n = 100;
	double R[3][n+1]; double Sum = 0;

	R[1][1] = h*b/2;

	cout << R[1][1] << endl;

	for(int i = 2; i <= n; i++)
	{
		for(int k = 1; k <= pow(2,(i-2)); k++)
		{
			Sum = Sum + (1/(1.1+cos((k-.5)*h)));
		}

		R[2][1] = .5*(R[1][1]+ h*Sum);
		Sum = 0;

		for(int j = 2; j <= i; j++)
		{
			R[2][j] = R[2][j-1] + (R[2][j-1] - R[1][j-1])/(pow(4,(j-1))-
1);
		}

		//cout << "i = " << i << ' '<< endl;

		for(int j = 1; j <= i; j++)
		{
		cout << std::setprecision(20) << R[2][j] << ' ';
```

```
                }

                cout << endl;

                I[i] = R[2][i];

                //cout << setprecision(20) << R[2][i] << endl;

                if(((R[1][i-1] - R[2][i]) < .0001)&&(-(R[1][i-1] - R[2][i])) < .0001)
                {
                        break;
                }

                h = h/2;

                for(int j = 1; j <= i; j++)
                {
                        R[1][j] = R[2][j];
                }
        }
        /*
        for(int k = 1; k < 5; k++)
        {
                Rn = (I[2*k] - I[k])/(I[4*k]-I[2*k]);
                cout << "R" << k << " = " << Rn << endl;
        }
         */
}
```

AdaptiveQuad.cpp

```
#include <iostream>
#include <iomanip>
#include <tgmath.h>

using namespace std;

main()
{
```

```
double a = 0; double b = 3.141592653589793238;
double TOL = .0001; int N = 1000; double APP = 0;
int i = 1;
double TOLi[N];
double ai[N];
double h[N];
double FA[N];
double FC[N];
double FB[N];
double S[N];
double L[N];

TOLi[i] = 10*TOL;
ai[i] = a;
h[i] = (b-a)/2;
FA[i] = 1/(1.1+cos(a));
FC[i] = 1/(1.1+cos(a+h[i]));
FB[i] = 1/(1.1+cos(b));
S[i] = h[i]*(FA[i]+4*FC[i] + FB[i])/3;
L[i] = 1;

double FD; double FE; double S1; double S2; double v[9];

while(i>0)
{
        FD = 1/(1.1+cos(ai[i]+h[i]/2));
        FE = 1/(1.1+cos(ai[i]+3*h[i]/2));
        S1 = h[i]*(FA[i] + 4*FD + FC[i])/6;
        S2 = h[i]*(FC[i] + 4*FE + FB[i])/6;

        v[1] = ai[i];
        v[2] = FA[i];
        v[3] = FC[i];
        v[4] = FB[i];
        v[5] = h[i];
        v[6] = TOLi[i];
        v[7] = S[i];
        v[8] = L[i];
```

```
                i = i-1;

                if(((S1+S2-v[7]) < v[6])&&(-(S1+S2-v[7])< v[6]))
                {
                        APP = APP + (S1+S2);
                }

                else if (v[8] >= N)
                {
                        cout << "Error " << endl;
                        break;
                }
                else
                {
                        i = i + 1;
                        ai[i] = v[1] + v[5];
                        FA[i] = v[3];
                        FC[i] = FE;
                        FB[i] = v[4];
                        h[i] = v[5]/2;
                        TOLi[i] = v[6]/2;
                        S[i] = S2;
                        L[i] = v[8] + 1;

                        i = i + 1;
                        ai[i] = v[1];
                        FA[i] = v[2];
                        FC[i] = FD;
                        FB[i] = v[3];
                        h[i] = h[i-1];
                        TOLi[i] = TOLi[i-1];
                        S[i] = S1;
                        L[i] = L[i-1];
                }
        }
        cout << setprecision(20) << APP << endl;
}
```

CompositeSimpsons output

6.5126620012944442805746803060173988342 29

Romberg output

4.93479

3.89539066415700308 3.5489229308593377432

4.3813141154932093713 4.5432885992719445056 4.6095796438327845124

5.426705910941601374 5.7751698427577320416 5.8572952589901179365

5.8771002687545204068

6.1354288446229503506 6.3716698225167336389 6.4114364878339999976

6.4202323803553316139

6.4223623102047469047

6.4954548330553603108 6.6154634958661633348 6.6317164074227923365

6.635212914082932123

6.6360559749995111645 6.6362648642124089093

6.6754726985764945013 6.7354786537502056021 6.7434796642758083607

6.745253684225856361

6.7456852166577894181 6.745792381116888059 6.74581912776326309

6.7654816721970565396 6.7954846634039105524 6.7994850640474906456

6.8003740386470408197

6.8005901968996731455 6.8006438674570945935 6.8006572622034537901

6.800660609461858 4635

6.8104861690520595019 6.8254876680037268599 6.8274878683103814581

6.8279323572669357034

6.8280404291046608023 6.8280672621762201757 6.8280739589759074093

6.8280756324604201524

6.828076050786 7544

6.8329884199805732692 6.8404891702900778583 6.8414892704425014358

6.841711514920 7887725

6.8417655508331565173 6.8417789673744353607 6.841782315776193002

6.8417831525189631847

6.8417833616823120479 6.841783413971 7532693

6.8442395460694545051 6.847989921432414917 6.8484899715085703065

6.848601093747713 9748

6.8486281117038982913 6.8486348199745439302 6.848636494175423195

6.848636912546807 8423

6.8486370171284818298 6.8486370432732019964 6.8486370498093380732

6.849865109270011132 6.8517402970035297116 6.8519903220416038536

6.8520458831611756878
6.8520593921392674019 6.8520627462745906655 6.8520635833750302979
6.8520637925607230656
6.8520638448515605035 6.8520638579239205868 6.8520638611919890693
6.8520638620090048576
6.8526778909093168934 6.8536154847890857766 6.8537404973081228476
6.8537682778679087647
6.8537750323569550659 6.8537767094246166977 6.8537771279748360698
6.8537772325676824536
6.8537772587131007285 6.8537772652492812142 6.8537772668833154555
6.8537772672918233496
6.8537772673939505452
6.8540842817387250818 6.8545530786818611446 6.8546155849413796801
6.8546294752212730828
6.8546328524657962333 6.8546336909996270492 6.8546339002747371794
6.8546339525711603713
6.8546339656438695087 6.8546339689119593075 6.8546339697289759 84
6.8546339699330229487
6.8546339699842926407 6.8546339699970584292
6.8547874771558721108 6.8550218756282541577 6.8550531287580138695
6.8550600738979605708
6.8550617625202221461 6.8550621817871375541 6.8550622864246921 75
6.8550623125729037 71
6.8550623191092583397 6.8550623207433032391 6.8550623211518120215
6.8550623212539392171
6.8550623212794707939 6.8550623212858541322 6.8550623212874501888
6.8551390748650593565 6.8552562741014551051 6.8552719006663354051
6.8552753732363091999
6.8552762175474395434 6.8552764271808968033 6.8552764794996745579
6.8552764925737799118
6.8552764958419576402 6.855276496658980534 6.8552764968632349252
6.8552764969142980789
6.8552764969270638673 6.8552764969302550924 6.8552764969310526766
6.8552764969312525167
6.8553148737197977525 6.8553734733380435884 6.8553812866204824061
6.8553830229054693035
6.8553834450610349194 6.8553835498777635493 6.8553835760371519825
6.8553835825742046595

6.8553835842082930796 6.8553835846168045265 6.8553835847189317221
6.8553835847444632989
6.8553835847508466372 6.8553835847524426939 6.855383584752841486
6.8553835847529409619
6.8553835847529658309
6.8554027731472109153 6.8554320729563489323 6.8554359795975692293
6.8554368477400622339
6.8554370588178450419 6.8554371112262098009 6.8554371243059044616
6.8554371275744312442
6.8554371283914754542 6.8554371285957307336 6.8554371286467938873
6.8554371286595596757
6.8554371286627509008 6.855437128663548485 6.8554371286637483252
6.8554371286637980631
6.8554371286638104976 6.8554371286638140504

AdaptiveQuad output

6.8555244126654928394