Daniel Purcell
MATH 4670
November 22, 2005

1. Problem Statement

Find the value of

$$\int_{-1}^{3} e^x \sin^2 x \, dx$$

using a) the composite trapezoidal rule and b) the composite Simpson's rule to within 10^{-10}. Which method required more evaluations of $e^x \sin^2 x$?

Note: According to my calculator, the "exact" value is 8.97050567528

1.a. Composite Trapezoidal Rule

$$\int_{-1}^{3} e^x \sin^2 x \, dx \approx \frac{h}{2}\left[f(-1) + f(3) + 2\sum_{j=1}^{n-1} f(x_j)\right],$$

C Code

```
#include<stdlib.h>
#include<math.h>

double f(double x);
void trapezoid(double a, double b);
void output(int n, double val, double exact);

int main()
{
   double a, b;
   a = -1;
   b = 3;
   trapezoid(a, b);
   return 0;
}

double f(double x)
{
   return exp(x) * pow(sin(x), 2);
}

void trapezoid(double a, double b)
{
```

```
   int n = 1000000;
   double sum, h, exact;
   sum = 0.0;
   int i;
   exact = 8.97050567528;
   while (exact - sum > 0.0000000001)
   {
      h = (b - a) / (n + 2);
      sum = 0.0;
      for (i = 1; i < n - 1; i++)
         sum += f(a + (i + 1) * h);
      sum = (h/2) * ( f(a) + f(b) + 2 * sum );
      output(n, sum, exact);
      n += 1000000;

   }
}

void output(int n, double val, double exact)
{
   printf("%d \t %3.11f \t %3.11f \n", n, val, exact - val);
}
```

Results

n	approximation	error
10000000	8.97050525109	0.00000042419
20000000	8.97050546318	0.00000021210
30000000	8.97050553388	0.00000014140
40000000	8.97050556923	0.00000010605
50000000	8.97050559044	0.00000008484
60000000	8.97050560458	0.00000007070
70000000	8.97050561468	0.00000006060
80000000	8.97050562226	0.00000005302
90000000	8.97050562815	0.00000004713
100000000	8.97050563286	0.00000004242

Obviously, the use of the composite trapezoidal rule is rather painful for this problem, as it still did not produce results accurate to within 10^{-10} even after 100,000,000 iterations. It appears to eventually converge, but I do not have a spare decade or two to confirm that.

1.b. Composite Simpson's Rule

$$\int_{-1}^{3} e^x \sin^2 x \, dx \approx \frac{h}{3}\left[f(-1) + 2\sum_{j=1}^{(n/2)-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(3)\right],$$

C Code

```
#include<stdlib.h>
#include<math.h>

double f(double x);
void simpson(double a, double b);
void output(int n, double val, double exact);
double max(double x, double y);
double min(double x, double y);

int main()
{
   double a, b;
   a = -1;
   b = 3;
   simpson(a, b);
   return 0;
}

double f(double x)
{
   return exp(x) * pow(sin(x), 2);
}

double max(double x, double y)
{
   return (x >= y) ? x : y;
}

double min(double x, double y)
{
   return (x <= y) ? x : y;
}

void simpson(double a, double b)
{
   int n = 10;
   double sum, sum_one, sum_two, h, exact;
   sum = 0.0;
```

```
   int i;
   exact = 8.97050567528;
   while (!(max(exact, sum) - min(exact, sum) < 0.00000000009
      && max(exact, sum) - min(exact, sum) >= 0))
   {
      h = (b - a) / n;
      sum = sum_one = sum_two = 0.0;
      for (i = 1; i <= ( (n/2) - 1) ; i++)
         sum_one += f(a + (2 * i) * h);
      sum_one *= 2;
      for (i = 1; i <= (n/2); i++)
         sum_two += f(a + (2 * i - 1) * h);
      sum_two *= 4;
      sum = (h/3) * (f(a) + sum_one + sum_two + f(b));
      output(n, sum, exact);
      n += 10;
   }
}

void output(int n, double val, double exact)
{
   printf("%d \t %3.11f \t %3.11f \n", n, val, exact - val);
}
```

Results

```
  n      approximation        error
---------------------------------------
1080     8.97050567541   -0.00000000013
1090     8.97050567541   -0.00000000013
1100     8.97050567540   -0.00000000012
1110     8.97050567540   -0.00000000012
1120     8.97050567539   -0.00000000011
1130     8.97050567539   -0.00000000011
1140     8.97050567539   -0.00000000011
1150     8.97050567538   -0.00000000010
1160     8.97050567538   -0.00000000010
1170     8.97050567538   -0.00000000010
1180     8.97050567537   -0.00000000009
1190     8.97050567537   -0.00000000009
1200     8.97050567537   -0.00000000009
```

The composite Simpson's rule only takes around 1180 or so iterations, thus it is more efficient to use Simpson's rule to solve this problem.

2. Problem Statement

Redo the integral from Problem One using Gaussian quadrature with $n = 1$, 2, 3, 4, 5.

Change of Variable

$$t = \frac{1}{2}(x+1) - 1$$

$$x = 2t + 1$$

$$dx = 2\,dt$$

$$\int_{-1}^{3} e^x \sin^2 x\,dx = \int_{-1}^{1} e^{2t+1} \sin^2(2t+1)\,2\,dt$$

C Code

```
#include <stdlib.h>
#include <math.h>

double gauss(int n);
double f(double x);

int main()
{
   int n;
   for (n = 2; n <= 5; n++)
       printf("%d \t %3.11f \n", n, gauss(n));
   return 0;
}

double gauss(int n)
{
   if (n == 2)
   {
     return f(0.5773502692) + f(-0.5773502692);
   }
   else if (n == 3)
   {
     return 0.5555555556 * f(0.7745966692) + 0.8888888889 * f(0.0000000000) +
            0.5555555556 * f(-0.7745966692);
   }
   else if (n == 4)
   {
     return 0.3478548451 * f(0.8611363116) + 0.6521451549 * f(0.3399810436) +
```

```
            0.6521451549 * f(-0.3399810436) + 0.3478548451 * f(-0.8611363116);
   }
   else if (n == 5)
   {
     return 0.2369268850 * f(0.9061798459) + 0.4786286705 * f(0.5384693101) +
            0.5688888889 * f(0.0000000000) + 0.4786286705 * f(-0.5384693101) +
            0.2369268850 * f(-0.9061798459);
   }
   else
     return 0.0;
}

double f(double x)
{
   return 2 * exp(2 * x + 1) * pow(sin(2 * x + 1), 2);
}
```

Results

```
n          approximation
------------------------
2         12.04855981792
3          8.02936122899
4          8.99497551338
5          8.97416372329
```

Gaussian quadrature appears to reach accurate results faster than the previous two methods. However, while the computer has to do less work, the programmer must do much more as n increases.

3. Problem Statement

Find the arc length along the curve $y = \frac{1}{x}$ from the point where $x = \frac{1}{4}$ to the point where $x = 7$.

Arc Length Equation

$$s = \int_a^b \sqrt{1 + (f'(x))^2}\, dx$$

$$s = \int_{1/4}^7 \sqrt{1 + (-x^2)^2}\, dx$$

C Code

I have decided to use a modification of the Simpson's rule program that I used for Problem 1, Part b.

```
#include<stdlib.h>
#include<math.h>

double f(double x);
void simpson(double a, double b);
void output(int n, double val);

int main()
{
   double a, b;
   a = (1.0/4.0);
   b = 7;
   simpson(a, b);
   return 0;
}

double f(double x)
{
   return sqrt(1 + pow(-pow(x, 2), 2));
}

void simpson(double a, double b)
{
   int n = 10;
   double sum, sum_one, sum_two, h, previous;
   sum = 0.0;
   previous = -1.0;
   int i;
   while (sum != previous)
   {
      previous = sum;
      h = (b - a) / n;
      sum = sum_one = sum_two = 0.0;
      for (i = 1; i <= ( (n/2) - 1) ; i++)
         sum_one += f(a + (2 * i) * h);
      sum_one *= 2;
      for (i = 1; i <= (n/2); i++)
         sum_two += f(a + (2 * i - 1) * h);
      sum_two *= 4;
      sum = (h/3) * (f(a) + sum_one + sum_two + f(b));
      output(n, sum);
      n += 10;
```

```
   }
}

void output(int n, double val)
{
   printf("%d \t %3.11f \n", n, val);
}
```

Results

```
 n         approximation
------------------------
10       115.23572951875
20       115.24764852813
30       115.24781785209
40       115.24784490226
50       115.24785292640
60       115.24785578154
 |             |
 |             |
1690     115.24785843073
1700     115.24785843073
1710     115.24785843073
1720     115.24785843073
1730     115.24785843073
```

The value converges to exactly 115.24785843073, according to my program. Using this integral in my calculator yields 115.247858431, which validates my solution, since the calculator rounds off, because it handles two less digits than my program.

4.a. Problem Statement

Find the exact value of

$$\int_1^4 \int_0^3 \cos x + 2xy \, dy \, dx$$

Solution

$$\int_1^4 \int_0^3 \cos x + 2xy \, dy \, dx =$$

$$\int_1^4 \left[\cos xy + xy^2\right]\Big|_0^3 \, dx =$$

$$\int_1^4 3\cos x + 9x \, dx =$$

$$\left[3\sin x + \frac{9}{2}x^2\right]\Bigg|_1^4 =$$

$$[3\sin 4 + 72] - \left[3\sin 1 + \frac{9}{2}\right] \approx 62.7051795597$$

4.b. Problem Statement

Find the value of the integral to within 10^{-5} by using the two dimensional version of the trapezoidal rule. Use $n = m = 6$, $n = m = 12$, $n = m = 24$, and $n = m = 48$.

C Code

```
#include <stdlib.h>
#include <math.h>

double f(double x, double y);
double trapezoid2d(int n, int m, double a, double b, double c, double d);


int main()
{
   int n, m;
   double a, b, c, d;
   a = 1;
   b = 4;
   c = 0;
   d = 3;
   for (m = n = 6; n <= 48; m = n = n * 2)
      printf("%d \t %d \t %3.15f \n", n, m, trapezoid2d(n, m, a, b, c, d));
   return 0;
}

double f(double x, double y)
{
   return cos(x) + 2 * x * y;
}

double trapezoid2d(int n, int m, double a, double b, double c, double d)
{
   double h, k, sum[3], i, j, xi, yj;
   h = (b - a) / n;
   k = (d - c) / m;
   sum[0] = sum[1] = sum[2] = 0.0;
```

```
   for (i = 1; i < n; i++)
   {
      xi = a + (double) i * h;
      sum[0] = sum[0] + f(xi, c) + f(xi, d);
   }

   for (j = 1; j < n; j++)
   {
      yj = c + (double) j * k;
      sum[1] = sum[1] + f(a, yj) + f(b, yj);
   }

   for (i = 1; i < n; i++)
   {
      for (j = 1; j < n; j++)
      {
         xi = a + (double) i * h;
         yj = c + (double) j * k;
         sum[2] = sum[2] + f(xi, yj);
      }
   }

   return (.25) * h * k * (2 * sum[0] + 2 * sum[1] + 4 * sum[2] + f(a, c)
          + f(a, d) + f(b, c) + f(b, d) );
}
```

Results

```
n        m          approximation
-----------------------------------
6        6       62.805490362276984
12       12      62.730178635116282
24       24      62.711424441887445
48       48      62.706740475223107
```

5. Problem Statement

Using the answers from Problem 4.b., carry out the Romberg process. Does it result in a greatly imrpoved approximation for the integral, or does it fail work?

C Code

```
#include <stdlib.h>
#include <math.h>
```

```
int main()
{
   int i, j;
   double romberg[4][4];
   romberg[0][0] = 62.805490362276984;
   romberg[1][0] = 62.730178635116282;
   romberg[2][0] = 62.711424441887445;
   romberg[3][0] = 62.706740475223107;

   for (j = 1; j < 4; j++)
       for (i = 1; i < 4; i++)
       {
          romberg[i][j] = (pow(4.0, j) * romberg[i][j-1] -
                           romberg[i-1][j-1]) / (pow(4.0, j) - 1.0);
       }

   for (i = 0; i < 4; i++)
       for (j = 0; j <= 1; j++)
           printf("%d \t %d \t %3.12f \n", i, j, romberg[i][j]);

   return 0;
}
```

Results

```
i        j        approximation
--------------------------------
0        0       62.805490362277
0        1       0.000000000000
1        0       62.730178635116
1        1       62.705074726063
2        0       62.711424441887
2        1       62.705173044144
3        0       62.706740475223
3        1       62.705179153002
```

The Romberg process appears to work correctly.

6. Problem Statement

For

$$V(a,b,c) = \int_{-1}^{1}\int_{-1}^{1} \frac{1}{\sqrt{(x-a)^2+(y-b)^2+c^2}}\,dy\,dx,$$

Find V(5, 7, 3).

C Code

I have decided to use a modification of the Gaussian quadrature program I used for Problem 2.

```
#include <stdlib.h>
#include <math.h>

double gauss(int n);
double f(double x);

int main()
{
   int n;
   for (n = 2; n <= 5; n++)
      printf("%d \t %3.11f \n", n, gauss(n));
   return 0;
}

double gauss(int n)
{
   if (n == 2)
   {
     return f(0.5773502692) + f(-0.5773502692);
   }
   else if (n == 3)
   {
     return 0.5555555556 * f(0.7745966692) + 0.8888888889 * f(0.0000000000) +
    0.5555555556 * f(-0.7745966692);
   }
   else if (n == 4)
   {
     return 0.3478548451 * f(0.8611363116) + 0.6521451549 * f(0.3399810436) +
    0.6521451549 * f(-0.3399810436) + 0.3478548451 * f(-0.8611363116);
   }
   else if (n == 5)
   {
     return 0.2369268850 * f(0.9061798459) + 0.4786286705 * f(0.5384693101) +
    0.5688888889 * f(0.0000000000) + 0.4786286705 * f(-0.5384693101) +
    0.2369268850 * f(-0.9061798459);
   }
   else
     return 0.0;
}

double f(double x)
{
```

```
   return 1.0 / ( sqrt(pow(x - 5.0, 2) + pow(x - 7.0, 2) + pow(3.0, 2) ) );
}
```

Results

```
n          approximation
------------------------
2          0.22094713416
3          0.22095182309
4          0.22095182155
5          0.22095182140
```

I assume that $V(5, 7, 3) \approx 0.22095182140$.