AMS 326 | FALL 2012 | STONY BROOK UNIVERSITY

Homework 2

Dinia Grace Gepte

10/2/2012

Problem 1

(1) Write a program for using two different methods to compute the following integral to achieve five digits of accuracy

$$\int_0^{\pi/4} \left(\frac{\sin(x^3)}{x^3} + \exp\left(-\frac{\sin x}{x}\right)\right)^3 dx$$

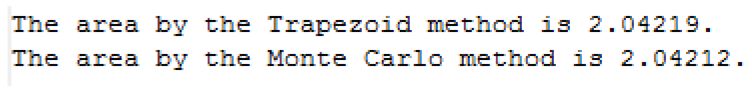
(2) State your reasons to use such two methods and describe their convergence properties.

(3) Estimate the number of operations for each of the two methods for achieving such results.

- The source code of the program is in `Integral.java` which is included in the zip file. A copy of the code is provided below:

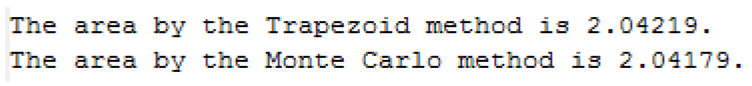
```
public class Integral
{
        // CONSTANTS FOR THIS FUNCTION
        public static final int N = 10000;    // ARBITRARILY CHOSEN
        public static final double A = 0;
        public static final double B = Math.PI / 4;
        public static final double DELTA = (B - A) / N;

        public static void main(String[] args)
        {
                // Format class to display area up to 5 digits of accuracy
                DecimalFormat df = new DecimalFormat("#.#####");

                // PRINT THE AREA USING THE TRAPEZOIDAL METHOD
                System.out.println("The area by the Trapezoid method is " +
                               df.format(trapezoid()) + ".");

                // PRINT THE AREA USING THE MONTE CARLO METHOD
                System.out.println("The area by the Monte Carlo method is " +
                               df.format(monteCarlo()) + ".");
        }

        public static double function(double x)
        {
                // FOR THIS FUNCTION, WHEN X = 0 THERE IS A SPECIAL VALUE FOR IT.
                // IF THIS IS NOT CHECKED, THE FUNCTION WILL EVALUATE TO NaN,
                // A VALUE PRODUCED WHEN THE DENOMINATOR IS 0.
                if (x == 0)
                        return (Math.pow(1 + Math.E, 3) / Math.pow(Math.E, 3));
                return Math.pow((Math.sin(Math.pow(x, 3))/Math.pow(x, 3)) +
                               (Math.exp(-(Math.sin(x)/x))), 3);
        }

        public static double trapezoid()
        {
                double area = 0;
                for (int i = 0; i < N; i++)
                        area += 0.5 * (function(getX(i)) + function(getX(i+1))) * DELTA;
                return area;
        }

        public static double getX(int n)
        {
                return A + n*DELTA;
        }

        public static double monteCarlo()
        {
                double area = 0;
                for (int i = 0; i < N; i++ )
                        area += function(randomNumberBetweenAandB()) * DELTA;
                return area;
        }

        public static double randomNumberBetweenAandB()
        {
                return Math.random() * (B-A) + A;      // THIS WILL NEVER GENERATE B
        }
}
```

- The program will produce the following output:



```
The area by the Trapezoid method is 2.04219.
The area by the Monte Carlo method is 2.04212.
```

Figure 1. Run 1



```
The area by the Trapezoid method is 2.04219.
The area by the Monte Carlo method is 2.04179.
```

Figure 2. Run 2

- As shown in Fig. 1 and 2, running the same program will output different results by the Monte Carlo Method while the trapezoid method always outputs the same answer. This is because the program fires N random numbers between A and B and evaluates the function at those points. Since they're random, it is possible that the first run's set of points is populated more to the left than the second run's set of points, or vice versa. These two methods were chosen because they have different concepts of evaluating the integral. The trapezoidal method relies on equally-even step x values while the Monte Carlo method relies on randomness.
- Generating a random number between A (inclusive) and B (exclusive) poses a minor problem in which the upper bound of the integral, i.e. B, will never be generated. However, since the N number is not too big, it is highly unlikely that the upper bound will be generated. I could have included the upper bound in the random generator but it will require more operations than the already convenient method `Math.random()` available.
- In this case, both methods have a complexity of O(N). In the trapezoid method, x_i is determined by the number of steps (N) it has from A to B, and in each step the function is evaluated. The Monte Carlo is the same thing but instead of determining equally spaced steps, N random numbers are generated and evaluated to the function.
- Due to the length of the method descriptions, they are omitted in the sample code above but are in `Integral.java`.

Problem 2

For the following given function with given domain in which the function is defined:

$$u(x,y) = (1-\cos\pi x)[1.23456+\cos(1.06512*y)]^2\, e^{\{-x^2-y^2\}}$$

where $x \in [-1,1],\ y \in [-2,2]$.

Please do the following:

(1) Divide the domain into 10x20 equally spaced mesh points and evaluate the function at these points and plot the function values by any tool that can make 2D plots;

(2) Among the 200 values you found above, please find two highest values and two lowest values. Identify the four of the 200 mesh points that have these four values;

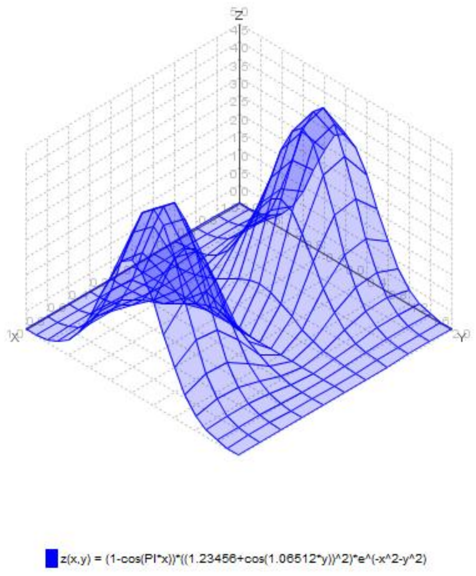
(3) Compute the L2 norms of the derivatives (gradients) $\left\|\left(\frac{\partial u}{\partial x},\frac{\partial u}{\partial y}\right)\right\|_2$ at the above special mesh points identified above.

- The source code of the program is in `Plot.java` which is included in the zip file. A copy of the main class is provided below:

```java
public class Plot
{
        // DELTA VARIABLE USED TO FIND THE PARTIAL DERIVATIVES
        private static final double DELTA = 0.0001;

        // VARIABLES THAT STORE THE x,y,z coordinates and corresponding gradient ACCDG TO
THE z-value
        //      double[0]: x-coord
        //      double[1]: y-coord
        //      double[2]: z-coord
        //      double[3]: grad(z)
        private static double[] highest = new double[4];
        private static double[] secondHighest = new double[4];
        private static double[] lowest = new double[4];
        private static double[] secondLowest = new double[4];

        public static void main(String[] args)
        {
                // PLOT DATA FOR X AND Y. DOMAIN IS A 10x20 GRID.
                double[] x = increment(-1.0, 0.2, 1.2);//INCREMENTS X = [-1.0, 1.2) BY 0.2
                double[] y = increment(-2.0, 0.2, 2.2);//INCREMENTS Y = [-2.0, 2.2) BY 0.2

                // THE increment METHOD OF THE DoubleArray CLASS GIVES INCREMENTED
                // DOUBLE VALUES THAT AREN'T ENTIRELY ACCURATE DUE TO THE FORMULA USED.
                // THIS FIXES THAT FOR THIS PROGRAM WHERE WE ONLY NEED 1 DECIMAL PLACE.
                DecimalFormat df = new DecimalFormat("#.#");
                for (int i = 0; i < x.length; i++)
                {
                        x[i] = Double.valueOf(df.format(x[i]));
                        y[i] = Double.valueOf(df.format(y[i]));
                }

                // PLOT DATA FOR z(x,y)
                double[][] z = function(x, y);


                // PANEL THAT SHOWS THE PLOT WITH A LEGEND AT THE BOTTOM
                Plot3DPanel plot = new Plot3DPanel("SOUTH");

                // ADD THE VALUES OF X, Y, Z TO THE PANEL WITH LEGEND
                plot.addGridPlot("z(x,y) = (1-cos(PI*x))*((1.23456+cos(1.06512*y))^2)*e^(-
                        x^2-y^2)", x, y, z);

                // PUT THE PANEL INTO A JFrame
                JFrame frame = new JFrame("A Plot");
                frame.setSize(600, 600);
                frame.setContentPane(plot);
                frame.setVisible(true);

                // FIND THE GRADIENTS
                highest[3] = gradient(partialX(highest[0], highest[1]),
                        partialY(highest[0], highest[1]));
        secondHighest[3] = gradient(partialX(secondHighest[0], secondHighest[0]),
                        partialY(secondHighest[0], secondHighest[0]));
        lowest[3] = gradient(partialX(lowest[0], lowest[1]),
                        partialY(lowest[0], lowest[1]));
        secondLowest[3] = gradient(partialX(secondLowest[0], secondLowest[1]),
                        partialY(secondLowest[0], secondLowest[1]));

        // PRINT A TABLE IN THE CONSOLE
        DecimalFormat out = new DecimalFormat("#.####");
        System.out.println("\t\tX\t|\tY\t|\tZ\t|\tGrad");
        System.out.println("Highest:\t" + highest[0] + "\t|\t" + highest[1] +
                        "\t|\t" + out.format(highest[2]) + "\t|\t" +
                        out.format(highest[3]));
        System.out.println("2nd Highest:\t" + secondHighest[0] + "\t|\t" +
                        secondHighest[1] + "\t|\t" + out.format(secondHighest[2]) +
                        "\t|\t" + out.format(secondHighest[3]));
        System.out.println("Lowest: \t" + lowest[0] + "\t|\t" + lowest[1] +
                        "\t|\t" + out.format(lowest[2]) + "\t|\t" +
                        out.format(lowest[3]));
        System.out.println("2nd Lowest:\t" + secondLowest[0] + "\t|\t" +
                        secondLowest[1] + "\t|\t" + out.format(secondLowest[2]) +
                        "\t|\t" + out.format(secondLowest[3]));
}

/*
 * Function definition: z(x,y) = (1-cos(PI*x))*((1.23456+cos(1.06512*y))^2)*e^(-
        x^2-y^2)
 */
public static double function(double x, double y)
{
        return (1-cos(PI * x)) * pow((1.23456 + cos(1.06512 * y)), 2) *
                        exp(-(pow(x, 2)) - (pow(y, 2)));
}

/*
 * Grid of the function. This also determines the highest and lowest 2 values.
 */
public static double[][] function(double[] x, double[]y)
{
        double[][] z = new double[y.length][x.length];
        for (int i = 0; i < x.length; i++)
                for (int j = 0; j < y.length; j++)
                {
                        // EVALUATE THE FUNCTION
                        z[j][i] = function(x[i], y[j]);

                        // MAKE THE FIRST VALUE OF THE TABLE TO BE THE HIGHEST &
                        // LOWEST
                        if (j == 0 && i == 0)
                        {
                                highest[0] = x[0];
                                highest[1] = y[0];
                                highest[2] = z[j][i];
                                lowest[0] = x[0];
                                lowest[1] = y[0];
                                lowest[2] = z[j][i];
                        }
                        // THERE IS A NEW HIGHEST
                        if (z[j][i] > highest[2])
                        {
                                // MAKE THE SECOND HIGHEST TAKE THE VALUES OF THE
                                // OLD HIGHEST
                                secondHighest[0] = highest[0];
                                secondHighest[1] = highest[1];
                                secondHighest[2] = highest[2];

                                // MAKE THE HIGHEST TAKE THE VALUES OF THE NEW
                                // HIGHEST
                                highest[0] = x[i];
                                highest[1] = y[j];
```

```
                                        highest[2] = z[j][i];
                                }
                                // THERE IS A NEW LOWEST
                                if (z[j][i] < lowest[2])
                                {
                                        secondLowest[0] = lowest[0];
                                        secondLowest[1] = lowest[1];
                                        secondLowest[2] = lowest[2];

                                        lowest[0] = x[i];
                                        lowest[1] = y[j];
                                        lowest[2] = z[j][i];
                                }
                        }

                return z;
        }

        public static double gradient(double partialX, double partialY)
        {
                return sqrt(pow(partialX, 2) + pow(partialY, 2));
        }

        public static double partialX(double x, double y)
        {
                return (function(x+DELTA, y) - function(x-DELTA, y)) / (2*DELTA);
        }

        public static double partialY(double x, double y)
        {
                return (function(x, y+DELTA) - function(x, y-DELTA)) / (2*DELTA);
        }
}
```

- When run, the program will output the following graph:



- The data table can also be found in the program when executed. A copy of it is in the file `data.txt`, included in the zip file.
- In the console, the program will output the following data about the two highest and lowest points:

```
                    X       |       Y       |       Z       |       Grad
Highest:            -0.8    |       0.0     |       4.763   |       2.7589
2nd Highest:        -0.8    |       -0.2    |       4.4841  |       4.5322
Lowest:             0.0     |       -2.0    |       0       |       0
2nd Lowest:         -0.2    |       -2.0    |       0.0017  |       0.0189
```

  where x, y, and z are the coordinates of the points and their corresponding gradient. The two highest points are actually {(-0.8, 0), (+0.8, 0)} but neither of them can be second highest.
- This program uses free source packages from http://jmathtools.berlios.de/doku.php, namely `jmatharray.jar` and `jmathplot.jar`, to plot the function in 3D and produce a list of coordinates. These packages are included in the zip file. It is said in the website that these packages are not high performance computing engines and plotters but because of the convenience to have the data points in the program to find the highest and lowest points, and gradients, I opted to plot the function in my program instead of doing it externally.
- Since it's only a 200-mesh point, the highest and lowest points need not be the local minima/maxima, so their corresponding gradients don't necessarily have to be 0.
- Due to the length of method descriptions, they are excluded in the sample code above but are in the zip file.