МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧЕРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет информационных технологий и программирования

Кафедра информационных систем

Проект

Аппроксимация плоских кривых методами динамического программирования

Выполнил студент
группы № M3303
Яцко Полина Олеговна

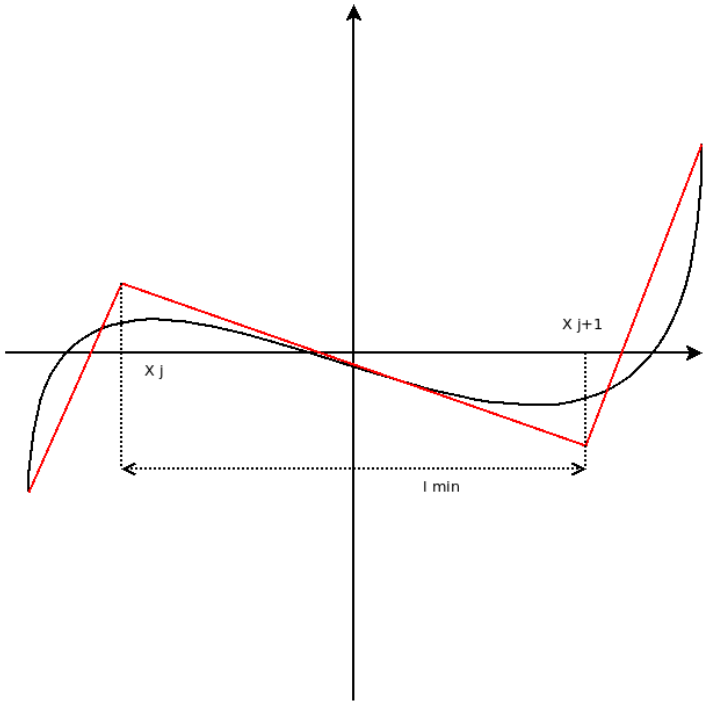
САНКТ-ПЕТЕРБУРГ
2018

Оглавление

Введение 3
Алгоритм решения 4
Реализация алгоритма на языке Java 7
Результаты работы алгоритма с разными критериями оценки выгодности пути 13
Список использованной литературы 19

Введение

Допустим, у нас есть некоторая плоская кривая. Аппроксимирующая ее кривая может быть ломаной линией, но точки ее изгиба должны быть расположены друг относительно друга не менее, чем на заданном расстоянии, чтобы при стремлении его к нулю изначальная кривая не была воспроизведена аппроксимирующей полностью. Другими словами, появляется ограничение на длины элементов аппроксимирующей ломаной (или на разность абсцисс точек изгиба $X_{j+1} - X_j > l_{min}$). Таким образом, речь в данной работе пойдёт о кусочно-линейной аппроксимации с ограничениями.



В качестве критерия близости исходной и аппроксимирующей кривых могут рассматриваться различные показатели, например, интеграл от модуля их разности или сумма квадратов отклонений в каких-то заранее заданных точках (квадратическое приближение), а также максимум модуля разности (равномерное приближение) и другие. Исследуем в данной работе приведенные выше показатели и их влияние на положение аппроксимирующих кривых.

Алгоритм решения

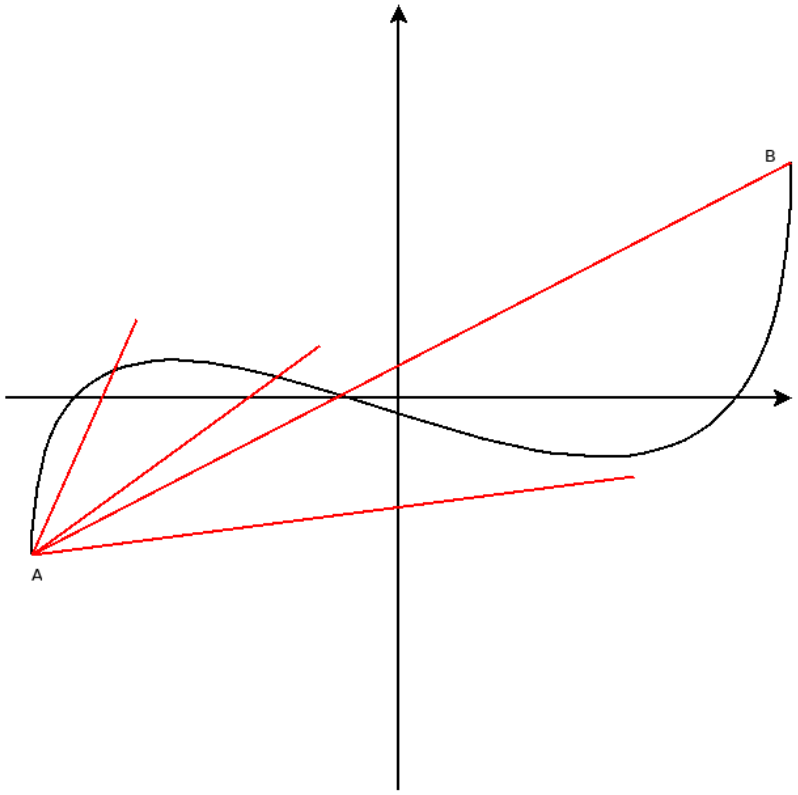
Решаем задачу при помощи динамического программирования.

Обозначим неизвестные пока абсциссы точек перелома как X_i, ординаты как Y_i, где i принадлежит множеству (1; 2; … n), где n — количество точек изгиба ломаной.

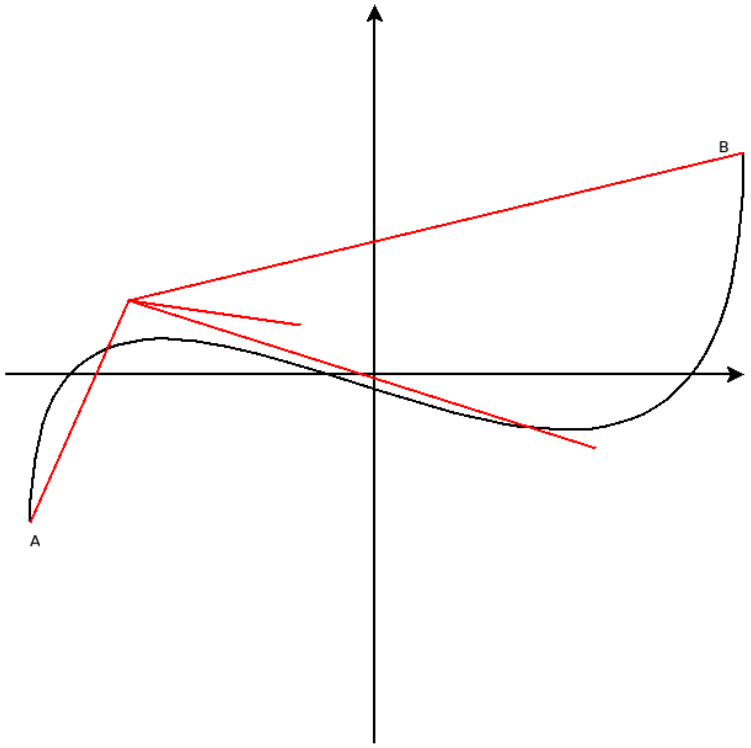
Ещё мы знаем начальную (A) и конечную (B) точки кривой. Мы будем стартовать из точки A и двигаться в процессе построения аппроксимирующей кривой к точке B.

Также введем l_{min} – минимальное расстояние между соседними абсциссами.

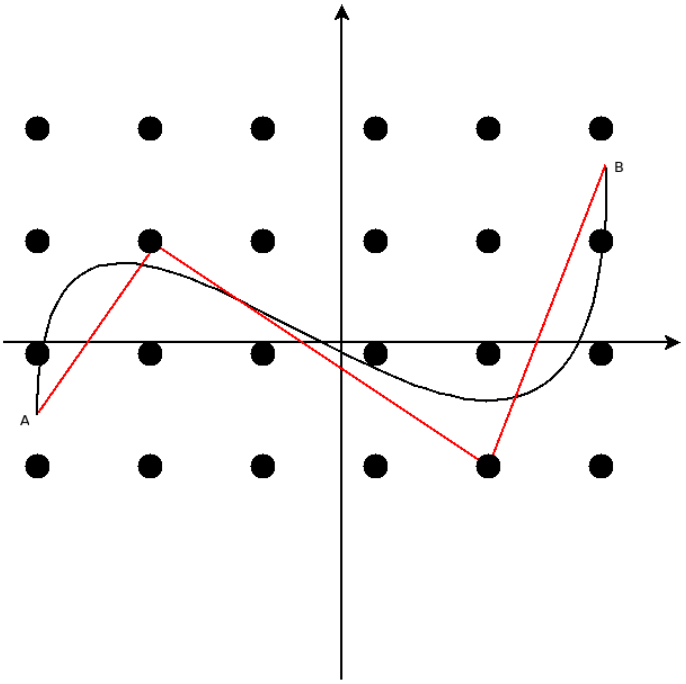
Из точки A мысленно проведем все возможные прямые, чьи абсциссы концов принадлежат $[X_A + l_{min}; X_B]$. Для каждой полученной конечной точки по выбранному критерию оценим эффективность проведения такого отрезка кривой.



Далее по очереди из каждой полученной точки построим аналогичный веер линий и посчитаем для каждой точки суммарную ее оценку по уже двум построенным кусочкам. При этом из всех допустимых путей, сходящихся в одной точке, оставляем один, самый оптимальный.



Конечно, рассчитать совершенно все точки пространства безумно долго, поэтому выбирается сетка из возможных точек, в которых могут находиться изгибы ломаной. Узлы сетки, допустим, можно задать через количество отрезков по горизонтали и вертикали между точками A и B. Тогда, по сути, l_{min} представляет из себя длину по оси абсцисс одной ячейки сетки.



На рисунке выше проиллюстрирована возможная сетка из точек. Строится она по следующему принципу: расстояние по оси абсцисс между точками A и B делится на нужное количество отрезков, таким образом, по своим координата абсцисс точки A и B «лежат в сетке». По оси ординат можно выбрать минимальное значение и максимальное значение — края сетки. Конечно, эти значения должны быть выбраны так, чтобы вся аппроксимируемая функция находилась внутри, и желательно даже с запасом. Потом этот интервал по оси ординат делится на нужное количество отрезков. При таком способе построения сетки ординаты A и B не обязаны «лежать на сетке», но это и не требуется, поскольку эти точки фиксированы.

Для точек сетки можно будет считать их по столбцам, то есть посчитать итоговые наивыгоднейшие пути к точкам первого столбца пользуясь только исходной точкой A, второго столбца — из первого и точки A и так далее. Такая возможность связана с тем, что прийти в точку можно только из более левой, вертикальные переходы запрещены по условию. Таким образом, каждая точка будет знать самый выгодный путь в себя основываясь на стоимости перехода в себя из предыдущей левой и информации левой точки о стоимости пути в ту. Важно, что к моменту пересчета стоимости правой части сетки информация в текущей точке будет окончательной.

Основным принципом, на котором базируется оптимизация многошагового процесса и особенности вычислительного метода динамического программирования, является принцип оптимальности Р. Беллмана.

Приведем его формулировку: оптимальное поведение обладает тем свойством, что каковы бы ни были начальное состояние и начальное решение, последующие решения должны составлять оптимальное поведение относительно состояния, полученного в результате начального решения.

Этот принцип можно сформулировать и по-другому: оптимальное поведение в многошаговом процессе обладает тем свойством, что какими бы ни было решение, принятое на последнем шаге, и состояние процесса перед последним шагом, предыдущие решения должны составлять оптимальное относительно этого состояния поведение.

Принцип оптимальности имеет конструктивный характер и непосредственно указывает процедуру нахождения оптимального решения. Математически он записывается выражением вида

$$f_{n-l}(S_l) = optimum_{U_{l+1}}[R_{l+1}(S_l, U_{l+1}) + f_{n-l-1}(S_{l+1})], \tag{2}$$

$l = 0, 1, ..., n-1$, где $U_l = (u_l^{(1)}; ...; u_l^{(m)})$ - решение (управление), выбранное на l-м шаге; $S_l = (s_l^{(1)}; \ldots; s_l^{(m)})$ – состояние системы на l-м шаге; R_l – непосредственный эффект, достигаемый на l-м шаге; f_{n-l} – оптимальное значение эффекта, достигаемого за $n \check{} l$ шагов; n - количество шагов (этапов). «Optimum» в выражении (2) означает максимум или минимум в зависимости от условия задачи. Формула (2) носит название уравнения Беллмана или рекуррентного соотношения. Процесс вычислени f_{n-l}, $l = 0, \ldots, n \check{} 1$, осуществляется при естественном начальном условии $f_0(S_n) = 0$, которое означает, что за пределами конечного состояния системы эффект равен нулю.

Реализация алгоритма на языке Java

Файл Main.java:

```
import java.util.ArrayList;
import java.util.Random;

public class Main {

    // -------- parameters -------------
    private static final Point a = new Point(0.0, f(0.0)); // left border point
    private static final Point b = new Point(Math.PI, f(Math.PI)); // right border
point

    private static final int xSectors = 60; // the amount of sectors for x coordi-
nate (approximation will be more accurate)
    private static final int ySectors = 60;

    // the step to find the maximum diff in the diffModes.MAX_DIFF
    private static final double STEP = 0.001;

    private static final int amountOfPointsForSquares = 1000;

    private static final double ymax = 1.5;
    private static final double ymin = 0;

    private static final diffModes mode = diffModes.MAX_DIFF;

    private static double f(double x) { // function to approximate
        return Math.sin(x);
    }

    private static double integral(double x) { // depends on f!!! it is F
        return -Math.cos(x);
    }

    // ----------parameters end -----------



    private static final double xSectorLength = Math.abs(b.getX() - a.getX()) /
xSectors;
    private static final double ySectorLength = Math.abs(ymax - ymin) / ySectors;

    private static Node[][] nodes;

    private enum diffModes {
        INTEGRAL, SUM_OF_DIFF_SQUARES, MAX_DIFF
    }

    private static final ArrayList <Double> pointsForSumOfDiffSquares = new Ar-
rayList<>();
    static {
        Random r = new Random();
        double half = 0.5 * (b.getX() - a.getX());
        for(int i = 0; i < amountOfPointsForSquares; i++) {
            pointsForSumOfDiffSquares.add(
            r.nextGaussian() /* the num from -1 to 1 */ * half /*make it from -
half to half*/
```

```
                    + (a.getX() + half) // the center between b and a
            );
        }
    }

    private static double integral(double x1, double x2) { // the area under the f
from x1 to x2
        return integral(x2) - integral(x1);
    }

    /*
     * get k coefficient in y = kx + b line by two points
     */
    private static double getK(Point p1,Point p2) {
        // line function:
        // y = ((x - x2)* (y2-y1)/(x2-x1)) + y2 = (y2-y1)/(x2-x1) * x - (y2-
y1)/(x2-x1) * x2 + y2
        return (p2.getY() - p1.getY()) / (p2.getX() - p1.getX());
    }

    /*
     * get b coefficient in y = kx + b line by two points
     */
    private static double getB(Point p1, Point p2) {
        // y = kx - kx2 + y2 = kx + y2 - kx2
       return p2.getY() - getK(p1, p2) * p2.getX();
    }

    /*
     * get result value of g (line by two points) of x
     */
    private static double g(double x, Point p1, Point p2) {
        return getK(p1, p2) * x + getB(p1, p2);
    }

    /*
     * the area under the line from p1 to p2
     */
    private static double intergralOfLine(Point p1, Point p2) {
        double k = getK(p1,p2);
        double b = getB(p1, p2);

        // y = kx + b
        // integral is (k * x * x /2.0 + b * x)

        return (k * p2.getX() * p2.getX() /2.0 + b * p2.getX())
                    - (k * p1.getX() * p1.getX() /2.0 + b * p1.getX());
    }

    private static double getMaxDiff(Point p1, Point p2) {
        double result = 0;
        for (double i = p1.getX(); i < p2.getX(); i += STEP) {
            double diffAtI = Math.abs(f(i) - g(i, p1, p2));
            result = Math.max(diffAtI, result);
        }
        return result;

    }
```

```
    private static double getSumOfDiffSquares(Point p1, Point p2) {
        double sum = 0;
        for(int i = 0; i < pointsForSumOfDiffSquares.size(); i++) {
            if(pointsForSumOfDiffSquares.get(i) > p1.getX() && pointsForSumOfD-
iffSquares.get(i) <= p2.getX()) {
                sum += Math.pow(
                        f(pointsForSumOfDiffSquares.get(i)) -
                                g(pointsForSumOfDiffSquares.get(i), p1, p2),
                        2
                );
            }
        }
        return sum;
    }

    /*
     * the value we want - the difference between the line and the function
     * (line from p1 to p2, function from x1 to x2)
     */
    private static double difference(Point p1, Point p2) {
        switch (mode) {
            case INTEGRAL:
                return Math.abs(integral(p1.getX(), p2.getX()) - inter-
gralOfLine(p1, p2));
            case SUM_OF_DIFF_SQUARES:
                return getSumOfDiffSquares(p1, p2);
            case MAX_DIFF:
            default:
                return getMaxDiff(p1, p2);
        }
    }

    /*
     * the approximate line cant be vertical, because there are no vertical func-
tions
     */
    private static Node[][] getNodesArray() {
        Node[][] arr = new Node [xSectors] [ySectors + 1];
        for(int i = 0; i <= xSectors - 2; i++) { // x
            for (int j = 0; j <= ySectors; j++) { // y
                arr [i][j] = new Node(
                        new Point(a.getX() + (i + 1) * xSectorLength, ymin + j *
ySectorLength),
                        Double.MAX_VALUE,
                        new Point()
                );
            }
        }
        arr[xSectors -1][0] = new Node(b, Double.MAX_VALUE, new Point()); // the
point b
        return arr;
    }

    private static int getXIndex(Point p) {
        return (int) ((p.getX() - a.getX()) / xSectorLength) - 1;
    }

    private static int getYIndex(Point p) {
        return (int) ((p.getY() - ymin) / ySectorLength);
```

```
    }

    private static void refreshNodesInfo(Node from) {
        int begXNumber = getXIndex(from.getCoordinate()) + 1;
        for (int i = begXNumber; i < xSectors; i++) {
            for (int j = 0; j <= ySectors; j++) {
                if (nodes[i][j] != null) {
                    double diff = difference(from.getCoordinate(),
nodes[i][j].getCoordinate());
                    double newPotentialCost = mode.equals(diffModes.MAX_DIFF) ?
                            Math.max(from.getCost(), diff) :
                            from.getCost() + diff;
                    if (newPotentialCost < nodes[i][j].getCost()) {
                        nodes[i][j].setCost(newPotentialCost);
                        nodes[i][j].setPrevious(from.getCoordinate());
                    }
                }
            }
        }
    }

    private static double round(double x) {
        return Math.round(x * 100) / 100.0;
    }

    private static void printThePathFromTheNode(Node current) {
        System.out.println(round(current.getCoordinate().getX()) + " " + round(cur-
rent.getCoordinate().getY()));
        int prevXIndex = getXIndex(current.getPrevious());
        int prevYIndex = getYIndex(current.getPrevious());
        if (prevXIndex != -1) {
            printThePathFromTheNode(nodes[prevXIndex][prevYIndex]);
        } else {
            System.out.println(round(a.getX()) + " " + round(a.getY()));
        }
        return;
    }

    public static void main (String args[]) {
        nodes = getNodesArray();

        // we start at the a
        refreshNodesInfo(new Node(a, 0, null));

        for (int i = 0; i < xSectors; i++) {
            for (int j = 0; j <= ySectors; j++ ) {
                if(nodes[i][j] != null)
                    refreshNodesInfo(nodes[i][j]);
            }
        }

        // print from the "b" node
        printThePathFromTheNode(nodes[xSectors - 1][0]);

       System.out.println("----------------------");
        // points to draw the function
        for (double i = a.getX(); i <= b.getX(); i+=0.05) {
            System.out.println(round(i) + " " + round(f(i)));
        }
```

```
    }
}
```

Файл Node.java (класс описывает узел сетки):

```
class Node {
    private Point coordinate; //coordinate of the node
    private double cost; //the cost of the path to the node from a
    private Point previous; // the previous point in the path

    Node (Point coordinate, double cost, Point previous) {
        this.coordinate = coordinate;
        this.cost = cost;
        this.previous = previous;
    }

    Point getCoordinate() {
        return coordinate;
    }

    double getCost() {
        return cost;
    }

    void setCost(double cost) {
        this.cost = cost;
    }

    Point getPrevious() {
        return previous;
    }

    void setPrevious(Point previous) {
        this.previous = previous;
    }
}
```

Файл Point.java (описывает точку на плоскости в декартовых координатах):

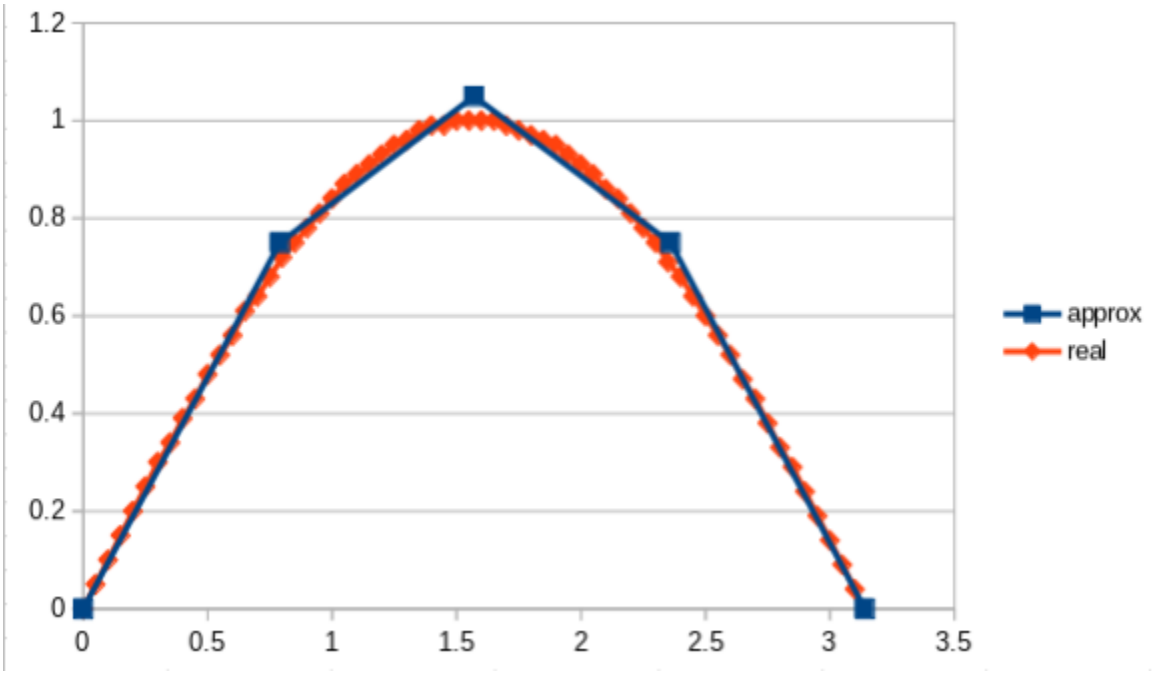
```
class Point {

    private double x;
    private double y;

    Point () {
        this.x = 0;
        this.y = 0;
    }

    Point (double x,double y) {
        this.x = x;
        this.y = y;
    }

    double getX() {
        return x;
    }
```

```
    double getY() {
        return y;
    }
}
```

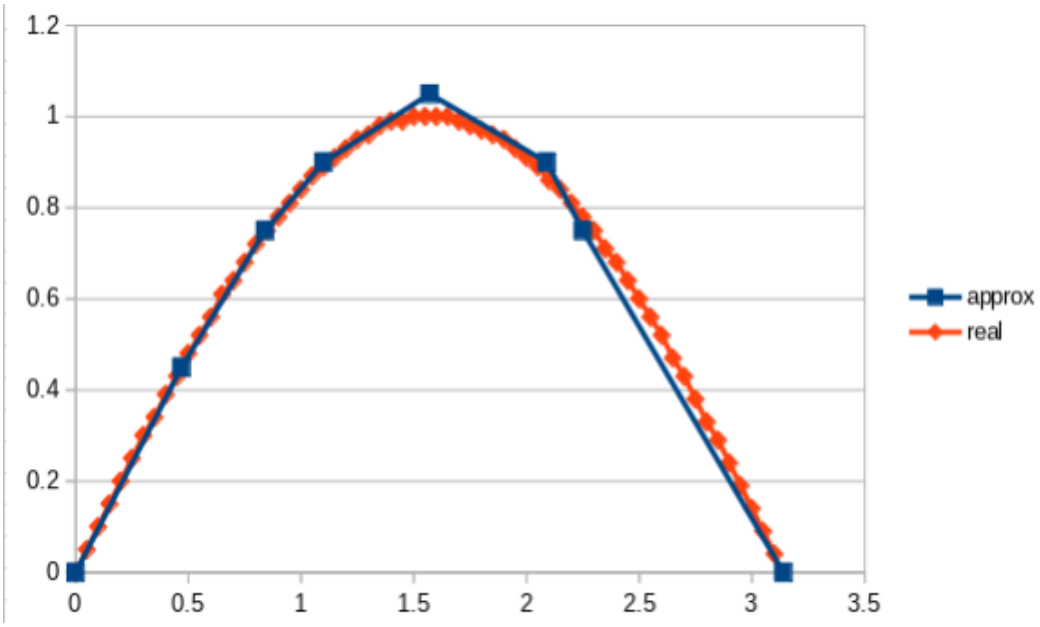
Результаты работы алгоритма с разными критериями оценки выгодности пути

Для начала сравнение пойдет на функции синуса от 0 до пи, с возможными границами по оси ординат 0 и 1.5. Я выбрала сетку 60 отрезков по оси абсцисс и 10 отрезков по оси ординат.

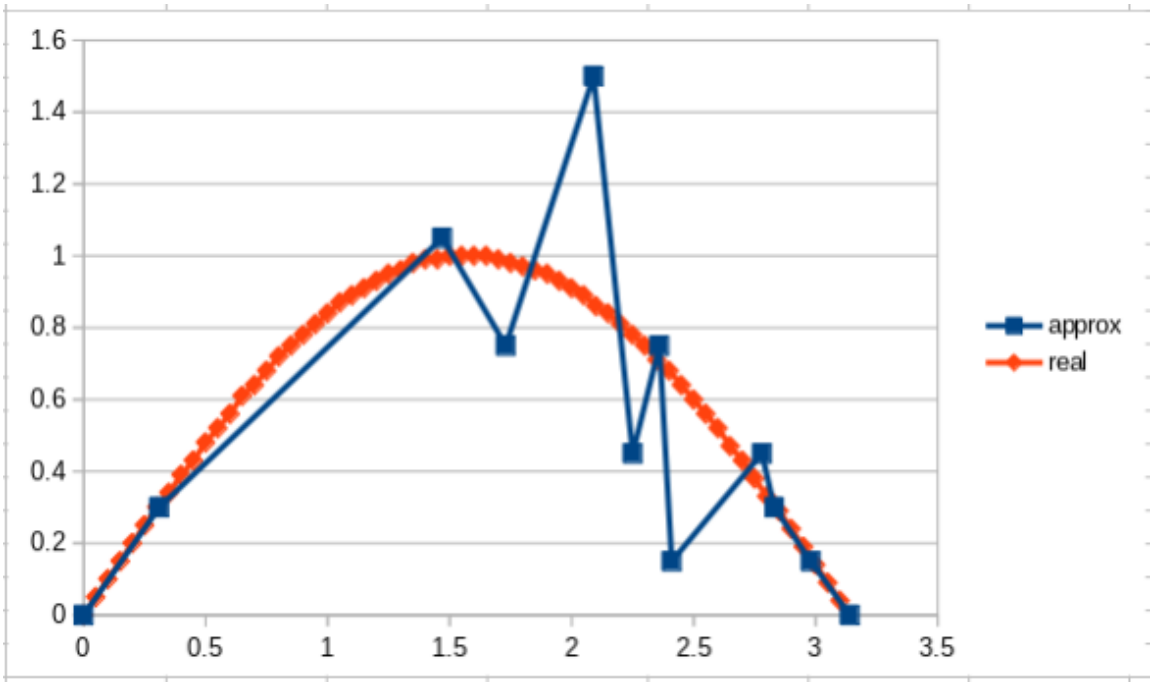
Такую картину выдало интегральное приближение:



Такую — построение точек с наименьшим максимальным отклонением на всей протяженности функции при тех же параметрах:

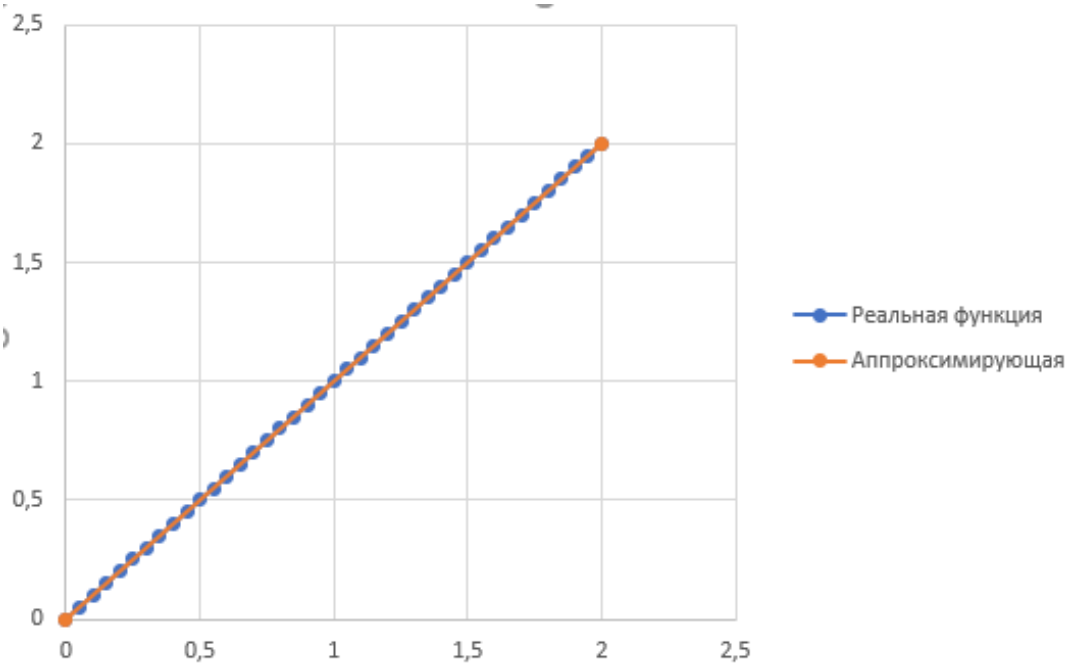


Запуск алгоритма с критерием, основывающимся на сумме квадратов разностей в точках, дал интересный результат. Я беру список точек, которые генерируются случайно и все лежат от A до B. Алгоритм работает так, чтобы в этих точках провести прямые по возможности через функцию, а на остальные не обращает особого внимания. Примерно так может выглядеть приближение при использовании 10 случайных точек при прочих равных условиях:

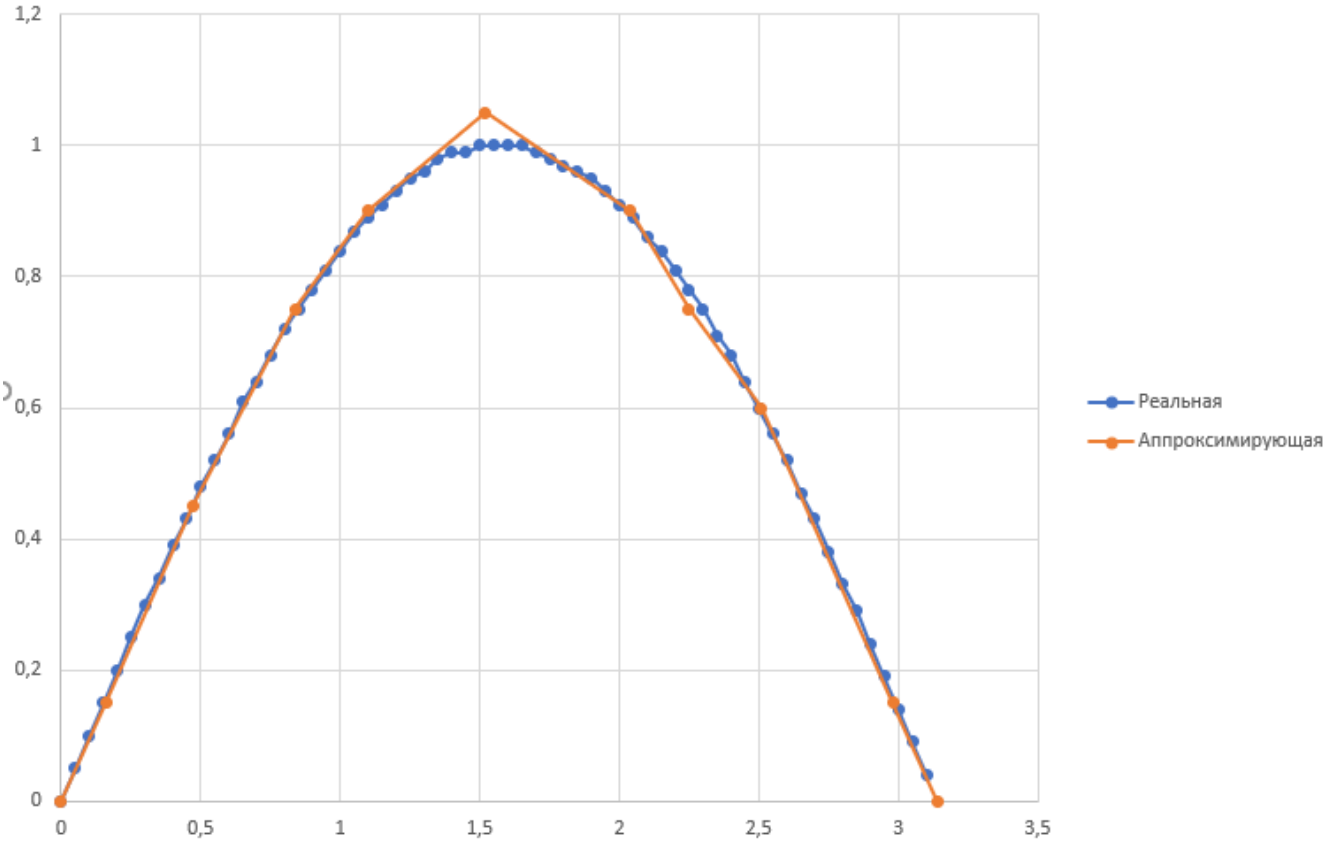


Как видно, уже на первый взгляд сумма квадратов отклонений в заданных точках ощутимо проигрывает остальным критериям. Оно и логично, метод наименьших квадратов хорош при приближении набора известных точек функцией известного вида, предположительно совпадающего с функцией источника точек. В данном случае учитываются 10 точек, все другие известные точки синусоиды игнорируются, а вид аппроксимирующей функции (кусочно — линейной) не совпадает с видом исходной функции.

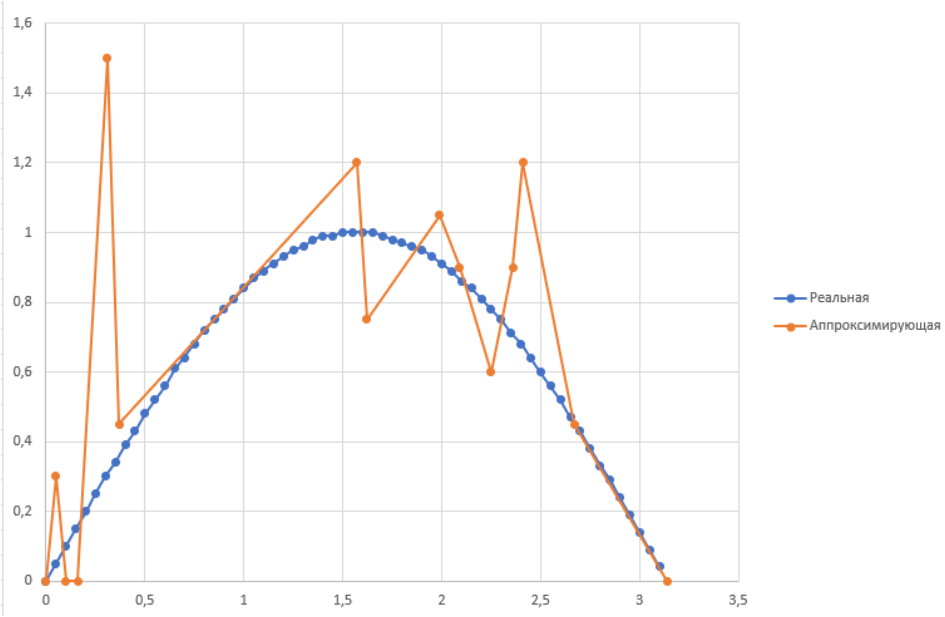
Такие результаты все же довольно любопытны. Если попробовать запустить построение аппроксимирующей ломаной с таким критерием приближения на прямой y = x, то получим полное совпадение (что в целом достаточно логично):



Попробуем поставить больше точек (1000) и «раскидать» их не случайным образом, а нормально. Получим следующую картину:

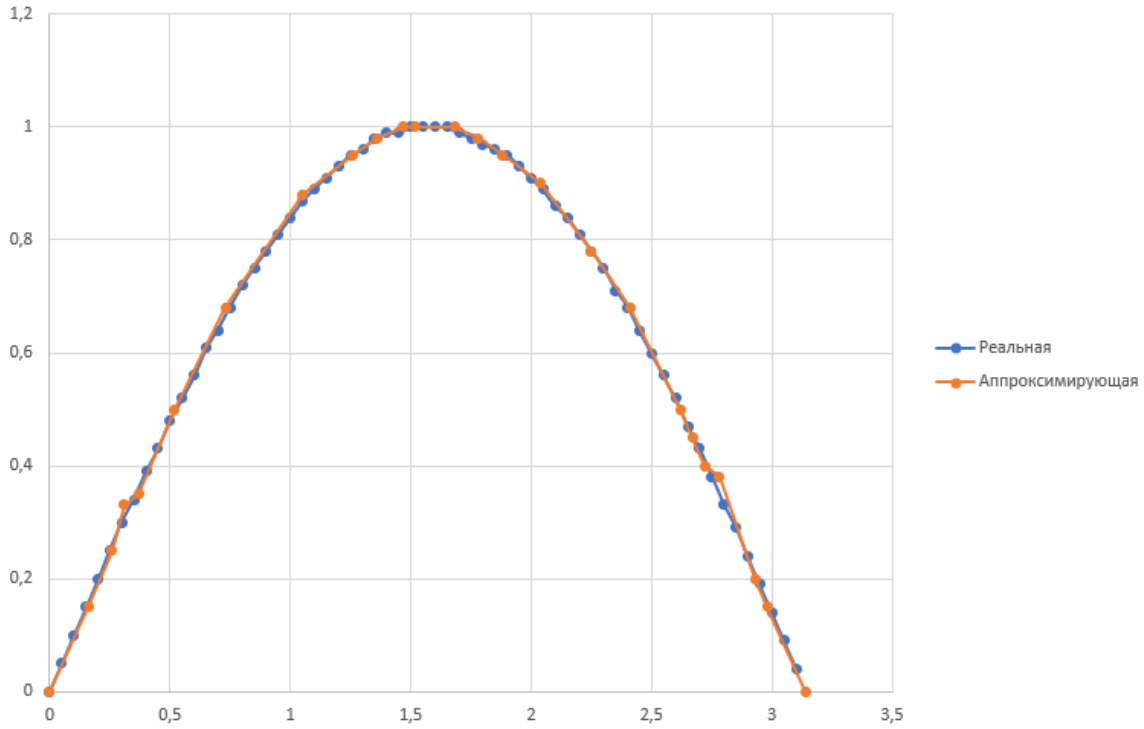


Этот алгоритм, получается, сильно зависит от количества точек. Выше был скриншот с десятью случайными точками, и отклонение было колоссальным. Если же эти десять точек поставить нормально, легче не становится:

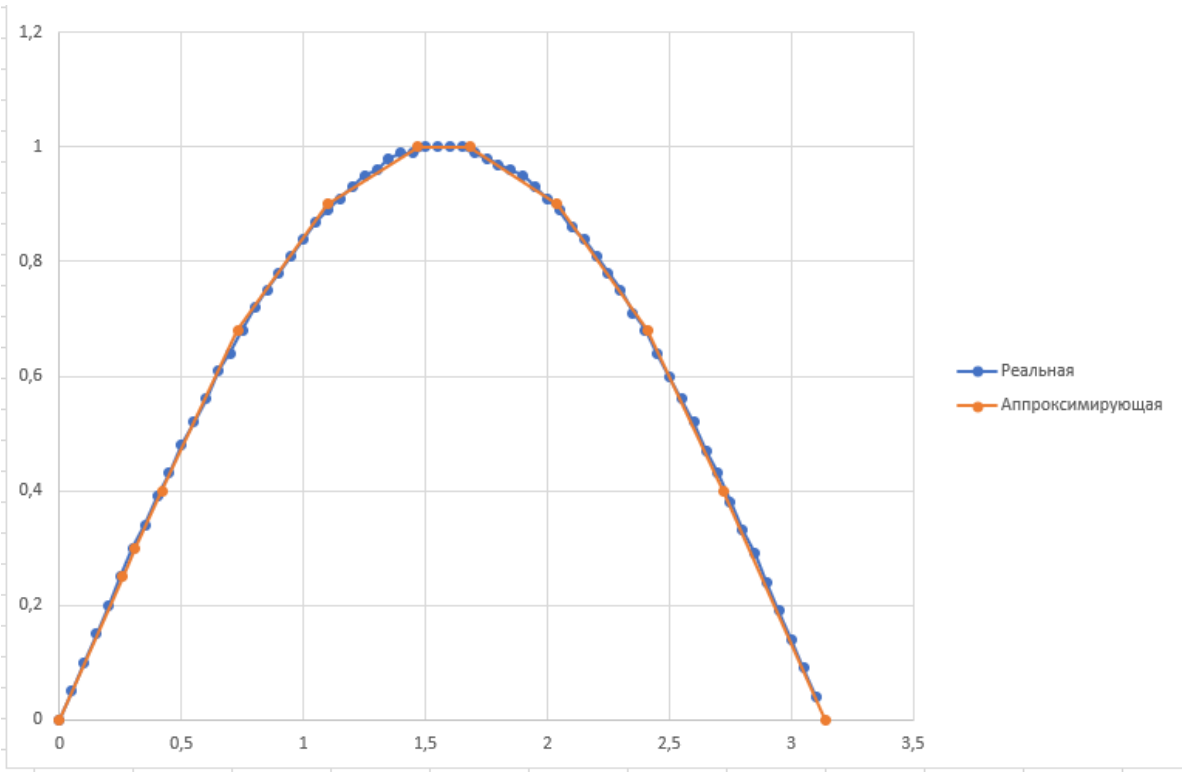


Как я уже писала, программа учитывает только приближение к функции в установленных точках, соответственно, если в случайных точках на каком-то участке есть «проплешина», то там наверняка ломаная будет сильно отличаться. А этих самых «проплешин» становится меньше с увеличением количества точек, что логично.

Еще более приближенный к исходной функции результат дает увеличение количества отрезков по оси ординат до 60 (сетка 60 х 60, 1000 точек, приближение методом наименьших квадратов отклонений):



При такой же сетке метод максимального отклонения (для полноты картины):



Как видно, все эти критерии оценки имеют право на существование, все они работают хорошо. Интегральный достаточно сложен в реализации (по сравнению в двумя другими) и требует больше знаний, но работает быстрее. Метод максимального отклонения самый простой, но сложно реализуется: в нем либо нужно с каким-то маленьким шагом «протыкивать» точки, либо решать в общем виде уравнение с производными (что я, к сожалению, сделать не смогла). Касательно метода наименьших квадратов некоторые рассуждения я уже привела в ходе работы – он сильно зависит от количества и расположения точек и, как по мне, более хорош для обратной задачи – проведения функции заданного вида по известным точкам. Однако и с такой задачей вполне себе справляется.

Список использованной литературы

- Струченков В. И. «Методы оптимизации в прикладных задачах»
- Беллман Р. Э. «Динамическое программирование»