Mandelbrot

Erik Bjäreholt, D13 (dat13ebj@student.lu.se)
Carl Ericsson, D13 (dat13cer@student.lu.se)

Inlämningsdatum: 26 november 2013

1 Bakgrund

Uppgiften är att beräkna mandelbrotsmängden som är en så kallad fraktal och rita upp den. En fraktal är en geometrisk figur vars mönster ser likadan ut när man zoomar in och ut i olika delar av den. För att illustrera detta ska vi använda oss av 3 olika klasser: `Mandelbrot`, `Complex`, och `Generator` där `Mandelbrot` innehåller `main`-metoden. Fönstret som ritar upp bilden ska vara av typen `MandelbrotGUI` som är en färdigskriven klass som ingår i kursen och har paketnamnet `se.lth.cs.ptdc.fractal.MandelbrotGUI`

Git-repository finnes här: `https://github.com/ErikBjare/Mandelbrot`

2 Modell

Programmet är som nämndes ovan uppbyggt av de tre klasserna `Mandelbrot`, `Complex` och `Generator`. Mandelbrot innehåller endast `main`-metoden som hanterar kommandon från MandelbrotGUI. Vissa av dessa kommandon körs sedan av `Generator`, bl.a. uträkning av mandelbrot-mängden och dess färgmatris (bild). Klassen `Complex` används för att representera komplexa tal.

Namn	*Beskrivning*
`Mandelbrot`	Huvudklass för programmet med `main`-metoden, instansierar `Complex` och `Generator`.
`Complex`	En klass som representerar ett komplext tal och innehålle en variabel för realdelen kallad re, och en för dess imaginärdel kallad im. Klassen innehåller även en del metoder för att behandla det komplexa talet. Används av `Generatorn`
`Generator`	En klass som räknar ut mandelbrotsmängden och ritar upp en bild efter mandelbrotspunkterna i fönstret `Mandelbrot GUI`

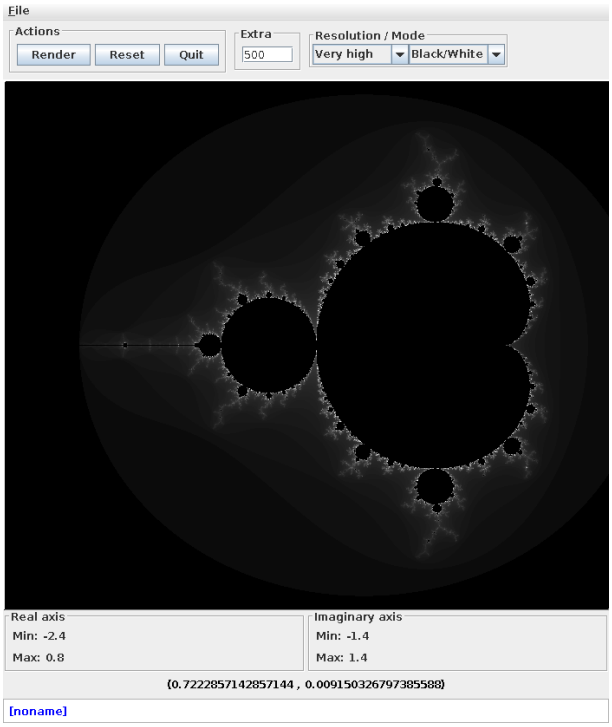
3 Brister och kommentarer

3.1 Förlust av kontrast

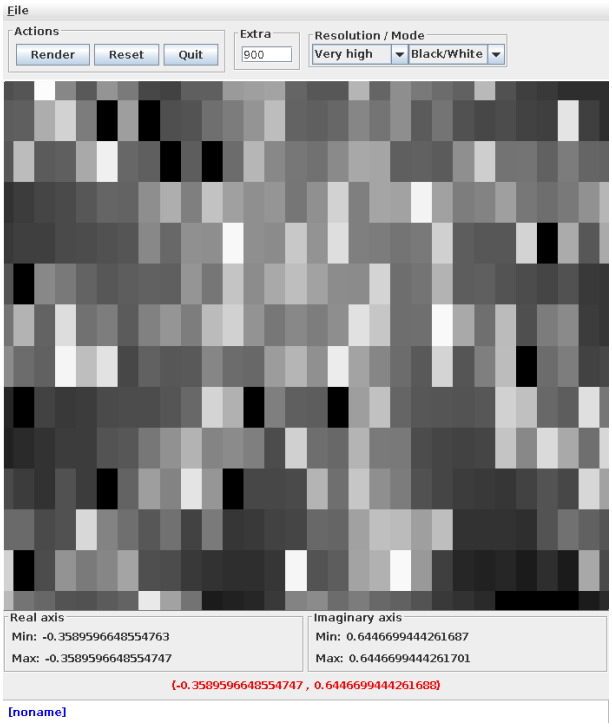
Vid inzoomning hade en enkel implementation som fungerade genom fixerad mappning av k till färger lett till ett mindre färgspann vid inzoomning. Detta löste vi genom att använda en metod likt den som användes i laboration 8 där vi hittade det största respektive minsta k i det aktuella området av vilket vi räknade ut ett intervall för vilket vi mappade alla tillgängliga färger.

3.2 Double's begränsningar

På grund av double's begränsningar så korrupteras bilden då minimum och maximum på axlarna närmar sig varandra så att de blir såpass lika att double inte klarar av att beräkna en korrekt mellanskillnad. Talintervallets precision minskar med antalet inzoomningar vilket syns tydligt i figur 2. Detta fenomen hade kunnat gå att åtgärda om man istället för double använder BigDecimal men detta testades inte då det ansågs opraktiskt prestandamässigt.



Figur 1: Beräkning av mandelbrotmängden med max-iterationer satt till 500



Figur 2: Exempel på hur precisionen av double inte räcker till

4 Programlistor

4.1 Mandelbrot

```
import se.lth.cs.ptdc.fractal.MandelbrotGUI;

public class Mandelbrot {

    public static void main(String[] args) {
        MandelbrotGUI w = new MandelbrotGUI();
        Generator gen = new Generator();

        boolean drawn = false;

        while (true) {
            int cmd = w.getCommand();
            switch (cmd) {
                case (MandelbrotGUI.RENDER):
                    gen.render(w);
                    drawn = true;
                    break;
                case (MandelbrotGUI.RESET):
                    w.resetPlane();
                    gen.reset();
                    drawn = false;
                    break;
                case (MandelbrotGUI.QUIT):
                    System.exit(0);
                    break;
                case (MandelbrotGUI.ZOOM):
                    if (drawn) gen.render(w);
                    break;
                default:
                    System.out.println(cmd);
                    break;
            }
        }
    }
}
```

4.2 Complex

```
public class Complex {
    double re;
    double im;

    public Complex (double re, double im) {
        this.re = re;
        this.im = im;
    }

    /**
     *
     * @return String-representation av objektets innehåll
```

```
 */
@Override
public String toString() {
    return re + " + " + im + " i";
}

/**
 *
 * @return Reella delen av det komplexa talet
 */
public double getRe() {
    return re;
}

/**
 *
 * @return Imaginära delen av det komplexa talet
 */
public double getIm() {
    return im;
}

/**
 *
 * @return Returnerar komplexa talets absolutbelopp i kvadrat
 */
public double getAbs2() {
    return Math.pow(re, 2) + Math.pow(im, 2);
}

/**
 *
 * @return Returnerar komplexa talets absolutbelopp
 */
public double getAbs() {
    return Math.sqrt(getAbs2());
}

/**
 * Adderar med det komplexa talet c
 * @param c Talet att addera med
 */
public void add(Complex c) {
    re += c.getRe();
    im += c.getIm();
}

/**
 * Multiplicerar med det komplexa talet c
 * @param c Talet att multiplicera med
 */
public void mul(Complex c) {
    // (a+bi)(c+di) = (ac-bd) + (bc+ad)i
    double newRe = (re*c.getRe() - im*c.getIm());
```

```
        double newIm = (im*c.getRe() + re*c.getIm());
        re = newRe;
        im = newIm;
    }
}
```

4.3 Generator

```
import se.lth.cs.ptdc.fractal.MandelbrotGUI;

import java.awt.*;

public class Generator {
    Color[] greyScale;
    Color[] colorScale;
    Color voidColor = new Color(0,0,0);
    int maxIterations = 200;
    int lowK;
    int highK;

    public Generator() {
        this.reset();

        this.greyScale = new Color[256];
        for (int i=0; i<255; i++) {
            greyScale[i] = new Color(i,i,i);
        }

        this.colorScale = new Color[256];
        int r = 0;
        int g = 0;
        int b = 255;
        int i = 0;
        while (r < 255) {
            this.colorScale[i] = new Color(r, g, b);
            r += 3;
            i++;
        }
        while (b > 0) {
            this.colorScale[i] = new Color(r, g, b);
            b -= 3;
            i++;
        }
        while (g < 81) {
            this.colorScale[i] = new Color(r, g, b);
            g += 1;
            i++;
        }
    }

    /**
     * Återställer vissa variabler
     */
    public void reset() {
```

```
    lowK = maxIterations;
    highK = 0;
}

/**
 * Ritar upp mandelbrot i en MandelbrotGUI ‘w‘
 * @param w
 */
public void render(MandelbrotGUI w) {
    w.disableInput();

    String extra = w.getExtraText();
    if (!extra.isEmpty())
        maxIterations = Integer.parseInt(w.getExtraText());

    lowK = maxIterations;
    highK = 0;

    double minRe = w.getMinimumReal(), maxRe = w.getMaximumReal();
    double minIm = w.getMinimumImag(), maxIm = w.getMaximumImag();
    int height = w.getHeight(), width = w.getWidth();
    int res = w.getResolution();

    int pix = 1;
    if      (res == MandelbrotGUI.RESOLUTION_VERY_HIGH) pix=1;
    else if (res == MandelbrotGUI.RESOLUTION_HIGH)      pix=3;
    else if (res == MandelbrotGUI.RESOLUTION_MEDIUM)    pix=5;
    else if (res == MandelbrotGUI.RESOLUTION_LOW)       pix=7;
    else if (res == MandelbrotGUI.RESOLUTION_VERY_LOW)  pix=9;

    Complex[][] complex = mesh(minRe, maxRe, minIm, maxIm, width, height);
    int[][] mandelK = getMandelK(complex, pix);

    boolean color = false;
    if (w.getMode() == MandelbrotGUI.MODE_COLOR) color = true;

    // Experimental
    //Color[][] colors = apply(mandelK);
    Color[][] colors = getImage(mandelK, color);
    w.putData(colors, pix, pix);

    maxIterations += lowK;

    w.enableInput();
}

/**
 * Returnerar en matris med k-värdena för varje complext tal i matrisen ‘complex‘
 * @param complex En matris med flera komplexa talplanet
 * @param res Precisionen som ska tas hänsyn till
 * @return Matris med k-värden
 */
private int[][] getMandelK(Complex[][] complex, int res) {
    int[][] mandelK = new int[complex.length/res][complex[0].length/res];
```

```
    for (int icol = 0; icol < complex[0].length / res; icol++)
        for (int irow = 0; irow < complex.length / res; irow++)
            mandelK[irow][icol] = getK(complex[irow * res + res / 2][icol * res + res / 2]);
    return mandelK;
}

/**
 * Returnerar en matris med färger från en matris med k-värden ‘mandelK‘
 * @param mandelK En matris med k-värden
 * @param color Specifierar om en färgskala ska användas, annars gråskala
 * @return Färgmatris baserad på k-värdena
 */
private Color[][] getImage(int[][] mandelK, boolean color) {
    Color[][] image = new Color[mandelK.length][mandelK[0].length];
    for (int icol = 0; icol < mandelK[0].length; icol++)
        for (int irow = 0; irow < mandelK.length; irow++)
            if (color) {
                image[irow][icol] = getColor(mandelK[irow][icol]);
            } else {
                image[irow][icol] = getGreyscale(mandelK[irow][icol]);
            }
    return image;
}

/**
 *
 * @param k k-värdet att hämta färg för
 * @return Punktens färg
 */
private Color getColor(double k) {
    if (k == maxIterations) {
        return voidColor;
    } else {
        int colorIndex = (int)( Math.pow((k-lowK)/(highK-lowK), 0.5) * 255);
        return colorScale[colorIndex];
    }
}

/**
 *
 * @param k k-värdet att hämta färg för
 * @return Punktens färg
 */
private Color getGreyscale(double k) {
    if (k == maxIterations) {
        return voidColor;
    } else {
        int colorIndex = (int)( Math.pow((k-lowK)/(highK-lowK), 0.5) * 255);
        return greyScale[colorIndex];
    }
}

/**
 *
```

```
 * @param c Räknar ut k-värde för något complext tal c
 * @return Punktens färg
 */
private int getK(Complex c) {
    Complex z = new Complex(0,0);
    z.add(c);
    for (int k=0; k<maxIterations; k++) {
        z.mul(z);
        z.add(c);
        if (z.getAbs() > 2) {
            if (k < lowK) {
                System.out.println("New lowK: " + k);
                lowK = k;
            } else if (k > highK) {
                System.out.println("New highK: " + k);
                highK = k;
            }
            return k;
        }
    }
    return maxIterations;
}

/**
 *
 * @param minRe Vart den reella axeln ska börja
 * @param maxRe Vart den reella axeln ska slute
 * @param minIm Vart den imaginära axeln ska börja
 * @param maxIm Vart den imaginära axeln ska sluta
 * @param width Bredd i pixlar
 * @param height Höjd i pixlar
 * @return Det komplexa talplanet
 */
private Complex[][] mesh(double minRe, double maxRe,
                         double minIm, double maxIm,
                         int width, int height) {
    Complex[][] plane = new Complex[height][width];

    double stepRe = (maxRe-minRe)/width;
    double stepIm = (maxIm-minIm)/height;

    for (int icol=0; icol<width; icol++) {
        double real = minRe+stepRe*icol;
        for (int irow=0; irow<height; irow++) {
            double imaginary = maxIm-stepIm*irow;
            plane[irow][icol] = new Complex(real, imaginary);
        }
    }

    return plane;
}


/**
```

```
 * Experimental, not used!
 *
 * @param k
 * @return
 */
public Color[][] apply(int[][] k) {
    //double cutOff = paramValue/100;
    double cutOff = 0.01;

    int height = k.length;
    int width = k[0].length;
    int pixels = height*width;

    int[] histogram = new int[256];

    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            histogram[k[i][j]] += 1;
        }
    }

    int[] cuts = computeCuts(histogram, cutOff, pixels);

    Color[][] outPixels = new Color[height][width];
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            int newk = 255 * (k[i][j] - cuts[0]) / (cuts[1] - cuts[0]);
            if (newk >= 0 && newk <= 255) {
                outPixels[i][j] = greyScale[newk];
            } else if (newk > 255) {
                outPixels[i][j] = voidColor;
            } else if (newk < 0) {
                outPixels[i][j] = greyScale[0];
            }
        }
    }

    return outPixels;
}

/**
 * Experimental, not used!
 *
 * @param histogram
 * @param cutOff
 * @param pixels
 * @return
 */
private int[] computeCuts(int[] histogram, double cutOff, int pixels) {
    int[] histAccum = new int[maxIterations+1];
    histAccum[0] = histogram[0];
    histAccum[maxIterations] = pixels;

    int lowCut = 0;
```

```
        int highCut = 0;
        for (int i=1; i<maxIterations+1; i++) {
            histAccum[i] = histAccum[i-1] + histogram[i];
            if (histAccum[i] >= 1) {
                lowCut = i;
                break;
            }
        }
        for (int i=maxIterations-1; i>0; i--) {
            histAccum[i] = histAccum[i+1] - histogram[i];
            if (histAccum[i] <= pixels*(1-cutOff)) {
                highCut = i;
                break;
            }
        }

        return new int[] {lowCut, highCut};
    }
}
```