University of Antwerp

Scientific Programming

Data Fitting Exercise 1

Armin Halilovic - s0122210

October 30, 2015

Contents

1 Problem

We are given the Runge function

$$f(x) = \frac{1}{1 + 25x^2} \quad x \in [-1, 1]$$

We will construct graphs that approximate this function using a few interpolation techniques.
First, we will use Newton's divided differences interpolation on a set of equidistant and a set of non equidistant points. Then, cubic spline interpolation will be used, followed by spline interpolation with modified boundary conditions.

All of this will be done using C++ and the GNU Scientific Library. In the solutions, we will briefly describe what we did with excerpts from the code, generate graphs using the interpolation methods, and look at how close each one of them got to the Runge function. The first solution will contain more information than the other ones, as it will explain most of the code.

2 Using the code

All of the C++ code can be found in the "main.cpp" file and in appendix A of this document. main.cpp comes accompanied by createImages.sh, which contains all of the neccessary UNIX commands to generate the graph images. This file relies on the graph program in the GNU plotutils package to plot graphs, so make sure that it is installed.

To compile and run the program, execute the following commands in the build/ directory:

```
cmake ..
make
chmod +x ./createImages.sh
./data_fitting
```

All of the graphs should be present in the build/images/ directory. If they are not there, make createImages.sh executable with "chmod +x" and run it to create them.

3 Solutions

In figure 1, we can see what the Runge function looks like.

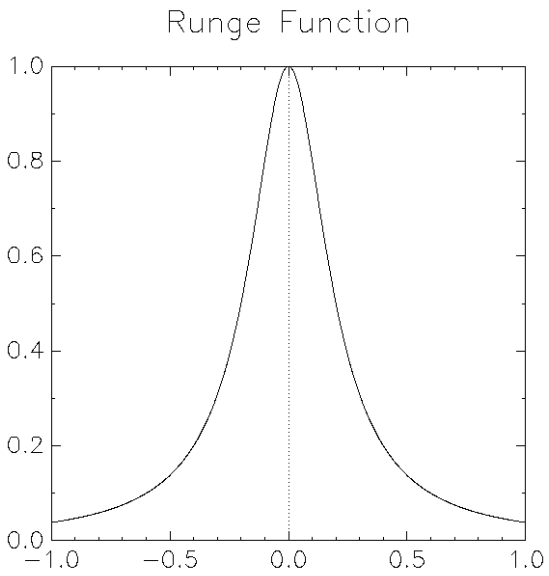

Figure 1: Runge function

3.1 Polynomial through equidistant points

We will start with Newton's divided differences interpolation to approximate the graph. A set of 17 equidistant points from -1 to 1 will be used do to the interpolation on, these will be generated as follows:

```
int i = 0, size = 17;
double xi, xa1[size], ya1[size];
for (xi = -1; xi <= 1; xi = xi + ( 2 / (double) 16)) {
    xa1[i] = xi;
    ya1[i] = runge_f(xi);
    i++;
}
```

To execute the interpolation, we need to initialize a couple of GSL structs.

```
gsl_interp_accel *acc = gsl_interp_accel_alloc();
gsl_spline *interp_poly = gsl_spline_alloc(gsl_interp_polynomial, size);
gsl_spline_init(interp_poly, xa1, ya1, size);
```

Here, the gsl_spline functions do not mean we are working with splines, despite what their name may suggest. They simply provide a higher level interface so that we can write less code later on.
The type of the interpolation we will work with is passed into the gsl_spline_alloc

function. In this case, gsl_spline_init will use gsl_poly_dd_init to initialize the polynomial we will use. gsl_poly_dd_init uses the Newton polynomial, which is what we need to calculate new interpolation points.

Now, we can start plotting graphs. The gsl_spline_eval function is used to calculate new points.
For each solution, we will plot 3 graphs: The result of the interpolation, and the absolute and relative difference with respect to the Runge function:

```
double interpValue, realValue;
for (xi = xa1[0]; xi < xa1[16]; xi += 0.001) {
    interpValue = gsl_spline_eval(interp_poly, xi, acc);
    realValue = runge_f(xi);
    file1 << xi << " " << interpValue << std::endl;
    file2 << xi << " " << interpValue - realValue << std::endl;
    file3 << xi << " " << (interpValue - realValue) / realValue <<
        std::endl;
}
```

The result of the interpolation and the difference with respect to the Runge function can be seen in figures 2 and 3.

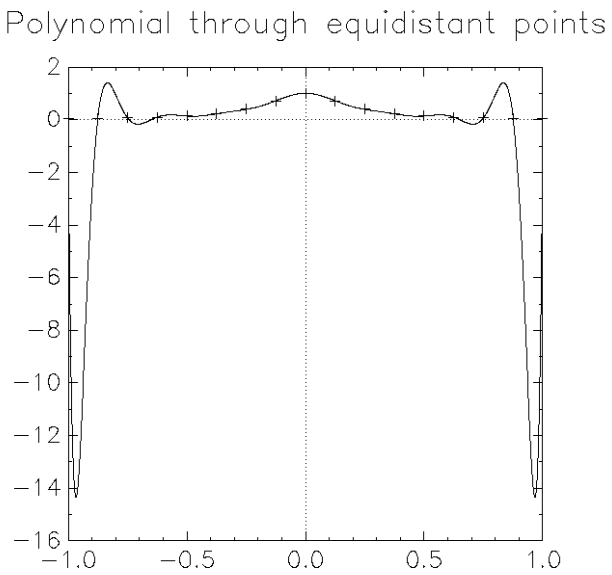

Figure 2: Interpolation through equidistant points

At first glance, this approximation looks horrible. However, this is purely because of how the oscillations at the endpoints of the graph skew the graph. In the middle part of the graph, the approximation is pretty accurate. This is typical for interpolation polynomials of higher powers.

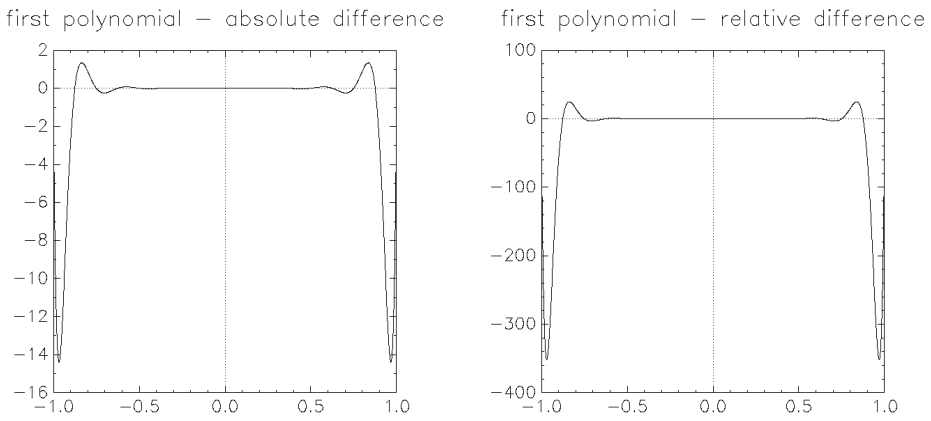

Figure 3: Differences for interpolation through equidistant points

In figure 3, it is clearly visible that the approximations are at their worst around the endpoints of the interpolation.

3.2 Polynomial through non equidistant points

Now, we will apply the same method as in the previous section. The only difference lies in the set of points we use the interpolation on. This set is decided by

$$x_i = \cos(\frac{(i+0.5)\pi}{17}) \quad where\ i \in \{0, 1, ..., 16\}$$

```
for (i = 0; i <= 16; i++) {
    xa[i] = cos( ((double)i + 0.5) * (M_PI / 17) );
}
```

The rest of the code is analogue to section 3.1.
The result of using non equidistant points is instantly apparent in figures 4 and 5. Compare them to figures 2 and 3 and you will see that using non equidistant points will dramatically improve the approximation, which is in line with what we have learned during the lectures on Scientific Programming. The oscillation is still present, but it is much less extreme now.

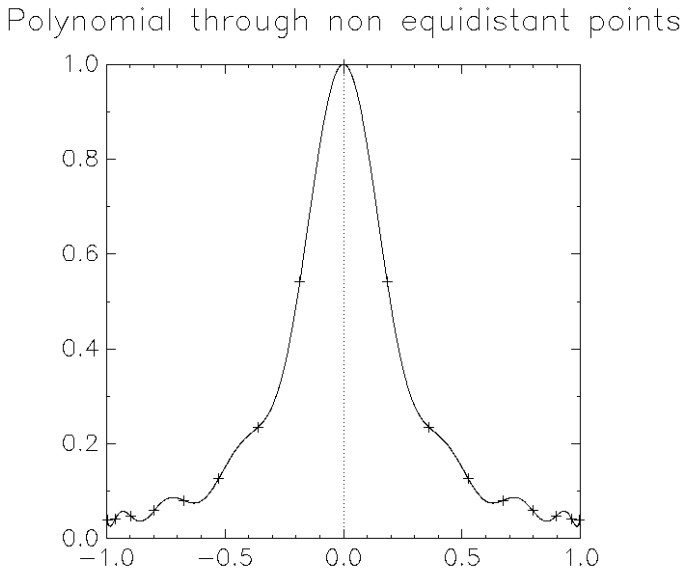

Figure 4: Polynomial interpolation through non equidistant points

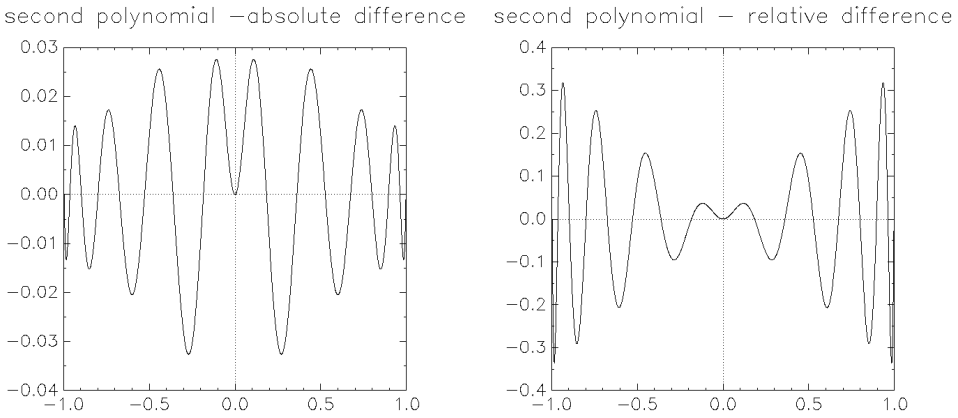

Figure 5: Differences for polynomial interpolation through non equidistant points

3.3 Natural cubic spline

GSL makes it simple to use various interpolation techniques. For natural cubic spline interpolation, the only difference we need in the code lies in the initialization of the interpolation struct. We can simply pass gsl_interp_cspline into gsl_spline_alloc instead of gsl_interp_polynomial. This will generate figures 6 and 7.

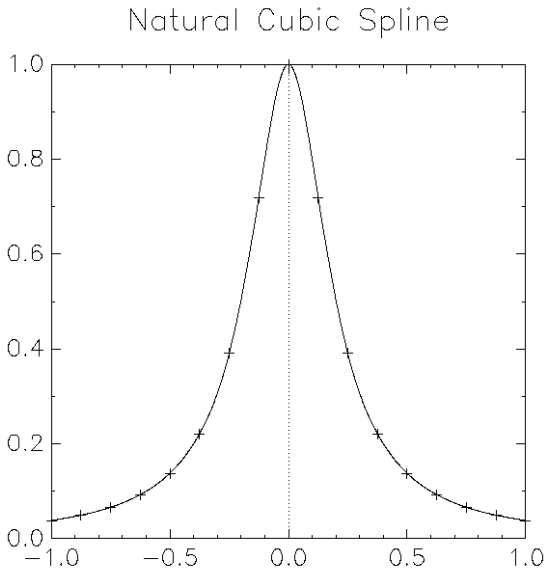

Figure 6: Cubic spline interpolation through equidistant points

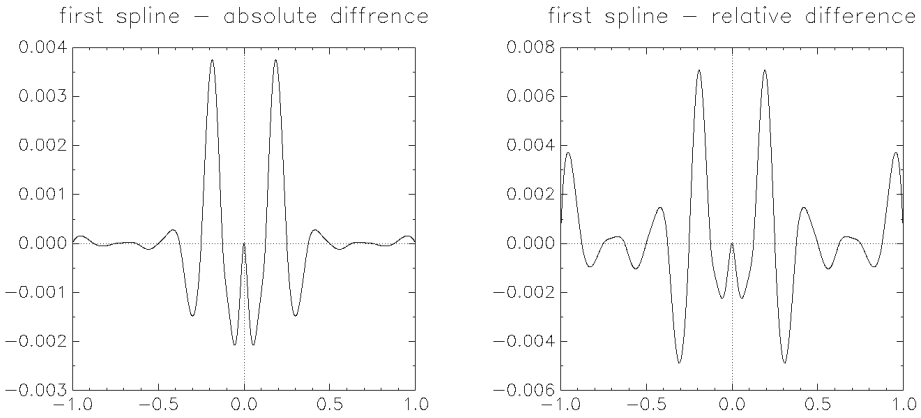

Figure 7: Differences for cubic spline interpolation

Spline interpolation can be used to avoid the extreme oscillations at the ends of intervals with polynomial interpolation. In figures 6 and 7, We can see that the result of using a natural cubic spline looks almost identical to the runge function. The spline has produced a much better graph than both of the polynomials: its errors have a supremum of roughly 0.004, while for the polynomials this number is 14 and 0.03.

3.4 Cubic spline with modified conditions

We now wish to execute a cubic spline interpolation with modified boundary conditions. We want to change the conditions to

$$S''(-1) = f''(-1) = \frac{925}{4394}$$

and

$$S''(1) = f''(1) = \frac{925}{4394}$$

However, there is no provided functionality to do this in GSL, so we will have to modify some of its code. In "cspline.c" in GSL's source code, we can see that in the cspline_init function the boundary conditions are set to 0 for the cubic spline.

```
106| state->c[0] = 0.0;
107| state->c[max_index] = 0.0;
```

In our code we simply change these values after initializing the spline. To do this, we first need to copy the declaration for cspline_state_t into our code so we can access the conditions.

```
gsl_spline *interp_spline = gsl_spline_alloc(gsl_interp_spline, size);
gsl_spline_init(interp_spline, xa1, ya1, size);
cspline_state_t *state = (cspline_state_t *)
    interp_cspline->interp->state;
state->c[0] = (double) 925 / 4394;
state->c[size - 1] = (double) 925 / 4394;
```

The results are visible in figures 8 and 9. The comparison of to the polynomial interpolation counts here as well, as not much has changed with respect to the natural cubic spline. Only the endpoints of the interval have changed, which is to be expected after changing the boundary conditions. The difference between the natural and this spline is only clear when looking at the graphs of the differences. In this case, we see that the difference with respect to the Runge function became worse when we modified the boundary conditions, which makes the natural cubic spline a better approximation.

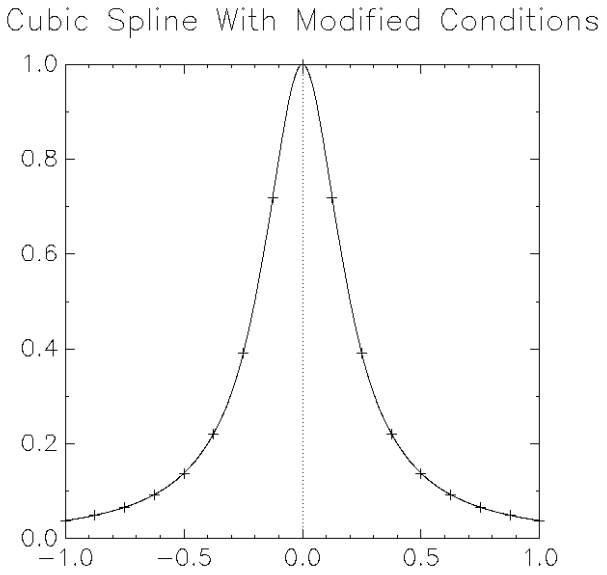

Figure 8: Spline interpolation through equidistant points

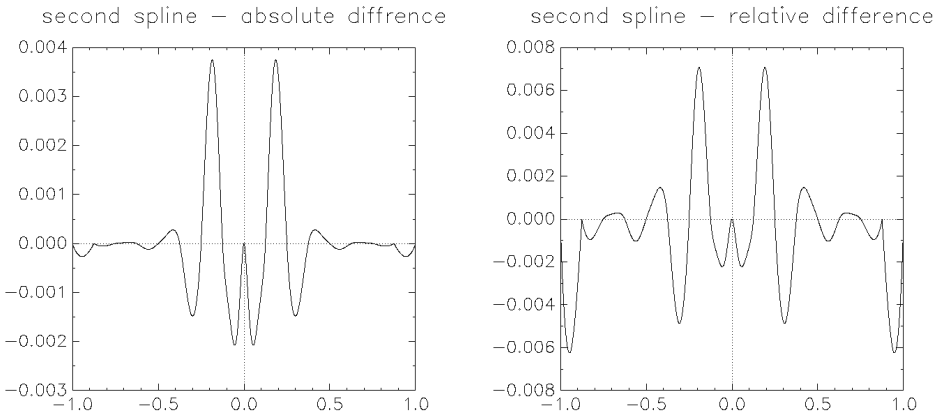

Figure 9: Differences for spline interpolation

4 Conclusion

After conducting the interpolations, we can conclude that using a cubic spline is the best way to approximate the Runge function, if we had to choose between polynomial and spline interpolation. This conforms to what we learned during the lecture on data fitting.

Appendices

A main.cpp

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_deriv.h>
#include <gsl/gsl_spline.h>
#include <algorithm>
#include <fstream>
#include <iostream>

//The given Runge function
double runge_f(double x) {
    return 1 / (1 + 25 * gsl_pow_2(x));
}

//Struct from gsl's "cspline.c" to be able to change the conditions on the endpoints of the
      spline
typedef struct
{
    double * c;
    double * g;
    double * diag;
    double * offdiag;
} cspline_state_t ;


int main (void) {
    int i = 0, size = 17;
    double xi, xa1[size], ya1[size], xa2[size], ya2[size];

    //Files to write data points for graphs to
    std::ofstream runge, poly1, poly2, spline1, spline2;
    std::ofstream diff1abs, diff1rel, diff2abs, diff2rel, diff3abs, diff3rel, diff4abs,
        diff4rel, truncError;
    runge.open("images/runge.dat");
    poly1.open("images/poly1.dat"); poly2.open("images/poly2.dat");
    spline1.open("images/spline1.dat"); spline2.open("images/spline2.dat");
    diff1abs.open("images/diff1abs.dat"); diff1rel.open("images/diff1rel.dat");
    diff2abs.open("images/diff2abs.dat"); diff2rel.open("images/diff2rel.dat");
    diff3abs.open("images/diff3abs.dat"); diff3rel.open("images/diff3rel.dat");
    diff4abs.open("images/diff4abs.dat"); diff4rel.open("images/diff4rel.dat");
    truncError.open("images/trunError.dat");

    if (!runge.is_open()) {
        std::cout << "Could not open file 'runge.dat', make sure the images folder exists"
            << std::endl;
        return 0;
    }

    //Mark the following points on the graph with a plus sign
    poly1 << "#m=0,S=2\n";
    poly2 << "#m=0,S=2\n";
    spline1 << "#m=0,S=2\n";
    spline2 << "#m=0,S=2\n";

    //17 equidistant points from -1 to 1
    for (xi = -1; xi <= 1; xi = xi + ( 2 / (double) 16)) {
```

```
    xa1[i] = xi;
    ya1[i] = runge_f(xi);
    poly1 << xi << " " << ya1[i] << std::endl;
    spline1 << xi << " " << ya1[i] << std::endl;
    spline2 << xi << " " << ya1[i] << std::endl;
    i++;
}


//17 non equidistant points from -1 to 1
for (i = 0; i <= 16; i++) {
    xa2[i] = cos( ((double)i + 0.5) * (M_PI / 17) );
}
//The points have to be sorted for the evaluation function
std::sort(xa2, xa2 + size);
for (i = 0; i <= 16; i++) {
    ya2[i] = runge_f(xa2[i]);
    poly2 << xa2[i] << " " << ya2[i] << std::endl;
}

 gsl_interp_accel *acc = gsl_interp_accel_alloc ();
//First polynomial for the equidistant points
 gsl_spline *interp_poly1 = gsl_spline_alloc (gsl_interp_polynomial, size);
//Second polynomial for the non equidistant points
 gsl_spline *interp_poly2 = gsl_spline_alloc (gsl_interp_polynomial, size);
//Natural cubic spline through the equidistant points
 gsl_spline *interp_cspline1 = gsl_spline_alloc ( gsl_interp_cspline , size);
//Cubic spline with modified requirements in the endpoints
 gsl_spline *interp_cspline2 = gsl_spline_alloc ( gsl_interp_cspline , size);

// Initialize the polynomials
 gsl_spline_init (interp_poly1, xa1, ya1, size);
 gsl_spline_init (interp_poly2, xa2, ya2, size);
 gsl_spline_init ( interp_cspline1 , xa1, ya1, size);
 gsl_spline_init ( interp_cspline2 , xa1, ya1, size);

//Change the conditions on the endpoints of the second cspline
 cspline_state_t *state = ( cspline_state_t *) interp_cspline2->interp->state;
state->c[0] = (double) 925 / 4394;
state->c[size - 1] = (double) 925 / 4394;

//Points of the runge function
for (xi = -1; xi < 1; xi += 0.001) {
    runge << xi << " " << runge_f(xi) << std::endl;
}

//Connect the following data points with a line
poly1 << "#m=1,S=0\n";
poly2 << "#m=1,S=0\n";
spline1 << "#m=1,S=0\n";
spline2 << "#m=1,S=0\n";

//Data points for the first interpolation polynomial and both of the csplines
double interpValue, realValue;
for (xi = xa1[0]; xi < xa1[16]; xi += 0.001) {
    interpValue = gsl_spline_eval (interp_poly1, xi, acc);
    realValue = runge_f(xi);
    poly1 << xi << " " << interpValue << std::endl;
    diff1abs << xi << " " << interpValue - realValue << std::endl;
     diff1rel << xi << " " << (interpValue - realValue) / realValue << std::endl;

    interpValue = gsl_spline_eval ( interp_cspline1 , xi, acc);
```

```
        spline1 << xi << " " << interpValue << std::endl;
        diff3abs << xi << " " << interpValue - realValue << std::endl;
        diff3rel << xi << " " << (interpValue - realValue) / realValue << std::endl;

        interpValue = gsl_spline_eval (interp_cspline2, xi, acc);
        spline2 << xi << " " << interpValue << std::endl;
        diff4abs << xi << " " << interpValue - realValue << std::endl;
        diff4rel << xi << " " << (interpValue - realValue) / realValue << std::endl;
    }

    //Data points for the second interpolation polynomial
    for (xi = xa2[0]; xi < xa2[16]; xi += 0.001) {
        interpValue = gsl_spline_eval (interp_poly2, xi, acc);
        realValue = runge_f(xi);
        poly2 << xi << " " << interpValue << std::endl;
        diff2abs << xi << " " << interpValue - realValue << std::endl;
        diff2rel << xi << " " << (interpValue - realValue) / realValue << std::endl;
    }

    //Free the memory and close the files
    gsl_interp_accel_free (acc);
    gsl_spline_free (interp_poly1);
    gsl_spline_free (interp_poly2);
    gsl_spline_free (interp_cspline1);
    gsl_spline_free (interp_cspline2);
    runge.close();
    poly1.close(); poly2.close();
    spline1.close(); spline2.close();
    diff1abs.close(); diff1rel.close();
    diff2abs.close(); diff2rel.close();
    diff3abs.close(); diff3rel.close();
    diff4abs.close(); diff4rel.close();
    truncError.close();

    //Create the images with the gsl graph application
    system("./createImages.sh");
    return 0;
}
```

B createImages.sh

```
#!/bin/bash

DIR="images"

graph -T png -F HersheySans -L "Runge Function" --bitmap-size 820x820 <
    $DIR/runge.dat > $DIR/runge.png

graph -T png -F HersheySans -L "Natural Cubic Spline" --bitmap-size 820x820 <
    $DIR/spline1.dat > $DIR/spline1.png
graph -T png -F HersheySans -L "Cubic Spline With Modified Conditions" --bitmap-size
    820x820 < $DIR/spline2.dat > $DIR/spline2.png

graph -T png -F HersheySans -L "Polynomial through equidistant points" --bitmap-size
    820x820 < $DIR/poly1.dat > $DIR/poly1.png
graph -T png -F HersheySans -L "Polynomial through non equidistant points"
    --bitmap-size 820x820 < $DIR/poly2.dat > $DIR/poly2.png

graph -T png -F HersheySans -L "first polynomial - absolute difference" --bitmap-size
    820x820 < $DIR/diff1abs.dat > $DIR/diff1abs.png
graph -T png -F HersheySans -L "second polynomial -absolute difference" --bitmap-size
    820x820 < $DIR/diff2abs.dat > $DIR/diff2abs.png
graph -T png -F HersheySans -L "first spline - absolute diffrence" --bitmap-size 820x820
    < $DIR/diff3abs.dat > $DIR/diff3abs.png
graph -T png -F HersheySans -L "second spline - absolute diffrence" --bitmap-size
    820x820 < $DIR/diff4abs.dat > $DIR/diff4abs.png

graph -T png -F HersheySans -L "first polynomial - relative difference" --bitmap-size
    820x820 < $DIR/diff1rel.dat > $DIR/diff1rel.png
graph -T png -F HersheySans -L "second polynomial - relative difference" --bitmap-size
    820x820 < $DIR/diff2rel.dat > $DIR/diff2rel.png
graph -T png -F HersheySans -L "first spline - relative difference" --bitmap-size 820x820
    < $DIR/diff3rel.dat > $DIR/diff3rel.png
graph -T png -F HersheySans -L "second spline - relative difference" --bitmap-size
    820x820 < $DIR/diff4rel.dat > $DIR/diff4rel.png
```