University of Antwerp

Scientific Programming

# Second Session
## Exercise 2

Armin Halilovic - s0122210

August 22, 2016

# Contents

## 1 Problem

We are given data points $(x_i, f_i), i = 0, ..., 20$ with $x_i = i - 10$ and $f_i = (-1)^i$ and are tasked do the following:

1. Calculate a polynomial interpolant through the data
2. Calculate the natural cubic spline for the data
3. Calculate a least squares approximant
4. Calculate the trigonometric polynomial interpolant
5. Calculate a trigonometric least squares approximant

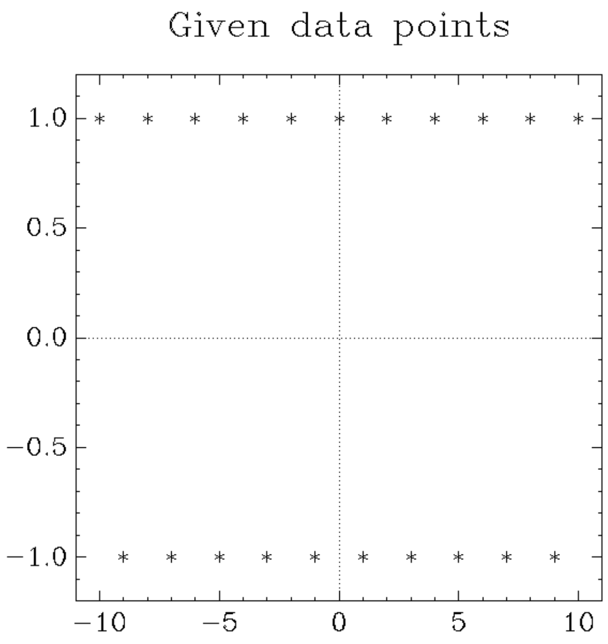


**Figure 1:** The given data points

All of this will be done using C++ and the GNU Scientific Library. In the solutions, we briefly describe what we did with excerpts from the code, generate graphs using the interpolation and approximation methods and discuss the results.

## 2 Using the code

All of the C++ code can be found in the .cpp files and in appendix A of this document. main.cpp comes accompanied by createImages.sh, which contains all of the necessary 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_smoothing_and_fn_approx.bin
```

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

## 3 Solutions

### 3.1 Polynomial interpolant

We use Newton's divided differences interpolation to do the polynomial interpolation through the data.

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, m);
gsl_spline_init(interp_poly, x_i, y_i, m);
```

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 Newton's divided differences to build the polynomial, which is what we need to calculate new interpolation points.

Now, we can plot a graph.

```
double interpValue;
    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    for (double x = minArray(x_i, m); x <= maxArray(x_i, m); x += 0.001) {
        interpValue = gsl_spline_eval(interp_poly, x, acc);
        polynomialInterp << x << " " << interpValue << "\n";
    }
```

The result of the interpolation can be seen in figure 2.

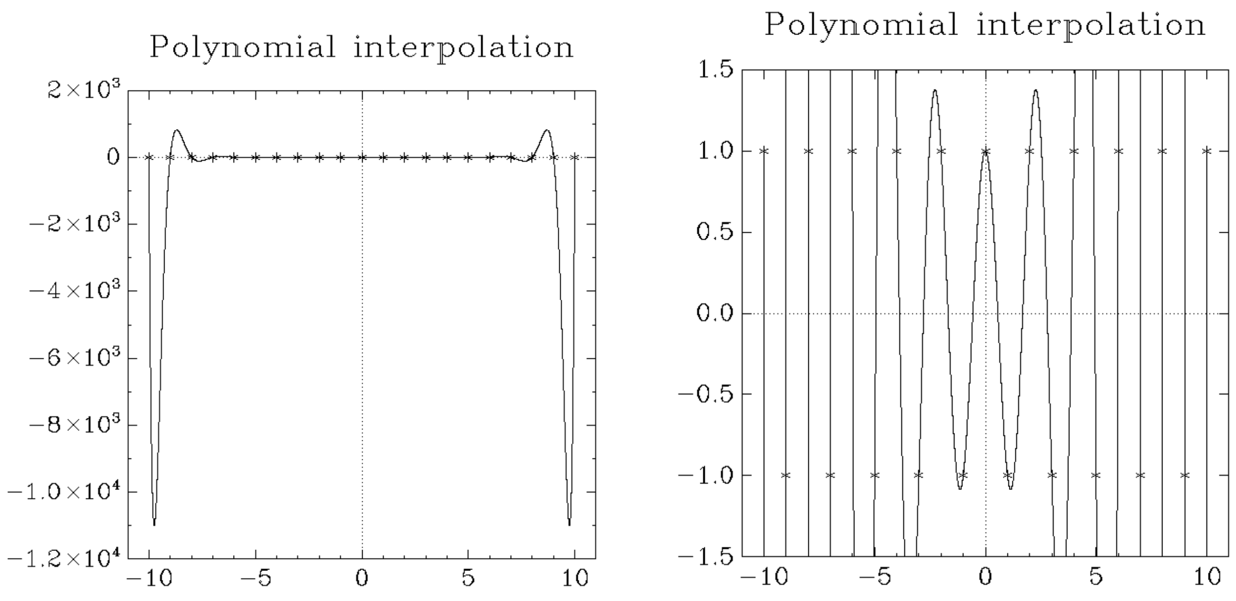


**Figure 2:** Polynomial interpolation through the given data points. Zoomed in image on the right.

The polynomial interpolant is of degree 20. We see that near the endpoints of the interpolation, the approximations become very large values. This happens because of high degree of the polynomial together with the fact that we are using equidistant points. A way to improve this interpolation would be using Chebyshev nodes as the $x_i$ points. These points are spread out more towards the endpoints of the interval, which would reduce this extreme behaviour.

## 3.2 Natural cubic spline

For natural cubic spline interpolation, the only difference in the code lies in the initialization of the interpolation struct. We simply pass gsl_interp_cspline into gsl_spline_alloc instead of gsl_interp_polynomial and plot the grap. The result is visible in figure 9.

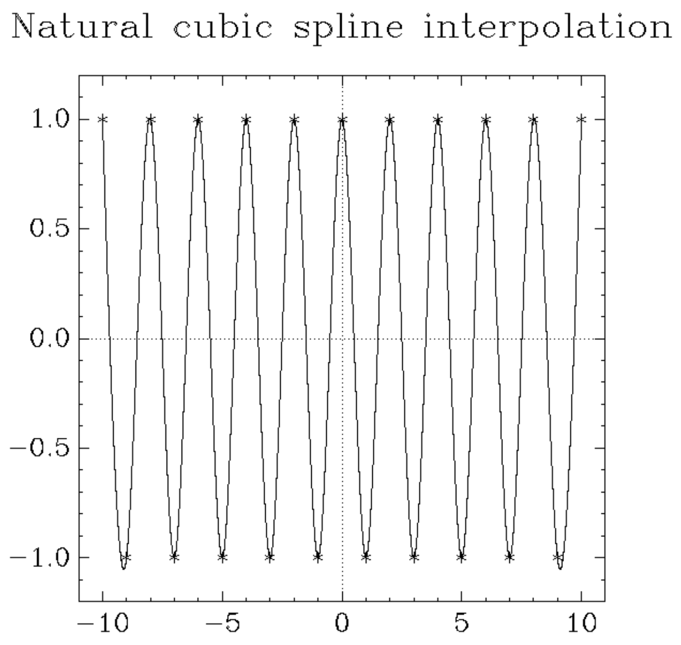


**Figure 3:** Natural cubic spline interpolation through the given data points

We get a much nicer result than the polynomial interpolation. However, we see that near the end points the interpolation goes beyond -1. This leads us to believe that for the given data, the end conditions of the cubic spline are not satisfied.

We achieve more favorable results by using a cubic spline with periodic boundary conditions. This is done by passing gsl_interp_cspline_periodic into the gsl spline allocator.

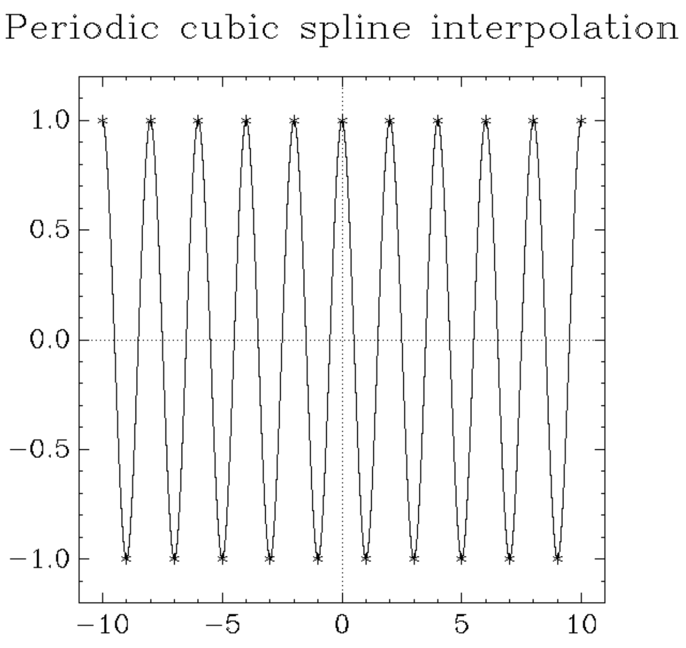


**Figure 4:** Periodic cubic spline interpolation through the given data points

## 3.3 Least squares approximation

To get a least squares approximation, we find an optimal solution for the equation

$$\lambda_1 f_1(x_i) + ... + \lambda_n f_n(x_i) = y_i \quad where \quad i = 1, ..., m \gg n$$

where m stands for the amount of given data points and n for the amount of unknowns we want to calculate. This equation can be written as an overdetermined system of equations of the form $A\lambda = y$. The system is constructed in the code as follows:

```
gsl_matrix *A = gsl_matrix_alloc(size, n);
gsl_vector *Y = gsl_vector_alloc(size);

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        gsl_matrix_set(A, i, j, gsl_pow_int(x_rescaled[i], j));
    }
    gsl_vector_set(Y, i, y_i[i]);
}
```

We note here that as basis function we used $f_i(x) = x^i$.

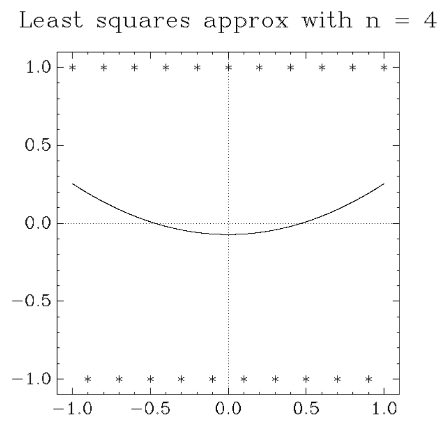
### 3.3.1 QR factorization

Since the system $A\lambda = y$ is overdetermined, we cannot solve it exactly, so we find the least squares solution for it. The least squares solution will minimize the Euclidean norm of the residual, $||A\lambda - y||$. Before we do that, QR factorization is used to decompose the matrix $A$ into a product $A = QR$ of an orthogonal matrix $Q$ and an upper triangular matrix $R$:

```
gsl_linalg_QR_decomp(QR, tau);
gsl_linalg_QR_lssolve(QR, tau, Y, X, R);
```

### 3.3.2 Results

We calculated the least squares solutions for the system for n = 4, 7, 10, 15, 20, and 21. Detailed output of the solutions for the systems can be found in the output/ directory. We show the graphs for these solutions here:



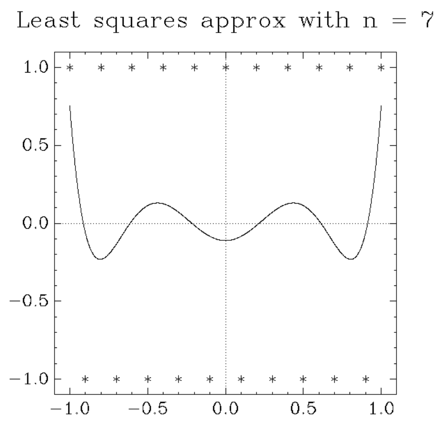
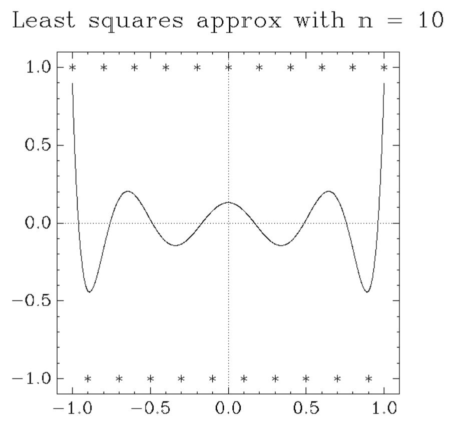


**Figure 5:** Approximations for n = 4 and n = 7



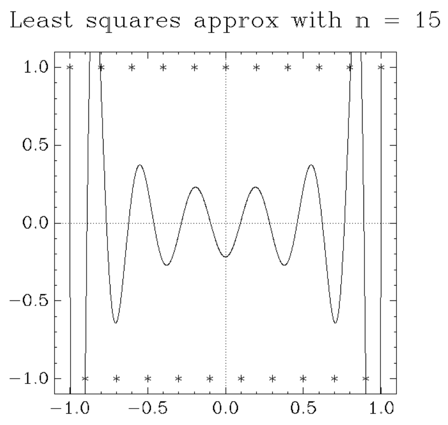
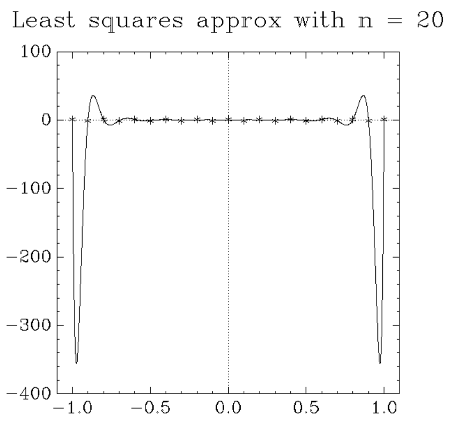


**Figure 6:** Approximations for n = 10 and n = 15



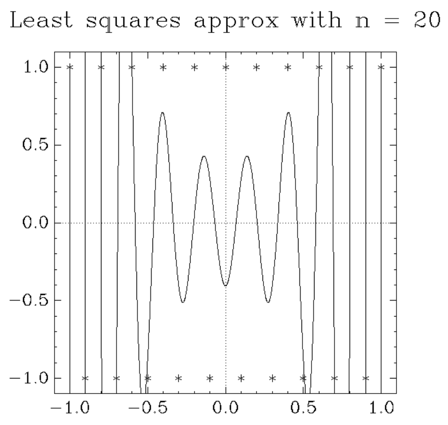
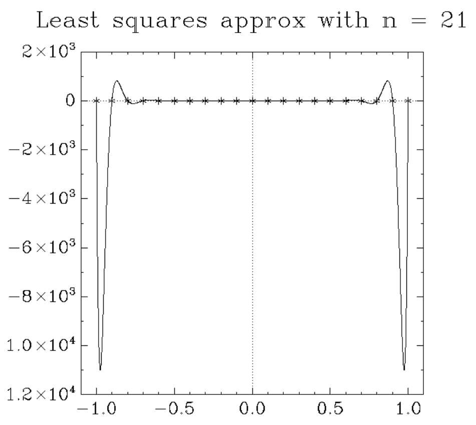


**Figure 7:** Approximation for n = 20. Zoomed in image on the right



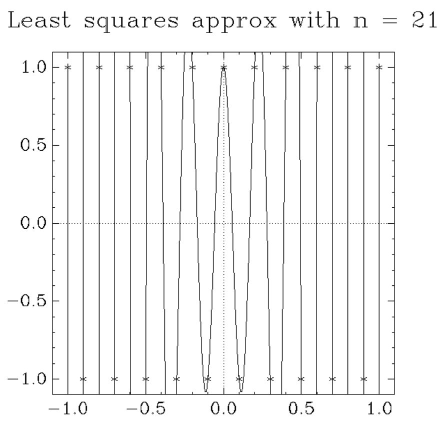


**Figure 8:** Approximation for n = 21. Zoomed in image on the right

Note that when n = 21, we get the same result as with the polynomial interpolation. This happens because at n = 21 we are solving the same 21x21 system that the polynomial interpolation solves. Thus, at n = 21 we are doing interpolation.

### 3.3.3 Condition number of $A$

We calculate the condition number of $A$ at each approximation to have some kind of measure for the accuracy of our solutions. The condition number is not a direct representation of the accuracy of the solutions, but it plays a big role in it. A condition number with a high order of magnitude means that a small change in the input matrix $A$ *could* cause a significant change in the solutions, whereas the opposite counts for condition numbers with smaller orders of magnitude.
To calculate the condition number of A, we use the following definition:

$$\kappa(A) = \frac{|max(S)|}{|min(S)|}$$

where S is the vector of singular values of A. In GSL, we can find S by using gsl_linalg_SV_decomp.

```
gsl_linalg_SV_decomp(U, V, S, work);
double condNumber, minS, maxS;
minS = gsl_vector_get(S, 0);
maxS = gsl_vector_get(S, 0);
for (int j = 0; j < n; j++) {
    if (gsl_vector_get(S, j) < minS) minS = gsl_vector_get(S, j);
    if (gsl_vector_get(S, j) > maxS) maxS = gsl_vector_get(S, j);
}
condNumber = fabs(maxS) / fabs(minS);
```

We find the following condition numbers

```
n = 4:  Condition number: 1.6514
n = 7:  Condition number: 91.935
n = 10: Condition number: 1294.9
n = 15: Condition number: 1.8592e+05
n = 20: Condition number: 1.2585e+08
n = 21: Condition number: 8.3138e+08
```

More details for each system can be found in the output/leastSquares_degree_*.txt files

A way to improve our results and lower the condition numbers would be to use the Legendre polynomials as basis functions instead of $f_i(x) = x^i$, but that was not done in these solutions.

## 3.4 Trigonometric polynomial interpolant

We find a trigonometric polynomial interpolant by using a function of the form:

$$f(t) = \frac{a_0}{2} + a_1 \cos(t) + a_2 \cos(2t) + ... + a_m \cos(mt) + b_1 \sin(t) + b_2 \sin(2t) + ... + b_m \sin(mt)$$

Because the given data points are evenly spaced, we can use a simple function to calculate the unknown $a_i$'s and $b_i$'s:

```
double fourierCoefficient(int j, int n, double *t_i, double *x_i, bool a) {
    double result = 0;

    for (int k = 0; k < n; k++) {
        result += x_i[k] * ((a) ? gsl_sf_cos(j * t_i[k]) : gsl_sf_sin(j * t_i[k]));
    }

    result *= 2.0/(double) n;
    return result;
}
```

We fill in the function with $n = amount\ of\ given\ data\ points = 21$ and $m = 10$. With $m = 10$, there are $2m + 1 = 21$ coefficients, so we can do interpolation. Using the function, we get the following graph:

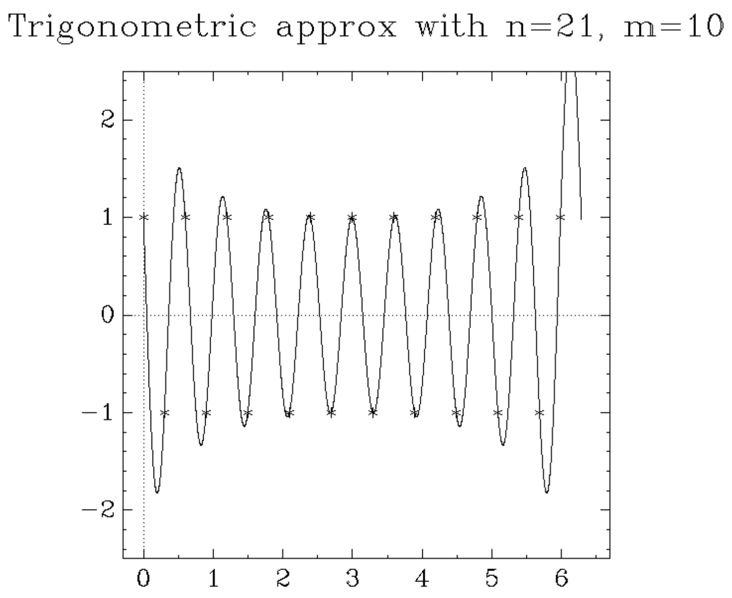


**Figure 9:** Trigonometric interpolation through the given data points

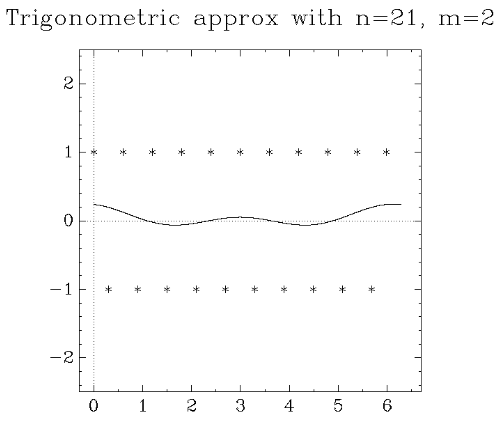
## 3.5 Trigonometric least squares approximant

We use the same function

$$f(t) = \frac{a_0}{2} + a_1 \cos(t) + a_2 \cos(2t) + ... + a_m \cos(mt) + b_1 \sin(t) + b_2 \sin(2t) + ... + b_m \sin(mt)$$

with the same n = 21. This time we choose m such that $n > 2m + 1$ holds true. The choice of an even vs. an uneven amount of terms does not matter here. The only time it matters is when

$n = 2m$. In that case, the function $f(t)$ changes slightly. Using the function, we get the following graphs for m = 2, 4, and 7:



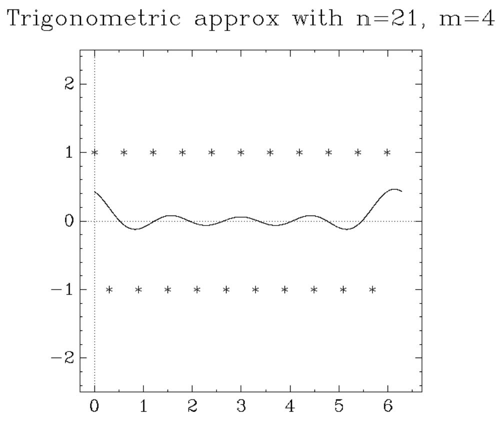
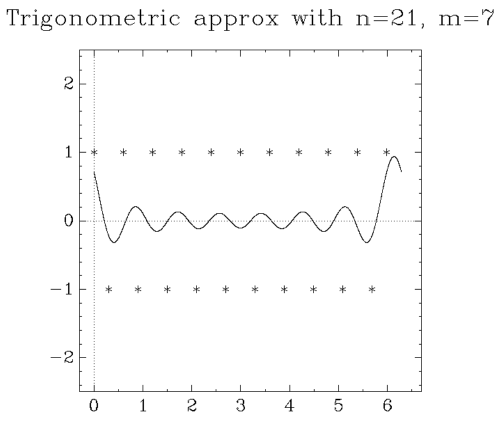


**Figure 10:** Trigonometric least squares approximation for the given data points with m = 2 and 4



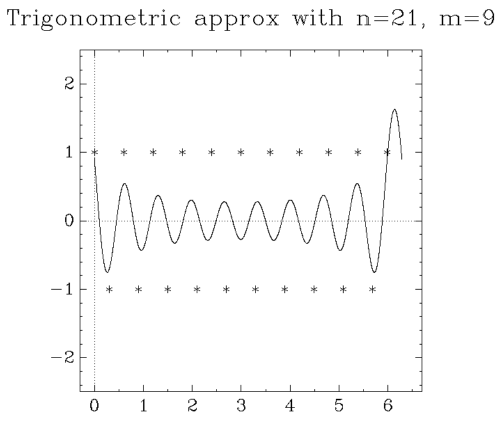


**Figure 11:** Trigonometric least squares approximation for the given data points with m = 7 and 9

# Appendices

## A main.cpp

```
#include <iostream>
#include "math.h"
#include "./functions.h"

int main (void) {
    const int m = 21;
    double x_i[m], y_i[m];
    fillFArrays(x_i, y_i, m);    // x_i = i - 10, y_i = (-1)^i, i = 0..20
    double a = minArray(x_i, m); // -10
    double b = maxArray(x_i, m); // 10

    // write given points to dataPoints.dat
    writeDataPoints(x_i, y_i, m);

    // polynomial interpolation and spline interpolation
    polynomialAndSplineSolutions(m, x_i, y_i);

    // least squares approximations
    int leastSquares[] = {2, 3, 4, 5, 7, 10, 15, 20, 21};
    for (int i = 0; i < 9; i++) {
        leastSquaresApproximation(m, leastSquares[i], x_i, a, b, y_i);
    }

    // trigonometric polynomial interpolation and trigonometric least squares approximation0
    trigonometricApproximation(m, 2, x_i, a, b, y_i);
    trigonometricApproximation(m, 4, x_i, a, b, y_i);
    trigonometricApproximation(m, 7, x_i, a, b, y_i);
    trigonometricApproximation(m, 9, x_i, a, b, y_i);
    trigonometricApproximation(m, 10, x_i, a, b, y_i);

    // 4 examples of fourier approximation from the book, works perfectly...
    /*
    double tt[] = {0, 2*M_PI/5, 4*M_PI/5, 6*M_PI/5, 8*M_PI/5}, xx[] = {1, 3, 2, 0, -1};
    trigonometricApproximation(5, 1, tt, 0, 2*M_PI, xx);
    trigonometricApproximation(5, 2, tt, 0, 2*M_PI, xx);

    double ttt[] = {0 * (2*M_PI)/8, 1 * (2*M_PI)/8, 2 * (2*M_PI)/8, 3 * (2*M_PI)/8, 4 * (2*M_PI)/8, 5 * (2*M_PI)/8, 6 *
        (2*M_PI)/8, 7 * (2*M_PI)/8}, xxx[] = {1, 1, 1, 1, 0, 0, 0, 0};
    trigonometricApproximation(8, 2, ttt, 0, 2*M_PI, xxx);
    trigonometricApproximation(8, 4, ttt, 0, 2*M_PI, xxx);
    */

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

## B functions.h

```
#ifndef PROJECT_FUNCTIONS_H
#define PROJECT_FUNCTIONS_H

#include <fstream>
#include <gsl/gsl_vector_double.h>
#include <gsl/gsl_matrix_double.h>

void printVector(const gsl_vector *, std::string);
void printVector(const gsl_vector *, std::string, std::ostream &);
void printVectorCoutAndFile(const gsl_vector *, std::string, std::ostream &);
void printMatrix(const gsl_matrix *, std::string);
void printMatrix(const gsl_matrix *, std::string, std::ostream &);
void printMatrixCoutAndFile(const gsl_matrix *, std::string, std::ostream &);

void printArray(const double *x, const int m);
double minArray(double *array, int n);
double maxArray(double *array, int n);

void writeDataPoints(double *x_i, double *y_i, int m);
void fillFArrays(double *x_i, double *y_i, int m);
void polynomialAndSplineSolutions(int m, double *x_i, double *y_i);
void leastSquaresApproximation(int m, int n, double *x_i, double a, double b, double *y_i);
void trigonometricApproximation(int n, int m, double *x_i, double a, double b, double *y_i);

#endif //PROJECT_FUNCTIONS_H
```

## C functions.cpp

```
#include "./functions.h"

#include <cmath>
#include <iomanip>
#include <iostream>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_spline.h>
#include <algorithm>
#include <sstream>
#include <gsl/gsl_sf_trig.h>

const int PRINT_WIDTH = 13;
const int PRINT_PRECISION = 5;

void printVector(const gsl_vector * v, std::string string, std::ostream &out) {
    out << "Vector " << string << ":\n";
    for (unsigned int i = 0; i < v->size; i++) {
        out << std::setw(PRINT_WIDTH) << std::setprecision(PRINT_PRECISION) << gsl_vector_get(v, i) << "\n";
    }
    out << "\n";
}

void printVectorCoutAndFile(const gsl_vector * v, std::string string, std::ostream &out) {
    std::cout << "Vector " << string << ":\n";
    out << "Vector " << string << ":\n";
    for (unsigned int i = 0; i < v->size; i++) {
        std::cout << std::setw(PRINT_WIDTH) << std::setprecision(PRINT_PRECISION) << gsl_vector_get(v, i) <<
            "\n";
        out << std::setw(PRINT_WIDTH) << std::setprecision(PRINT_PRECISION) << gsl_vector_get(v, i) << "\n";
    }
    std::cout << "\n";
    out << "\n";
}
```

```
void printMatrix(const gsl_matrix *m, std::string string, std::ostream &out) {
    out << "Matrix " << string << ":\n";
    for (unsigned int i = 0; i < m->size1; i++) {
        for (unsigned int j = 0; j < m->size2; j++) {
            out << std::setw(PRINT_WIDTH) << std::setprecision(PRINT_PRECISION) << gsl_matrix_get(m, i, j);
        }
        out << "\n";
    }
    out << "\n";
}

void printMatrixCoutAndFile(const gsl_matrix *m, std::string string, std::ostream &out) {
    std::cout << "Matrix " << string << ":\n";
    out << "Matrix " << string << ":\n";
    for (unsigned int i = 0; i < m->size1; i++) {
        for (unsigned int j = 0; j < m->size2; j++) {
            std::cout << std::setw(PRINT_WIDTH) << std::setprecision(PRINT_PRECISION) << gsl_matrix_get(m, i, j);
            out << std::setw(PRINT_WIDTH) << std::setprecision(PRINT_PRECISION) << gsl_matrix_get(m, i, j);
        }
        std::cout << "\n";
        out << "\n";
    }
    std::cout << "\n";
    out << "\n";
}

void printArray(const double *x, const int m) {
    for (int i = 0; i < m; i++) {
        std::cout << x[i];
        if (i != m-1) std::cout << ", ";
        else std::cout << std::endl;
    }
}

double minArray(double *array, int n)
{
    size_t i;
    double minimum = array[0];
    for (i = 1; i < n; ++i) {
        if (minimum > array[i]) {
            minimum = array[i];
        }
    }
    return minimum;
}

double maxArray(double *array, int n)
{
    size_t i;
    double maximum = array[0];
    for (i = 1; i < n; ++i) {
        if (maximum < array[i]) {
            maximum = array[i];
        }
    }
    return maximum;
}

int f(double i)
{
    return (int) pow(-1, i);
}

// write given points to dataPoints.dat
void writeDataPoints(double *x_i, double *y_i, int m)
{
    std::ofstream dataPoints;
    dataPoints.open("images/dataPoints.dat");

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

    dataPoints << "#m=0,S=3\n";
    for (int i = 0; i < m; i++) {
        dataPoints << x_i[i] << " " << y_i[i] << "\n";
    }

    dataPoints.close();
}

void fillFArrays(double *x_i, double *y_i, int n)
{
    for (int i = 0; i < n; i++) {
        x_i[i] = i - 10;
        y_i[i] = f(i - 10);
    }
}

void polynomialAndSplineSolutions(int m, double *x_i, double *y_i)
{
    std::ofstream polynomialInterp, cubicSplineNatural, cubicSplinePeriodic;
    polynomialInterp.open("images/polynomialInterp.dat");
    cubicSplineNatural.open("images/cubicSplineNatural.dat");
    cubicSplinePeriodic.open("images/cubicSplinePeriodic.dat");

    if (!polynomialInterp.is_open()) {
        return;
    }

    // SETUP INTERPOLANTS

    // allocate polynomial interpolant
    gsl_spline *interp_poly = gsl_spline_alloc(gsl_interp_polynomial, m);
    // allocate natural cubic spline interpolant
    gsl_spline *interp_cspline_n = gsl_spline_alloc( gsl_interp_cspline , m);
    gsl_spline *interp_cspline_p = gsl_spline_alloc( gsl_interp_cspline_periodic , m);

    // initialize interpolants
    gsl_spline_init (interp_poly, x_i, y_i, m);
    gsl_spline_init ( interp_cspline_n , x_i, y_i, m);
    gsl_spline_init ( interp_cspline_p , x_i, y_i, m);

    // WRITE POINTS TO FILES

    // mark following points with a plus
    polynomialInterp    << "#m=0,S=3\n";
    cubicSplineNatural  << "#m=0,S=3\n";
    cubicSplinePeriodic << "#m=0,S=3\n";
    for (int i = 0; i < m; i++) {
        polynomialInterp    << x_i[i] << " " << y_i[i] << std::endl;
        cubicSplineNatural  << x_i[i] << " " << y_i[i] << std::endl;
        cubicSplinePeriodic << x_i[i] << " " << y_i[i] << std::endl;
    }

    // connect the following points with a line
    polynomialInterp    << "#m=1,S=0\n";
    cubicSplineNatural  << "#m=1,S=0\n";
    cubicSplinePeriodic << "#m=1,S=0\n";

    double interpValue;
    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    for (double x = minArray(x_i, m); x <= maxArray(x_i, m); x += 0.001) {
        interpValue = gsl_spline_eval(interp_poly, x, acc);
        polynomialInterp << x << " " << interpValue << "\n";

        interpValue = gsl_spline_eval( interp_cspline_n , x, acc);
        cubicSplineNatural << x << " " << interpValue << "\n";

        interpValue = gsl_spline_eval( interp_cspline_p , x, acc);
        cubicSplinePeriodic << x << " " << interpValue << "\n";
    }
    // free the memory and close the files
    gsl_interp_accel_free (acc);
    gsl_spline_free (interp_poly);
    gsl_spline_free ( interp_cspline_n );
    gsl_spline_free ( interp_cspline_p );
    polynomialInterp.close();
    cubicSplineNatural.close();
    cubicSplinePeriodic.close();
}

// solve overdetermined matrix where m = amount of points, n = amount of unknowns, x_i = array of x values in [a, b], y_i
    = array of y values
void leastSquaresApproximation(int m, int n, double *x_i, double a, double b, double *y_i)
{
    std::stringstream ss; ss << n;

    double x_rescaled[m];
    // rescale the x values we will work with from [a, b] to [-1, 1]
    for (int i = 0; i < m; i++) {
        x_rescaled[i] = ((1 - -1) * (x_i[i] - a) / (b - a)) - 1;
    }
    a = -1;
    b = 1;

    // files to write data points for graphs to
    std::ofstream output, approximation;
    output.open("output/leastSquares_degree_" + ss.str() + ".txt");
    approximation.open("images/leastSquares_degree_" + ss.str() + ".dat");

    if (!output.is_open()) {
        std::cout << "Could not open file 'leastSquares_degree_.txt', make sure the output folder exists" << std::endl;
        return;
    }

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

    // initialize matrices A and Y for the equation A lamda = Y
    gsl_matrix *A = gsl_matrix_alloc(m, n);
    gsl_vector *Y = gsl_vector_alloc(m);

    // put the input data into matrix A and vector Y
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            gsl_matrix_set(A, i, j, gsl_pow_int(x_rescaled[i], j));
        }
        gsl_vector_set(Y, i, y_i[i]);
    }

    // initialize work matrices and vectors
    gsl_vector *tau = gsl_vector_alloc(n),
               *X = gsl_vector_alloc(n),
               *R = gsl_vector_alloc(m),
               *S = gsl_vector_alloc(n),
               *work = gsl_vector_alloc(n);
    gsl_matrix *QR = gsl_matrix_alloc(m, n),
               *U = gsl_matrix_alloc(m, n),
               *V = gsl_matrix_alloc(n, n);

    gsl_matrix_memcpy(QR, A);
    gsl_matrix_memcpy(U, A);

    gsl_linalg_QR_decomp(QR, tau);
    gsl_linalg_QR_lssolve(QR, tau, Y, X, R);

    printMatrix(A, "input A", output);
    printVector(Y, "input Y", output);
    printMatrix(QR, "QR, received by QR decomposition", output);
    printVector(X, "X, solution found by solving after QR decomposition", output);
    printVector(R, "residual R = y − Ax", output);

    // add following invisible points for better automatic graph generation
    approximation << "#m=0,S=0\n";
    approximation << 1.1 << " " << 1.1;
    approximation << 1.1 << " " << −1.1;
    approximation << −1.1 << " " << 1.1;
    approximation << −1.1 << " " << −1.1;

    // mark the following points on the graph with a plus sign
    approximation << "#m=0,S=3\n";

    // output given data points to files
    for (int i = 0; i < m; i++) {
        approximation << x_rescaled[i] << " " << y_i[i] << std::endl;
    }

    // connect the following data points with a line
    approximation << "#m=1,S=0\n";

    // calculate new values for the graphs
    for (double x = a; x <= b + 0.001; x = x + 0.001) {
        double y = gsl_vector_get(X, n−1);
        // use horners method to calculate the y values
        for (int i = n−1; i > 0; i−−) {
            y = y∗x + gsl_vector_get(X, i−1);
        }

        approximation << x << " " << y << std::endl;
    }

    // the condition number we will use is max(S) / min(S)
    gsl_linalg_SV_decomp(U, V, S, work);
    double condNumber, minS, maxS;
    minS = gsl_vector_get(S, 0);
    maxS = gsl_vector_get(S, 0);
    for (int j = 0; j < n; j++) {
        if ( gsl_vector_get (S, j) < minS) minS = gsl_vector_get(S, j);
        if ( gsl_vector_get (S, j) > maxS) maxS = gsl_vector_get(S, j);
    }
    condNumber = fabs(maxS) / fabs(minS);
    output << "Calculating condition number by: abs(max(singular values)) / abs(min(singular values)):\n\t";
    output << "Condition number: " << condNumber << std::endl;

    // free the memory and close the files
    gsl_matrix_free (A);
    gsl_matrix_free (QR);
    gsl_matrix_free (U);
    gsl_matrix_free (V);
    gsl_vector_free (Y);
    gsl_vector_free (tau);
    gsl_vector_free (X);
    gsl_vector_free (R);
    gsl_vector_free (S);
    gsl_vector_free (work);
    output.close();
    approximation.close();
}

double fourierCoefficient (int j, int n, double ∗t_i, double ∗x_i, bool a) {
    double result = 0;

    for (int k = 0; k < n; k++) {
        result += x_i[k] ∗ ((a) ? gsl_sf_cos (j ∗ t_i [k]) : gsl_sf_sin (j ∗ t_i [k]));
    }

    result ∗= 2.0/(double) n;
    return result;
}

void fourierSolve (int n, double ∗t_i, double ∗x_i, int m, std::ofstream &out) {
    double coeff[2 ∗ m + 1];
    coeff [0]  =  fourierCoefficient (0,  n,  t_i ,  x_i ,  true)  /  2;

    // generate  coefficients
    for  (int  i  = 1; i  <= m; i++) {
        int  j  = 2 * i;
        coeff [j−1] =  fourierCoefficient (i ,  n,  t_i ,  x_i ,  true);
        coeff [j]     =  fourierCoefficient (i ,  n,  t_i ,  x_i ,  false );

        // if  n == 2m, divide last a  coefficient  by 2.  last  b  coefficient   will  be 0
        if  (i  == m and n == 2*m) {
            coeff [j]   /= 2;
            coeff [j+1] = 0;
        }
    }

    // print  out the approximation function
    /*std :: cout << "n = " << n << " approximation: \n\t";
    std :: cout << "f(x) = " << coeff[0] << " + " ;
    for  (int  i  = 1; i  <= m; i++) {
        int  j  = 2 * i;
        std :: cout << coeff[j−1] << "*cos(" << i << "x) + ";
        std :: cout << coeff[j]     << "*sin(" << i << "x)";
        std :: cout << ((i != m) ? " + " : "\n");
    }
    std :: cout << std::endl;*/

    out << "#m=1,S=0\n";
    // generate approximated values
    for  (double x = 0; x < 2*M_PI + 0.001; x = x + 0.001) {
        double y = coeff [0];
        for  (int  i  = 1; i  <= m; i++) {
            int  j  = 2 * i;
            y += coeff[j−1] *  gsl_sf_cos (i  * x);
            y += coeff[j]     *  gsl_sf_sin (i  * x);
        }

        out << x << " " << y << std::endl;
    }
}

// n given points,  equidistant  points  t_i ,  corresponding data values  x_i
void trigonometricApproximation(int n, int m, double *t_i ,  double a,  double b,  double *x_i)  {
    std :: stringstream mm, nn;
    mm << m;
    nn << n;

    std :: ofstream out;
    out.open("images/trigApprox_n_" + nn.str() + "_m_" + mm.str() + ".dat");
    out << "#m=0,S=3\n";

    double  t_rescaled [n];
    // rescaled  value  = (value − (old min)) * (new range)/(old range)) + (new min)
    // new min = 0
    double new_max;
    if  (b − a > 2 * M_PI) {
        new_max = (n−1) * (2*M_PI / n);
    }  else  {
        new_max = 2*M_PI;
    }
    double range_multiplier  = (new_max − 0) / (b − a); // (new range) / (old range)

    for  (int  i  = 0; i  < n; i++) {
        t_rescaled [i]  = (t_i [i]  − a) * range_multiplier ;
        out << t_rescaled[i]   << " " << x_i[i]  << std::endl;
    }

    //std :: cout << "n = " << n << " equidistant points as time values in [" << a << ", " << b << "] rescaled to [0 ,
          2pi]: \n\t";
    //printArray(t_rescaled ,  n);

    fourierSolve (n,  t_rescaled ,  x_i ,  m, out);
    out. close () ;
```

```
}
```

## D createImages.sh

```
#!/bin/bash

DIR="images"
IMAGEFORMATTING="-T png --bitmap-size 820x820 -h 0.8 -w 0.8 -r 0.125 -u 0.10 -f 0.028 --title-font-size
    0.055"
IMAGEFORMATTING="-T png --bitmap-size 820x820"

echo "Creating images ..."

graph -L "Given data points"                -x -11.0 11.0 -y -1.2 1.2 $IMAGEFORMATTING <
    $DIR/dataPoints.dat > $DIR/dataPoints.png
graph -L "Polynomial interpolation"         -x -11.0 11.0            $IMAGEFORMATTING <
    $DIR/polynomialInterp.dat > $DIR/polynomialInterp.png
graph -L "Polynomial interpolation"         -x -11.0 11.0 -y -100 100 $IMAGEFORMATTING <
    $DIR/polynomialInterp.dat > $DIR/polynomialInterp_zoomed_1.png
graph -L "Polynomial interpolation"         -x -11.0 11.0 -y -1.5 1.5 $IMAGEFORMATTING <
    $DIR/polynomialInterp.dat > $DIR/polynomialInterp_zoomed_2.png
graph -L "Natural cubic spline interpolation"  -x -11.0 11.0 -y -1.2 1.2 $IMAGEFORMATTING <
    $DIR/cubicSplineNatural.dat > $DIR/cubicSplineNatural.png
graph -L "Periodic cubic spline interpolation" -x -11.0 11.0 -y -1.2 1.2 $IMAGEFORMATTING <
    $DIR/cubicSplinePeriodic.dat > $DIR/cubicSplinePeriodic.png

for n in 2 4 7 10
do
   graph -L "Least squares approx with n = "$n -x -1.1 1.1 -y -1.1 1.1 $IMAGEFORMATTING <
       $DIR/leastSquares_degree_$n.dat > $DIR/leastSquares_degree_$n.png
done

for n in 15 20 21
do
   graph -L "Least squares approx with n = "$n -x -1.1 1.1           $IMAGEFORMATTING <
       $DIR/leastSquares_degree_$n.dat > $DIR/leastSquares_degree_$n"_zoomed_out".png
   graph -L "Least squares approx with n = "$n -x -1.1 1.1 -y -1.1 1.1 $IMAGEFORMATTING <
       $DIR/leastSquares_degree_$n.dat > $DIR/leastSquares_degree_$n.png
done

for m in 2 4 7 9 10
do
   graph -L "Trigonometric approx with n=21, m="$m -x -0.3 6.7 -y -2.5 2.5 $IMAGEFORMATTING <
       $DIR/trigApprox_n_21_m_$m.dat > $DIR/trigApprox_n_21_m_$m.png
done

#graph -x -0.3 6.7 $IMAGEFORMATTING < $DIR/trigApprox_n_5_m_1.dat > $DIR/trigApprox_n_5_m_1.png
#graph -x -0.3 6.7 $IMAGEFORMATTING < $DIR/trigApprox_n_5_m_2.dat > $DIR/trigApprox_n_5_m_2.png

#graph -x -0.3 6.7 $IMAGEFORMATTING < $DIR/trigApprox_n_8_m_2.dat > $DIR/trigApprox_n_8_m_2.png
#graph -x -0.3 6.7 $IMAGEFORMATTING < $DIR/trigApprox_n_8_m_4.dat > $DIR/trigApprox_n_8_m_4.png

echo "Done"
```