Wetenschappelijk Programmeren: Opdracht 2de zit

Ruben Van Assche - S0122623

23 augustus 2017

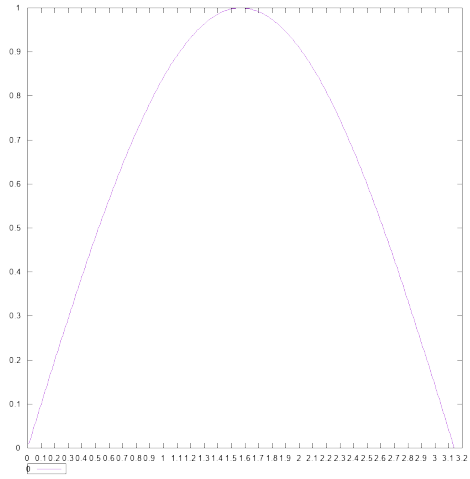
1 Opgave 1

Bereken een een polynomische interpolant en een spline over het interval [0,π] voor de functie sin(x). Probeer de absolute fout $\leq 10^{-10}$ te krijgen.

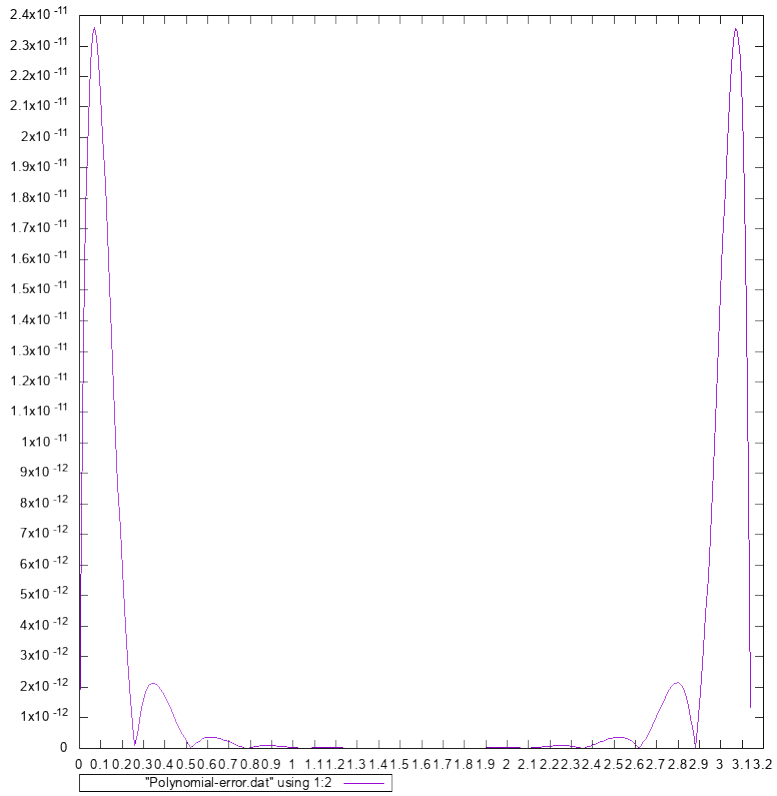
1.1 Polynomische Interpolant

Hierbij wordt gebruik gemaakt van de GSL Spline functies om een juiste interpolant $p(x)$ te bekomen. Vervolgens wordt $p(x)$ uitgerekend op het interval [0, π], waarbij de x waarde met stappen van 0.01 verhoogd wordt. Telkens wanneer de fout : $|f(x) - p(x)| \geq 10^{-10}$ wordt de procedure stilgelegd en een interpolant $p(x)$ met 1 datapunt meer dan de vorige interpolant wordt gegeneerd. Hiera de interpolant opnieuw wordt berekend op het interval [0, π] totdat $|f(x) - p(x)| \leq 10^{-10}$ waar is.

Deze methode levert een interpolant van de 12de graad op waarbij gebruik wordt gemaakt van 13 datapunten. De totale absolute fout voor deze interpolant is gelijk aan 2.396666307598349446 6e-12. Hieronder staan de grafieken met de geplotte interpolant en de foutenkromme.



Figuur 1: Polynomishe interpolant van 12de graad voor sin(x)



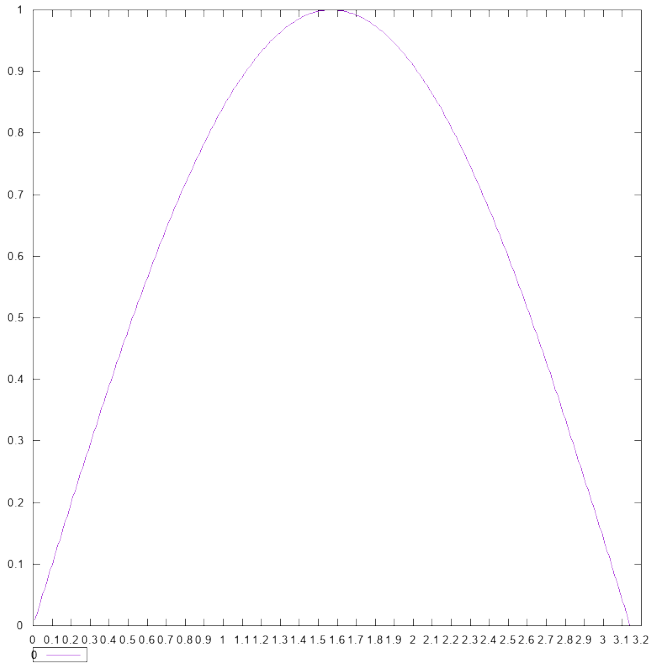
Figuur 2: Foutenkromme voor de polynomische interpolant

Bij de foutenkromme zien we duidelijk het Runghe fenomeen optreden. Aan de buitenkanten van het interval begint de foutenkromme te oscilleren. We zullen zien bij een spline dat ze oscillatie langs de buitenkant veel kleiner is. Een andere oplossing was het verminderen van datapunten(wat niet nuttig was in dit geval gezien dat deze een hogere fout opleveren), een paar datapunten weg laten of een andere basis kiezen zoals bijvoorbeeld de chebyshev basis.

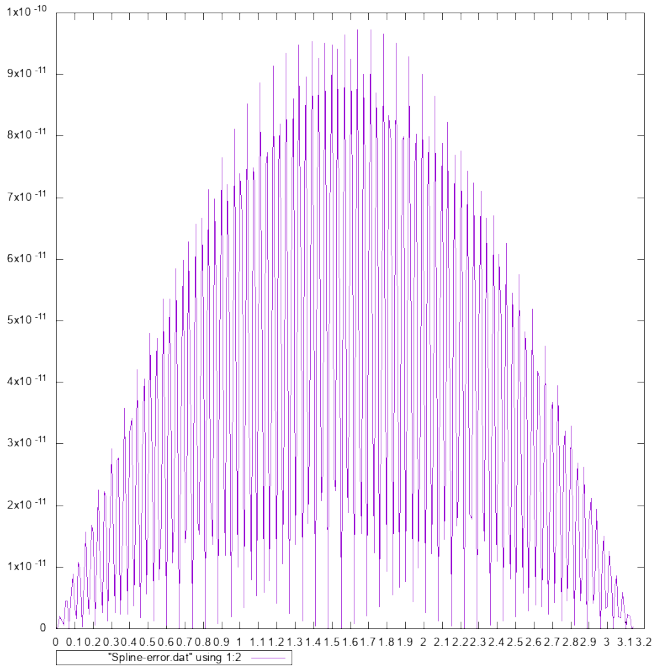
1.2 Natuurlijke Kubische Spline

Een andere manier om sin(x) te benaderen gaat via een natuurlijke kubische spline. We gebruiken dezelfde procedure als bij de polynomische interpolant maar gebruiken dan de GSL functies voor het tekenen van een spline die we $s(x)$ noemen. Wat opvalt is dat we deze keer geen gebruik maken van 13 datapunten maar wel 226!

Een natuurlijke kubische spline heeft in haar begin en eindpunt als 2de afgeleide 0. Dit is een goed idee ten eerste omdat het de berekening van een spline makkelijker maakt. We zouden een waarde kunnen geven voor de 2de afgeleide in het begin en eindpunt maar dit zou de functie meer convex of concaaf maken naar het einde toe(=de boog wordt groter). Hieronder de geplotte spline en zijn foutenkromme.



Figuur 3: Een natuurlijke kubische spline die sin(x) benadert met 226 datapunten



Figuur 4: Foutenkromme voor de spline

We zien dat de geplotte spline de sin(x) functie mooi benadert. Wanneer we kijken naar de foutenkromme van de spline merken we veel meer oscillaties dan de polynomische interpolant maar deze zijn veel beter gedistribueerd over het interval. Dit maakt dat de fout ongeveer overal hetzelfde is t.o.v. de polynomische interpolant waarbij de fout enkel langs de uiteinden zit. Wat wel op te merken valt is dat de fout in de spline lichtjes groter is dan die van de polynomische interpolant.

2 Opgave 2

Berken nu met deze n + 1 datapunten van de natuurlijke kubische spline een kleinste kwadraten parabool benadering van de vorm $q(x) = a + bx + cx^2$ en bespreek.

2.1 De kleinste kwadraten parabool benadering

Door gebruik te maken van de GSL Multifit Lineair functies is het zeer simpel om een kleinste kwadraten benadering te plotten van de vorm $q(x) = a + bx + cx^2$. We geven volgende matrix mee waarop we de kleinste kwadraten procedure uitvoeren. De x waarden komen voort uit de gegeven n+1 datapunten.

$$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix}$$

Als y waarden wordt er gebruik gemaakt van volgende vector:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

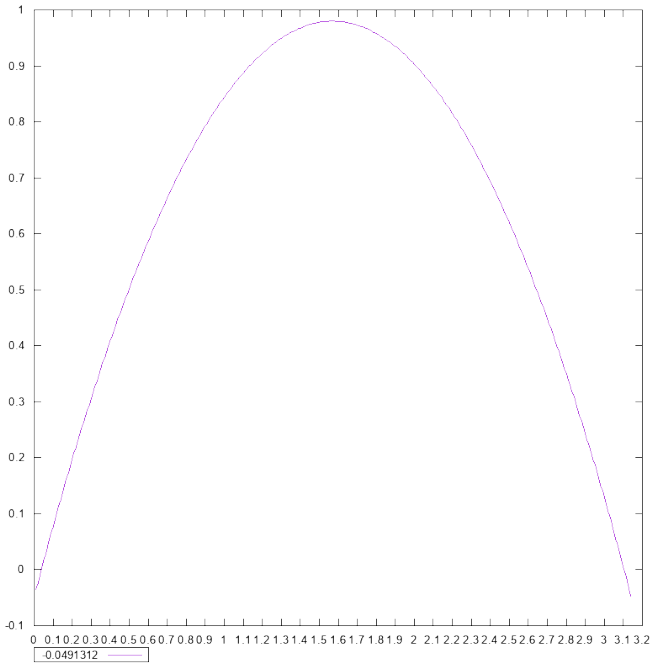
Dit levert dan volgend stelsel op:

$$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} * \begin{bmatrix} c_0 \\ c_1 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

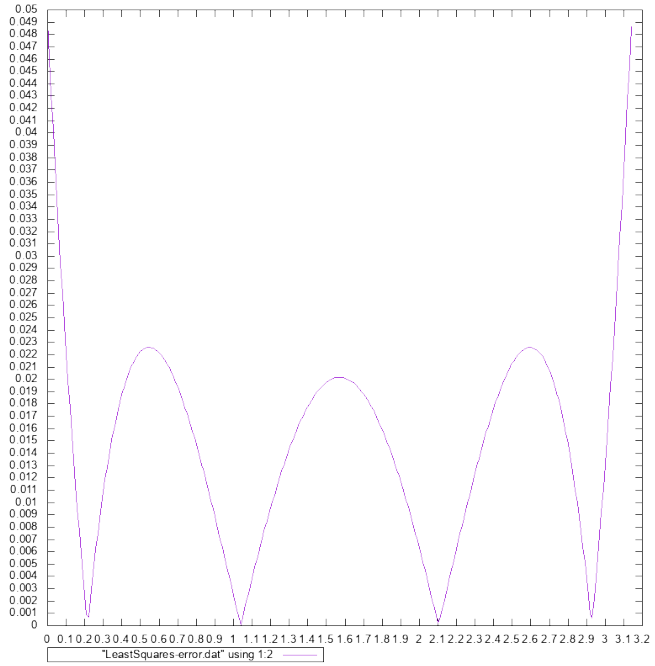
Wanneer we GSL dit stelsen laten oplossen d.m.v. de kleinste kwadraten methode krijgen we hetvolgende polynoom:

$$q(x) = -0.049131235912031057111 + 1.3101111168458918765x - 0.417021320491334779 31x^2$$

Wanneer we deze functie $q(x)$ dan plotten over het interval [0, π] verkrijgen we volgende plot + tevens ook de foutenkromme $|f(x) - q(x)|$:



Figuur 5: Een kleinste kwadraten benadering van sin(x)



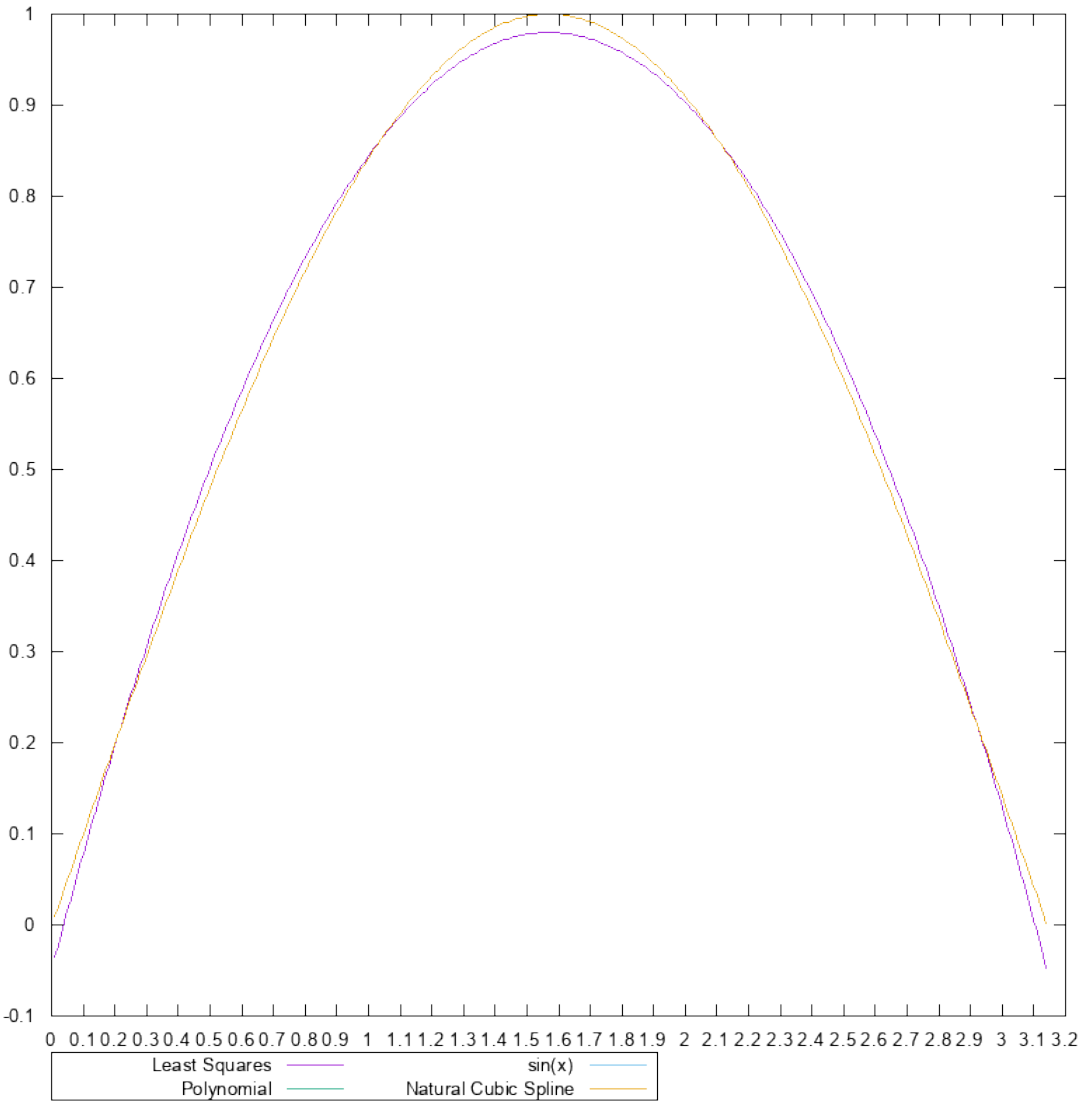
Figuur 6: Foutenkromme voor de kleinste kwadraten benadering

We zien dat de foutenkromme een speciale vorm heeft, met momenten is de

fout nul en met andere momenten is de fout hoog. Zeer hoog zelfs de fout licht een stuk hoger dan de foout van de polynomische interpolant en de natuurlijke kubische spline. Gezien de absolute fout van de absolute waarde van de fout neemt(en er dus geen y-waarden onder 0 zijn). Kunnen we veronderstellen dat de plot van de kleinste kwadraten functie soms een positieve en soms een negatieve fout zal hebben.

2.1.1 Vergelijkingen

Wanneer we nu de 3 benaderingen plotten op 1 grafiek + een plot van sin(x) bekomen we volgende plot:



Figuur 7: Alle benaderingen + sin(x)

Wat opvalt is dat de polynimische interpolant, natuurlijke kubische spline en sin(x) samenvallen. Natuurlijk is dit niet zo, we hebben al eerder gezien dat de natuurlijke kubische spline haar fout anders gedistribueerd is + lichtjes groter is. Dit zijn immers kleine verschillen welke niet opvallen op de huidige grafiek, mocht men deze vergtoeten zou men duidelijker een verschil zien tussen deze 3 functies.

De functie die er wel uit springt is de kleinste kwadraten parabool benadering, deze loopt niet gelijk met de 3 andere en maakt zoals eerder aangehaald een zeer grote fout. Wat opvalt is dat de fout soms ook 0 is: dit zijn de punten waar de sin(x) en de kleinste kwadraten benaderin elkaar snijden. Zoals eerder aangehaald is de fout soms postitief en soms negatief. Dit zien we ook in de plot: soms light de kleinste kwadraten benadering boven de sin(x) functie en soms eronder.

Er is geen samenvattende plot van de foutenkrommes van de 3 benaderingen omdat deze niet duidelijk genoeg zou zijn en aan de hand van de plot van de 3 benaderingen al veel uitgelegd kan worden.

3 Opgave 3

Functie $f(x)$ is gelijk aan -1 op het interval [-π, 0] en +1 op het interval [0, π], bereken enkele benaderingen. Maak gebruik van $n \geq 120$ equidistante punten.

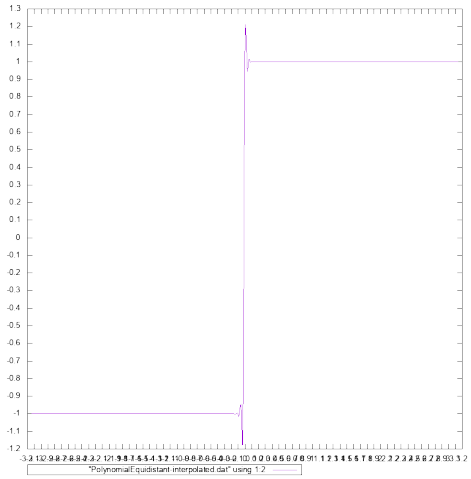
3.1 Veelterm Interpolanten

Er werd gevraagd om enkele veelterm interpolanten te berekenen. Om te bepalen welke graad te gebruiken berekenen we telkens de relative fout :

$$\left| \frac{x_{berekend} - x}{x} \right|$$

Uiteindelijk kiezen we 2 manieren voor het selecteren van punten(zie opgave 1, hoe meer punten geselecteerd, hoe harder het runghe fenomeen kan optreden).

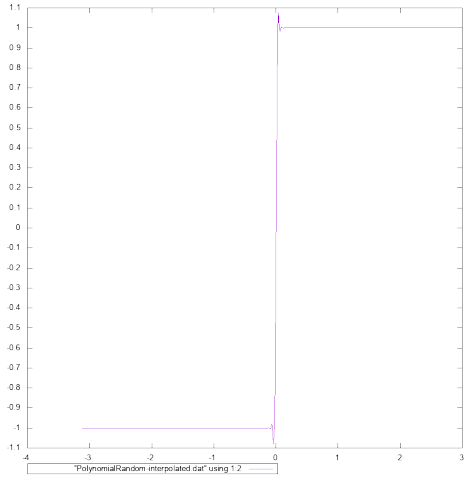
Ten eerste selecteren we punten op een equidistante afstand van elkaar, we laten deze afstand steeds kleiner worden en dus in het begin worden niet alle punten gebruikt. Wanneer we deze procedure uitvoeren op 241 punten valt op dat de relatieve fout het kleinst is wanneer we een polynomische interpolant van graad 240 construeren met 241 datapunten. Dit levert relatieve fout op van : 4.2881891672892731293 .



Figuur 8: Een polynomische equidistante interpolant van f(x) van graad 240

De interpolant benadert de functie vrij goed behalve rond de x-waarde 0.

Een andere manier van punten selecteren is door deze niet equidistant te kiezen maar random over het interval te selecteren. Wanneer we deze procedure toepassen verkijgen we een polynoom van de graad 183 met 182 datapunten waarbij de relative fout 2.5545165758846741788 is. We zien dit ook op de plot van de interpolant:



Figuur 9: Een polynomische random interpolant van f(x) van graad 240

Waarbij de equidistante punten de y-waarden rond 0 nog tot ongeveer 1.3 lopen door osccilaties, lopen ze bij random gekozen punten maar tot 1.1 een duidelijk verschil! Daarom kiezen we dus voor een polynomishce interpolant met random gekozen punten inplaats van equidistante punten.

3.2 Kleinste kwadraten veeltermbenadering

TODO

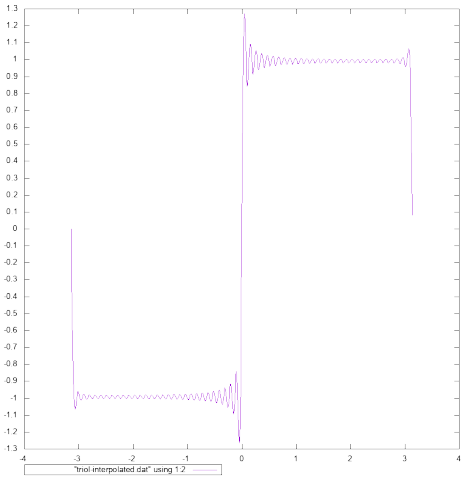
3.3 Trigonometrische veelterminterpolant

Voor de trigoniometrische veelterminterpolant berekenen we enkele coefficienten met $i = 1 \dots m$:

$$a_i = \frac{2}{n} \sum_{k=0}^{n-1} (y_k * cos(i * x_k))$$

$$b_i = \frac{2}{n} \sum_{k=0}^{n-1} (y_k * sin(i * x_k))$$

Bij trigonometrische veelterminterpolatie stellen we dat n = 2*m dus in dit geval kiezen we voor n = 122 en m = 61. Hierdoor verkrijgen we een interpolant met als relatieve fout : 19.13715151604820619. Welke tamelijk groot is t.o.v. de polynomische interpolant. Hieronder een plot:



Figuur 10: Een trigonometrische veelterminterpolant van f(x)

Om de plot te te tekenen wordt gebruik gemaakt van volgende functie :

$$p(x) = \frac{a_0}{2} + \sum_{i=1}^{m} (a_i * cos(i * x)) + \sum_{i=1}^{m} (b_i * sin(i * x))$$

Wat handig is dat de opgave bestaat uit equidistante punten : TODO

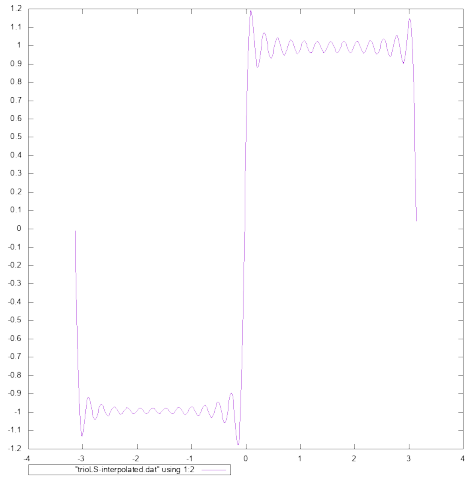
3.4 Trigonometrische kleinste-kwadraten approximant

Hierbij stellen we dat $n \geq 2 * m$ en dus hebben we gekozen voor n = 121 en m = 25. We kiezen hierbij voor een oneven aantal termen(25) hierbij worden de coefficienten $a_i enb_i$ niet meer berekend van $i = 1 \ldots m$ maar van $i = 1 \ldots m-1$. Tevens komt er een extra factor bij : $\frac{1}{2} * a_m * cos(m * x)$. Dit levert volgende functie op:

$$p(x) = \frac{a_0}{2} + \sum_{i=1}^{m-1} (a_i * cos(i * x)) + \sum_{i=1}^{m-1} (b_i * sin(i * x)) + \frac{1}{2} * a_m * cos(m * x)$$

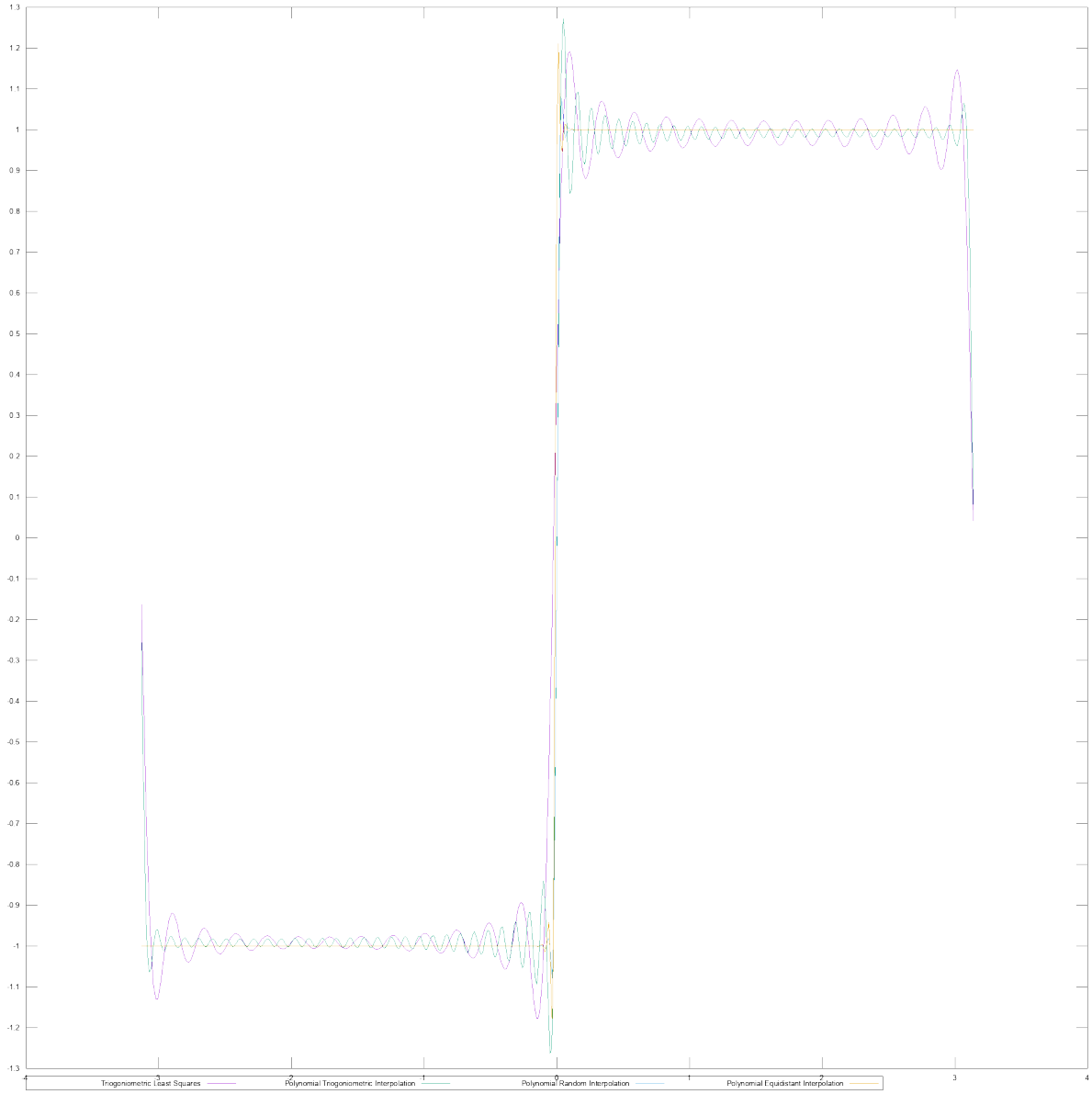
Dit verschil tussen een oneven/even aantal termen zorgt ervoor dat : TODO

De relative fout van deze benadering is : 34.4556406247306981O7. Wanneer we de functie $p(x)$ plotten krijgen we volgende functie :



Figuur 11: Trigonometrische kleinste-kwadraten approximant van f(x)

3.5 Vergelijken



Figuur 12: Alle benaderingen van opgave 4

Wanneer we nu alle benaderingen naast elkaar zetten op een afbeelding zien we dat de Polynomische Interpolant met random gekozen punten f(x) het beste benadert, gevolgd door de Polynomische interpolant met equidistante punten.

Daarna de trigonometrische interpolant en vervolgens de Trigonometrische kleinstekwadraten approximant. Dit zagen we ook eerder in de fouten die de beanderingen hadden.

4 Opgave 4

Gebruik de n+1 datapunten die de polynomische interpolant p(x) in opgave 1 opleverde en stel een stelsel interpolatievoorwaarden op, bereken het conditiegetal en los het op met GEPP. Het stelsel zal volgende vorm hebben:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots x_0^n \\ 1 & x_1 & x_1^2 & \dots x_1^n \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \dots x_n^n \end{bmatrix} * \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Hierbij zijn de x en y waarden al gekend uit de n + 1 datapunten uit opgave 1. Dit levert het volgende stelsel op:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3.1 & 9.9 & 31 & 97 & 3.1e+02 & 9.6e+02 & 3e+03 & 9.5e+03 & 3e+04 & 9.4e+04 & 2.9e+05 & 9.2e+05 \\ 1 & 0.5 & -2.5 & -12 & -43 & -1.4e+02 & -4.7e+02 & -1.5e+03 & -4.7e+03 & -1.5e+04 & -4.7e+04 & -1.5e+05 & -4.6e+05 \\ 1 & 0.75 & 0.75 & -1.5 & -10 & -49 & -2e+02 & -7.5e+02 & -2.6e+03 & -9e+03 & -3e+04 & -9.8e+04 & -3.2e+05 \\ 1 & 0.17 & 0.56 & -0.99 & -2.6 & -20 & -1e+02 & -4.2e+02 & -1.6e+03 & -5.6e+03 & -1.9e+04 & -6.4e+04 & -2.1e+05 \\ 1 & 0.92 & 0.31 & 0.68 & 0.2 & -1.2 & -13 & -83 & -4.3e+02 & -2e+03 & -8.3e+03 & -3.3e+04 & -1.2e+05 \\ 1 & 0.33 & 0.89 & -0.79 & 0.57 & 0.65 & 1.4 & 17 & 1.2e+02 & 6.3e+02 & 3e+03 & 1.3e+04 & 5.2e+04 \\ 1 & 0.083 & 0.31 & -0.68 & 0.79 & -0.44 & -0.98 & 1.1 & 13 & 93 & 5.1e+02 & 2.4e+03 & 1.1e+04 \\ 1 & 0.83 & 0.56 & 0.99 & 0.14 & 0.65 & 0.14 & 0.29 & 0.76 & 11 & 92 & 5.9e+02 & 3.2e+03 \\ 1 & 0.58 & 0.97 & 0.43 & -0.12 & -0.35 & -0.31 & -0.32 & -0.73 & 0.44 & 7.1 & 66 & 4.6e+02 \\ 1 & 0.25 & 0.75 & -1 & 0.87 & 0.49 & 0.87 & -0.29 & -0.23 & 0.72 & -0.33 & -5.6 & -54 \\ 1 & 0.67 & 0.89 & 0.79 & -0.11 & -0.39 & -0.26 & -0.35 & -0.93 & 0.85 & -0.3 & 0.13 & 2.4 \\ 1 & 0.42 & 0.97 & -0.43 & 0.25 & 0.42 & 0.56 & 0.19 & 0.29 & -0.67 & -0.46 & 0.63 & -0.063 \end{bmatrix}$$

Dit levert volgend stelsel op :

$$A * \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0.25881904510252073948 \\ 0.49999999999999994449 \\ 0.70710678118654746172 \\ 0.86602540378443859659 \\ 0.96592582628906820119 \\ 1 \\ 0.96592582628906831221 \\ 0.86602540378443881863 \\ 0.70710678118654757274 \\ 0.50000000000000033307 \\ 0.25881904510252101703 \\ 1.2246467991473532072e-16 \end{bmatrix}$$

Vervolgens berekenen we het conditiegetal van de matrix A om vervolgens te kijken naar de orde van de relatieve fout van de oplossing. Het berekenen van het conditiegetal gebeurt d.m.v. een singuliere waarden decompositie(GSL Linalg SV Decomp) waarbij de minimale en maximale singuliere waarde wordt gezocht en deze door elkaar worden gedeeld. Wanneer dit wordt uitgevoerd komen we op volgende:

$$\kappa(A) = 418891964854.25164795 \approx 4 * 10^{12}$$

GEPP is niet stabiel:

$$\|\tilde{x} - x^\star)\| \leq n^3 * ULP * \kappa(A) * \|x^\star\|$$

Hieruit halen we dat de onvermijdelijke relative afrondingsfout $n^3 * ULP$ wordt opgeblazen op $x^\star$ (de exacte oplossing) met een factor $\kappa(A)$, het conditiegetal. Dit grote conditiegetal zorgt ervoor dat de fout wordt vergroot met een factor 10^{12}. Op mijn computer was ULP $= 2.2 * 10^{-16}$ en $n^3 = 2197 = 2 * 10^2$. Dit zorgt ervoor dat de fout van GEPP ongeveer gelijk is aan:

$$10^2 * 0^{12} * 10^{-16} = 10^{-2} = 0.01$$

Dit is een zeer hoge fout en er kan dus verwacht worden dat resultaten niet altijd even correct zijn.

Als laatste nog de uitkomstenvector van de GEPP procedure, hiermee kan vervolgens een interpolatiepolynoom $p(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$ worden opgesteld om zo de sin(x) functie benaderen.

$$C = \begin{bmatrix} 0 \\ 0.99999999913751358438 \\ 1.0761642333038017988e-08 \\ -0.16666672378000291221 \\ 1.7343027474583990613e-07 \\ 0.0083329935383840807173 \\ 4.5700695425411687 47e-07 \\ -0.0001988490828597052 2837 \\ 3.0059277657056513061e-07 \\ 2.6061122447290966013e-06 \\ 5.309884908907600906e-08 \\ -3.8025144634222871813e-08 \\ 2.0172966316920489529e-09 \end{bmatrix}$$

4.1 Opgave 5

Bereken het volume van een cylinder met als grondvlak $\Omega = (x - 0.5)^2 + (y - 0.5)^2 \leq 0.25$ en als hoogte $ln(x + y)$ tot 2 beduidende cijfers correct.

4.1.1 MAPLE

Om te kunnen weten of de berekende oplossing tot 2 beduidende cijfers correct is wordt gebruik gemaakt van MAPLE om een exacte oplossing voor het volume van de cylinder te vinden. Dit kan d.m.v. volgende code:

```
with(VectorCalculus);
int(abs(ln(x+y)), [x, y] = Circle({0.5,0.5}, .5));
```

Wanneer we dit uitrekenen bekomen we dat het volume $I = 0.2550197403$.

4.1.2 Berekening

Om het volume te bereken maken we gebruik van Monte Carlo Integratie deze gebruikt random samples en kijkt of deze (x, y) coordinaten in het volume liggen of niet. Om deze conditie te checken geven we volgende functie aan GSL mee :

```
double func (double x[], size_t dim, void * p){
    if (dim != 2)
```

```
    {
        fprintf (stderr, "error: dim != 2");
        abort ();
    }

    if(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2) <= 0.25){
        return fabs(log(x[0] + x[1]));
    }else{
        return 0.0;
    }
}
```

Indien het(x,y) coordinaat in het grondvlak ligt, zal het een hoogte mee krijgen bepaald door$ln(x + y)$ ligt het niet in het grondvlak, dan krijgt het de hoogte 0 mee. Wanneer we dit uitrekenen en met een seed = 123 en random generator = mrg. Is met 691 samples de inhoud gelijk aan 0.25936620629454121278 wat de gelijkenis van 0.25 heeft met de MAPLE berekening.

5 Code

../one.cpp - Opgave 1, 2 en 4

```
#include <iostream>
#include <list>
#include <vector>
#include "utils.h"

// GSL
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_linalg.h>

double calculateConditionNumber(gsl_matrix* input, int size){
    gsl_vector* vector = gsl_vector_alloc(size);
    gsl_matrix* tempmatrix = gsl_matrix_alloc(size,size);
    gsl_matrix* m = gsl_matrix_alloc(size,size);
    gsl_matrix_memcpy(m, input);
    gsl_vector* work = gsl_vector_alloc(size);

    gsl_linalg_SV_decomp(m, tempmatrix, vector, work);

    double min = std::numeric_limits<double>::max();
    double max = -std::numeric_limits<double>::min();
    for(int i = 0; i < size;i++){
        double result = gsl_vector_get(vector, i);
        if(result > max){
            max = result;
        }
```

```
        if(result < min){
            min = result;
        }
    }

    gsl_vector_free(vector);
    gsl_matrix_free(tempmatrix);
    gsl_vector_free(work);

    return max/min;
}

// n >= 2
std::list<std::pair<double, double>> getDataPoints(int n){
    n = n - 1;
    std::list<std::pair<double, double>> points;

    double a = 0;
    double b = Utils::PI;
    double h = (a+b)/n;

    for(int i = 0;i <= n;i++){
        double x = a + i*h;
        points.push_back(std::make_pair(x, sin(x)));
    }

    return points;
}

void interpolate(int n, std::string type, const gsl_interp_type * T){
    auto points = getDataPoints(n);
    n = points.size();

    std::cout << type << " interpolation, points: " <<  n
<< std::endl;

    double x[points.size()];
    double y[points.size()];

    // Load Points
    int counter = 0;
    for(auto point : points){
        x[counter] = point.first;
        y[counter] = point.second;
        counter++;
    }

    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *interpolation = gsl_spline_alloc(T, n);
    gsl_spline_init(interpolation, x, y, n);
```

```
    double totalError = 0;
    counter = 0;

    std::list<std::pair<double, double>> plotFunction;
    std::list<std::pair<double, double>> interpolatedFunction;
    std::list<std::pair<double, double>> errorFunction;

    for(double xi = 0;xi <= Utils::PI;xi += 0.01){
        // Calculate y by using the interpolation
        double yi = gsl_spline_eval(interpolation, xi, acc);
        double fi = sin(xi);

        // Calculate the error
        double error = fabs(fi - yi);
        totalError += error;
        if(error > pow(10, -10)){
            // Try with a higher amount of samples
            interpolate(n + 1, type,  T);
            return;
        }

        plotFunction.push_back(std::make_pair(xi, fi));
        interpolatedFunction.push_back(std::make_pair(xi, yi));
        errorFunction.push_back(std::make_pair(xi, error));

        counter++;
    }

    // Write points
    Utils::writePointsFile(plotFunction, "Onegraphs/Sin-Plot.dat");
    Utils::writePointsFile(interpolatedFunction, "Onegraphs/" + type + "-interp
    Utils::writePointsFile(errorFunction, "Onegraphs/" + type + "-error.dat");


    gsl_spline_free(interpolation);
    gsl_interp_accel_free(acc);

    totalError = totalError/counter;
    std::cout << "Total Error : " << totalError << std::endl;
}

void leastSquares(int n){
    double chisq;

    gsl_matrix* X = gsl_matrix_alloc(n, 3);
    gsl_vector* y = gsl_vector_alloc(n);

    gsl_vector* q = gsl_vector_alloc(3);
    gsl_matrix* cov = gsl_matrix_alloc(3, 3);
```

```
auto points = getDataPoints(n);

int counter = 0;
for(auto point : points){
    // Set matrix
    gsl_matrix_set(X, counter, 0, 1.0);
    gsl_matrix_set(X, counter, 1, pow(point.first, 1));
    gsl_matrix_set(X, counter, 2, pow(point.first, 2));

    // Set y Vector
    gsl_vector_set (y, counter, point.second);

    counter++;
}


gsl_multifit_linear_workspace * work = gsl_multifit_linear_alloc(n, 3);
gsl_multifit_linear(X, y, q, cov, &chisq, work);
gsl_multifit_linear_free (work);

double a = gsl_vector_get(q,0);
double b = gsl_vector_get(q,1);
double c = gsl_vector_get(q,2);

std::cout << "Least Squares: a+bx+cx^2" << std::endl;
std::cout << "a : " << a << std::endl;
std::cout << "b : " << b << std::endl;
std::cout << "c : " << c << std::endl;


std::list<std::pair<double, double>> interpolatedFunction;
std::list<std::pair<double, double>> errorFunction;

for(double xi = 0;xi <= Utils::PI;xi += 0.01){
    // Calculate y by using the interpolation
    double yi = a + pow(xi, 1)*b + pow(xi, 2)*c;
    double fi = sin(xi);

    // Calculate the error
    double error = fabs(fi - yi);

    interpolatedFunction.push_back(std::make_pair(xi, yi));
    errorFunction.push_back(std::make_pair(xi, error));

    counter++;
}

// Write points
```

```
    Utils::writePointsFile(interpolatedFunction, "Onegraphs/LeastSquares-interp
    Utils::writePointsFile(errorFunction, "Onegraphs/LeastSquares-error.dat");

    gsl_matrix_free(cov);
    gsl_vector_free(q);

    gsl_vector_free(y);
    gsl_matrix_free(X);
}

void useGEPP(int n){
    std::vector<std::pair<double, double>> points;
    for(auto point : getDataPoints(13)){
        points.push_back(std::make_pair(point.first, point.second));
    }

    gsl_matrix* A = gsl_matrix_alloc(n,n);
    for(int i = 0;i < n;i++){
        for(int j = 0;j < n;j++){
            gsl_matrix_set(A, i, j, pow(points.at(i).first, j));
        }
    }

    double conditionNumber = calculateConditionNumber(A, n);
    std::cout << "Matrix A has condition number: " << conditionNumber << std::e

    gsl_vector *y = gsl_vector_alloc(n);
    for(int i =0;i < n;i++){
        gsl_vector_set(y, i, points.at(i).second);
        std::cout << points.at(i).second << std::endl;
    }

    gsl_vector *x = gsl_vector_alloc(n);
    int s;
    gsl_permutation * p = gsl_permutation_alloc(n);

    gsl_linalg_LU_decomp (A, p, &s);
    gsl_linalg_LU_solve (A, p, y, x);

    //Print the function values
    std::cout << "GEPP x results : " << std::endl;
    for(int i = 0;i < n;i++){
        std::cout << gsl_vector_get(x, i) << std::endl;
    }

    std::cout << "Matrix A : " << std::endl;
    Utils::matrixToLatex(A, n, 2);

    gsl_permutation_free(p);
    gsl_vector_free(x);
```

```
    gsl_vector_free(y);
    gsl_matrix_free(A);
}

int main() {
    std::cout.precision(20);
    interpolate(3, "Polynomial", gsl_interp_polynomial);
    interpolate(3, "Spline", gsl_interp_cspline);
    leastSquares(226);
    useGEPP(13);

    std::cout << std::numeric_limits<double>::epsilon() << std::endl;
}
```

../one.cpp - Opgave 3

```
#include <iostream>
#include <list>
#include <vector>
#include "utils.h"

// GSL
#include <gsl/gsl_spline.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_chebyshev.h>

enum TYPE{RANDOM, EQUIDISTANT};

double function(double x){
    if(-Utils::PI - 1 <= x and x < 0.0){
        return -1.0;
    }else if(0.0 <= x and x < Utils::PI + 1){
        return 1.0;
    }else{
        throw std::runtime_error("Unkown number for range");
    }
}


// n >= 120 gets one more
std::vector<std::pair<double, double>> getDataPoints(int n){
    if(n < 120){
        throw std::runtime_error("n should be more then 120");
    }

    std::vector<std::pair<double, double>> points;

    double a = -Utils::PI;
    double b = Utils::PI;
    double h = (b-a)/n;
```

```
    for(int i = 0;i <= n;i++){
        double x = a + i*h;
        points.push_back(std::make_pair(x, function(x)));
    }

    return points;
}

// Returns the error
std::vector<std::pair<double, double> > polynomialInterpolant(std::vector<std::
TYPE equidistant, double *error, bool *stopped){
    int n = degree + 1; // We need degree + 1 points for polynomial interpolati

    std::vector<std::pair<double, double>> points;
    if(equidistant == EQUIDISTANT){
        double a = 0;
        double b = generatedPoints.size();
        int h = ceil((b-a)/n);

        for(int i = 0;i < n;i++){
            int x = a + i*h;

            if(x >= generatedPoints.size()){
                // Couln't make this polynom
                *stopped = true;
                return points;
            }

            std::pair<double, double> pair = generatedPoints.at(x);
            points.push_back(pair);
        }
    }else if(equidistant == RANDOM){
        for(int i = 0;i < n;i++){
            auto pair = Utils::select_randomly(generatedPoints.begin(), generat
            points.push_back(*pair);
            generatedPoints.erase(pair);
        }

        std::sort(points.begin(), points.end());
    }else{

    }

    double x[n];
    double y[n];

    // Load Points
    int counter = 0;
    for(auto point : points){
```

```
        x[counter] = point.first;
        y[counter] = point.second;
        counter++;
    }

    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *interpolation = gsl_spline_alloc(gsl_interp_cspline, n);
    gsl_spline_init(interpolation, x, y, n);

    *error = 0.0;

    for(double xi = points.front().first;xi <= points.back().first;xi += 0.01){
        double yi = gsl_spline_eval(interpolation, xi, acc);


        *error += fabs((yi - function(xi))/function(xi));
    }


    if(equidistant == EQUIDISTANT){
        //std::cout << "Degree(" << degree <<") Polynomial Interpolation Equids
    }else if(equidistant == RANDOM){
        //std::cout << "Degree(" << degree <<") Polynomial Interpolation Random
    }else{

    }

    return points;
}

void generatePointsFileFromPolynomialInterpolated(std::vector<std::pair<double,
    int n = points.size();

    double x[n];
    double y[n];

    // Load Points
    int counter = 0;
    for(auto point : points){
        x[counter] = point.first;
        y[counter] = point.second;
        counter++;
    }

    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *interpolation = gsl_spline_alloc(gsl_interp_cspline, n);
    gsl_spline_init(interpolation, x, y, n);

    std::list<std::pair<double, double>> interpolatedFunction;
```

```
    for(double xi = points.front().first;xi <= points.back().first;xi += 0.01){
        double yi = gsl_spline_eval(interpolation, xi, acc);

        interpolatedFunction.push_back(std::make_pair(xi, yi));
    }

    Utils::writePointsFile(interpolatedFunction, "Twographs/" + name + "-interp
}

double cheb(int grade, double x){
    if(grade == 0){
        return 1.0;
    }else if(grade == 1){
        return x;
    }else if(grade > 1){
        return 2.0 * x * cheb(grade - 1, x) - cheb(grade - 2, x);
    }else{

    }
}

double polynomialLeastSquares(int n, double given){
    /*
    auto points = getDataPoints(120);
    int m = points.size();
    double a = -Utils::PI;
    double b = Utils::PI;

    // Scale to chebychev, not nesairy
    std::vector<double> scalers;
    for(int k = 1;k <= m;k++){
        double scaler = -cos(((2.0*k - 1.0)/(2.0*m))*Utils::PI);
        scalers.push_back(scaler);
    }

    // Scale to interval [ 1 , 1]
    for(int k = 1;k <= m;k++){
        double x = points.at(k-1).first;
        //double scaled = -1 + 2*((x + Utils::PI)/(2*Utils::PI));
        double scaled = (scalers.at(k-1) + 1.0)*((b - a)/2.0) + a;
        points.at(k-1).first = scaled;
        points.at(k-1).second = function(scaled);
    }


    std::vector<double> coeff;
    for(int i = 0;i <= n;i++){
        double upper = 0.0;
        double lower = 0.0;
```

```
        for(int k = 1;k <= m;k++){
            upper += (points.at(k-1).second * cheb(i, scalers.at(k-1)));
            lower += (pow(cheb(i, scalers.at(k-1)), 2));
        }

        std::cout << upper << " , " << lower << std::endl;

        coeff.push_back(upper/lower);
    }

    double result;
    // Let's calculate
    for(int i = 0;i <= n;i++){
        //result += coeff.at(i) * cheb(i, 2*((given-a)/(b-a) - 1));
        result += coeff.at(i)*given;
    }
    */

    std::vector<double> cj;
    for(int j = 0;j <= n - 1;j++){
        double result = 0.0;

        for(int k = 1;k <= n;k++){
            double xk = cos( (Utils::PI * (k - 0.5))/(n) );
            double xkj = cos( (j*Utils::PI * (k - 0.5))/(n) );
            result += function(xk)*xkj;
        }

        result = result/2.0;

        cj.push_back(result);
    }

    double result;
    for(int k = 0;k <= n-1;k++){
        result += cj.at(k)*cheb(k, given);
    }

    result -= 0.5*cj.at(0);

    return result;
}

void calculatePolynomialLeastSquares(){
    double error = 0.0;
    int n = 3;

    std::list<std::pair<double, double>> interpolatedFunction;
```

```
    for(double xi = -Utils::PI;xi <= Utils::PI;xi += 0.1){
        double yi = polynomialLeastSquares(n, xi);

        interpolatedFunction.push_back(std::make_pair(xi, yi));

        error += fabs((yi - function(xi))/function(xi));
    }

    Utils::writePointsFile(interpolatedFunction, "Twographs/PolynomialLS-interp

    std::cout << "Polynomial Least Squares n: " << n << ", error: " << error <<
}

void calculatePolynomialInterpolant(){
    double minError = pow(10,100000);
    int degree = 0;
    std::vector<std::pair<double, double> > equidistantPoints;
    for(int i = 3;i <= 240;i++){
        double error = 0.0;
        bool stopped = false;
        auto temp = polynomialInterpolant(getDataPoints(240), i, EQUIDISTANT, &
        if(error < minError and stopped == false){
            minError = error;
            degree = i;
            equidistantPoints = temp;
        }
    }

    std::cout << "Polynomial Equidistant function with degree: " << degree << "


    minError = pow(10,100000);
    degree = 0;
    std::vector<std::pair<double, double> > randomPoints;
    for(int i = 3;i <= 240;i++){
        double error = 0.0;
        bool stopped = false;
        auto temp = polynomialInterpolant(getDataPoints(240), i, RANDOM, &error
        if(error < minError and stopped == false){
            minError = error;
            degree = i;
            randomPoints = temp;
        }
    }

    // Generate file
    generatePointsFileFromPolynomialInterpolated(equidistantPoints, "Polynomia
    generatePointsFileFromPolynomialInterpolated(randomPoints, "PolynomialRando

    std::cout << "Polynomial Random function with degree: " << degree << " and
```

```
}



double aj(std::vector<std::pair<double, double>> points, int j){
    double result = 0.0;
    int n = points.size();

    for(int k = 0;k < n;k++){
        result += points.at(k).second*cos(j * points.at(k).first);
    }

    result = result * (2.0/n);

    return result;

}



double bj(std::vector<std::pair<double, double>> points, int j){
    double result = 0.0;
    int n = points.size();

    for(int k = 0;k < n;k++){ // till n-1 otherwise periodic
        result += points.at(k).second*sin(j * points.at(k).first);
    }

    result = result * (2.0/n);

    return result;

}

double a0(std::vector<std::pair<double, double>> points){
    return aj(points, 0);
}

// m -> amount of coef
// x -> x value for generated function
double triGoniometric(std::vector<std::pair<double, double>> points, int m, dou
    int n = points.size();


    double approx = a0(points)/2.0;

    if(n == 2*m){
        for(int i = 1;i <= m - 1;i++){
            approx += aj(points, i)*cos(i * x);
            approx += bj(points, i)*sin(i * x);
```

```
        }
        approx += aj(points, m)*0.5*cos(m * x);
    }else{
        for(int i = 1;i <= m;i++){
            approx += aj(points, i)*cos(i * x);
            approx += bj(points, i)*sin(i * x);
        }
    }

    return approx;
}

void calculateTriogoniometricLeastSquares(){
    // Least Squares -> n > 2m + 1

    auto points = getDataPoints(120);
    int m = 25; // 121 > 41
    double error = 0.0;

    std::list<std::pair<double, double>> interpolatedFunction;

    for(double xi = -Utils::PI;xi <= Utils::PI;xi += 0.01){
        double yi = triGoniometric(points, m, xi);

        error += fabs((yi - function(xi))/function(xi));

        interpolatedFunction.push_back(std::make_pair(xi, yi));
    }

    Utils::writePointsFile(interpolatedFunction, "Twographs/trioLS-interpolated

    std::cout << "Triogoniometric Least Squares with m : " << m << ", n : " <<
}

void calculateTriogoniometricInterpolant(){
    // Least Squares -> n > 2m + 1

    auto points = getDataPoints(121);
    int m = 61; // 121 > 41
    double error = 0.0;

    std::list<std::pair<double, double>> interpolatedFunction;

    for(double xi = -Utils::PI;xi <= Utils::PI;xi += 0.01){
        double yi = triGoniometric(points, m, xi);

        error += fabs((yi - function(xi))/function(xi));

        interpolatedFunction.push_back(std::make_pair(xi, yi));
    }
```

```
    Utils::writePointsFile(interpolatedFunction, "Twographs/trioI-interpolated.

    std::cout << "Triogoniometric Interpolant with m : " << m << ", n : " << po
}

int main() {
    std::cout.precision(20);

    calculatePolynomialInterpolant();
    calculateTriogoniometricLeastSquares();
    calculateTriogoniometricInterpolant();
    calculatePolynomialLeastSquares();
}
```

../three.cpp - Opgave 5

```
#include <iostream>
#include <list>
#include <vector>
#include "utils.h"

// GSL
#include <gsl/gsl_math.h>
#include <gsl/gsl_monte.h>
#include <gsl/gsl_monte_plain.h>

double func (double x[], size_t dim, void * p){
    if (dim != 2)
    {
        fprintf (stderr, "error: dim != 2");
        abort ();
    }

    if(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2) <= 0.25){
        return fabs(log(x[0] + x[1]));
    }else{
        return 0.0;
    }
}

// x 0 -> 1
// y 0 -> 1
// MAPLE
// with(VectorCalculus);
// int(abs(ln(x+y)), [x, y] = Circle(   0.5,0.5    , .5));
// .2550197403

double calculate(int samples){
    double res, err;
```

```
    double xl[2] = {0.0, 0.0};
    double xu[2] = {1.0, 1.0};

    const gsl_rng_type *T;
    gsl_rng *r;

    gsl_monte_function G = { &func, 2, 0 };

    gsl_rng_env_setup ();

    T = gsl_rng_default;
    r = gsl_rng_alloc (T);

    gsl_monte_plain_state *s = gsl_monte_plain_alloc (2);
    gsl_monte_plain_integrate (&G, xl, xu, 2, samples, r, s, &res, &err);
    gsl_monte_plain_free (s);
    gsl_rng_free (r);

    return res;
}

int main() {
    std::cout.precision(20);

    int samples = 1;
    while(true){
        double result = calculate(samples);
        std::cout << "Samples: " << samples << ", result: " << result << std::e

        int test = result*100;
        if(test == 25){
            break;
        }else{
            samples += 10;
        }
    }

}
```