

Praktikumsbericht 1. Praxissemester

Norman Walter

Matrikelnummer: 728297

FHTE Esslingen
Studiengang
Softwaretechnik/Medieninformatik

September 2005 - Februar 2006

Durchgeführt bei

ERICSSON 

GmbH Backnang
Gerberstraße 33
71522 Backnang

Abteilung: Wireless Product Unit (WPU/HWA4)

Teil 3: Interpolation_NG

Betreuer: Dr. Stefan Kern

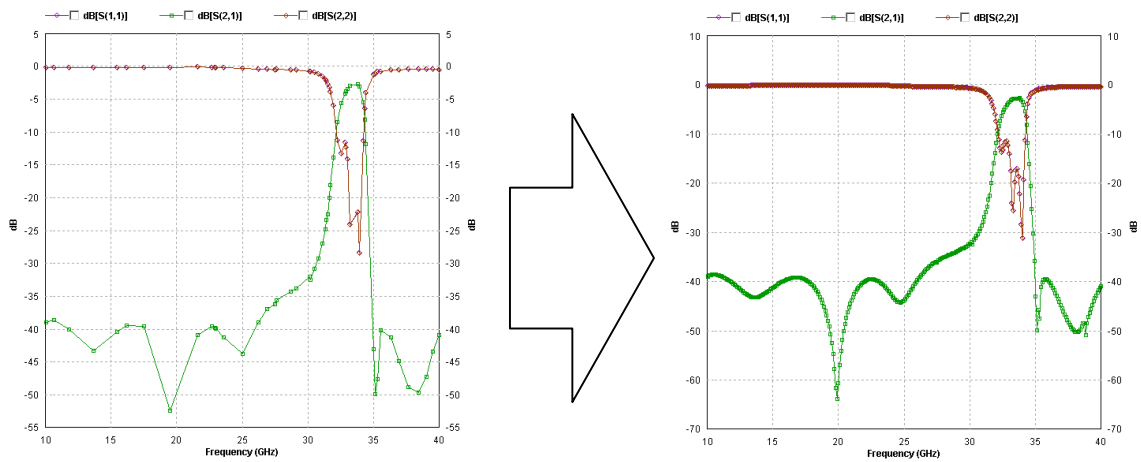
Inhaltsverzeichnis

Einleitung	4
Feldsimulation von Hochfrequenzschaltungen	5
Ist-Analyse	6
Use Case	7
Programmablauf	8
Das Touchstone Dateiformat	9
Die Klasse spt-File	11
Das Modul tokenizer	17
Das Modul deg_rad	19
Die Klasse frequenz	20
Die Klasse curve	22
Die Klasse curves	25
Die Klasse interpolation	29
Das Modul ratint	33
Alternative Fehlerrechnung	35
Benchmark	37
Fazit	45
Zeitplan	46
Literaturverzeichnis	48

Einleitung

Um die Feldsimulation integrierter Mikrowellenschaltkreise zu beschleunigen, wurde eine Queue entworfen, welche die zu berechnenden Frequenzen auf verschiedene Rechner verteilt.

Das Interpolationsprogramm hat innerhalb der Queue folgende Aufgaben zu erfüllen: Zum einen erhält der Entwickler durch die Interpolation schon eine Vorschau auf das Ergebnis, bevor alle zu rechnenden Frequenzen berechnet wurden. Zum anderen wird für jede Frequenz eine Fehlerfunktion berechnet. Mit Hilfe der Fehlerfunktion kann man ermitteln, welche der noch nicht berechneten Frequenzen die Kurve am stärksten beeinflusst. Die Queue rechnet dann den wichtigsten Punkt als nächstes. Sobald für alle Frequenzen eine bestimmte Fehlergrenze unterschritten ist, hört die Queue auf, neue Frequenzen zu berechnen. Dadurch wird die Simulation einer Schaltung wesentlich beschleunigt, da nur noch wenige Frequenzen tatsächlich vom Feldsimulator gerechnet und die übrigen Frequenzen interpoliert werden.



Feldsimulation von Hochfrequenzschaltungen

Die s-Parameter (Streu-Parameter, engl.: Scattering-parameter) beschreiben die wichtigsten Eigenschaften einer HF-Schaltung. Jeder s-Parameter ist das Verhältnis aus einer bei der Schaltung ankommenden und von ihr abgehenden Welle.

Im folgenden Beispiel soll ein 2-Tor betrachtet werden.

Die s-Parameter berechnet man aus vier Spannungen (a , b) an der Schaltung. Da die Spannungen komplexe Größen sind, müssen sie nach Betrag und Phase bekannt sein.

a_1 : Amplitude und Phase der ankommenden Spannung am Tor 1

a_2 : Amplitude und Phase der ankommenden Spannung am Tor 2

b_1 : Amplitude und Phase der abgehenden Spannung am Tor 1

b_2 : Amplitude und Phase der abgehenden Spannung am Tor 2

Bei einem 2-Tor erhalten wir 4 s-Parameter als Verhältnis der ankommenden und der abgehenden Spannungen::

$$\text{Eingangsreflektionsfaktor: } S_{11} = \left. \frac{b_1}{a_1} \right|_{a_2=0}$$

$$\text{Vorwärtsübertragungsfaktor: } S_{21} = \left. \frac{b_2}{a_1} \right|_{a_2=0}$$

$$\text{Rückwärtsübertragungsfaktor: } S_{12} = \left. \frac{b_1}{a_2} \right|_{a_1=0}$$

$$\text{Ausgangsreflektionsfaktor: } S_{22} = \left. \frac{b_2}{a_2} \right|_{a_1=0}$$

Bei der Feldsimulation wird nun ermittelt, welche s-Parameter sich für bestimmte Frequenzen innerhalb eines Intervalls ergeben. Für das obige Beispiel ergeben sich so vier s-Parameter pro Frequenzpunkt.

Ist-Analyse

Die letzte Version des Interpolationsprogramms von Jens Timmermann wurde in C++ geschrieben, wobei aber abgesehen von der Verwendung der Klasse `complex` aus der C++ Standardbibliothek, sowie der `fstream`-Klasse keine weiteren C++ Sprachelemente verwendet werden – der Rest des Programms ist einfaches C. Das gesamte Programm steht samt aller Funktionen in einer einzigen Datei. Nur eine einzige eigene Header-Datei wird inkludiert. Zur Speicherung diverser Daten werden Arrays mit fest vorgegebener Größe verwendet, was das Programm aber leider ziemlich unflexibel macht, denn damit wird die Menge der Eingabedaten (*konkret: die Anzahl der möglichen Frequenzpunkte*) beschränkt.

Für die eigentliche Interpolation wird eine modifizierte Version des Bulirsch Stoer Algorithmus verwendet. Der Algorithmus wurde dahingehend modifiziert, dass er statt mit reellen Werten mit komplexen Werten rechnet.

Mein Vorgänger Jens Timmermann hatte die Aufgabe, den Algorithmus durch Wiederverwendung von Zwischenergebnissen zu beschleunigen. Dabei stellte sich heraus, dass man einen Faktor von ca. 2 in der Rechenzeit erreichen kann. Leider ist das Programm nicht stabil und auch die Fehlerfunktion funktioniert nicht wie erwartet.

Gewünschter Soll-Zustand

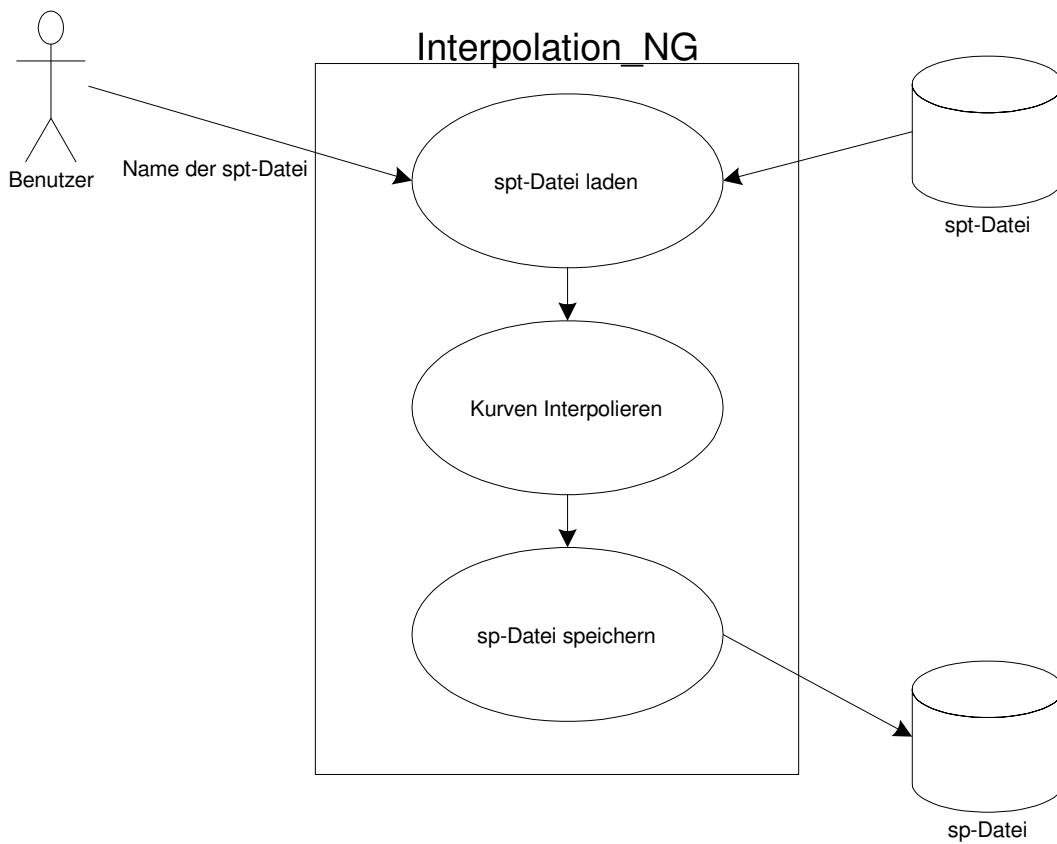
Das Interpolationsprogramm soll stabil sein und die Fehlerfunktion soll korrekte Werte liefern. Außerdem wäre eine Verbesserung des Laufzeitverhaltens wünschenswert.

Ich habe mich dazu entschlossen, das Programm komplett neu zu schreiben. Trotzdem habe ich die Grundlegenden Ideen (z.B. die Fehlerfunktion) von Jens Timmermann übernommen. In einer zweiten Programmversion wurde eine alternative Fehlerfunktion verwendet.

Das neue Programm hat folgende Eigenschaften:

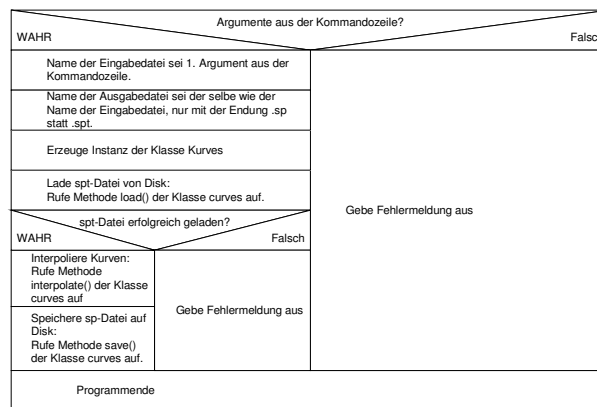
- Programmiersprache ist C++
- Objektorientierte Programmierung
- Aufteilung in klar abgegrenzte Module
- Konsequente Verwendung der *Standard Template Library (STL)*
- Interpolation mittels *Bulirsch Stoer Algorithmus (modifiziert)*
- Beliebige Anzahl von Frequenzpunkten möglich

Use Case



Der Benutzer übergibt in der Kommandozeile den Namen der spt-Datei als Argument. Das Interpolationsprogramm interpoliert dann alle noch nicht gerechneten Frequenzpunkte und rechnet jeweils eine Fehlerfunktion dazu aus. Das Resultat wird in einer sp-Datei gespeichert.

Programmablauf



Implementierung

```

/* Interpolation_NG
 *
 * Interpolationsprogramm für Touchstone Dateien.
 * Interpoliert die nicht bekannten Frequenzpunkte
 * und rechnet eine Fehlerfunktion aus.
 *
 * Autor: Norman Walter
 *
 */

#include <cstdlib>
#include <iostream>
#include <string>

#include "frequenzen.h"
#include "spt_file.h"
#include "curves.h"

using namespace std;

int main(int argc, char *argv[])
{
    string infile,outfile; // Namen der Ein- und Ausgabedatei

    if (argc < 2)
    {
        cout << "Argument fehlt" << endl;
        exit(EXIT_FAILURE);
    }

    infile = argv[1];

    // Der Name der Ausgabedatei entspricht dem Namen der Eingabedatei,
    // nur mit der Endung .sp statt .spt
    outfile=infile;
    outfile.replace(outfile.find(".spt"),4, ".sp");

    curves Kurven; // Instanz der curves-Klasse

    // Kurvensatz von Disk laden
    if (!Kurven.load(infile.c_str()))
    {
        cout << "Fehler: Konnte " << infile << " nicht laden." << endl;
        exit(EXIT_FAILURE);
    }

    Kurven.interpolate(); // Kurvensatz interpolieren
    Kurven.save(outfile.c_str()); // Kurvensatz speichern

    return EXIT_SUCCESS;
}

```

Das Touchstone Dateiformat

Eine Touchstone Datei ist eine ASCII Datei, die aus einem Header und einem Datenteil besteht. Ein Ausrufezeichen (!) leitet einen Kommentar ein.

Der Header sieht wie folgt aus:

```
# HZ|KHZ|MHZ|GHZ|THZ G|H|S|Y|Z MA|DB|RI [R x]
```

Das Zeichen # markiert den Anfang des Headers.

```
HZ | KHZ | MHZ | GHZ | THZ
```

Gibt die Einheit für die Frequenzen an.

```
G | H | S | Y | Z
```

Gibt den Parametertyp an.

```
MA | DB | RI
```

Gibt an, wie komplexe Zahlen in der Datei repräsentiert werden. MA bzw. DB bedeutet, dass die Parameter in Polarform (Betrag und Winkel) angegeben sind. Der Winkel wird dabei im Gradmaß angegeben. Bei DB wird der Betrag mittels $20 \cdot \log(\text{Betrag})$ abgebildet. RI bedeutet, dass die Parameter als Real- und Imaginärteil dargestellt werden.

```
[R x]
```

Gibt die referenz-Impedanz an (optional). Die referenz Impedanz wird nur für s-Parameter benötigt. Wird keine referenz-Impedanz angegeben, so wird 50 Ohm angenommen.

Beispiele für Header:

```
# GHZ S MA R 50
```

```
# MHZ S DB
```

```
# HZ Z RI
```

Für den Datenteil gelten folgende Regeln:

- Die Matrizen der Netzwerkparameter sind nach Reihen geordnet, mit Ausnahme von 2-Ports dort sind die Netzwerkparameter nach Spalten geordnet.
- Jeder Netzwerkparameter ist eine komplexe Zahl, dargestellt durch zwei aufeinanderfolgende reelle Zahlen.
- Jede Zeile darf aus maximal 4 Netzwerkparametern (8 reellen Zahlen) bestehen. Falls die Matrix mehr als 4 Netzwerkparameter pro Zeile hat, wird die Zeile umgebrochen.
- Jede Zeile der Matrix von Netzwerkparametern fängt in einer neuen Zeile an.
- Die erste Zeile einer Netzwerkparameter Matrix fängt mit der Frequenz an.

Die Syntax für die Netzwerkdaten ist:

```
<Frequenz 1> <Zeile 1>
               [<Zeile 1 Fortsetzung>]
               <Zeile 2>
               [<Zeile 2 Fortsetzung>]
               .....
               <Zeile n>
               [<Zeile n Fortsetzung>]

<Frequenz 2> <Zeile 1>
               [<Zeile 1 Fortsetzung>]
               <Zeile 2>
               [<Zeile 2 Fortsetzung>]
               .....
               <Zeile n>
               [<Zeile n Fortsetzung>]

...
<Frequenz m> <Zeile 1>
               [<Zeile 1 Fortsetzung>]
               <Zeile 2>
               [<Zeile 2 Fortsetzung>]
               .....
               <Zeile n>
               [<Zeile n Fortsetzung>]
```

Beispiel:

```
! IE3D 8.12 Simulator
! Geometry File: c:/Queue_Local/bk5052/1054_f301/Bandpass_oHaube_10_001_alle2.geo
! Simulation Date: Thu Feb 21 15:09:55 2002
!
! Accuracy Enhancement = 2
! 3D Accuracy Enhancement = 2
! Gridding Freq = 40 (GHz)
! Matrix Solution = 5
# GHZ S MA R undefined
! Nport = 2
10 0.9697870359 30.876824125 0.011213483406 -34.62362757 0.011213483406 -34.62362757 0.96978635997 30.87686226
! elapsed time -interpolated...- ~imqs_error_function: 0
10.1 0.96968085016993 27.551019453101 0.011391767417874 -38.257421112173 0.011391767417874 -38.257421112173 0.96968013303026 27.550975598877
! elapsed time -interpolated...- ~imqs_error_function: 0.00020053111007965
10.2 0.96955899108871 24.224735783397 0.011538039767363 -41.956820020401 0.011538039767363 -41.956820020401 0.96955825340176 24.224613190277
! elapsed time -interpolated...- ~imqs_error_function: 0.00034596167397456
10.3 0.96942714145566 20.8981187593 0.011649106371294 -45.698713870988 0.011649106371294 -45.698713870988 0.96942640085181 20.897920968237
! elapsed time -interpolated...- ~imqs_error_function: 0.00044459801011563
10.4 0.96929058290844 17.571279807852 0.011722938688509 -49.458461209627 0.011722938688509 -49.458461209627 0.96928985415976 17.571010663816
! elapsed time -interpolated...- ~imqs_error_function: 0.00050407284480701
10.5 0.96915417866573 14.244298444007 0.011758822435175 -53.211100473827 0.011758822435175 -53.211100473827 0.96915347389268 14.243962106061
```

Die Klasse spt-File

Die Klasse spt-file repräsentiert eine spt-Datei.

spt-file
Attribute: private: vector<string> comments; // Kommentare der Touchstone Datei string unit; // Hz, MHz, KHz oder GHz? string r_string; // Widerstand, auf den normiert wrden soll unsigned int nport; // Anzahl der Ports freq_vector *frequenzen; // Zeiger auf Liste der Frequenzen
Methoden: public: // Anzahl der Ports zurückgeben inline unsigned int Get_nport(void) { return nport; }; // Einheit der Frequenz zurückgeben inline string Get_unit(void) { return unit; }; // Wieviele Frequenzen (berechnete und nicht berechnete) // enthält die Datei? inline unsigned int Get_NumFreq(void) { return frequenzen->size(); }; // spt Datei von Disk laden // Eingabeparameter: char *filename - Name der Datei // Rückgabe: bool - true, falls Datei erfolgreich geladen // werden konnte, sonst false bool load(const char *filename); // Inhalt der Datei ausgeben void print(void); // Liefert einen Zeiger auf den vector mit den Frequenzen inline freq_vector* GetFreqs(void) { return frequenzen; }; // Maximale Frequenz ermitteln double GetMaxFreq(void); // Minimale Frequenz ermitteln double GetMinFreq(void); // Maximalen Betrag aller s-Parameter ermitteln double GetMaxMagnitude(void); // Minimalen Betrag aller s-Parameter ermitteln double GetMinMagnitude(void);

Der Datentyp `freq_vector` ist in der Header-Datei `spt_file.h` als ein Synonym für `vector<frequenz>` definiert:

```
typedef vector<frequenz> freq_vector;
```

Im STL `vector` comments werden alle Kommentare aus der `spt`-Datei bis zum Zeichen `#` abgelegt. Jede Kommentarzeile ist dabei ein einzelner String z.B.:

! IE3D 8.12 Simulator
! Geometry File: c:/Queue_Local/bk5052/1054_f301/Bandpass_oHaube_10_001_alle2.geo
! Simulation Date: Thu Feb 21 15:09:55 2002
!
! Accuracy Enhancement = 2
! 3D Accuracy Enhancement = 2
! Gridding Freq = 40 (GHz)
! Matrix Solution = 5

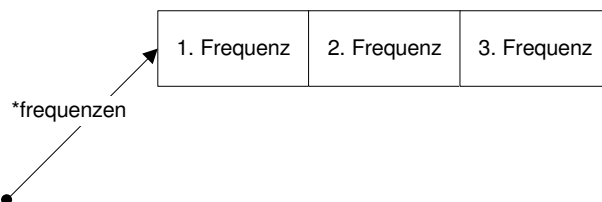
Der String `unit` entspricht der in der `spt`-Datei angegebenen Einheit für die Frequenz (z.B. Hz, kHz, Mhz...), `r_string` ist der angegebene Widerstand als String (kann auch „undefined“ sein).

Die Anzahl der Ports ist als spezieller Kommentar in `spt`-Dateien vermerkt:

```
! Nport = 2
```

bedeutet z.B., dass die Schaltung 2 Ports hat. Dies wird im Attribut `nport` vermerkt.

Das Attribut `*frequenzen` ist ein Zeiger auf einen STL `vector`, der nach dem Laden der `spt`-Datei mit Hilfe der Methode `load()` Instanzen der Frequenz-Klasse enthält:



Die Instanzen der Frequenz-Klasse enthalten jeweils die in der `spt`-Datei enthaltenen Frequenzen, sowie ihre zugehörige `s`-Parameter (Details siehe Klasse `frequenz`).

In einer `spt`-Datei sind noch nicht berechnete Frequenzen als Kommentar mit spezieller Semantik vermerkt, z.B.

```
! imqs_freq 1.0100000000000000e+001
```

Dies bedeutet, dass für die Frequenz 10,1 (GHz) noch keine Simulation durchgeführt wurde. Für solche, noch nicht berechneten Frequenzen wird ebenfalls eine Instanz der `frequenz`-Klasse erzeugt und zum `freq_vector` hinzugefügt. Allerdings enthalten solche Instanzen dann keine `s`-Parameter.

Für eine Schaltung mit n Ports gibt es n^2 `s`-Parameter.

$$\begin{pmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,j} \\ S_{2,1} & \ddots & \cdots & \vdots \\ \vdots & \cdots & \ddots & \vdots \\ S_{i,1} & \cdots & \cdots & S_{i,j} \end{pmatrix} \quad \begin{matrix} 1 \leq i \leq n \\ 1 \leq j \leq n \end{matrix}$$

aus physikalischen Gründen kann man bei passiven Schaltungen davon ausgehen, dass die an der Hauptdiagonalen der oben dargestellten Matrix gespiegelten s-Parameter identisch sind. D.h. $S_{2,1}$ entspricht $S_{1,2}$, $S_{3,2}$ entspricht $S_{2,3}$ usw. .

Ein Teil der Matrix wird also redundant:

$$\begin{pmatrix} S_{1,1} & S_{1,2} & \dots & S_{1,j} \\ 0 & \ddots & \dots & \vdots \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & S_{i,j} \end{pmatrix}$$

Für die übriggebliebenen Elemente der Matrix gilt folgende Regel: $S_j \leq S_i$

Beim Laden der spt-Datei werden nur solche s-Parameter zu den Instanzen der Frequenz-Klasse hinzugefügt, welche dieser Regel entsprechen. Alle anderen s-Parameter sind redundant und werden beim Laden ignoriert.

Dies bringt eine Verbesserung der polynomialen Zeitkomplexität, da im Programm nur Kurven der nicht redundanten s-Parameter über der Frequenz interpoliert werden müssen. Die Kurven der redundanten s-Parameter wären damit deckungsgleich.

Würde man die Kurven aller s-Parameter interpolieren, so würde sich eine polynomiale Zeitkomplexität von

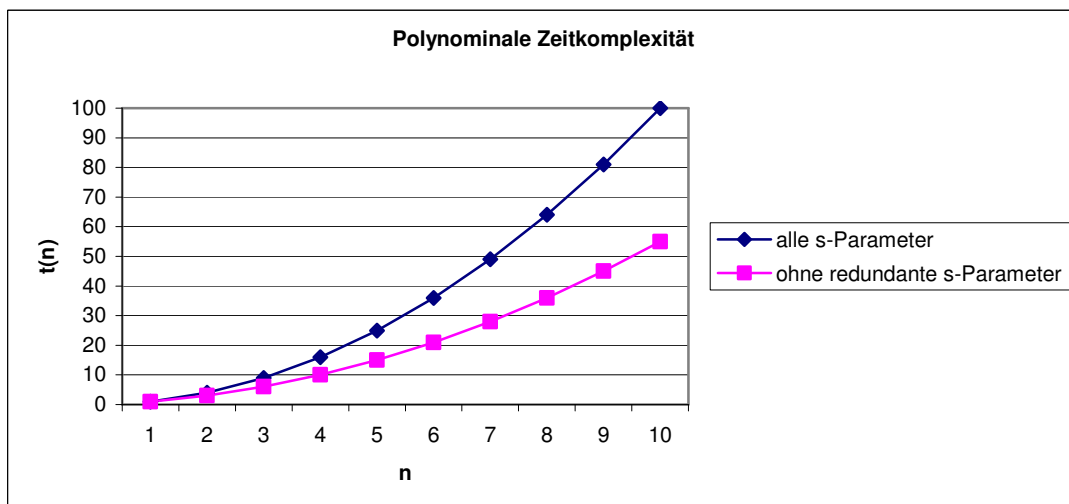
$$O_{(n)} = n^2$$

n = Anzahl der Ports

ergeben. Durch Weglassen der redundanten s-Parameter kann man diese immerhin auf

$$O_{(n)} = \frac{n^2 - n}{2} + n$$

verbessern. Somit spart man bei einem 2-Port 1 Durchlauf pro nicht berechnete Frequenz, bei einem 3-Port spart man 3 Durchläufe, bei einem 4-Port 6 usw.



Später, beim Speichern der interpolierten Werte in der sp-Datei spiegelt man einfach die Elemente an der Hauptdiagonalen der Matrix.

Implementierung der Methode load():

```
// spt Datei von Disk laden
// Eingabeparameter: char *filename - Name der Datei
// Rückgabe:      bool - true, falls Datei erfolgreich geladen
//              werden konnte, sonst false
//
bool spt_file::load(const char *filename)
{
    string line;          // Puffer für Zeile
    unsigned long line_nr = 0; // Zeilennummer
    int assign_pos = 0;    // Position des Zuweisungszeichens
    unsigned int Si,Sj;    // Zähler für s-Parameter

    ifstream in(filename,ios::in | ios::binary | ios::out);

    if (!in.good())
    {
        cout << "Error while loading " << filename << endl;
        in.close();
        return false;
    }

    // Header
    while (getline(in,line,'\n'))
    {
        // Die Spezifikationszeile beginnt mit dem Zeichen #
        // und hat das folgende Format:
        // [#][Ghz MHz KHz Hz][S][MA DB RI][R n]
        if (line.find("#") == 0)
        {
            token_vector tokens;

            // Zeile in Tokens zerlegen
            tokens = TokenizeString(line);

            // Die Spezifikationszeile muß genau 6 Elemente haben
            if (tokens.size()!=6)
            {
                cout << "Error while loading " << filename << ": "
                    << "Wrong number of tokens in specification line." << endl
                    << "Found " << tokens.size()
                    << " tokens (must be 6)." << endl;

                in.close();
                return false;
            }

            // Einheit (Hz, MHz, KHz, GHz) auslesen
            unit = tokens[1];

            // Nur die Magnitude/Angle Darstellung für s-Parameter
            // ist implementiert
            if (tokens[3]!="MA")
            {
                cout << "Error while loading " << filename << ": "
                    << tokens[3] << " not implemented." << endl;
                in.close();
                return false;
            }

            r_string = tokens[5];

            break;
        }
        else
        {
            // Kommentarzeilen im vector comments speichern
            comments.push_back(line);
        }
    }
}
```

```

}

// Lese Anzahl der Ports aus
while (getline(in,line,'\n'))
{
    // Anzahl der Ports auslesen
    if (strncmp("! Nport =",line.c_str(),9) == 0)
    {
        assign_pos = line.find("=");
        line = line.assign(line,assign_pos+1,line.length());
        // Anzahl der Ports einlesen
        sscanf (line.c_str(), "%d", &nport);

        break;
    }
}

// Gehe alle Zeilen durch
while (getline(in,line,'\n'))
{
    Si=1;
    Sj=1;

    // Frequenzen lesen
    if ( strncmp("! imqs_freq",line.c_str(),10) == 0 )
    {
        double imqs_freq;
        frequenz freq;      // Instanz der Frequenz-Klasse

        assign_pos = line.find("! imqs_freq");
        line = line.assign(line,assign_pos+12,line.length());
        sscanf (line.c_str(), "%lf", &imqs_freq);

        freq.set_freq(imqs_freq,unit);

        // Diese Frequenz zur Liste der Frequenzen hinzufügen
        frequenzen->push_back(freq);
    }
    // Frequenzblock einlesen
    // Falls das erste Zeichen der Zeile nicht "!" ist...
    else if (!line.empty() && line.find("!") != 0)
    {
        token_vector tokens;
        token_vector tokens_new;
        token_vector::iterator p;
        string token;
        double tempvalue,rho,theta;
        unsigned long i = 0;
        frequenz freq;      // Instanz der Frequenz-Klasse
        bool freq_ready = false;    // Frequenz wurde eingelesen?
        // Wieviele tokens gehören zu einer Frequenz?
        unsigned int tokens_per_freq = 1+nport*nport*2;
        unsigned int tokens_count;

        // Zeile in Tokens zerlegen
        tokens = TokenizeString(line);

        // Falls die Zeile überhaupt Tokens enthält
        if (!tokens.empty())
        {
            // Für jede Frequenz muß eine bestimmte Anzahl von Tokens
            // eingelesen werden. Falls diese Zahl noch nicht erreicht wurde,
            // lese nächste Zeile ein
            while (tokens.size() < tokens_per_freq)
            {
                getline(in,line,'\n');

                tokens_new = TokenizeString(line);

                // Gehe alle neuen Tokes durch
                for (p=tokens_new.begin();p!=tokens_new.end();p++)
                {
                    tokens.push_back((*p));
                }
            }
        }
    }
}

```

```

}

// Gehe alle Tokes durch
for (p=tokens.begin();p!=tokens.end();p++)
{
    token = *p; // aktuelles Token

    // Versuche Fließkommazahl aus dem Token zu lesen.
    // Falls keine Fließkommazahl gelesen werden konnte,
    // sind die Tokens dieser Zeile keine gültige
    // Frequenz und die Schleife wird abgebrochen.
    if (sscanf (token.c_str(), "%f", &tempvalue)==EOF)
    {
        freq_ready=false;
        break;
    }

    if (i==0)
    {
        // Das erste Element der Zeile enthält die Frequenz
        freq.set_freq(tempvalue,unit);
    }
    else if (i%2==0)
    {
        // Geradzahlige Elemente der Zeile enthalten
        // die Phase der s-Parameter (im Gradmaß).
        // Der Wert wird als Bogenmaß im s-Parameter abgelegt.
        theta=DEG2RAD(tempvalue);

        // Hat man Betrag und Phase, so kann man
        // den s-Parameter zu dieser Frequenz hinzufügen

        // Auf Grund der Symmetrie wird ein Teil der s-Parameter
        // redundant. Welche s-Parameter identisch sind,
        // kann man anhand des Index herausfinden
        if (Sj>=Si)
        {
            sparameter sp = polar(rho,theta);
            freq.insert_sparameter(sp);
        }

        // Zähle s-Parameter hoch
        if (Sj < nport)
        {
            Sj++;
        }
        else
        {
            Si++;
            Sj=1;
        }

        freq_ready=true;
    }
    else
    {
        // Die ungeradzahligen Elemente der Zeile
        // enthalten den Betrag der s-Parameter
        rho=tempvalue;
    } // if

    i++;
} // for

// Falls diese Frequenz erfolgreich eingelesen werden
// konnte, wird sie zur Liste der Frequenzen hinzugefügt
if (freq_ready) frequenzen->push_back(freq);

} // if

    line_nr++;
} // while

in.close();

return true;
}

```

Das Modul tokenizer

Für das Einlesen der spt-Dateien wurde ein Hilfsmodul namens tokenizer mit dem Definitionsmodul tokenizer.h und Implementierungsmodul tokenizer.cpp implementiert. Dieses Hilfsmodul hat die Aufgabe Strings in einzelne Tokens zu zerlegen. Tokens sollen hier durch Leerzeichen getrennte Teile eines Strings sein.

So soll z.B. der String

```
# GHZ S MA R undefined
```

in 6 Tokens zerlegt werden:

#	GHZ	S	MA	R	undefined
---	-----	---	----	---	-----------

Als Container für solche Tokens wurde in der Header-Datei tokenizer.h der Datentyp token_vector vereinbart. Dieser Container enthält die einzelnen Teilstrings.

Die Funktion TokenizeString() erhält als Eingabeparameter einen String und gibt einen token_vector zurück der die Tokens des Eingabestrings enthält.

Hier das Definitionsmodul:

```
/* tokenzier.h
 * Autor: Norman Walter
 */

#ifndef TOKENIZER_H
#define TOKENIZER_H

#include <vector>
#include <string>

using namespace std;

// vector-Typ für tokens:
// Jeder Token darin ist ein String
typedef vector<string> token_vector;

// Zerlegt einen String in einzelne Tokens und
// liefert sie in einem token_vector zurück.
//
// Eingabeparameter: const string instring - Eingabestring
// Rückgabe      : token_vector - enthält die einzelnen tokens
//
token_vector TokenizeString(const string instring);

#endif
```

Die Implementierung sieht wie folgt aus:

```
/* tokenzier.cpp
 * Autor: Norman Walter
 */

#include "tokenizer.h"

// Zerlegt einen String in einzelne Tokens und
// liefert sie in einem token_vector zurück.
//
// Eingabeparameter: const string instring - Eingabestring
// Rückgabe      : token_vector - enthält die einzelnen tokens
//
token_vector TokenizeString(const string instring)
{
    token_vector tokens; // Hier kommen die einzelnen tokens hinein
    string token;       // Ein einzelner token
    char ch;            // Einzelnes Zeichen
```

```

// Gehe den gesamten Eingabestring Zeichen für Zeichen durch
for(int i=0;instring[i] != '\0';i++)
{
    ch=instring[i];
    // Falls das aktuelle Zeichen kein Leerzeichen ist,
    // wird es zum aktuellen token hinzugefügt
    if (!isspace(ch))
    {
        token += ch;
    }
    else
    {
        // Token in token_vector einfügen
        if (!token.empty())
        {
            tokens.push_back(token);
            token = "";
        }
    }
}

return tokens;
}

```

Das Modul deg_rad

Das Modul deg_rad besteht nur aus einer Header-Datei namens deg_rad.h. Diese enthält zwei Macros zur Umwandlung vom Gradmaß ins Bogenmaß und umgekehrt. Falls die Symbolische Konstante PI nicht definiert wurde, wird diese zusätzlich definiert.

```
/* deg_rad.h
 * Makros für die Umrechnung Grad - Bogenmaß (Radiant)
 * Autor: Norman Walter
 * Datum: 13.1.2006
 *
 */

#ifndef DEG_RAD_H
#define DEG_RAD_H

// Falls PI nicht definiert ist
#ifndef PI
#define PI 3.14159265358979323846
#endif

#define DEG2RAD(a) (PI/180.0*a) // Grad nach Bogenmaß
#define RAD2DEG(x) (180.0/PI*x) // Bogenmaß nach Grad

#endif
```

Umrechnung von Grad nach Bogenmaß:

$$Winkel_{\text{Bogenmaß}} = \frac{Winkel_{\text{Gradmaß}} \cdot \pi}{180}$$

Umrechnung von Bogenmaß nach Gradmaß:

$$Winkel_{\text{Gradmaß}} = \frac{Winkel_{\text{Bogenmaß}} \cdot 180}{\pi}$$

Die Klasse frequenz

frequenz
Attribute: private: double f; // Frequenz string unit; // Einheit (GHz, MHz, KHz oder Hz) vector<sparameter> sparameter_list; // Liste der sparameter für dieser Frequenz double error; // Fehler für diese Frequenz
Methoden: public: // Frequenz und Einheit setzen // Eingabeparameter: double freq - Frequenz // string u - Einheit für die Frequenz inline void set_freq(double freq, string u) { f = freq; unit = u; }; // Frequenz setzen // Eingabeparameter: double freq - Frequenz inline void set_freq(double freq) { f = freq; }; // Liefert den Wert der Frequenz inline double get_freq(void) { return f; }; // Fehler für diese Frequenz setzen inline void set_error(double e) { error=e; }; // Fehler für diese Frequenz inline double get_error(void) { return error; }; // Wurden für diese Frequenz schon s-Parameter berechnet? inline bool calculated(void) { return !sparameter_list.empty(); }; // s-Parameter einfügen // Eingabeparameter: struct sparameter sp - s-Parameter, // bestehend aus Betrag und Phase void insert_sparameter(sparameter sp); // Wieviele s Parameter sind für diese Frequenz vorhanden? inline unsigned int get_num_sparameters(void) { return sparameter_list.size(); }; // Holt den n-ten s-Parameter aus der Frequenz heraus inline sparameter* get_sparameter(unsigned int n) { return (n < sparameter_list.size()) ? &sparameter_list[n] : NULL; }; // Setzt s-Parameter sp an index n in sparameter_list inline bool set_sparameter(unsigned int n, sparameter sp) { if (n > sparameter_list.size()) return false; sparameter_list[n]=sp; return true; } // Diese Frequenz ausgeben void print(void); // Maximalen Betrag aller s-Parameter ermitteln // Falls für diese Frequenz keine s-Parameter vorliegen, wird 0 zurückgegeben. double GetMaxMagnitude(void); // Minimalen Betrag aller s-Parameter ermitteln // Falls für diese Frequenz keine s-Parameter vorliegen, wird 0 zurückgegeben. double GetMinMagnitude(void);

Eine Instanz der Klasse `frequenz` repräsentiert jeweils eine Frequenz samt ihren zugehörigen `s`-Parametern, so wie sie in einer `spt`-Datei vorkommt.

In der Header-Datei `frequenzen.h` sind folgende Datentypen als Synonyme definiert:

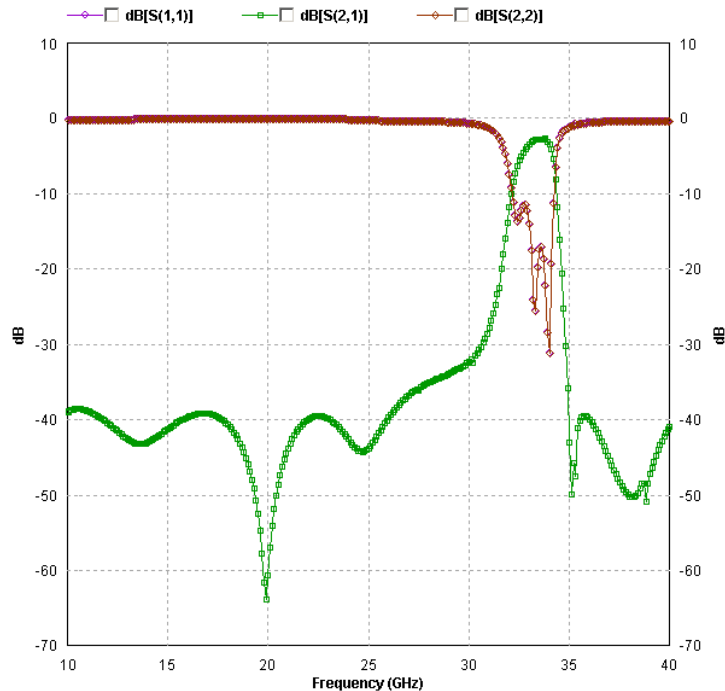
```
typedef vector<double> double_vector;  
  
// Datentyp für s-Parameter  
typedef complex<double> sparameter;
```

Ein einzelner `s`-Parameter wird also als Instanz der Klasse `complex<double>` dargestellt. `Complex` ist ein Template der C++ Standardbibliothek.

Das Attribut `f` der Klasse `frequenz` enthält den Wert der Frequenz, `unit` enthält die Einheit für die Frequenzen, so wie in der `spt`-Datei angegeben (als `String`). Der STL-vector `sparameter_list` enthält alle `s`-Parameter zu dieser Frequenz. Im Attribut `error` wird das Ergebnis der Fehlerfunktion nach der Interpolation abgelegt.

Die Klasse curve

Die Klasse curve repräsentiert jeweils eine einzelne Kurve, wie sie in Diagrammen aus der Hochfrequenztechnik gebräuchlich ist. Dabei sind z.B. alle S11 Parameter über der Frequenz aufgetragen.



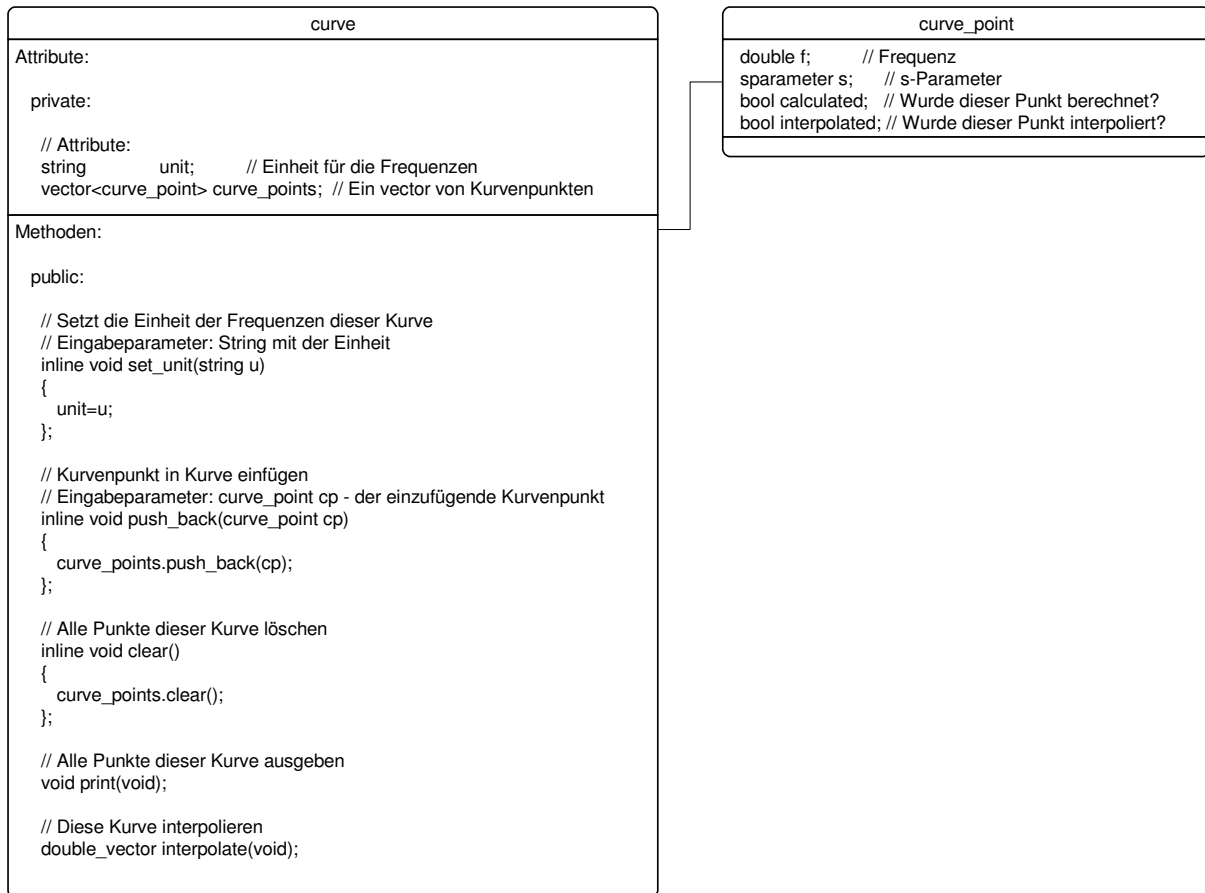
Der in der Header-Datei curve.h vereinbarte Strukturtyp curve_point stellt dabei einen einzelnen Punkt einer solchen Kurve dar:

```
// Strukturtyp für Kurvenpunkt
typedef struct
{
    double f;      // Frequenz
    sparameter s; // s-Parameter
    bool calculated; // Wurde dieser Punkt berechnet?
    bool interpolated; // Wurde dieser Punkt interpoliert?
} curve_point;
```

Da in einer solchen Kurve sowohl berechnete, nicht berechnete als auch interpolierte Punkte vorkommen können, muss man diese Punkte unterscheiden. Dies geschieht hier, indem die Attribute calculated bzw. interpolated gesetzt werden.

Im STL-vector curve_points werden diese Kurvenpunkte nach folgendem Schema eingetragen:

curve_points	Sij für Frequenz 1	Sij für Frequenz 2	Sij für Frequenz 3	...	Sij für Frequenz n
--------------	-----------------------	-----------------------	-----------------------	-----	-----------------------



Über die Methode `interpolate()` kann eine Kurve sich selbst interpolieren. Dabei werden alle nicht berechneten Punkte interpoliert. Zusätzlich wird für die zugehörige Frequenzen eine Fehlerfunktion berechnet. Der Rückgabewert der Methode `interpolate()` ist ein STL-vector aus `double`-Werten. Jedes Element dieses vectors stellt dabei den Fehler für einen bestimmten Frequenzpunkt dar. Für berechnete Frequenzpunkte ist der Fehler 0.

```

// Diese Kurve interpolieren
double_vector curve::interpolate(void)
{
    interpolation rat(this);          // Instanz der Klasse interpolation
    sparameter temp_s;              // temporärer s-Parameter zum Vergleich
    double diff_s_error_n_max;      // Betragmäßige Differenz zweier s-Parameter
    double_vector errors(curve_points.size(),0.0); // vector für Fehler der Frequenzen

    // Gehe alle Punkte dieser Kurve durch (also alle Frequenzen)
    vector<curve_point>::iterator p;
    vector<curve_point>::iterator q;

    unsigned int fn = 0;
    p=curve_points.begin();

    for (p;p!=curve_points.end();p++)
    {
        error_n_max = 0.0; // Maximaler Fehler für diesen Frequenzpunkt der Kurve

        // Falls dieser Punkt noch nicht berechnet oder interpoliert wurde...
        // Interpoliere den Punkt für diese Frequenz
        if (! (p->calculated || p->interpolated))
        {
            p->s=rat.interpolate(p->f); // s-Parameter interpolieren
            p->interpolated=true;      // Punkt als berechnet markieren
        }

        // Pro Durchlauf soll jeweils ohne einen bestimmten Stützpunkt
  
```

```

// interpoliert werden und dann das Ergebnis mit der Interpolation
// über alle Punkte verglichen werden. Die maximale Abweichung
// der Beträge der s-Parameter wird im Attribut error der Frequenz
// vermerkt.

// Falls dieser Frequenzpunkt interpoliert wurde, soll der
// Fehler ausgerechnet werden.
if (p->interpolated)
{
    // Es soll vom 2. bis zum vorletzten Frequenzpunkt iteriert werden.
    unsigned int k=0;
    q=curve_points.begin()+1;
    for (q;q!=curve_points.end()-1;q++)
    {
        // Falls dieser Frequenzpunkt berechnet wurde, ist
        // er eine Stützstelle.
        if (q->calculated)
        {
            // Berechne die Differenz des Funktionswerts der Interpolationsfunktion,
            // und der n-1 Interpolation
            temp_s = rat.interpolate_n1(p->f,k);
            diff_s = fabs(abs(p->s)-abs(temp_s));

            // Ist der berechnete Fehler größer wie das bisherige Maximum?
            if (diff_s > error_n_max) error_n_max = diff_s;

            k++;
        }
    }
}

errors[fn]=error_n_max;
fn++;

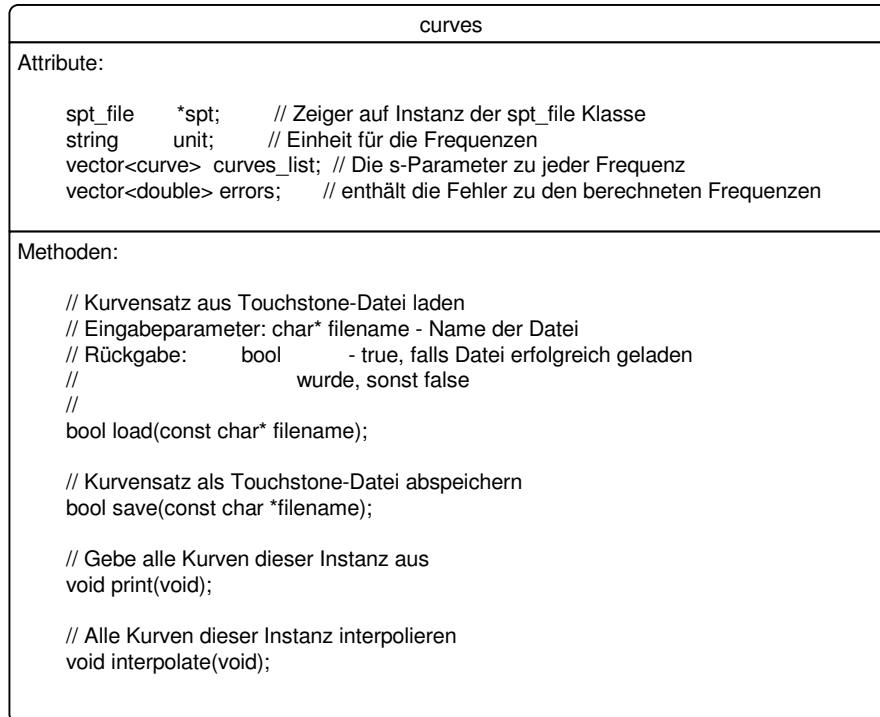
}

return errors;
}

```

Die Klasse curves

Die Klasse curves ist die übergeordnete Klasse zu curve, d.h. sie stellt einen Container für Instanzen der Klasse curve dar. Zusätzlich bietet curves die Möglichkeit, einen Kurvensatz von Disk zu laden bzw. diesen (nach der Interpolation) als sp-Datei zu speichern.



Zum Laden des Kurvensatzes erzeugt die Methode load() der Klasse curves zunächst eine Instanz der Klasse spt_file und ruft deren Methode load() auf. Somit wird die spt-Datei geladen. Aus dem Inhalt der spt-Datei werden dann die einzelnen Kurven konstruiert.

```
// Kurvensatz aus Touchstone-Datei laden
// Eingabeparameter: char* filename - Name der Datei
// Rückgabe:      bool      - true, falls Datei erfolgreich geladen
//                wurde, sonst false
//
//
bool curves::load(const char* filename)
{
    // Versuche die Touchstone-Datei zu laden
    if (!spt->load(filename))
    {
        cout << "curves::load() : Unable to open " << filename << endl;
        return false;
    }

    unit = spt->Get_unit(); // Hz, KHz, MHz oder Ghz?

    curve_point cp; // Einzelner Kurvenpunkt
    curve cv;       // Einzelne Kurve aus Kurvenpunkten
    sparameter *sp; // Zeiger auf einzelnen s-Parameter

    freq_vector *file_freqs = spt->GetFreqs();
    freq_vector::iterator fr;

    cv.set_unit(unit); // Einheit für Frequenzen dieser Kurve setzen

    // Für alle s-Parameter
    for (unsigned int n=0; n < file_freqs->begin()->sparameter_list.size(); n++)
    {
```

```

// Für alle Frequenzen aus der Datei
for (fr=file_freqs->begin();fr!=file_freqs->end();fr++)
{
    cp.f=fr->get_freq(); // Frequenz
    cp.interpolated = false; // Punkt wurde nicht interpoliert

    // versuche, n-ten s-Parameter herauszuholen
    // also z.B. S11
    sp = fr->get_sparameter(n);

    // Falls dieser existiert...
    if (sp!=NULL)
    {
        // Kurvenpunkt zusammenstellen:
        cp.s=*sp; // s-Parameter
        cp.calculated=true; // dieser Punkt wurde berechnet
    }
    else
    {
        cp.calculated=false; // dieser Punkt wurde noch nicht berechnet
    }

    // Kurvenpunkt eintragen:
    cv.push_back(cp);
}

// Füge die Kurve für diese Sparameter zum Kurvensatz hinzu.
curves_list.push_back(cv);
cv.clear();
}

// Fehler für alle Punkte auf Null setzen
fr=spt->frequenzen->begin();
for (fr;fr!=spt->frequenzen->end();fr++)
{
    errors.push_back(0.0);
}

return true;
}

```

Die Methode `interpolate()` der Klasse `curves` interpoliert einen kompletten Kurvensatz, indem sie jede in `curves_list` enthaltene Instanz der Klasse `curve` anweist, sich selbst zu interpolieren.

```

// Alle Kurven dieser Instanz interpolieren
void curves::interpolate(void)
{
    double_vector curve_errors; // Fehler einer einzelnen Kurve

    vector<curve>::iterator cv; // Iterator über alle Kurven
    double_vector::iterator e; // Iterator über Fehler aller Kurven

    // Für alle Kurven
    for (cv=curves_list.begin();cv!=curves_list.end();cv++)
    {
        // Kurve interpolieren und vector mit Fehlern zurückgeben
        curve_errors=cv->interpolate();

        // Gleiche Fehler dieser Kurve mit den Fehlern
        // aller Kurven ab.

        e=errors.begin();
        unsigned int k=0;

        for (e;e!=errors.end();e++)
        {
            if (curve_errors[k]>(*e)) (*e) = curve_errors[k];
            k++;
        }
    }
}

```

Dabei wird gleichzeitig die Fehlerfunktion der Kurve mit der Fehlerfunktion der bisher berechneten Kurven abgeglichen. Falls für eine Frequenz der aktuell berechneten Kurve ein größerer Fehler ermittelt wurde, wird dieser als neuer Fehler für diesen Frequenzpunkt aller Kurven eingetragen. Es wird also für jede Frequenz der maximale Fehler ermittelt. Nach der Interpolation aller Kurven enthält das Attribut errors die maximalen Fehler für jede Frequenz.

Die Methode save() der Klasse curves speichert das Ergebnis der Interpolation als sp-Datei. Zusätzlich werden die Kommentare aus der spt-Datei übernommen, aus der der Kurvensatz ursprünglich gelesen wurde.

Da beim Laden der spt-Datei die redundanten s-Parameter ignoriert wurden, müssen diese beim Speichern aus den vorhandenen Werten rekonstruiert werden.

Folgende s-Parameter wurden aus der spt-Datei übernommen:

$$\begin{pmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,j} \\ 0 & \ddots & \cdots & \vdots \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & S_{i,j} \end{pmatrix}$$

Eigentlich brauchen wir aber alle s-Parameter:

$$\begin{pmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,j} \\ S_{2,1} & \ddots & \cdots & \vdots \\ \vdots & \cdots & \ddots & \vdots \\ S_{i,1} & \cdots & \cdots & S_{i,j} \end{pmatrix}$$

Dabei gehen wir davon aus, dass wir die fehlenden s-Parameter durch Spiegelung der Matrix an der Hauptdiagonalen rekonstruieren können. Das bedeutet z.B. $S_{2,1}$ entspricht $S_{1,2}$. Also können wir für $S_{1,2}$ an Stelle von $S_{2,1}$ in die sp-Datei schreiben.

```
// Kurvensatz als Touchstone-Datei abspeichern
bool curves::save(const char *filename)
{
    //cout << "curves::save()" << endl;

    unsigned int num_freq = spt->Get_NumFreq(); // Anzahl der Frequenzen
    curve_point *cp;
    frequenz *freq;

    vector<curve>::iterator cv; // Iterator über alle Kurven
    unsigned int i = 0;
    unsigned int Si,Sj; // Zähler für s-Parameter

    // Öffne Ausgabedatei zum schreiben
    fstream out(filename,ios::out | ios::binary);

    if (!out.good())
    {
        cout << "Error while saving " << filename << endl;
        out.close();
        return false;
    }

    out.precision(14);

    // Kommentarzeilen ausgeben
    vector<string>::iterator comment;
    comment=spt->comments.begin();
    for (comment;comment!=spt->comments.end();comment++)
    {
        out << (*comment) << endl;
    }
}
```

```

}

out << "# " << spt->unit << " S M A R " << spt->r_string << endl;
out << "! Nport = " << spt->nport << endl;

// Für alle Frequenzen
for (unsigned int n=0; n<num_freq;n++)
{
    // Frequenz schreiben
    out << (spt->frequenzen->begin()+n)->f;

    i = 0;
    unsigned int s_per_line = 0;
    unsigned int curve_n;

    // Über alle Sij der s-Parameter iterieren
    for(Si=1; Si <= spt->nport; Si++)
    {
        for(Sj=1; Sj <= spt->nport; Sj++)
        {
            // In welcher Kurve sind s-Parameter Sij enthalten?
            if (Sj==1)
            {
                curve_n = Si-1;
            }
            else if (Sj > Si)
            {
                curve_n++;
            }
            else
            {
                curve_n = curve_n + spt->nport - Sj + 1;
            } // if

            cv=curves_list.begin()+curve_n;

            cp = &cv->curve_points[n];

            if (cp != NULL)
            {
                if (cp->calculated || cp->interpolated)
                {

                    // Betrag und Phase in Datei schreiben
                    out << " " << abs(cp->s)
                        << " " << RAD2DEG(arg(cp->s));

                    s_per_line++;

                    // Sorgt für Zeilenumbrüche
                    // Nur maximal 4 s-Parameter pro Zeile
                    if (s_per_line >= 4)
                    {
                        out << endl << "\t\t";
                        s_per_line = 0;
                    }
                }
            }

        } // for
    }

    out << endl;

    // Fehler für die n-te Frequenz ausgeben
    out << "! elapsed time -interpolated...- ~imqs_error_function: "
        << errors[n] << endl;

}

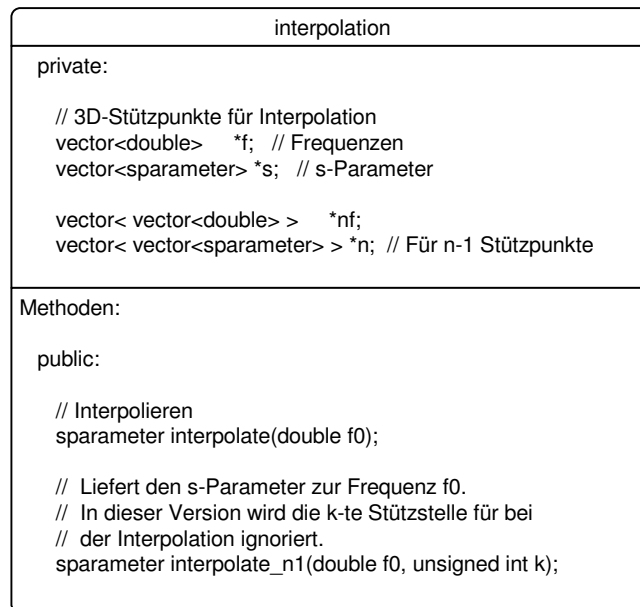
out.close();

return true;
}

```

Die Klasse interpolation

Die Klasse Interpolation enthält in Ihren Attributen die für die Interpolation einer Kurve nötigen Stützstellen einer Kurve. Eine Stützstelle besteht jeweils aus einer Frequenz (reell) und einem s-Parameter (komplex). *f zeigt dabei auf einen STL-vector, der die Frequenzen der Stützstellen enthält, *s zeigt auf einen STL-vector, der alle s-Parameter der Stützstellen enthält. Die Methoden interpolate() und interpolate_n1() führen verschiedene Arten von Interpolationen durch.



Der Konstruktor der Klasse Interpolation erwartet als Eingabeparameter einen Zeiger auf eine Instanz der curve Klasse:

```
// Konstruktor
interpolation(class curve *cv);
```

Jede Instanz der Klasse interpolation ist also für jeweils eine bestimmte Kurve zuständig.

Die für die Interpolation benötigten Stützstellen werden vom Konstruktor dieser Klasse aus den Punkten der jeweiligen Kurve herausgefiltert – Stützstellen sind bereits berechnete Punkte. Dieser Vorgang geschieht einmal bei der Inkarnation eines Objekts der Klasse interpolation. Anschließend können Frequenzpunkte anhand dieser Stützstellen interpoliert werden.

```
// Konstruktor
interpolation::interpolation(class curve *cv)
{
    f = new vector<double>;
    s = new vector<sparameter>;
    n = new vector< vector<sparameter> >;
    nf = new vector< vector<double> >;

    // Gehe alle Punkte dieser Kurve durch
    vector<curve_point>::iterator p=cv->curve_points.begin();
    for (p;p!=cv->curve_points.end();p++)
    {
        // Die Frequenzen und s-Parameter von allen bekannten
        // Kurvenpunkten in getrennten vektoren ablegen
        if (p->calculated)
        {
            f->push_back(p->f);
            s->push_back(p->s);
        }
    }
}
```

```

// Fülle n und nf vectoren.
// in diesen vectoren fehlt jeweils die k-te Stützstelle.
unsigned int k = 1;
p=cv->curve_points.begin()+1;
for (p;p!=cv->curve_points.end()-1;p++)
{
    if (p->calculated)
    {
        vector<double> tempf = (*f);
        tempf.erase(tempf.begin()+k); // Frequenz von Stützstelle k entfernen
        nf->push_back(tempf);

        vector<sparameter> temp = (*s);
        temp.erase(temp.begin()+k); // s-Parameter von Stützstelle k entfernen
        n->push_back(temp);

        k++;
    }
}
};

```

Die Methode `interpolate()` nutzt alle vorhandenen Stützstellen zur Interpolation des s-Parameters an der Stelle `f0`.

```

// Liefert den s-Parameter zur Frequenz f0.
sparameter interpolation::interpolate(double f0)
{
    sparameter sp;
    sparameter dy;

    // Bulirsch Stoer Algorithmus für komplexe Zahlen
    ratint(*f, *s, f0, sp, dy);

    return sp;
}

```

Die Methode `interpolate_n1()` wird zur Berechnung der Fehlerfunktion benötigt. Dabei wird der k-te Stützpunkt zwischen dem ersten und letzten Stützpunkt nicht bei der Interpolation berücksichtigt.

```

// Liefert den s-Parameter zur Frequenz f0.
// In dieser Version wird die k-te Stützstelle für bei
// der Interpolation ignoriert.
sparameter interpolation::interpolate_n1(double f0, unsigned int k)
{
    sparameter sp;
    sparameter dy;

    vector < vector<double> >::iterator p = nf->begin()+k;
    vector < vector<sparameter> >::iterator q = n->begin()+k;

    // Bulirsch Stoer Algorithmus für komplexe Zahlen
    ratint(*p, *q, f0, sp, dy);

    return sp;
}

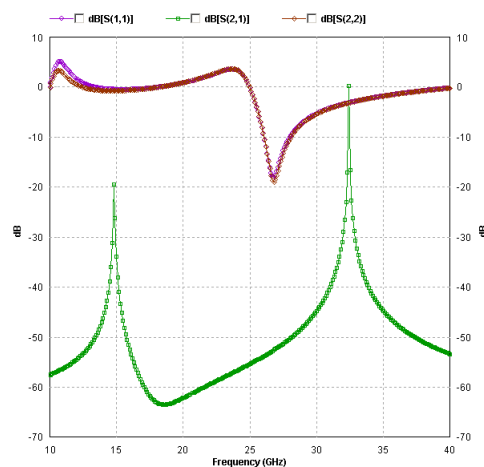
```

Die Fehlerfunktion

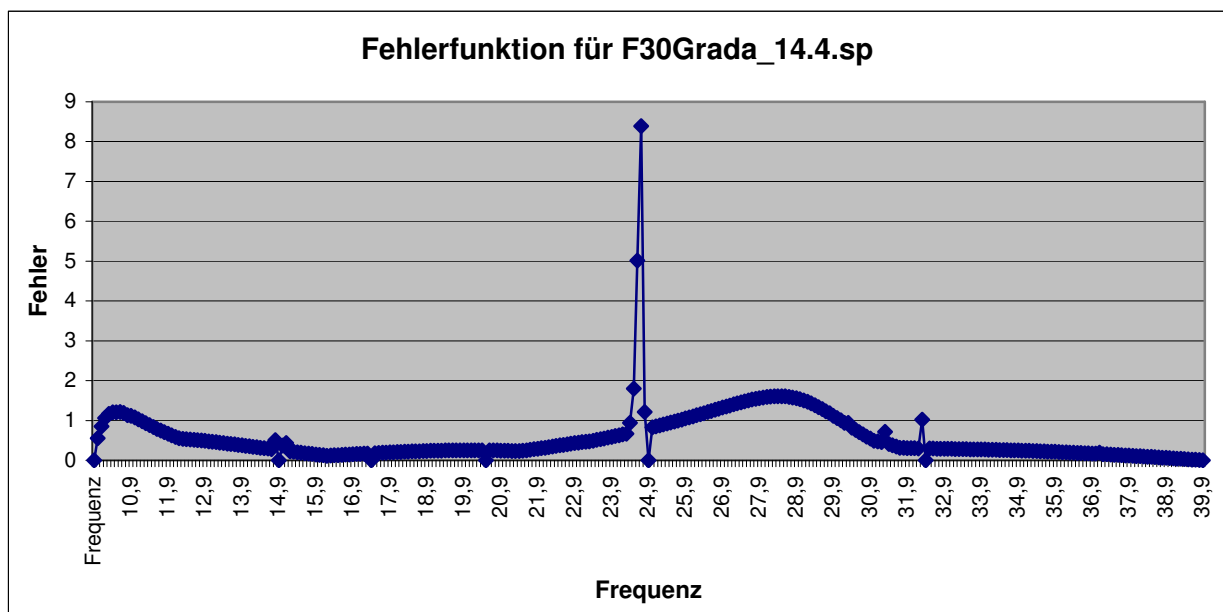
Damit die Queue effizient arbeitet, soll immer derjenige Punkt als nächstes gerechnet werden, der den Verlauf der Kurve am meisten beeinflusst. Dieser Punkt wird mit Hilfe einer „Fehlerfunktion“ ermittelt.

Zu jedem zu interpolierenden Kurvenpunkt wird ein Fehlerwert bestimmt. Dazu wird die gesamte Kurve zunächst unter Verwendung aller Stützstellen interpoliert. Dann interpoliert wird die selbe Kurve nochmals interpoliert – diesmal ohne einen bestimmten Stützpunkt. Die beiden Werte werden miteinander verglichen und das Maximum gespeichert. Dies wird für alle Stützstellen von zweiten bis zum vorletzten Kurvenpunkt wiederholt. So erhält man zu jeder Frequenz einen Fehlerwert. Für Frequenzen, die simuliert wurden ist der Fehler 0.

Im folgenden Beispiel wurden schon eine bestimmte Anzahl von Stützstellen per Simulation der entsprechenden Frequenzen ermittelt und anschließend eine Interpolation der resultierenden Kurven durchgeführt:

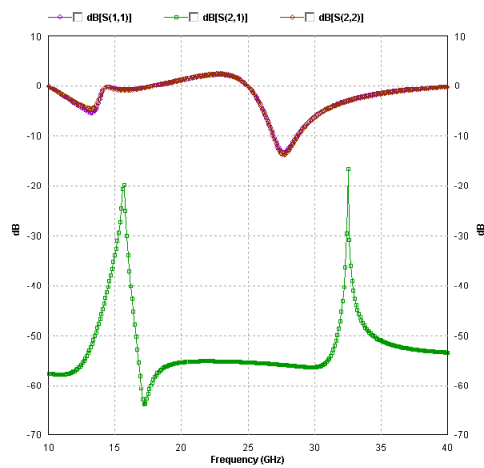


In dem folgenden Diagramm sind die Fehlerwerte über der Frequenz aufgetragen:

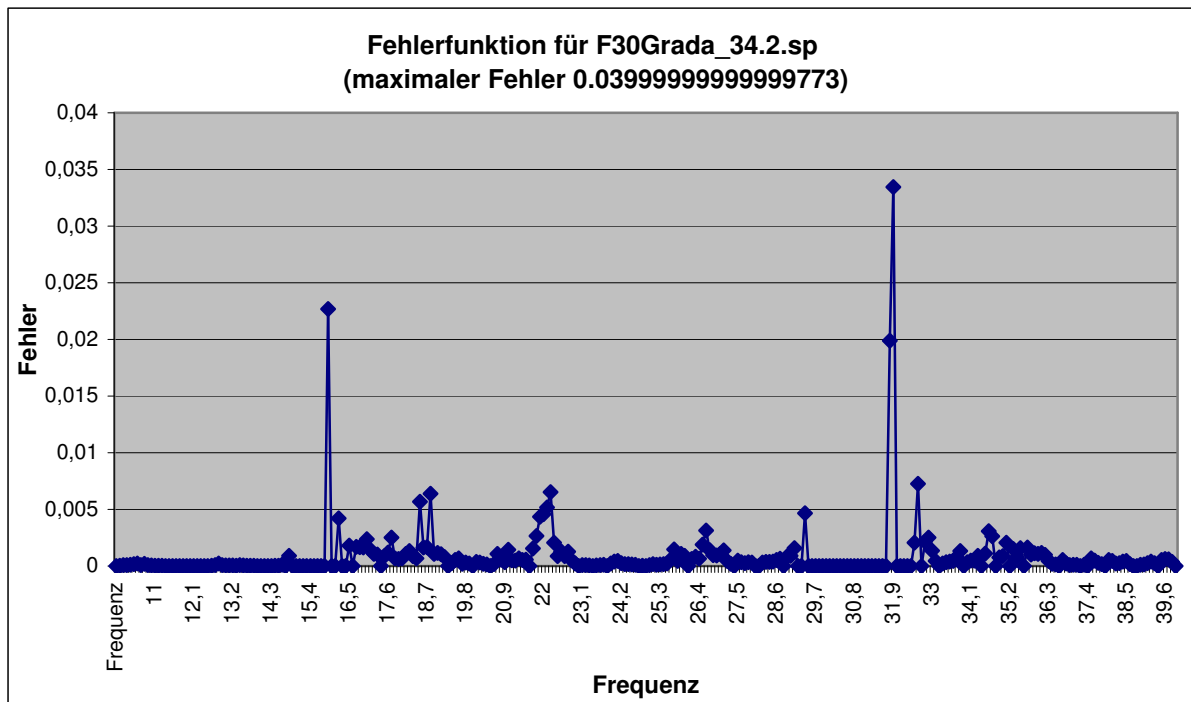


Wie aus den Diagramm ersichtlich, ist der größte Fehler bei der Frequenz 24,8 GHz. Also wird die Queue diese Frequenz als nächstes simulieren.

Nach der Simulation der Frequenz 24,8 GHz ändert sich der Kurvenverlauf der interpolierten Kurve:



Diese Schritte werden von der Queue so lange fortgesetzt, bis das Maximum der Fehlerfunktion einen vorgegebenen Grenzwert (hier 0.03999999999999773) unterschritten. Das Ergebnis der Interpolation ist dann ausreichend nahe an der tatsächlichen Funktion.



Das Modul ratint

Das Modul ratint besteht aus dem Definitionsmodul ratint.h und dem Implementierungsmodul ratint.cpp. Es enthält den eigentlichen Interpolationsalgorithmus.

Als Grundlage für den Interpolationsalgorithmus wurde der Bulirsch-Stoer Algorithmus (die Version als „Numeric Reciepes in C++“) gewählt. Dieser wurde so modifiziert, dass er mit komplexen statt mit reellen Stützpunkten rechnen kann.

Der Bulirsch-Stoer Algorithmus löst folgendes Problem:

Gegeben seien die Stützstellen x_0, \dots, x_n und die zugehörigen Werte f_0, \dots, f_n .

Zu berechnen ist der Wert einer interpolierenden rationalen Funktion an der Stelle x .

Das Definitionsmodul ratint.h:

```
/* ratioint.h
 * Interpolationsalgorithmus nach Bulirsch und Stoer.
 * Dies ist die modifizierte Version für komplexe Zahlen.
 *
 * Autor: Norman Walter (Original aus Numerical Reciepes in C++)
 */

#ifndef RATINT_H
#define RATINT_H

#include <iostream>
#include <cmath>
#include <vector>
#include <complex>

using namespace std;

/* Bulirsch Stoer Algorithmus für komplexe Zahlen
 *
 * Eingabeparameter: vector<double> &xa : reelle x Werte der Stützpunkte
 *                   vector<complex> &ya : komplexe y Werte der Stützpunkte
 *                   const double x : (reeller) x-Wert der gesuchten Stelle
 *                   complex<double> &y : Referenz auf die Adresse, an der f(x)
 *                                     abgelegt werden soll.
 *                   complex<double> &dy : Referenz auf die Adresse, an der der
 *                                     Fehler von f(x) abgelegt werden soll.
 *
 */
void ratint(vector<double> &xa, vector< complex <double> > &ya, const double x, complex<double> &y, complex<double>
&dy);

#endif
```

Das Implementierungsmodul ratint.cpp:

```
/* ratioint.cpp
 * Interpolationsalgorithmus nach Bulirsch und Stoer.
 * Dies ist die modifizierte Version für komplexe Zahlen.
 *
 * Autor: Norman Walter (Original aus Numerical Reciepes in C++)
 */

#include "ratint.h"

using namespace std;

/* Bulirsch Stoer Algorithmus für komplexe Zahlen
 *
 * Eingabeparameter: vector<double> &xa : reelle x Werte der Stützpunkte
 *                   vector<complex> &ya : komplexe y Werte der Stützpunkte
 *                   const double x : (reeller) x-Wert der gesuchten Stelle
 *                   complex<double> &y : Referenz auf die Adresse, an der f(x)
```

```

*           abgelegt werden soll.
*           complex<double> &dy : Referenz auf die Adresse, an der der
*           Fehler von f(x) abgelegt werden soll.
*
*/
void ratint(vector<double> &xa, vector< complex <double> > &ya, const double x, complex<double> &y, complex<double>
&dy)
{
    const double TINY=1.0e-25;
    int m,i,ns=0;
    complex<double> w,t,hh,h,dd;

    int n=xa.size();

    vector<complex <double> > c(n),d(n);

    // hh=fabs(x-xa[0]);
    hh=polar(x-xa[0]);
    for (i=0;i<n;i++)
    {
        // h=fabs(x-xa[i]);
        h=polar(x-xa[i]);
        if (h == 0.0)
        {
            y=ya[i];
            dy=0.0;
            return;
        }
        else if (abs(h) < abs(hh))
        {
            ns=i;
            hh=h;
        }
        c[i]=ya[i];
        d[i]=ya[i]+TINY;
    }
    y=ya[ns-];
    for (m=1;m<n;m++)
    {
        for (i=0;i<n-m;i++)
        {
            w=c[i+1]-d[i];
            h=xa[i+m]-x;
            t=(xa[i]-x)*d[i]/h;
            dd=t-c[i+1];
            // Falls das Nennerpolynom Null wird,
            // ist hier eine Polstelle vorhanden
            if (dd == 0.0)
            {
                // Workarround: Setze Betrag auf 10 und Phase auf 0
                y=polar(10.0,0.0);
                return;
            }
            dd=w/dd;
            d[i]=c[i+1]*dd;
            c[i]=t*dd;
        }
        y += (dy=(2*(ns+1) < (n-m) ? c[ns+1] : d[ns--]));
    }
}

```

Alternative Fehlerrechnung

Bisher wurde für die Fehlerrechnung jeweils ein Stützpunkt aus der Kurve herausgenommen und dann die Interpolation erneut durchgeführt und zwar vom zweiten bis zum vorletzten Punkt der Kurve.

Als Alternative wurde ein Algorithmus ersonnen, der dies nur noch in einem bestimmten Bereich in der Nähe des zu interpolierenden Stützpunktes durchführt. Die Idee dahinter ist, dass sich die Stützpunkte in der Nähe des zu interpolierenden Punktes am stärksten auf das Ergebnis der Interpolation auswirken.

Dazu wurde die Methode `interpolate()` der Klasse `curve` entsprechend modifiziert:

```
// Diese Kurve interpolieren
double_vector curve::interpolate(void)
{
    interpolation rat(this);          // Instanz der Klasse interpolation
    sparameter temp_s;              // temporärer s-Parameter zum Vergleich
    double diff_s,error_n_max;       // Betragmäßige Differenz zweier s-Parameter
    double_vector errors(curve_points.size(),0.0); // vector für Fehler der Frequenzen

    // Gehe alle Punkte dieser Kurve durch (also alle Frequenzen)
    vector<curve_point>::iterator p;
    vector<curve_point>::iterator q;

    unsigned int fn = 0;
    p=curve_points.begin();

    for (p!=curve_points.end();p++)
    {

        error_n_max = 0.0; // Maximaler Fehler für diesen Frequenzpunkt der Kurve

        // Falls dieser Punkt noch nicht berechnet oder interpoliert wurde...
        // Interpoliere den Punkt für diese Frequenz
        if (! (p->calculated || p->interpolated))
        {
            p->s=rat.interpolate(p->f); // s-Parameter interpolieren
            p->interpolated=true;      // Punkt als berechnet markieren
        }

        // Pro Durchlauf soll jeweils ohne einen bestimmten Stützpunkt
        // interpoliert werden und dann das Ergebnis mit der Interpolation
        // über alle Punkte verglichen werden. Die maximale Abweichung
        // der Beträge der s-Parameter wird im Attribut error der Frequenz
        // vermerkt.

        // Falls dieser Frequenzpunkt interpoliert wurde, soll der
        // Fehler ausgerechnet werden.
        if (p->interpolated)
        {
            // Das Intervall, über welches iteriert werden soll...
            vector<curve_point>::iterator l,r;

            // Wieviel Punkte sollen im Intervall sein?
            unsigned int range_max = 8;
            unsigned int count = 0;
            unsigned int left_points, right_points, l_sp, r_sp;;
            left_points=0;
            right_points=0;
            l_sp=0;
            r_sp=0;

            l=curve_points.begin()+(fn-1);
            r=curve_points.begin()+(fn+1);

            // Ermittle grenzen des Intervalls
            while (count < range_max)
            {
                // linke Seite
                if (l != curve_points.begin())
                {
                    // Nur berechnete Punkte sind Stützstellen
                    if (l->calculated)
                    {
```

```

        count++;
        l_sp++;
    }

    left_points++;
    l--;
}

if (count >= range_max)
{
    break;
}

// rechte Seite
if (r != curve_points.end())
{
    // Nur berechnete Punkte sind Stützstellen
    if (r->calculated)
    {
        count++;
        r_sp++;
    }

    right_points++;
    r++;
}

// Das Intervall ist die ganze Kurve,
// aber es gibt weniger Stützpunkte wie gefordert...
if ((l == curve_points.begin()) && (r == curve_points.end()))
{
    break;
}

} // while

vector<curve_point>::iterator start_pos, end_pos;
start_pos = curve_points.begin()+(fn-left_points);
end_pos = curve_points.begin()+(fn+right_points);

// Wieviele Stützstellen sind vor dem Intervall?
unsigned int sp = 0;
l=curve_points.begin();
for (l!=start_pos;l++)
{
    if (l->calculated) sp++;
}

// Es soll vom 2. bis zum vorletzten Frequenzpunkt
// des Intervalls iteriert werden.
unsigned int k=sp-1;
q=start_pos+1;
for (q;q!=end_pos-1;q++)
{
    // Falls dieser Frequenzpunkt berechnet wurde, ist
    // er eine Stützstelle.
    if (q->calculated)
    {
        // Berechne die Differenz des Funktionswerts der Interpolationsfunktion,
        // und der n-1 Interpolation
        temp_s = rat.interpolate_n1(p->f,k);

        diff_s = fabs(abs(p->s)-abs(temp_s));

        // Ist der berechnete Fehler größer wie das bisherige Maximum?
        if (diff_s > error_n_max) error_n_max = diff_s;

        k++;
    }
}
errors[fn]=error_n_max;
fn++;
}

return errors;
}

```

Benchmark

Um das Laufzeitverhalten meines Programms mit den Programmen meiner Vorgänger zu vergleichen, habe ich ein einfaches Benchmarkprogramm geschrieben:

```
/* benchmark.cpp
 * Autor: Norman Walter
 */

#include <iostream>

#include "Timer.h"

using namespace std;

int main(int argc, char *argv[])
{
    Timer T0,T1,T2,T3;

    string infile;
    string testproc_0 = "interpolation_nor_2 ";
    string testproc_1 = "interpolation_nor ";
    string testproc_2 = "interpolation_jens1 ";
    string testproc_3 = "interpolation_ratint ";

    if (argc < 2)
    {
        cout << "Argument fehlt" << endl;
        exit(EXIT_FAILURE);
    }

    infile = argv[1];

    T0.Start();
    system((testproc_0 + infile).c_str());
    T0.Stop();

    cout << testproc_0 << "benötigt " << T0.Difftime()
        << " Sekunden für " << infile << endl;

    T1.Start();
    system((testproc_1 + infile).c_str());
    T1.Stop();

    cout << testproc_1 << "benötigt " << T1.Difftime()
        << " Sekunden für " << infile << endl;

    T2.Start();
    system((testproc_2 + infile).c_str());
    T2.Stop();

    cout << testproc_2 << "benötigt " << T2.Difftime()
        << " Sekunden für " << infile << endl;

    T3.Start();
    system((testproc_3 + infile).c_str());
    T3.Stop();

    cout << testproc_3 << "benötigt " << T3.Difftime()
        << " Sekunden für " << infile << endl;

    return EXIT_SUCCESS;
}
```

Zum Messen der Zeit habe ich folgende, einfache Timer-Klasse benutzt:

```
// Timer.h
// Eine Klasse zur Zeitmessung
// Autor: Norman Walter

#ifndef TIMER_H
```

```

#define TIMER_H

#include <ctime>

class Timer
{
    // Attribute
private:
    clock_t starttime;
    clock_t endtime;

public:

    // Konstruktor
    Timer(void)
    {
        starttime = 0;
        endtime = 0;
    }

    // Destruktor
    ~Timer(void)
    {
    }

    // Methoden

    // Zeitmessung starten
    void Timer::Start(void);

    // Zeitmessung stoppen
    void Timer::Stop(void);

    // Zeitmessung Zurücksetzen
    void Timer::Reset(void);

    // Die gemessene Zeit zurückgeben
    double Timer::Difftime(void);

};

#endif

// Timer.cpp
// Eine Klasse zur Zeitmessung
// Autor: Norman Walter

#ifndef TIMER_H
#include "Timer.h"
#endif

void Timer::Start(void)
{
    starttime = clock();
}

void Timer::Stop(void)
{
    endtime = clock();
}

void Timer::Reset(void)
{
    starttime = 0;
    endtime = 0;
}

double Timer::Difftime(void)
{
    return double(endtime-starttime)/CLOCKS_PER_SEC;
}

```

Natürlich liefert die hier verwendete Art der Zeitmessung nur einen sehr groben Vergleich, da das Ergebnis von anderen Tasks, welche auf dem selben Rechner laufen, verfälscht wird. Dennoch kann

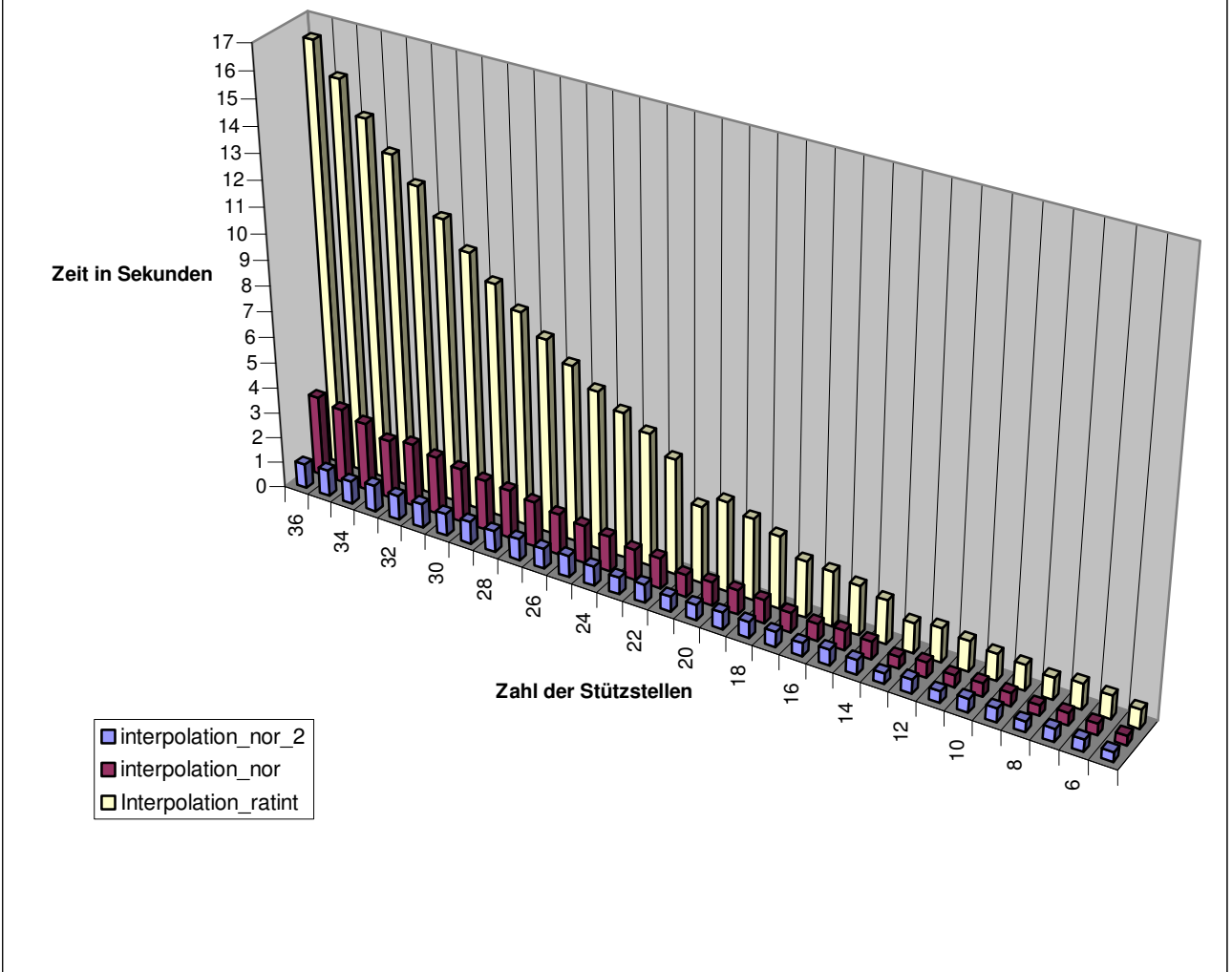
man damit erkennen, dass das mein neues Interpolationsprogramm wesentlich schneller als die Programme meiner Vorgänger ist.

Wenn das Interpolationsprogramm innerhalb der Queue verwendet wird, verschiebt sich im Laufe der Simulation das Verhältnis von der Anzahl der vorhandenen Stützpunkten (= berechnete Werte) zur Anzahl der zu interpolierenden Punkte (= nicht berechnete Werte). Am Anfang sind nur wenige Werte tatsächlich berechnet worden und es müssen viele interpoliert werden. Im weiteren Verlauf kommen immer mehr neue Stützstellen hinzu, dafür sinkt aber die Anzahl der zu interpolierenden Werte.

Um so mehr Stützstellen zur Verfügung stehen, um so näher kommt das Ergebnis der Interpolation an die tatsächliche Funktion. Allerdings braucht der Bulirsch Stoer Algorithmus um so länger, je mehr Stützstellen bei der Interpolation berücksichtigt werden.

Den stärksten Einfluss auf die Laufzeit hat die Fehlerrechnung. Je mehr Stützstellen vorhanden sind, umso länger dauert die Fehlerrechnung, da (bei meiner ersten Programmversion) für jede Stützstelle im Intervall vom zweiten bis zum vorletzten Kurvenpunkt die Interpolation nochmals durchgeführt werden muss, jeweils ohne einen bestimmten Stützpunkt.

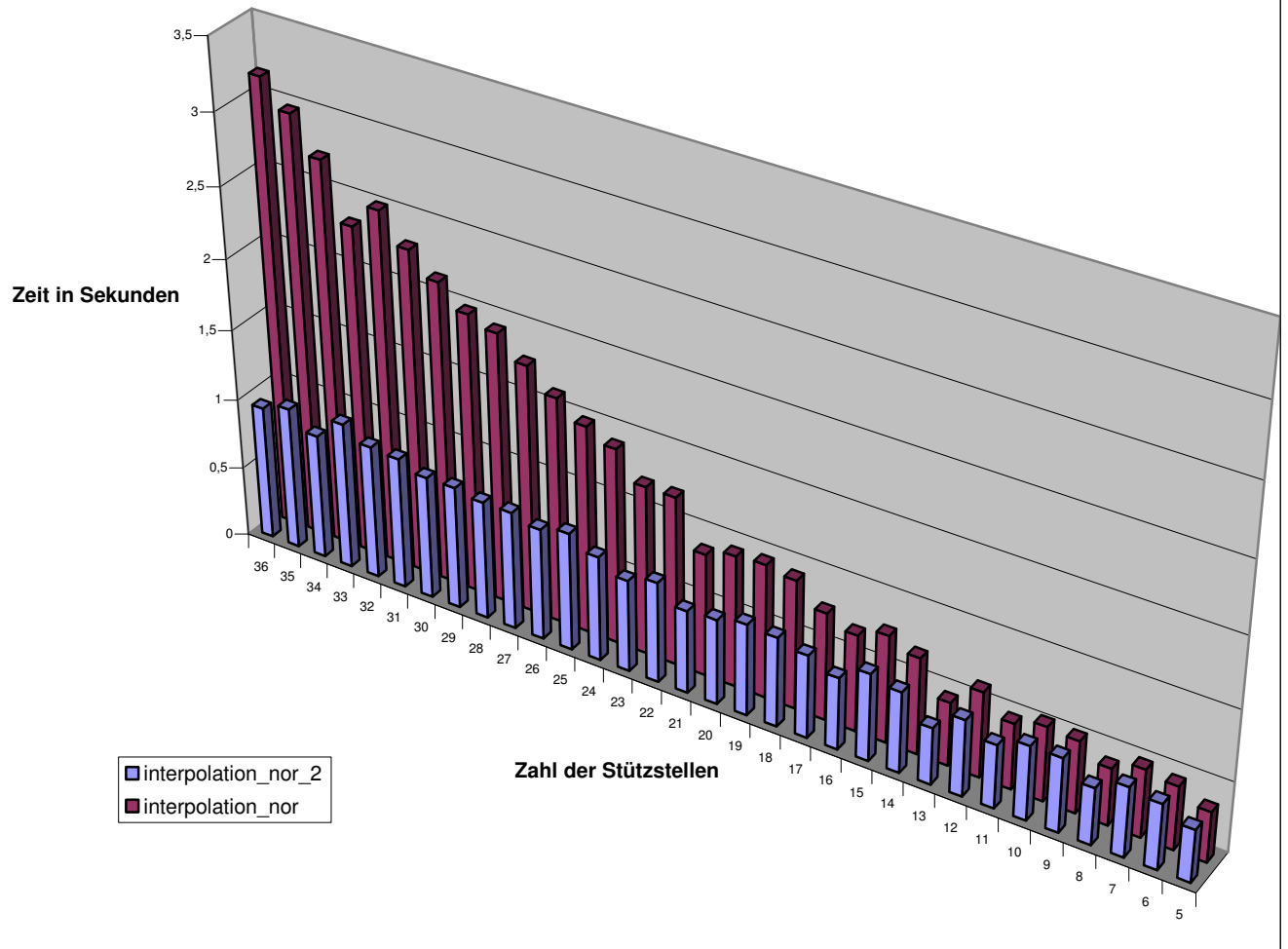
Benchmark für Interpolation (MDMS32_Bandpass.spt)



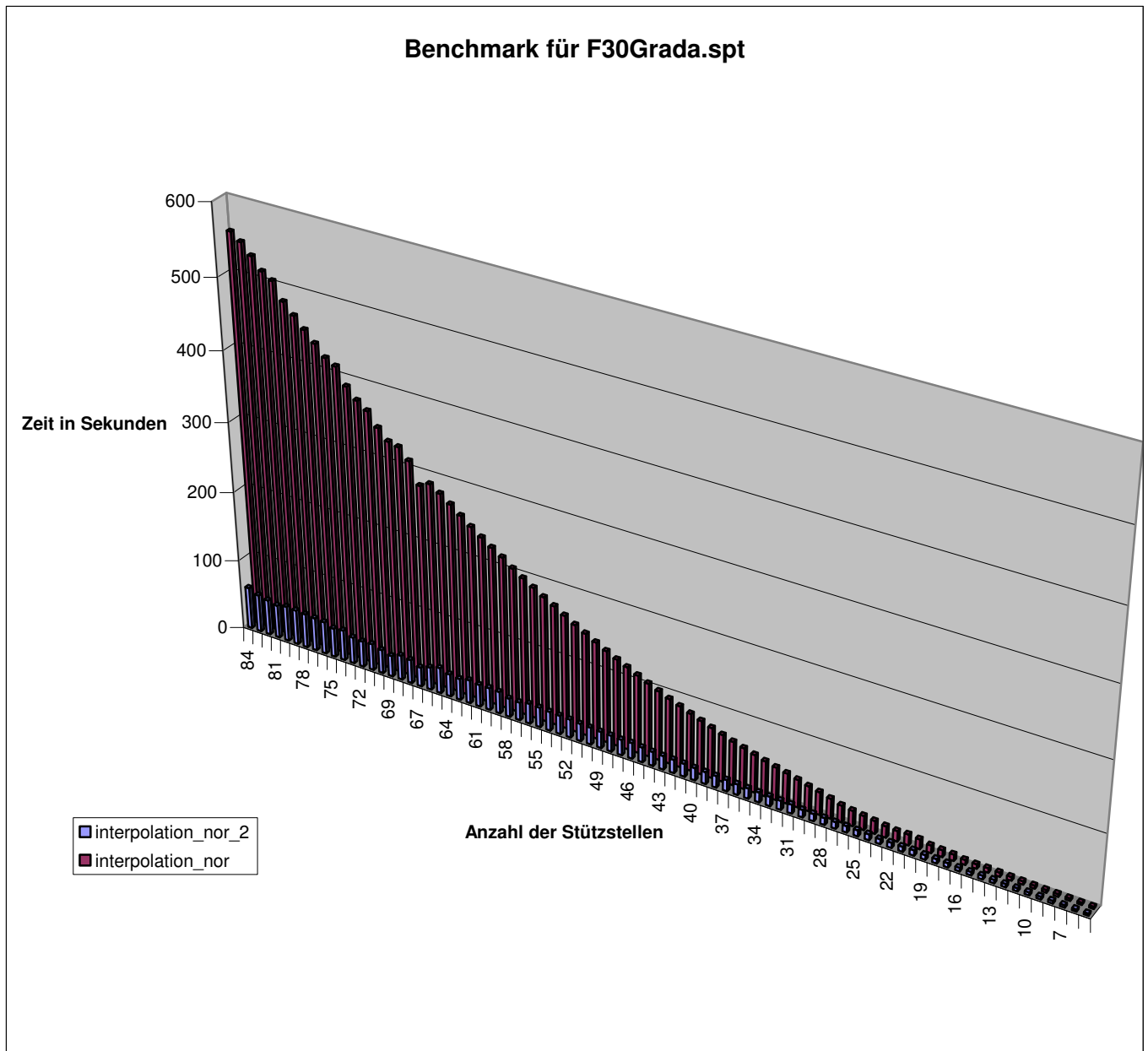
Zu Testzwecken wurde ein einfacher 2-Port simuliert. Mein neues Interpolationsprogramm ist hier mit „interpolation_nor“ gekennzeichnet. „interpolation_nor_2“ ist die modifizierte Version, bei der die Fehlerrechnung nur auf ein Intervall um den zu interpolierenden Punkt beschränkt ist. „interpolation_ratint“ ist das Interpolationsprogramm von Thomas Rolfes.

Das Diagramm zeigt, wie sich die Laufzeit der einzelnen Programme mit der Anzahl der bekannten Stützstellen verändert.

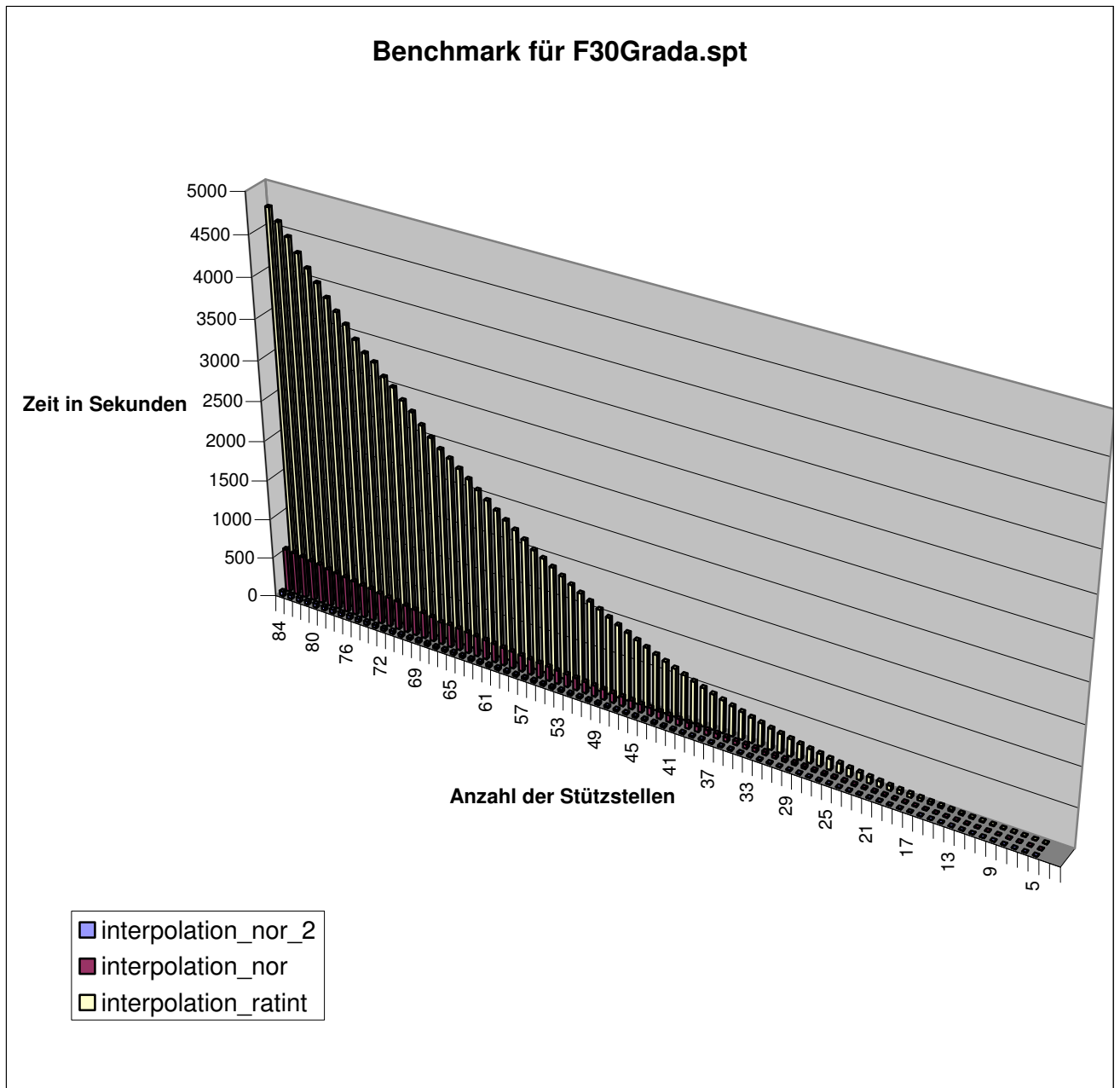
Direkter Vergleich zwischen interpolation_nor und interpolation_nor_2



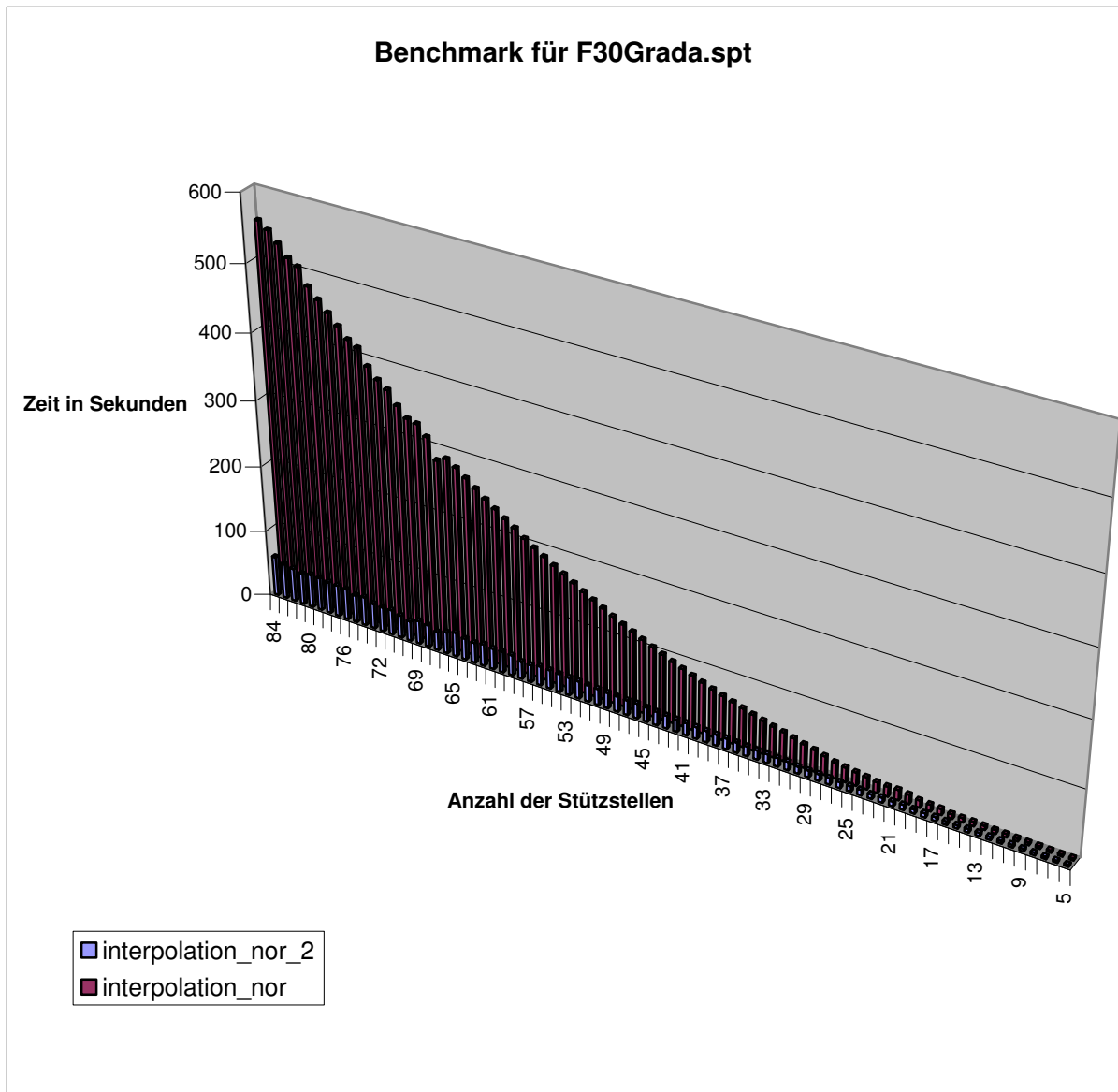
Bei meiner zweiten Programmversion („interpolation_nor_2“) steigt die Laufzeit mit der Anzahl der Stützpunkte weniger stark, da für die Fehlerfunktion jeweils nur eine feste Anzahl von Stützstellen betrachtet wird.



Für diesen Test wurde ein 8-Port mit besonders vielen Frequenzpunkten simuliert. Wie man anhand der obigen Grafik sehen kann, haben beide Programmversionen für bis zu 8 Punkten das selbe Laufzeitverhalten – was auch zu erwarten war. Bei steigender Anzahl von Stützstellen wird „interpolation_nor“ deutlich langsamer, weil dort für die Fehlerfunktion jede Stützstelle berücksichtigt wird. Bei „interpolation_nor_2“ werden hingegen nur 8 Stützstellen bei der Fehlerfunktion berücksichtigt. Trotzdem erhöht sich auch die Laufzeit von „interpolation_nor_2“ leicht, da der Bulirsch Stoer Algorithmus mit steigender Anzahl von Stützstellen auch etwas länger für die Interpolation braucht.



Für diesen Test wurde eine Simulation durchgeführt, bei der sehr viele (84) Frequenzpunkte tatsächlich simuliert wurden. Der Vergleich zeigt, dass die neuen Interpolationsprogramme dem alten „interpolation_ratint“ deutlich überlegen sind, vor allem dann, wenn sehr viele Stützstellen in die Rechnung miteinbezogen werden.



Der direkte Vergleich zwischen „interpolation_nor“ und „interpolation_nor_2“ zeigt, dass die modifizierte Fehlerrechnung, bei der nur ein Teilintervall von Stützstellen in der Nähe des zu interpolierenden Punktes berücksichtigt werden eine deutliche Verbesserung des Laufzeitverhaltens bringt. Während „interpolation_nor“ bei 84 Punkten über 9 Minuten für die Interpolation braucht, ist „interpolation_nor_2“ schon nach ca. einer Minute fertig.

Fazit

Die neue Version des Interpolationsprogramms ist wesentlich schneller als die Vorgängerversion, was sich insbesondere bei vielen Stützstellen positiv bemerkbar macht.

Der Code ist in übersichtliche Module unterteilt, was die Pflege stark vereinfacht.

Zeitplan

ID	Vorgangsname	Anfangsdatum	Enddatum	Dauer	2006		
					Januar	Februar	
1	Analyse	02.01.2006	06.01.2006	5t			
2	Planung (Version 1)	05.01.2006	11.01.2006	5t			
3	Implementierung (Version 1)	07.01.2006	26.01.2006	14t			
4	Test (Version 1)	23.01.2006	09.02.2006	14t			
5	Planung (Version 2)	02.02.2006	03.02.2006	2t			
6	Implementierung (Version 2)	03.02.2006	07.02.2006	3t			
7	Test (Version 2)	07.02.2006	09.02.2006	3t			

Literaturverzeichnis

Programmieren in C
von Brian W. Kernighan und Dennis M. Ritchie
ISBN 3-446-15497-3

Softwaretechnik in C und C++
von Rolf Isernhagen
ISBN 3-4461-8201-2

C kurz & gut
von Peter Prinz und Ulla Kirch-Prinz
ISBN 3-89721-238-2

C++ Der schnelle Einstieg
von Peter Wollschlaeger
ISBN 3-8272-6501-0

Die C++ Programmiersprache
von Bjarne Stroustrup
ISBN 3-8273-2058-5

Numerical Recipes in C++. The Art of Scientific Computing.
von William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery
ISBN 0521750334

Taschenbuch der Mathematik
von Ilja N. Bronstein, Konstantin A. Semendjajew, Gerhard Musiol
ISBN 3817120060

Elektronik für Ingenieure und Naturwissenschaftler
von Ekbert Hering, Klaus Bressler, Jürgen Gutekunst
ISBN 3540243097