```
private CZahl AddZahl(CZahl Z1, CZahl Z2)
{
    int i, zres;
    int dimdiff, len1, len2, lendiff;
    byte[] summand1, summand2, overflow, result;


    dimdiff = Zahl1.Dimension - Zahl2.Dimension;

    len1 = Zahl1.Numbers.GetUpperBound(0) + 1;
    len2 = Zahl2.Numbers.GetUpperBound(0) + 1;


    if (dimdiff > 0) len2 += Math.Abs(dimdiff);
    if (dimdiff < 0) len1 += Math.Abs(dimdiff);

    lendiff = len2 - len1;

    if (lendiff > 0) len1 += Math.Abs(lendiff);
    if (lendiff < 0) len2 += Math.Abs(lendiff);

    len1++; len2++; // probeweise
    if (dimdiff == 0) { len1++; len2++; }

    summand1 = new  byte[len1];
    summand2 = new byte[len1];
    overflow = new byte[len1];
    result = new byte[len1];

    summand1[0] = 0;
    summand2[0] = 0;
    overflow[0] = 0;
    result[0] = 0;
```


Jonas Haase (15)

Wettbewerb „Jugend Forscht“ 2015

Arbeitsgemeinschaft „Jugend Forscht“ des

Christian Gymnasium Hermannsburg

Betreuung: StD Thomas Biedermann

Inhaltsverzeichnis

Einleitung

Wie in meiner Kurzfassung beschrieben, haben wir uns im Mathematik Unterricht mit der Zahl Pi beschäftigt. Nach einigen Versuchen die Zahl mit dem Taschenrechner zu berechnen stellte ich fest, dass mein Taschenrechner welcher mit bis zu 100 Ziffern langen Zahlen rechnen kann nicht für diese Rechnungen ausreichte. Ich tätigte die ersten Überlegungen mit meinem Lehrer und testete auch andere Rechner wie z.B. jenen, der bei Windows vorinstalliert ist, welcher jedoch schnell an seine Grenzen stieß als ich probierte die Rechenmethode, welche ich vorher entwickelt hatte, auszuführen. Diese errechnet den Umfang des Einheitskreises also 2 x Pi pro Rechnung etwas genauer, indem sie das Ergebnis des vorherigen Rechenschrittes verwendet, wozu der Satz des Pythagoras genutzt wird.

Das Problem welches sich mir bei der Programmierung in den Weg stellte ist, dass das Zahlenformat welches man üblicherweise für Gleitkommazahlen in C# benutzt (double) für meine Rechnungen nicht ausreicht, da es mit ca. 8-20 Stellen zu ungenau für eine genauere Berechnung ist und auch kein Zahlenformat in diesem Programm vorhanden ist mit dem man Rechnungen ausführen kann und gleichzeitig die Zahlenlänge nahezu unbeliebig machen kann. Deshalb muss ich in meinem Programm ein Zahlenformat selbst "erstellen", welches die Ziffernfolge und die Position des Kommas speichert. Da man bei Zahlen mit einer unterschiedlichen Länge an Nachkommastellen nicht einfach die erste Ziffer der einen mit der ersten Ziffer der anderen Zahl zusammen rechnen kann ist dies ein komplizierter Vorgang der mit der Zahl geschehen muss. Es gibt natürlich schon Rechner die mit so komplexen Zahlen rechnen können, jedoch ist dies bei den meisten Taschenrechnern und Rechnern auf dem Computer nicht der Fall. Außerdem wird es mir, wenn der Rechner alle Rechenarten rechnen kann möglich sein, eine Iteration einzubauen und somit die Berechnung von Pi möglichst genau vorzunehmen, was eigentlich die Idee dieses Projektes entstehen ließ.

Mein Ziel ist es, dass der Rechner nach der Fertigstellung addieren subtrahieren, multiplizieren, dividieren, potenzieren und radizieren kann weil diese Rechenarten auch zur genaueren Bestimmung von Pi unter der Verwendung des Satzes von Pythagoras durch iterative Berechnung benötigt werden. Sollte mir das gelungen sein werde ich die Genauigkeit meines Ergebnisses mit anderen Ergebnissen vergleichen.

Vorgehensweise und Methoden

Wie bereits beschrieben habe ich bei meinem Projekt die Programmiersprache C# benutzt. Da die Zahlenformate welche von C# zum rechnen vorhanden sind in der Länge nicht ausreichen konvertierte ich die Zahl in ein Format, welches alle Informationen über die Zahl enthält, die ich zum rechnen benötige. Dazu strukturierte ich mit dem Befehl "struct" ein Zahlenformat in dem ich die Ziffernfolge (zur Kontrolle) als string, die Ziffernfolge in einem Byte-Array zur Rechnung um auf einzelne Ziffern besser zugreifen zu können, das Vorzeichen der Zahl als integer (1 oder -1) und die Dimension, welche die Position des Kommas gezählt vom Anfang der Zahl (bei Zahlen >1) angibt, speichere. Die Dimension spielt eine sehr wichtige Rolle wenn diese unterschiedlich ist, da wenn man eine Zahl addiert und dabei die einzelnen Ziffern addiert man durch diese die miteinander zu addierenden Ziffern bestimmen kann.

	Bsp. 1	**Bsp. 2**
string	123,45	0,004
Ziffernfolge	12345	0004
Byte-Array	1, 2, 3, 4, 5	4
Dimension	3	-2

```
struct CZahl
        {
            public string Zahl;
            public byte[] Numbers;
            public int Dimension;
            public int Vorzeichen;
        }
```

Nach dem Erstellen des Formates müssen aus der Ausgangszahl, welche in ein Textfeld eingegeben worden ist die benötigten Informationen gesammelt werden (Ziffernfolge als string und Byte-Array, Dimension und Vorzeichen). Dazu erstelle ich zunächst eine Funktion namens MakeZahl und in dieser die zur Berechnung benötigten Variablen.

```
private CZahl MakeZahl(string Ziffernfolge)
        {
            CZahl Z = new CZahl();
            byte[] ca;
            int i,j;
```

Anhand einer if Abfrage kann ich das Vorzeichen Bestimmen. Ich gehe anfangs davon aus, dass die Zahl positiv ist, da wenn kein Vorzeichen vorhanden ist dies zutrifft. Falls jedoch ein Minus-Zeichen vorhanden ist wird der integer in -1 geändert. Wenn Die Zahl 0 wäre würde das Vorzeichen als Plus gespeichert werden. Dies wäre zu vernachlässigen.

```
            Z.Vorzeichen = 1;
            if (Ziffernfolge[0] == '-')
            {
                Z.Vorzeichen = -1;
                Ziffernfolge = Ziffernfolge.Substring(1);
            }
            if (Ziffernfolge[0] == '+')
            {
                Ziffernfolge = Ziffernfolge.Substring(1);
```

Daraufhin werden führende Nullen entfernt, weil diese in dem neuen Format keine relevante Rolle spielen, wenn diese in die die Dimension einberechnet wurden (z.B. 0,0000034).

```
            if (Ziffernfolge.Length != 1)
                {
```

```
                while (Ziffernfolge[0] == '0') Ziffernfolge =
Ziffernfolge.Substring(1);
                }
```

Wenn eine Zahl kein Komma besitzt wir dies zur einfacheren Bestimmung der Dimension am Ende hinzugefügt.

```
            if (Ziffernfolge.IndexOf(",") == -1) Ziffernfolge += ",";
```

Nun beginnt die Bestimmung der Dimension. Die Dimension beschreibt bei Zahlen >=0,1 die Verschiebung des Kommas, welche benötigt wird, damit es vor der ersten Ziffer (von links) steht, welche mit der Position des Kommas übereinstimmt. Bei kleineren Zahlen als 0,1 werden die Nullen, welche auf das Komma folge gezählt bis die erste andere Ziffer kommt gezählt und diese Dimension ist dann negativ.

```
            if (Ziffernfolge[0] == ',')
            {
                for (i = 1; Ziffernfolge[i] == '0'; i++) ;
                Z.Dimension = -i + 1;
            }
            else Z.Dimension = Ziffernfolge.IndexOf(",");
```

Da nun die Dimension fertig bestimmt ist werden das Komma und führende Nullen bei Zahlen <0,1 nicht mehr benötigt und können entfernt werden, da ein Komma die Rechnung erschweren würde.

```
if (Ziffernfolge[0] == ',') Ziffernfolge = Ziffernfolge.Substring(1);
            while (Ziffernfolge[0] == '0') Ziffernfolge =
Ziffernfolge.Substring(1);

if (Ziffernfolge.IndexOf(",") != -1) Ziffernfolge =
Ziffernfolge.Remove(Ziffernfolge.IndexOf(","), 1);
```

Nun kann die Ziffernfolge einem bereits erstelltem Byte-Array zugewiesen werden und CZahl Z kann mit den nun zur Verfügung stehenden Werten zurückgegeben werden.

```
      for (i = 0; i < Ziffernfolge.Length; i++)
                  {
                      ca[i] = Convert.ToByte(Ziffernfolge[i]);
                      ca[i] -= 48;
                  }

      Z.Numbers = ca;
      return (Z);
        }
```

Anhand der Addition möchte ich nun eine Rechenweise genauer erläutern, da die Grundprinzipien der Rechenweise sich häufig überschneiden, es nur noch etwas komplizierter wird, da z.B. beim Subtrahieren auch auf negative Zahlen Rücksicht genommen werden muss und auch die Länge der Zahl stark verändert werden kann. Zunächst wird wieder eine Funktion erstellt namens AddZahl und benötigte Variable werden hinzugefügt.

```
        private CZahl AddZahl(CZahl Z1, CZahl Z2)
        {
            int i, zres, dimdiff, len1, len2, lendiff;
            byte[] summand1, summand2, overflow, result;
```

Um die Länge der zu Berechnenden Zahl zu bestimmen werden zunächst die Lägen der zwei Summanden bestimmt. Außerdem wir die Differenz der zwei Dimensionen bestimmt, um die Verschiebung die bei der Berechnung auftreten wird zu bestimmen und um sie auf die Länge einer Zahl zu addieren, weil der Dimensionsunterschied die Länge der Zahl des Ergebnisses beeinflusst. Wenn der Dimensionsunterschied positiv ist, wird er auf die Länge der 2. Zahl gerechnet, da zahl dann die größere Dimension haben muss. Wenn sie negativ ist wird sie auf die Länge der 1. Zahl gerechnet.

```
            len1 = Zahl1.Numbers.GetUpperBound(0) + 1;
            len2 = Zahl2.Numbers.GetUpperBound(0) + 1;

            dimdiff = Zahl1.Dimension - Zahl2.Dimension;

            if (dimdiff > 0) len2 += Math.Abs(dimdiff);
            if (dimdiff < 0) len1 += Math.Abs(dimdiff);
```

Ebenso geschieht es mit der Längendifferenz. Wenn diese positiv ist, also die Länge der 2. Zahl größer als die Länge der 1. Zahl ist wird diese auf die Länge der ersten Zahl und wenn sie negativ ist auf die Länge der zweiten Zahl gerechnet. Wenn die Dimensionsdifferenz 0 ist wird das Byte-Array um 1 vergrößert, da es dann möglich ist, dass wie bei z.B. 5+6 die Zahl um eine Ziffer verlängert wird da das Ergebnis 11 ist.

```
            lendiff = len2 - len1;

            if (lendiff > 0) len1 += Math.Abs(lendiff);     // lendiff > 0:
Stellen bei erster Zahl zufügen
            if (lendiff < 0) len2 += Math.Abs(lendiff);     // lendiff < 0:
Stellen bei zweiter Zahl zufügen

            len1++; len2++;

            if (dimdiff == 0) { len1++; len2++; }
```

Da len1 und len2 nun beide gleich sind und der Länge des für das Ergebnis benötigte Byte-Array entsprechen und diese Länge auch für die Byte-Arrays der zwei Summanden wichtig ist, weil es einfacher zu rechnen ist, wenn man die Ziffer auf der ersten Position von Zahl eins mit der Ziffer von der ersten Position von Zahl 2 addieren kann werden für die zwei Summanden, für das Ergebnis und für den Überschlag, welcher beim addieren zustande kommen kann und bei der nächsten Addition der folgenden Ziffern berücksichtigt werden muss und in diesem Byte-Array zwischengespeichert wird.

```
            summand1 = new  byte[len1];
            summand2 = new byte[len1];
            overflow = new byte[len1];
            result = new byte[len1];
```

Der ersten Stelle wird in allen Byte-Arrays eine 0 zugewiesen, da diese Stelle für eine mögliche Verlängerung der Zahl vorbehalten werden soll, falls am Ende ein Überschlag vorhanden sein sollte.

```
            summand1[0] = 0;
            summand2[0] = 0;
            overflow[0] = 0;
            result[0] = 0;
```

Wenn die Dimensionsdifferenz größer als Null ist wird Zahl 1 in summand1 übernommen ohne dass sich die Position verändert. Dementsprechend werden bei Zahl zwei an den “dimdiff“ vielen stellen Nullen eingesetzt, damit die Ziffern an gleicher Position in den Zahlen addiert werden können. Erst nachdem dies geschehen ist werden hinter die Nullen die eigentlischen Ziffern der Zahl gesetzt. Wenn die Dimensionsdifferenz 0 ist können die Zahlen übernommen werden ohne dass vorher Nullen eingefügt werden müssen.

```
            if (dimdiff > 0)
            {
                for (i = 0; i <= Zahl1.Numbers.GetUpperBound(0); i++)
summand1[i+1] = Zahl1.Numbers[i];
                for (i = 1; i < dimdiff; i++) summand2[i] = 0;
                for (i = 0; i <= Zahl2.Numbers.GetUpperBound(0); i++)
            {
            summand2[i + dimdiff+1] = Zahl2.Numbers[i]
            }
            }
            if (dimdiff == 0)
            {
                for (i = 0; i <= Zahl1.Numbers.GetUpperBound(0); i++)
summand1[i+1] = Zahl1.Numbers[i];
                for (i = 0; i <= Zahl2.Numbers.GetUpperBound(0); i++)
summand2[i+1] = Zahl2.Numbers[i];
            }
```

Wenn die Dimensionsdifferenz negativ ist geschieht selbiges wie wenn die DImensionsdifferenz positiv ist, nur dass statt Zahl1 und summand1 Zahl2 und summand2 (und andersrum) genutzt werden.

```
            if (dimdiff < 0)
            {
                for (i = 0; i <= Zahl2.Numbers.GetUpperBound(0); i++)
summand2[i+1] = Zahl2.Numbers[i];
```

```
                for (i = 1; i < -dimdiff; i++) summand1[i] = 0;
                for (i = 0; i <= Zahl1.Numbers.GetUpperBound(0); i++)
summand1[i - dimdiff + 1] = Zahl1.Numbers[i];
            }
```

Nun kann die eigentliche Addition folgen. Es wird von rechts nach links addiert, weshalb man mit der Ziffer auf Position len1 – 1 startet, diese Abfrage bis i kleiner als Null ist durchlaufen lässt und i pro Schleife verkleinert wird. zres ist das Zwischenresultat welches durch das Addieren der zwei Summanden plus den Überschlag zustande kommt. Wenn dieses höher als 10 ist muss für die nächste Ziffer der Überschlag 1 in dem Byte-Array overflow gespeichert werden. Außerdem muss dann von zres 10 abgezogen werden und als Ergebnis dieser Stelle eingesetzt werden. Ist das Ganze nicht der Fall kann zres übernommen werden

	Dimension	Dezimalzahl			
Summand 1	2	34	3	4	
Summand 2	1	7,9	+	7	9
Überschlag			1		
Ergebnis		41,9	4	1	9

```
            for (i = len1-1; i > 0; i--) {
                zres = summand1[i] + summand2[i] + overflow[i];
                if (zres > 9)
                {
                    result[i] = (byte)(zres - 10);
                    overflow[i - 1] = 1;
                }
                else
                    result[i] = (byte) zres;
            }
```

Wenn bei der letzten Teiladdition ein Überschlag entstanden ist wird ist die erste Ziffer des Byte-Arrays vom Ergebnis gleich der erste Ziffer des Byte-Arrays vom Überschlag.

```
            result[0] = overflow[0];
```

Es wird eine neue CZahl erstellt, damit ihr die neuen Eigenschaften des Ergebnisses zugeteilt werden können.

```
            CZahl Ergebnis = new CZahl();
```

Wenn an der ersten Stelle des Ergebnisses ein Überschlag hinzugefügt worden ist sind die Dimensionen gleich und somit kann die Dimension von Zahl2 übernommen werden jedoch muss +1 gerechnet werden, da die Zahl durch den Überschlag um eine Stelle verlängert wurde und deshalb die Dimension vergrößert werden muss.
Ergebnis.Numbers kann nun das Ergebnis (result) zugewiesen werden.

```
            if (result[0] != 0)
            {
                Ergebnis.Dimension = Zahl2.Dimension + dimdiff + 1;
                Ergebnis.Numbers = result;
            }
```

Wenn die Ziffer an Position eins (null) null ist wird ein neues Byte-Array erstellt welche, das Ergebnis um die führende Null gekürzt zugewiesen wird. Es wird die größere Dimension übernommen oder bei Gleichheit eine der beiden.
Ergebnis.Numbers kann nun das Ergebnis (overflow) zugewiesen werden.

```
            else
            {
                overflow = new byte[len1 - 1];
                for (i = 0; i < result.GetUpperBound(0); i++) overflow[i] =
result[i + 1];

                if (Zahl1.Dimension > Zahl2.Dimension)Ergebnis.Dimension =
Zahl1.Dimension;
                // Dimension von Zahl 1 ist größer --> Dimension des
Ergebnis ist Dimension von Zahl 1

                if (Zahl2.Dimension > Zahl1.Dimension)Ergebnis.Dimension =
Zahl2.Dimension;
                // Dimension von Zahl 2 ist größer --> Dimension des
Ergebnis ist Dimension von Zahl 2

                if (Zahl1.Dimension == Zahl2.Dimension) Ergebnis.Dimension
= Zahl2.Dimension;
                // Beide Dimensionene sind gleich --> irgendeine alte
Dimension wird übernommen

                Ergebnis.Numbers = overflow;
            }
```

Ergebnisse

Da die Fertigstellung des Rechners noch nicht erfolgt ist konnte er noch nicht auf die endgültige Genauigkeit überprüft werden. Momentan funktionieren die Addition und Subtraktion einwandfrei jedoch weisen Multiplikation und Division noch Fehler auf. Da ich die die funktionierenden Rechenarten auf Fehler überprüft habe und die Länge der Summanden keine Rolle spielt solange die Maximallänge eines Byte-Arrays, welche bei > 65000 liegt, nicht überschritten wird ist davon auszugehen, dass diese mit extrem hoher Genauigkeit rechnen können.

Zusammenfassung

Mein Rechner welcher ursprünglich zur möglichst genauen Errechnung von Pi dienen sollte kann momentan addieren, subtrahieren und vielleicht auch bald ohne Fehler multiplizieren und dividieren. Das Radizieren und Potenzieren sollte keine große Herausforderung sein wenn die Multiplikation funktioniert.

Mein direktes Ziel habe ich also zu diesem Zeitpunkt noch nicht erreicht jedoch habe ich eine Methode zur weiteren Entwicklung des Rechensystems uns sobald dies funktioniert werde ich die Berechnung von Pi vornehmen.

Hilfsmittel/Personen

Thomas Biedermann (AG-Leiter und Projekt Betreuer) Unterstützung bei der Themenwahl, Hilfe bei der Entwicklung der Rechenart und dem Aufbau des Programmes

Microsoft Visual C#