

ERA Praktikum SS 2017
Gleitkomma-Arithmetik
Realisierung von $\sin x$

ERA Praktikum SS 2017	1
1. Anwenderdokumentation	3
1.1 Einleitung	3
1.2 Beschreibung der Funktionalität des Programms	3
1.3 Bedienung	4
2. Entwicklerdokumentation	8
2.1 Einleitung	8
2.2 Lösungsidee	8
2.3 Implementierung	9
2.4 Test	18
2.5 Erweiterungsoptionen	19
3. Verfasser	19

1. Anwenderdokumentation

1.1 Einleitung

Im Rahmen des ERA Praktikums an der TU München wurde die Realisierung der Sinusfunktion in Assembler umgesetzt. Im Folgenden findet sich die Anwenderdokumentation, die Benutzern die Funktionsweise und Anwendung des Programms erläutern soll.

1.2 Beschreibung der Funktionalität des Programms

In Assembler wurde ein Algorithmus implementiert, der den Wert der Sinusfunktion zu beliebigen Eingaben annähert und hierfür ausschließlich die vier Grundrechenarten und die Negation verwendet.

Um den gesuchten Wert zu bestimmen, sind in einer Tabelle vorberechnete Sinuswerte gespeichert. Da sich die Sinusfunktion in 2π Intervallen wiederholt, ist es ausreichend Werte innerhalb eines Intervalls zu speichern.

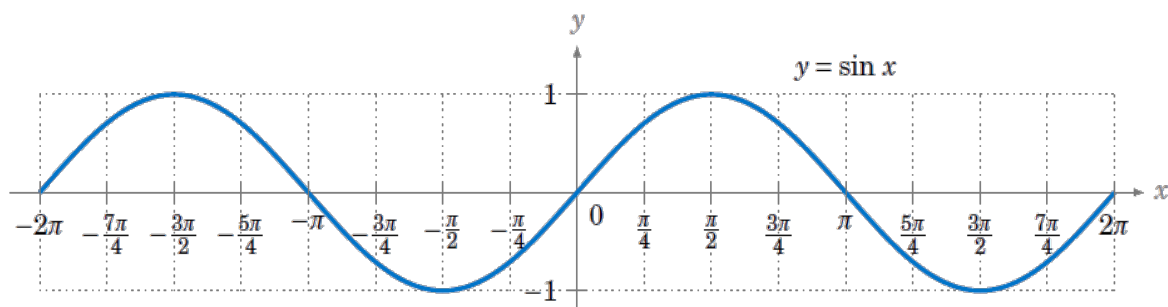


Abbildung 1: Sinusfunktion

Eingabewerte die über dieses Intervall hinausgehen, werden mithilfe einer Modulo-Rechnung mit 2π auf das Intervall abgebildet. (Die Modulo-Rechnung bestimmt den Rest bei ganzzahliger Division durch 2π .)

Anschließend sucht der Algorithmus in der Tabelle nach den benachbarten gespeicherten Werten zum gesuchten Eingabewert, also nach dem größten Wert der grade noch kleiner ist als der Eingabewert und nach dem kleinsten Wert der grade noch größer ist als der Eingabewert. Sind diese beiden Werte gefunden, wird die prozentuale Nähe des Eingabewerts zum kleineren benachbarten Wert berechnet (siehe Abbildung 2).



Abbildung 2

Auf Grundlage dieser prozentualen Nähe wird nun der Sinuswert für den Eingabewert aus den Sinuswerten der beiden gespeicherten Werte zusammengesetzt.

Die Sinusfunktion wiederholt sich nicht nur in 2π Intervallen, sondern ist auch im Intervall 0 bis 2π punktsymmetrisch am Punkt $(\pi, 0)$, sowie im Intervall 0 bis π spiegelsymmetrisch am Punkt $(\frac{\pi}{2}, 0)$. Um Platz bei der Speicherung der vorberechneten Werte zu sparen, wurden daher nur Werte im Intervall 0 bis $\frac{\pi}{2}$ in der Tabelle gespeichert und die übrigen Werte bis 2π durch Negation und Spiegelung hergeleitet. Nähe Details hierzu können der Entwicklerdokumentation entnommen werden.

1.3 Bedienung

Zur Ausführung des Assembler-Codes wurde ein C-Rahmenprogramm geschrieben, welches über das Terminal bedient werden kann. Da das C-Rahmenprogramm in Quellcode vorliegt, muss dieses zunächst kompiliert werden. Voraussetzungen hierfür sind unter macOS die Installation von x86 nasm, sowie des Clang Compilers.

Zur Kompilierung müssen sich die Dateien „Makefile“, „lut.asm“, „sin.asm“, „rahmenProgramm.c“ und „testProgramm.c“ im selben Verzeichnis befinden.

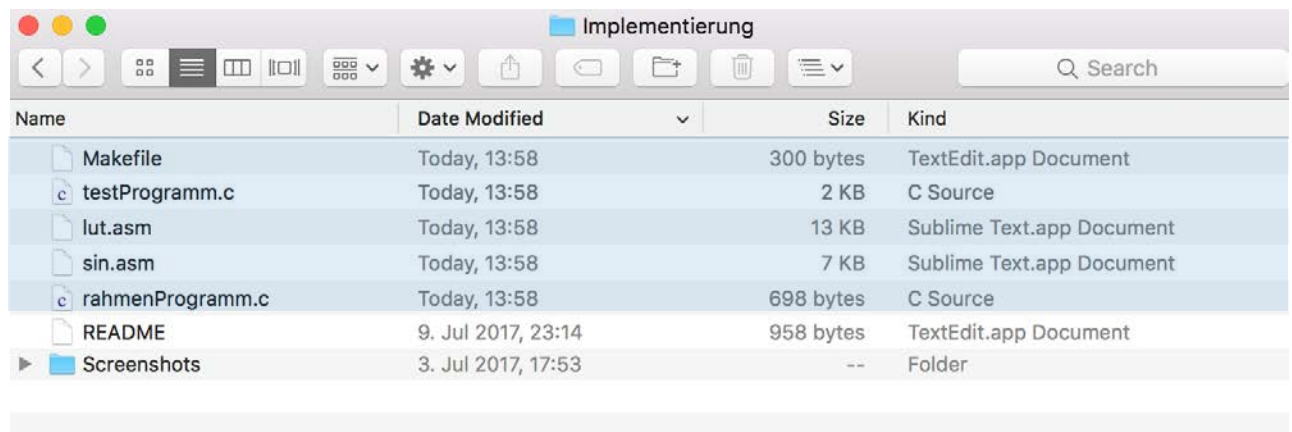
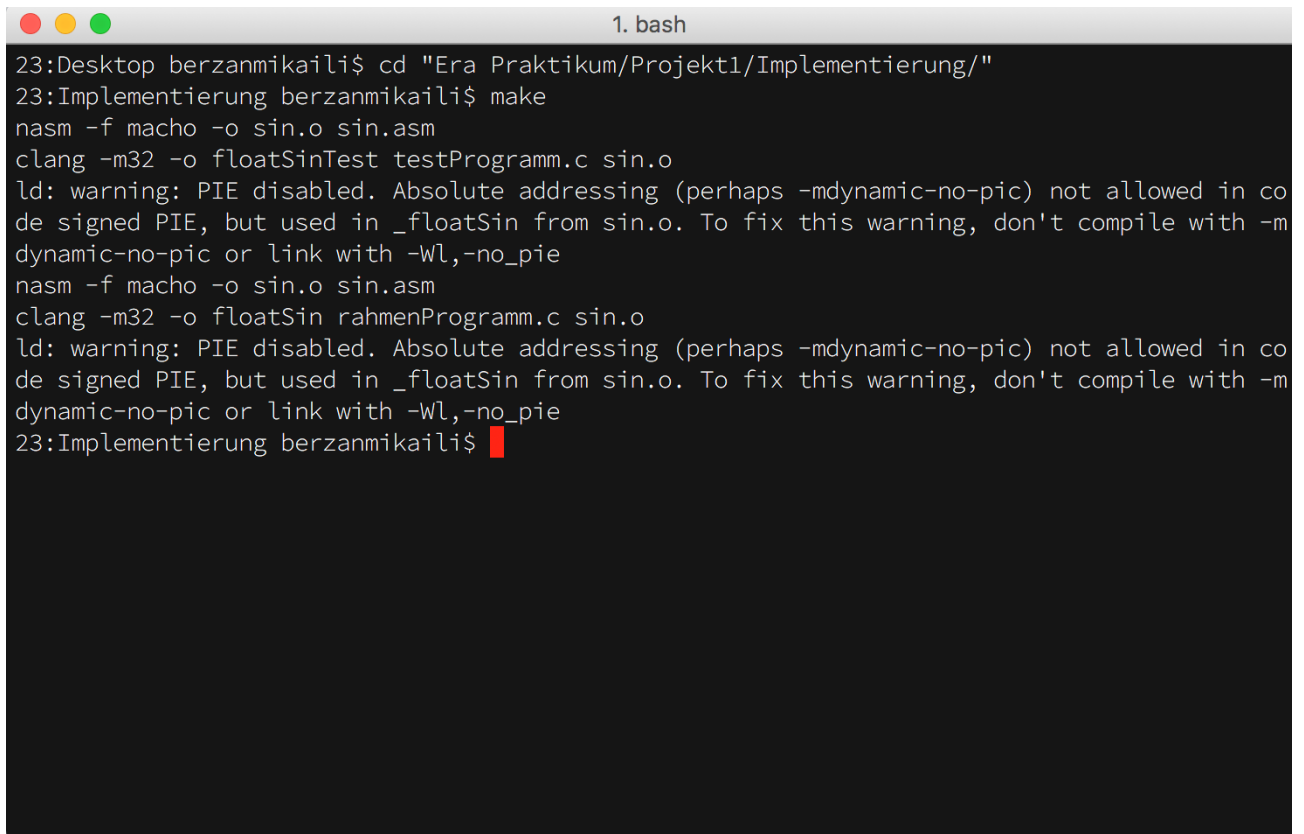


Abbildung 3: Benötigte Dateien sind farblich markiert

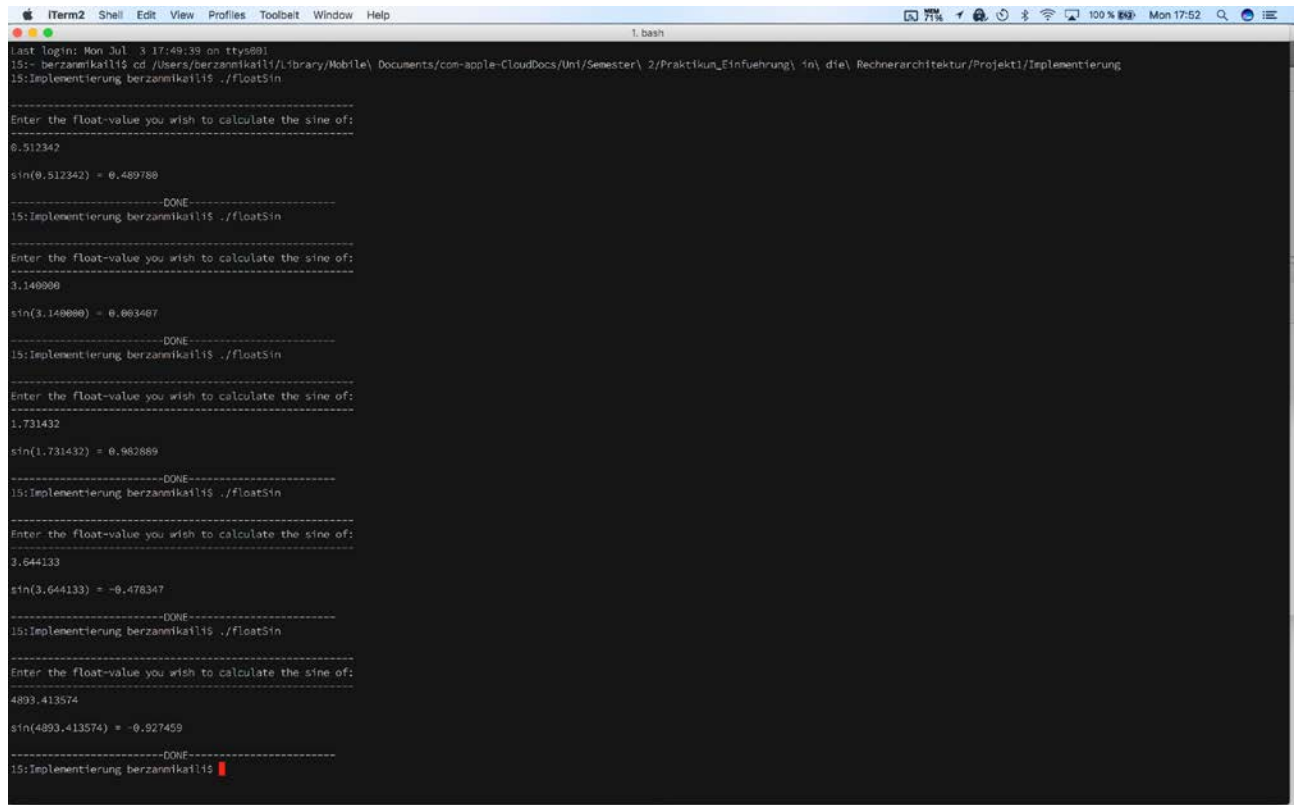
Im Terminal muss nun mithilfe des Befehls „cd [filepath]“ in eben dieses Verzeichnis navigiert werden, wobei [filepath] durch den Pfad zum Verzeichnis ersetzt werden muss. Anschließend kann das Programm mit dem Befehl „make“ kompiliert werden. (Siehe Abbildung 4.)

A screenshot of a terminal window titled "1. bash". The terminal shows a sequence of commands and their outputs. The user starts at a prompt "23:Desktop berzanmikaili\$". They enter "cd "Era Praktikum/Projekt1/Implementierung/"", and the prompt changes to "23:Implementierung berzanmikaili\$". Then they enter "make". The output shows "nasm -f macho -o sin.o sin.asm" and "clang -m32 -o floatSinTest testProgramm.c sin.o". Then they enter "ld: warning: PIE disabled. Absolute addressing (perhaps -mdynamic-no-pic) not allowed in code signed PIE, but used in _floatSin from sin.o. To fix this warning, don't compile with -mdynamic-no-pic or link with -Wl,-no_pie". They then enter "nasm -f macho -o sin.o sin.asm" and "clang -m32 -o floatSin rahmenProgramm.c sin.o". The same warning message appears again. Finally, they enter "23:Implementierung berzanmikaili\$".

```
1. bash
23:Desktop berzanmikaili$ cd "Era Praktikum/Projekt1/Implementierung/"
23:Implementierung berzanmikaili$ make
nasm -f macho -o sin.o sin.asm
clang -m32 -o floatSinTest testProgramm.c sin.o
ld: warning: PIE disabled. Absolute addressing (perhaps -mdynamic-no-pic) not allowed in code signed PIE, but used in _floatSin from sin.o. To fix this warning, don't compile with -mdynamic-no-pic or link with -Wl,-no_pie
nasm -f macho -o sin.o sin.asm
clang -m32 -o floatSin rahmenProgramm.c sin.o
ld: warning: PIE disabled. Absolute addressing (perhaps -mdynamic-no-pic) not allowed in code signed PIE, but used in _floatSin from sin.o. To fix this warning, don't compile with -mdynamic-no-pic or link with -Wl,-no_pie
23:Implementierung berzanmikaili$
```

Abbildung 4: Kompilierung

Sobald das Programm kompiliert ist, lässt es sich durch einen Doppelklick auf die entstandene „floatSin“ Datei im Verzeichnis starten. (Alternativ lässt es sich auch durch den Befehl „./floatSin“ über das Terminal starten.) Im Terminal wird nun die Aufforderung angezeigt, den gewünschten Eingabewert einzutippen (Abb. 5). Mit Enter kann die Eingabe bestätigt werden und einen Augenblick später wird das Ergebnis im Terminal angezeigt.



```
iTerm2 Shell Edit View Profiles Toolbar Window Help
1. bash
Last login: Mon Jul 3 17:49:39 on ttys00
15:~: berzannikall1$ cd /Users/berzannikall1/Library/Mobile Documents/com-apple-CloudDocs/Uni/Semester\ 2/Praktikum_Einfuehrung\ in\ die\ Rechnerarchitektur/Projekt1/Implementierung
15:Implementierung berzannikall1$ ./floatSin

-----
Enter the float-value you wish to calculate the sine of:
0.512342
sin(0.512342) = 0.489788
-----
15:Implementierung berzannikall1$ ./floatSin

-----
Enter the float-value you wish to calculate the sine of:
3.149999
sin(3.149999) = 0.003487
-----
15:Implementierung berzannikall1$ ./floatSin

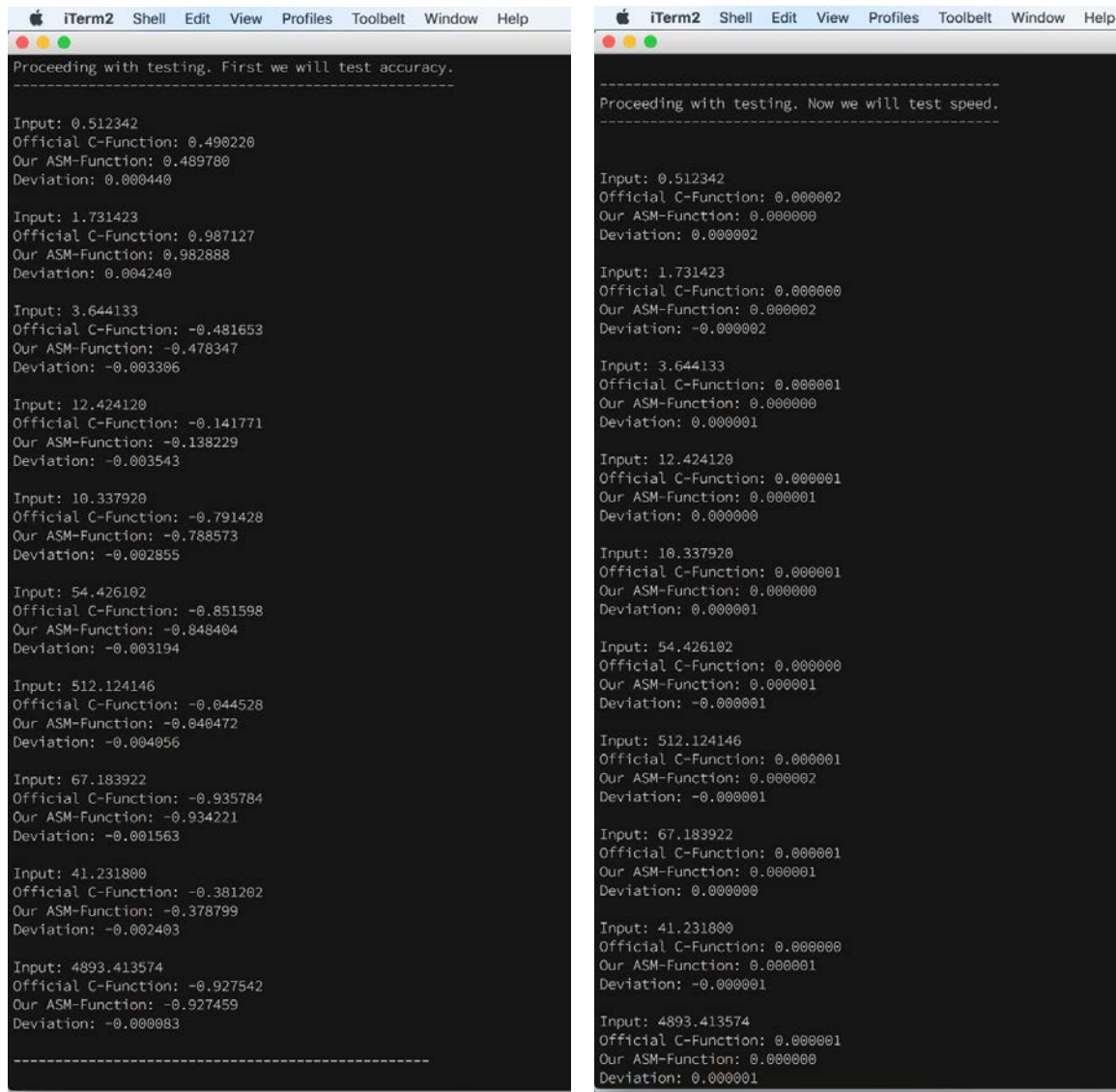
-----
Enter the float-value you wish to calculate the sine of:
1.731432
sin(1.731432) = 0.962889
-----
15:Implementierung berzannikall1$ ./floatSin

-----
Enter the float-value you wish to calculate the sine of:
3.644133
sin(3.644133) = -0.478347
-----
15:Implementierung berzannikall1$ ./floatSin

-----
Enter the float-value you wish to calculate the sine of:
4893.413574
sin(4893.413574) = -0.927459
-----
15:Implementierung berzannikall1$
```

Abbildung 5: Ausführung des Programms

Das Testprogramm kann durch Doppelklick auf die „floatSinTest“ Datei oder den Befehl „./floatSinTest“ gestartet werden. Hier werden 10 zufällig ausgewählte Werte in Hinsicht auf Genauigkeit und Geschwindigkeit mit der C-Sinusfunktion verglichen.



```
iTerm2 Shell Edit View Profiles Toolbelt Window Help
Proceeding with testing. First we will test accuracy.
-----
Input: 0.512342
Official C-Function: 0.490220
Our ASM-Function: 0.489780
Deviation: 0.000440

Input: 1.731423
Official C-Function: 0.987127
Our ASM-Function: 0.982888
Deviation: 0.004240

Input: 3.644133
Official C-Function: -0.481653
Our ASM-Function: -0.478347
Deviation: -0.003306

Input: 12.424120
Official C-Function: -0.141771
Our ASM-Function: -0.138229
Deviation: -0.003543

Input: 10.337920
Official C-Function: -0.791428
Our ASM-Function: -0.788573
Deviation: -0.002855

Input: 54.426102
Official C-Function: -0.851598
Our ASM-Function: -0.848404
Deviation: -0.003194

Input: 512.124146
Official C-Function: -0.044528
Our ASM-Function: -0.040472
Deviation: -0.004056

Input: 67.183922
Official C-Function: -0.935784
Our ASM-Function: -0.934221
Deviation: -0.001563

Input: 41.231800
Official C-Function: -0.381202
Our ASM-Function: -0.378799
Deviation: -0.002403

Input: 4893.413574
Official C-Function: -0.927542
Our ASM-Function: -0.927459
Deviation: -0.000083
-----

iTerm2 Shell Edit View Profiles Toolbelt Window Help
-----
Proceeding with testing. Now we will test speed.
-----
Input: 0.512342
Official C-Function: 0.000002
Our ASM-Function: 0.000000
Deviation: 0.000002

Input: 1.731423
Official C-Function: 0.000000
Our ASM-Function: 0.000002
Deviation: -0.000002

Input: 3.644133
Official C-Function: 0.000001
Our ASM-Function: 0.000000
Deviation: 0.000001

Input: 12.424120
Official C-Function: 0.000001
Our ASM-Function: 0.000001
Deviation: 0.000000

Input: 10.337920
Official C-Function: 0.000001
Our ASM-Function: 0.000000
Deviation: 0.000001

Input: 54.426102
Official C-Function: 0.000000
Our ASM-Function: 0.000001
Deviation: -0.000001

Input: 512.124146
Official C-Function: 0.000001
Our ASM-Function: 0.000002
Deviation: -0.000001

Input: 67.183922
Official C-Function: 0.000001
Our ASM-Function: 0.000001
Deviation: 0.000000

Input: 41.231800
Official C-Function: 0.000000
Our ASM-Function: 0.000001
Deviation: -0.000001

Input: 4893.413574
Official C-Function: 0.000001
Our ASM-Function: 0.000000
Deviation: 0.000001
```

Abbildung 6: Testprogramm

Über den Befehl „make clean“ kann das Verzeichnis anschließend wieder in die Ausgangslage versetzt werden.

2. Entwicklerdokumentation

2.1 Einleitung

Im Rahmen des ERA-Praktikums an der TU München wurde in einem dreimonatigen Projekt die Implementation der Sinusfunktion in Assembler umgesetzt. Im Folgenden findet sich die Entwicklerdokumentation, die Entwickler dabei unterstützen soll die Funktionsweise und Implementierung des Programms zu verstehen und eventuelle Adaptionen oder Erweiterungen schneller umsetzen zu können.

2.2 Lösungsidee

Die implementierte Lösungsidee basiert auf dem Einsatz einer Look-Up-Table, in der vorberechnete Funktionswerte gespeichert sind. Dieser Ansatz wurde aufgrund der enormen Vorteile im Bereich Effizienz und Geschwindigkeit gegenüber dem alternativen Ansatz der Reihenentwicklung vorgezogen. Unser Ansatz macht dabei Gebrauch von den periodischen Eigenschaften der Sinusfunktion. Zunächst wiederholt sich die Sinusfunktion in Perioden der Länge 2π . Daher kann durch eine Modularechnung jeglicher Eingabewert auf dieses Intervall $[0; 2\pi]$ abgebildet werden. Doch die Sinusfunktion wiederholt sich nicht nur in 2π Intervallen, sondern ist auch im Intervall $[0; 2\pi]$ punktsymmetrisch am Punkt $(\pi, 0)$, sowie im Intervall $[0; \pi]$ spiegelsymmetrisch am Punkt $(\frac{\pi}{2}, 0)$. Um weiteren Platz bei der Speicherung der vorberechneten Werte zu sparen, müssen daher nur Werte im Intervall 0 bis $\frac{\pi}{2}$ in der Tabelle gespeichert sein. Durch Spiegelung und Negation können aus den Funktionswerten der Sinusfunktion im Intervall $[0; \frac{\pi}{2}]$ alle anderen Funktionswerte hergeleitet werden.

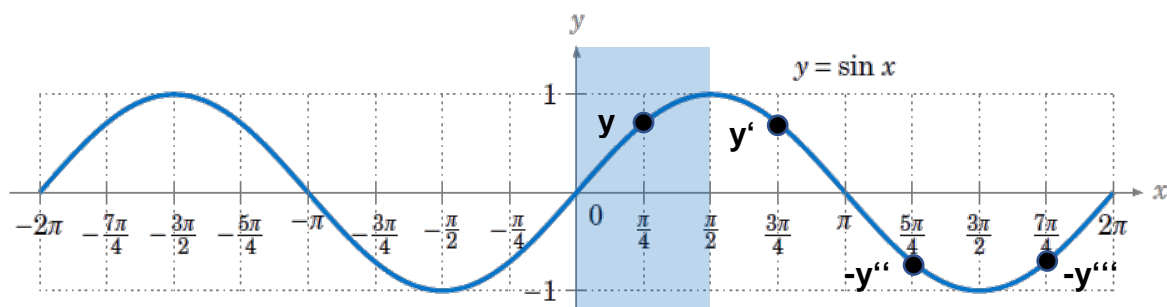


Abbildung 7

Da die Look-Up-Table geordnet im Speicher liegt, kann mit einer binären Suche sehr effizient nach Nachbarwerten zum Eingabewert gesucht werden. Mit Interpolation wird aus den Nachbarwerten der Funktionswert zum Eingabewert angenähert.

Um Interpolationsfehler zu vermeiden, sind in Bereichen, in denen die Sinusfunktion stärker steigt, mehr Stützstellen nötig, als an Hoch- und Tiefpunkten, an denen die Steigung merkbar kleiner ist. Daher ist der Abstand der gespeicherten X-Werte an Stellen starker Steigung deutlich dichter. Wie dies erreicht wurde, ist im Abschnitt 2.3 Implementierung unter „lut.asm“ beschrieben.

2.3 Implementierung

Die Implementierung teilt sich in zwei Assembler-Dateien („sin.asm“, „lut.asm“), sowie zwei C-Dateien („rahmenProgramm.c“, „testProgramm.c“). Im Folgenden wird nacheinander die Funktionsweise der verschiedenen Programme erläutert.

2.3.1 sin.asm

In „sin.asm“ befindet sich die die Hauptroutine des Programms.

Zunächst werden in der __data Section einige Speicherzellen initialisiert (Abb. 8). Um später bei den Berechnungen im FPU-Register darauf zugreifen zu können, wird am Label „two“ 2.0 als Konstante in den Speicher geschrieben. Die Speicherstelle am Label „sign“ dient später als Zwischenspeicherstelle für das Vorzeichen des Endergebnisses. Außerdem wird an dieser Stelle die Assembler-Datei „lut.asm“ eingebunden, in welcher die Look-Up-Table initialisiert wird. Weitere Details zur Look-Up-Table folgen im Abschnitt zu „lut.asm“.

```
9  section __DATA,__data
10
11  two: dd 2.0           ; 2 in den Speicher legen
12  sign: dw 1           ; Vorzeichen des Endergebnisses im Hauptspeicher ablegen
13  %include "lut.asm"
14
```

Abbildung 8

Als nächstes wird in der Section __bss Speicher für das Zwischenergebnis der Modulo-Rechnung, den abgebildeten Eingabewert, das Endergebnis und den ursprünglichen Eingabewert reserviert. (Abb. 9)

```
15  section __BSS,__bss
16
17  modResult: resd 1     ; Zwischenergebnis von Modulo
18  newValue: resd 1     ; Abgebildete Value
19  result: resd 1       ; Endergebnis
20  value: resd 1        ; Eingabewert
21
```

Abbildung 9

Die Section `__text` beginnt mit der Herstellung der Ausgangssituation. Um Verfälschungen am Ergebnis zu vermeiden, werden alle Register – sowohl die Register EAX, EBX, ECX, EDX und ESI, als auch die FPU-Register – auf null gesetzt. Gemäß den Calling Conventions wird der durch das C-Rahmenprogramm eingelesene Eingabewert auf dem Stack abgelegt. Um ihn effektiv weiterverarbeiten zu können, wird dieser Eingabewert nun an die Speicherstelle „value“ gelegt. (Abb. 10)

```
23  section __TEXT,__text
24
25  ;-----
26  ;-----setup-----
27  ;-----
28
29  finit
30  mov eax, 0
31  mov ebx, 0
32  mov ecx, 0
33  mov edx, 0
34  mov esi, 0
35
36  push ebp
37  mov ebp, esp
38  fld dword [ebp+8]
39  fstp dword [value]
40  pop ebp                ; Value x an Speicherstelle 0
41
```

Abbildung 10

Aufgrund der angesprochenen periodischen Eigenschaften der Sinusfunktion wird der Eingabewert nun auf das Intervall $[0; 2\pi]$ abgebildet. Hierzu wird zunächst der Eingabewert von der Speicherstelle „value“ in das FPU-Register geladen. An dieser Stelle ist zu erwähnen, dass die FPU-Register, im Gegensatz zu den üblichen Registern, nicht direkt beladen werden, sondern wie ein Stack funktionieren. Als nächstes wird also π in das FPU-Register geladen und befindet sich damit an oberster Stelle. Nun wird π verdoppelt, indem das oberste Register mit dem Wert an Speicherstelle „two“ multipliziert wird. (Zur Erinnerung, an diese Stelle wurde zu Beginn der Wert 2 gespeichert.) Anschließend beginnt der spannende Teil der Modulo-Rechnung. Um die Verständlichkeit zu wahren, soll im Folgenden nur auf das Prinzip der Rechnung eingegangen werden, statt jede Zeile einzeln zu erklären. In *Abbildung 11* findet sich zusätzlich der zugehörige Codeabschnitt inklusive ausführlicher Kommentare.

Zunächst wird also der Eingabewert durch 2π geteilt. Das Ergebnis wird abgerundet, indem der Gleitkommawert als Ganzzahl in den Speicher geschrieben und anschließend wieder in das FPU-Register geladen wird. Dieser Abgerundete Wert wird nun mit 2π multipliziert, womit man den Wert erhält, der ganzzahlig durch 2π teilbar ist. Dieses Ergebnis wird vom Ursprünglichen Eingabewert abgezogen, um den gewünschten Modulowert zu bestimmen.

Dieser Modulwert wird abschließend an die Speicherstelle „newValue“ geschrieben und im weiteren Verlauf des Programms für die Berechnung des Sinuswerts verwendet. Im Folgenden wird der Wert an Speicherstelle „newValue“ vereinfachend Eingabewert genannt.

```

41
42 fld dword [value]      ; Value x in FPU-Register pushen
43 fldpi                 ; Pi in FPU-Register pushen
44 fmul dword [two]       ; Pi*2 definiert den betrachteten Abschnitt
45
46 ;-----setup end-----
47
48
49 ;-----
50 ;-----modulo-----
51 ;-----
52
53 modulo:
54 fdiv st1, st0          ; Value durch 2pi, Ergebnis in ST1
55 fld st1               ; Ergebnis der Rechnung ist Top-Of-Stack
56 fisttp dword [modResult]; Rundet Ergebnis ab
57 fld dword [modResult] ; Integer Ergebnis in FPU-Register pushen
58 fmul st1              ; Integer Ergebnis * 2pi
59 fld dword [value]     ; Ursprüngliche Value x in FPU-Register pushen
60 fsub st0, st1          ; Ergebnis von Value x subtrahieren
61 fst dword [newValue]  ; Neue Value in Hauptspeicher ablegen
62
63 ;-----modulo end-----
64

```

Abbildung 11

Um weiteren Speicherplatz einzusparen, kann der Eingabewert, wie oben angesprochen, weiter auf das Intervall $\left[0; \frac{\pi}{2}\right]$ abgebildet werden. Hierzu wird zunächst das FPU-Register geleert, um Verfälschungen der Rechnungen zu vermeiden. Als nächstes werden π und der Eingabewert in das FPU-Register geladen. Nun wird überprüft, ob der Eingabewert im Intervall $[\pi; 2\pi]$ liegt. Ist dies der Fall, wird an Speicherstelle „sign“ zwischengespeichert, dass das Endergebnis später negiert werden muss. Außerdem wird in diesem Fall π von Eingabewert subtrahiert, um ihn ins Intervall $[0; \pi]$ abzubilden. Liegt der Eingabewert bereits im Intervall $[0; \pi]$ können diese beiden Schritte trivialerweise übersprungen werden. (Siehe Abbildung 12.)

```

66 ;-----
67 ;-----Abbildung auf Pi/2-----
68 ;-----
69
70 fninit                ; FPU-Register leeren
71 fldpi                 ; Pi in FPU-Register pushen
72 fld dword [newValue]  ; newValue in FPU-Register pushen
73 fcomi                 ; newValue und Pi vergleichen
74 jbe lowerEqual        ; Wenn <= dann zu lowerEqual
75
76 ; if newValue > pi then
77 mov word [sign], -1    ; Endergebnis später negieren
78 fsub st0, st1          ; newValue - Pi
79
80 lowerEqual:

```

Abbildung 12

Als nächstes wird π im FPU-Register durch zwei geteilt. Anschließend wird überprüft, ob der Eingabewert im Intervall $\left[0; \frac{\pi}{2}\right]$ liegt. Falls nicht, wird er in dieses Intervall abgebildet, indem der Eingabewert von π subtrahiert wird. Damit ist die Abbildung des Eingabewerts auf das Intervall $\left[0; \frac{\pi}{2}\right]$ abgeschlossen, und der aktualisierte Eingabewert kann an die Speicherstelle „newValue“ geschrieben werden. (Siehe Abbildung 13.)

```

80 lowerEqual:
81 ; else if newValue <= pi then
82 fld dword [two]        ; 2 in FPU-Register pushen
83 fdiv st2, st0          ; Pi / 2, Ergebnis in ST2
84 fstp dword [two]       ; 2 aus FPU-Register poppen
85 fcomi                 ; newValue und Pi/2 vergleichen
86
87 jbe updateValue        ; Falls newValue schon <= pi/2, dann nichts zu tun
88 ; if newValue > pi/2 then
89 fldpi                 ; Pi in FPU-Register pushen
90 fsub st0, st1          ; Pi/2 - newValue | newValue auf Bereich 0 bis Pi/2 abgebildet
91 updateValue:
92 fst dword [newValue]   ; Neue Value in Hauptspeicher ablegen
93
94 ;-----Abbildung auf Pi/2 end-----

```

Abbildung 13

Nun da sichergestellt ist, dass der Eingabewert im Bereich der in der Look-Up-Table gespeicherten Werte liegt, kann mit der Schrankensuche begonnen werden. Um die Effizienz zu maximieren, wurde hierfür eine binäre Suche implementiert. Als untere Schranke wird zunächst die Startadresse der Look-Up-Table im Register EAX gespeichert, als obere Schranke die Endadresse der eben genannten im Register EBX. Nun beginnt die binäre Suchschleife. Die mittlere Speicherzelle zwischen oberer und unterer Schranke wird als Pivotelement verwendet. Da in der Look-Up-Table 32 Bit Werte gespeichert sind, muss sichergestellt werden, dass die Pivotadresse am 8 Byte Raster ausgerichtet ist. Mit anderen Worten, der Offset zur Startadresse muss durch 4 teilbar sein.

Um dies zu garantieren, wird dieser Offset durch 8 geteilt und das ganzzahlige Ergebnis wieder mit 4 multipliziert. Diese Rechenoperationen werden effizient durch zweimaliges Rechtsshiften mit anschließendem zweimaligem Linksshiften realisiert. Durch anschließendes Addieren des Offsets zur Startadresse erhält man die ausgerichtete Pivotadresse.

```

97 ;-----
98 ;-----Binäre Intervallsuche -----
99 ;-----
100
101 find_interval:
102 mov eax, lutX          ; Startadresse in eax
103 mov ebx, lutX          ; Startadresse in ebx
104 add ebx, 1600          ; Endadresse in ebx
105
106 start:
107 ; Pivotadresse = (Obere Schranke - Untere Schranke)/2 + Untere Schranke
108 mov esi, ebx           ; Obere Schranke in esi
109 sub esi, eax           ; Obere Schranke - Untere Schranke, Ergebnis in esi
110 shr esi, 1             ; / 2
111 add esi, eax           ; + Startadresse, Ergebnis = Pivotadresse in esi
112
113 ; Pivotadresse an 4-Byte (32 Bit) Raster ausrichten
114 sub esi, lutX          ; Abstand von aktueller Adresse zur Startadresse
115 shr esi, 2             ; Pivot durch 4 teilen
116 shl esi, 2             ; Pivot mit 4 multiplizieren
117 add esi, lutX          ; Abstand + Startadresse
118

```

Abbildung 14

Jetzt wird der Wert an der Pivotadresse, sowie der Eingabewert ins FPU-Register geladen. Hier können die beiden Gleitkommazahlen verglichen werden. Falls der Eingabewert kleiner als der Pivotwert ist, wird die obere Schranke auf die Pivotadresse gesetzt. Falls er größer ist, wird die untere Schranke auf die Pivotadresse gesetzt. Nun wird überprüft, ob obere und untere Schranke noch mehr als eine Stelle auseinanderliegen. Ist dies der Fall, wird die Schleife erneut angesprungen. Ansonsten ist die Schrankensuche beendet. (Siehe Abbildung 15.)

```

118
119 fninit          ; FPU-Register leeren
120 fld dword [newValue] ; newValue in FPU-Register pushen
121 fld dword [esi]      ; Pivotwert in FPU-Register pushen
122
123 fcomi           ; newValue und Pivotwert vergleichen
124
125 jnb setLowerBound ; Falls newValue > Pivotwert, setze Untere Schranke auf Pivotadresse
126
127 ; else if newValue < Pivotwert then
128 mov ebx, esi      ; Obere Schranke auf Pivotadresse
129 jmp checkend      ; setLowerBound überspringen
130
131 setLowerBound:
132 ; else if newValue > Pivotwert then
133 mov eax, esi      ; Untere Schranke auf Pivotadresse
134
135 ; Abbruchbedingung prüfen
136 checkend:
137 mov edx, ebx      ; Obere Schranke in edx speichern
138 sub edx, eax      ; Obere Schranke - Untere Schranke, Ergebnis in edx
139 cmp edx, 4        ; Mit 4 vergleichen
140 jg start          ; Falls Abstand größer als 4 nochmal durchlaufen
141
142 endIntervalSearch:
143
144 ;-----Binäre Intervallsuche end-----
145

```

Abbildung 15

Im letzten Teilabschnitt folgt nun die Interpolation. Zunächst wird die prozentuale Nähe des Eingabewerts zur unteren Schranke bestimmt (Abb. 16). Hierzu werden der Wert der unteren Schranke, der Wert der oberen Schranke, sowie der Eingabewert in das FPU-Register geladen. Anschließend wird der Eingabewert von der unteren Schranke, und die untere Schranke von der oberen Schranke subtrahiert. Das Ergebnis der ersten Rechnung wird durch das Ergebnis der zweiten Rechnung dividiert und man erhält die prozentuale Nähe des Eingabewerts zur unteren Schranke. (Siehe Abbildung 17.)



Abbildung 16

$$\text{Prozentuale Nähe} = \frac{X - A}{B - A} = \frac{\text{Eingabewert} - \text{untere Schranke}}{\text{obere Schranke} - \text{untere Schranke}}$$

```
150
151 fninit                ; FPU-Register Leeren
152 fld dword [ebx-4]      ; x-Wert der unteren Schranke in FPU-Register pushen
153 fld dword [ebx]        ; x-Wert der oberen Schranke in FPU-Register pushen
154 fld dword [newValue]   ; newValue in FPU-Register pushen
155
156 ; Prozentuale Nähe ausrechnen
157 fsub st0, st2          ; newValue - Wert der unteren Schranke
158 fincstp               ; FPU-Stackpointer inkrementieren (st0 jetzt auf oberer Schranke)
159 fsub st0, st1          ; Obere Schranke - Untere Schranke
160 fdecstp               ; FPU-Stackpointer dekrementieren (st0 auf newValue - untere Schranke)
161 fdiv st0, st1          ; Prozentuale Nähe von newValue zu unterer Schranke jetzt in st0
162
```

Abbildung 17

Nachdem bisher nur mit den X-Werten hantiert wurde, wird nun erstmalig auf die Funktionswerte in der Look-Up-Table zugegriffen. Hierfür wird der Abstand der Adressen zur Startadresse der ersten Look-Up-Table berechnet und anschließend auf die Startadresse der zweiten Look-Up-Table addiert. Details zur Look-Up-Table folgen im nächsten Abschnitt.

Die Funktionswerte werden in das FPU-Register geladen. Nun wird der Funktionswert der unteren Schranke mit der prozentualen Nähe des Eingabewerts zur unteren Schranke multipliziert. Um die prozentuale Nähe des Eingabewerts zur oberen Schranke zu bestimmen, muss die Nähe zur unteren Schranke von 1 subtrahiert werden. Dann kann der Funktionswert der oberen Schranke mit der eben bestimmten prozentualen Nähe multipliziert werden. Die beiden Funktionswerte multipliziert mit ihren prozentualen Nähen ergeben addiert den gesuchten Funktionswert zum Eingabewert. Im letzten Rechenschritt wird das Ergebnis noch mit dem zuvor zwischengespeicherten Vorzeichen multipliziert und die Berechnung ist abgeschlossen. (Siehe Abbildung 18.)


```

164 ; Ergebnis berechnen
165 sub eax, lutX      ; Abstand von unterer Schranke zum Start der Lookup-Table X
166 add eax, lutY      ; Position in der Lookup-Table Y = Adresse von sin(untere Schranke)
167
168 sub ebx, lutX      ; Abstand von oberer Schranke zum Start der Lookup-Table Y
169 add ebx, lutY      ; Position in der Lookup-Table Y = Adresse von sin(obere Schranke)
170
171 fld dword [ebx]    ; Y-Wert der oberen Schranke in FPU-Register pushen
172 fld dword [eax]    ; Y-Wert der unteren Schranke in FPU-Register pushen
173
174 fmul st0, st2      ; Y-Wert der unteren Schranke * prozentuale Nähe von
175                    ; newValue zu Y-Wert der unteren Schranke
176
177 fld1               ; 1 in FPU-Register pushen
178 fsub st0, st3      ; 1 - prozentuale Nähe = Prozentuale Nähe zu oberer Schranke
179 fmul st0, st2      ; Y-Wert der oberen Schranke * prozentuale Nähe von newValue
180                    ; zu Y-Wert der oberen Schranke
181
182 fadd st0, st1      ; Prozentuale Werte der beiden Schranken ergeben
183                    ; addiert den interpolierten Wert
184
185 fild word [sign]   ; Vorzeichen in FPU-Register pushen
186 fmul st0, st1      ; Falls Value x im Bereich Pi bis 2Pi lag, wird das Ergebnis negiert
187
188 ;-----Interpolation end-----
189
190 ret

```

Abbildung 18

2.3.2 lut.asm

Als nächstes soll näher auf die Look-Up-Table eingegangen werden. Hier werden zu 400 Werten im Intervall $\left[0; \frac{\pi}{2}\right]$ die zugehörigen vorberechneten Funktionswerte definiert. Der Aufbau gliedert sich in zwei Teilbereiche. Zunächst werden am Label „lutX“ die 400 X-Werte definiert. Danach folgen am Label „lutY“ die 400 zugehörigen Funktionswerte.

```

2  lutX:
3  dd 0.002500002604173991
4  dd 0.005000020833567712
5  dd 0.0075000703142798445
6  dd 0.010000166674167114
7  dd 0.012500325543723647
8  .   ; Zur Übersichtlichkeit abgekürzt,
9  .   ; hier würde die Look-Up-Table weitergehen
10 .
11
12
13 lutY:
14 dd 0.0
15 dd 0.0025
16 dd 0.005
17 dd 0.0075
18 dd 0.01
19 .   ; Zur Übersichtlichkeit abgekürzt,
20 .   ; hier würde die Look-Up-Table weitergehen
21 .

```

Abbildung 19: Look-Up-Table

Da die Sinusfunktion in verschiedenen Bereichen verschiedene Steigungen hat, muss der Abstand der Stützstellen in manchen Bereichen enger sein als in anderen. Um dieses Problem zu lösen und einen optimalen Abstand der Stützstellen zu erreichen, wurde die Berechnung mit der Umkehrfunktion $\sin^{-1}(x)$ statt dem naheliegenden $\sin(x)$ angegangen. Wie in *Abbildung 19* zu sehen ist, haben die Funktionswerte in „lutY“ einen konstanten Abstand von 0,0025 zueinander. Zu jedem dieser Funktionswerte wurde mit $\sin^{-1}(x)$ der entsprechende X-Wert berechnet. So ist sichergestellt, dass die X-Werte in Bereichen in denen eine starke Steigung herrscht, einen kleinen Abstand zueinander haben, um Interpolationsfehler zu minimieren.

2.3.3 rahmenProgramm.c

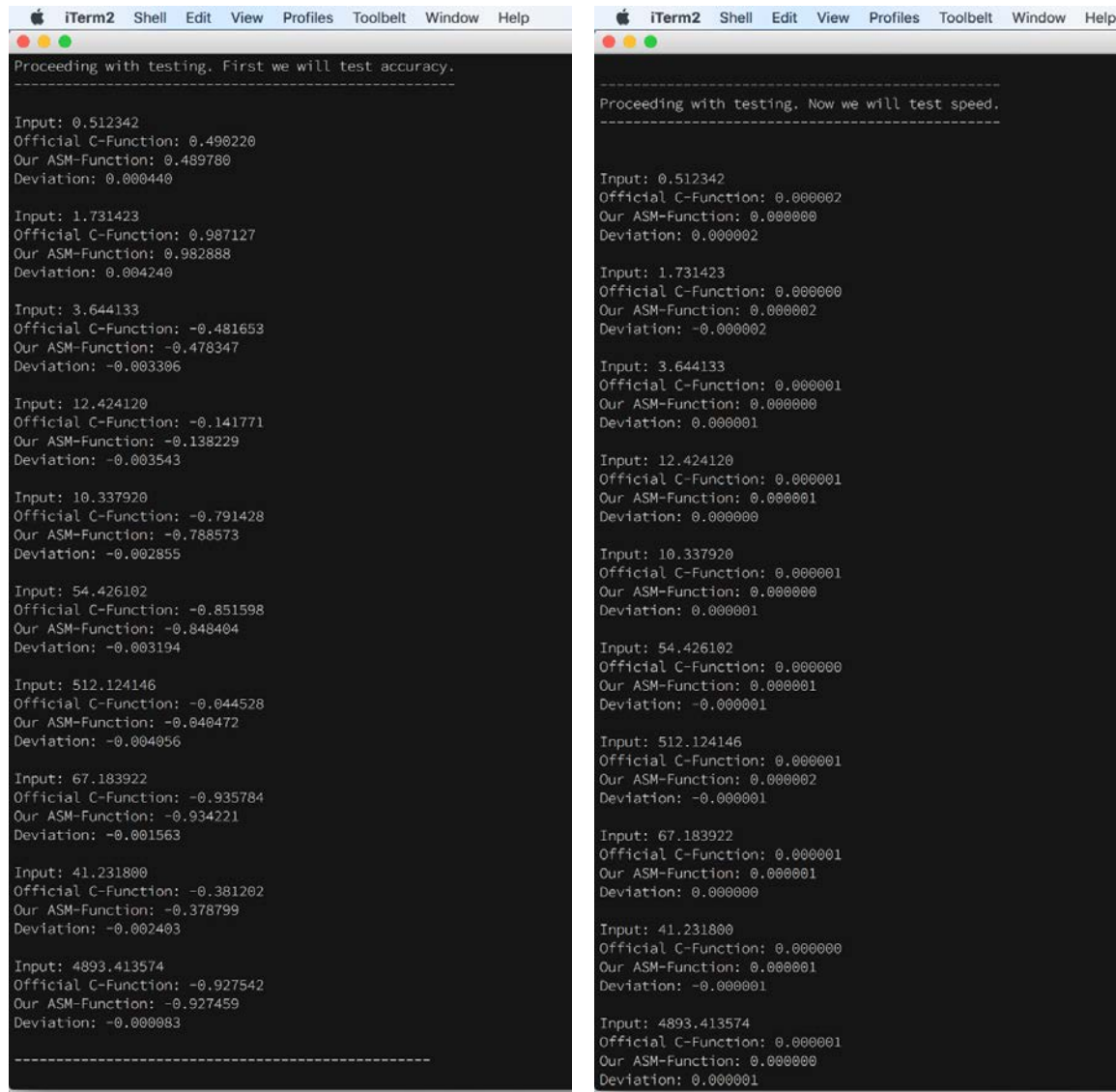
Das C-Rahmenprogramm bildet die Schnittstelle zwischen Benutzer und Assemblerprogramm. Nachdem der Benutzer den zu berechnenden Wert ins Terminal eingegeben hat, wird dieser Wert vom C-Programm beim Aufruf des Assemblerprogramms auf den Stack gelegt. Von dort aus kann das Assemblerprogramm darauf zugreifen. Nach Beendigung des Assemblerprogramms wird vom C-Programm der oberste Eintrag im FPU-Register ausgegeben, welcher das Ergebnis enthält.

2.3.4 testProgramm.c

Da sich der nächste Abschnitt allein Tests widmet, soll an dieser Stelle darauf verzichtet werden näher auf die Funktionsweise des Testprogramms einzugehen.

2.4 Test

Um Geschwindigkeit und Genauigkeit der Berechnung der Sinusfunktion zu überprüfen, wurde ein Testprogramm geschrieben. Dieses berechnet zu zufällig ausgewählten Testwerten den Sinus, einmal mit unserer Assemblerimplementierung und einmal mit der Sinusfunktion der C-Bibliothek. Die Geschwindigkeit wird getestet, indem die aktuelle Zeit vor und nach Aufruf der jeweiligen Sinus-Routine zwischengespeichert wird und abschließend voneinander Subtrahiert wird. **Abbildung 14** zeigt einen Durchlauf des Testprogramms.



```
iTerm2 Shell Edit View Profiles Toolbelt Window Help
Proceeding with testing. First we will test accuracy.
-----
Input: 0.512342
Official C-Function: 0.490220
Our ASM-Function: 0.489780
Deviation: 0.000440

Input: 1.731423
Official C-Function: 0.987127
Our ASM-Function: 0.982888
Deviation: 0.004240

Input: 3.644133
Official C-Function: -0.481653
Our ASM-Function: -0.478347
Deviation: -0.003306

Input: 12.424120
Official C-Function: -0.141771
Our ASM-Function: -0.138229
Deviation: -0.003543

Input: 10.337920
Official C-Function: -0.791428
Our ASM-Function: -0.788573
Deviation: -0.002855

Input: 54.426102
Official C-Function: -0.851598
Our ASM-Function: -0.848404
Deviation: -0.003194

Input: 512.124146
Official C-Function: -0.044528
Our ASM-Function: -0.040472
Deviation: -0.004056

Input: 67.183922
Official C-Function: -0.935784
Our ASM-Function: -0.934221
Deviation: -0.001563

Input: 41.231800
Official C-Function: -0.381202
Our ASM-Function: -0.378799
Deviation: -0.002403

Input: 4893.413574
Official C-Function: -0.927542
Our ASM-Function: -0.927459
Deviation: -0.000083
-----

iTerm2 Shell Edit View Profiles Toolbelt Window Help
-----
Proceeding with testing. Now we will test speed.
-----

Input: 0.512342
Official C-Function: 0.000002
Our ASM-Function: 0.000000
Deviation: 0.000002

Input: 1.731423
Official C-Function: 0.000000
Our ASM-Function: 0.000002
Deviation: -0.000002

Input: 3.644133
Official C-Function: 0.000001
Our ASM-Function: 0.000000
Deviation: 0.000001

Input: 12.424120
Official C-Function: 0.000001
Our ASM-Function: 0.000000
Deviation: 0.000001

Input: 10.337920
Official C-Function: 0.000001
Our ASM-Function: 0.000001
Deviation: 0.000000

Input: 54.426102
Official C-Function: 0.000000
Our ASM-Function: 0.000000
Deviation: 0.000001

Input: 512.124146
Official C-Function: 0.000001
Our ASM-Function: 0.000001
Deviation: -0.000001

Input: 67.183922
Official C-Function: 0.000001
Our ASM-Function: 0.000002
Deviation: -0.000001

Input: 41.231800
Official C-Function: 0.000000
Our ASM-Function: 0.000001
Deviation: 0.000000

Input: 4893.413574
Official C-Function: 0.000001
Our ASM-Function: 0.000000
Deviation: -0.000001
```

Abbildung 20: Testprogramm

Wie zu erkennen ist, liegt die Abweichung der Ergebnisse in den meisten Fällen unter 0,005. Oft sogar noch darunter. Bei der Geschwindigkeit lässt sich kein klarer Sieger feststellen, bei manchen aufrufen liegt die C-Routine leicht vorne und bei manchen unsere Implementierung. Die Differenzen spielen sich hier jedoch im Nanosekundenbereich ab.

2.5 Erweiterungsoptionen

Eine naheliegende Erweiterung wäre die Cosinusfunktion. Da der Verlauf der Cosinusfunktion genau dem um $\frac{\pi}{2}$ verschobenen Verlauf der Sinusfunktion entspricht, wäre hierfür nicht einmal eine Anpassung der Look-Up-Table nötig. Man könnte Cosinus-Eingabewert einfach um $\frac{\pi}{2}$ verschieben und anschließend die Routine der Sinusfunktion verwenden.

Eine weitere Erweiterungsmöglichkeit liegt in der Ermöglichung der Berechnung von verschiedenen Streckungen und Stauchungen der Sinusfunktion. Wenn hierbei allerdings auf die vorhandene Look-Up-Table zurückgegriffen wird, könnten bei starken Streckungen Ungenauigkeiten auftreten.

Allgemein lässt sich das Grundgerüst der vorliegenden Implementierung ohne allzu großen Aufwand an jegliche andere periodische Funktion anpassen. Hierzu muss lediglich eine entsprechende Look-Up-Table eingebunden werden und spezifische Passagen des Codes, wie das Intervall in das der Eingabewert abgebildet werden soll, angepasst werden.

3. Verfasser

Diese Dokumentation wurde von Florian Müller verfasst und ist das Ergebnis einer dreimonatigen Projektarbeit mit Berzan Yildiz und Thea Kramer. Sie wurde am 23.07.2017 fertiggestellt und eingereicht.