LEHRSTUHL FÜR RECHNERTECHNIK UND RECHNERORGANISATION

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Projektaufgabe - A307: Numerische Quadratur

James Li
Alexander Epple
Mingyang Li

Inhaltsverzeichnis

1 Einleitung **2**

2 Problemstellung und Spezifikation **3**
2.1 Theoretischer Teil 3
2.2 Praktischer Teil 3

3 Lösungsfindung **4**
3.1 Theoretischer Teil 4
3.2 Praktischer Teil 5

4 Dokumentation der Implementierung **6**
4.1 Benutzerdokumentation 6
4.2 Entwicklerdokumentation 6

5 Ergebnisse **8**
5.1 Güte der Näherung 8
5.2 Analyse von perf record 9
5.3 Vergleich mit der disassemblierten C-Version 9
5.4 Die Berechnungsdauer im Vergleich 9

6 Zusammenfassung und Ausblick **10**

1 Einleitung

Die Informatik ist mit der Mathematik eng verbunden. Gute Algorithmen und mathematische Methoden helfen uns, konkrete Probleme zu lösen.

Eines dieser Probleme ist das Berechnen von Integralen, welches nur approximiert werden kann. Die Aufgabe bestand daher darin, die Gauss-Formel für $n = 2$ zu entwickeln und zu vereinfachen und damit die Quadratur eines übergebenen Polynoms mithilfe eines Programms (in C und ARM-Assembler) zu implementieren.

Diese Aufgabe fordert den vielschichtigen Einsatz von Fähigkeiten wie der mathematischen Analyse, Programmierkenntnisse in C und Assembler sowie Teamwork.

Diese Dokumentation beschreibt das Problem, wie es gelöst wurde und wie die fertige Implementation genutzt werden kann. Zuletzt werden die Ergebnisse, also die Güte der Näherung sowie die Qualität und Performance des Programms präsentiert und eingeschätzt.

2 Problemstellung und Spezifikation

Die Aufgabe gliedert sich in zwei Teile, einem theoretischen und praktischen Bereich. Beide Teile sollen nun hier spezifiziert werden.

2.1 Theoretischer Teil

Für die gegebene Gauss-Legendre Formeln: Das Integral $I_n[p]$ (1), die Nullstellen $L_{n+1}(x)$ und die daraus resultierenden Stützstellen (2) sowie die Gewichte a_{in} (3), sollen für $n = 2$ entwickeln und das Ergebnis vereinfacht werden, was eine Formel $I_2[p]$ liefert. Diese Formel darf nur die vier Grundrechenarten und Funktionswerte enthalten. Im nächsten Schritt wird das $n + 1$-ten Legendre-Polynom mit den Nullstellen und den Gewichten a_{in} berechnet. Am Ende werden die Gewichte a_{in} in die Formel (1) eingesetzt.

$$I_n\left[p\right] = \int_{-1}^{1} p\left(x\right) \mathrm{d}x = \sum_{i=0}^{n} \alpha_{in} \cdot p\left(x_i\right) \tag{1}$$

$$L_{n+1}\left(x\right) = \frac{1}{2^{n+1} \cdot (n+1)!} \cdot \frac{\mathrm{d}^{n+1}}{\mathrm{d}x^{n+1}} \left(x^2 - 1\right)^{n+1} \tag{2}$$

$$\alpha_{in} = \int_{-1}^{1} \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \mathrm{d}x \tag{3}$$

2.2 Praktischer Teil

Das Polynom welches integriert werden soll, wird mithilfe einer Funktion double f(double x) über eine dynamischen Bibliothek übergeben. Diese Funktion bestimmt den Funktionswert f an den Stützstellen x. Mit der oben beschriebenen Formel $I_2[p]$ wird dann die Funktion double gauss_legendre_2(double (*p)(double) in Assembler implementiert. Diese liefert dann das Ergebnis der numerischen Quadratur. Die Implementierung wird außerdem gegen eine native C-Version sowie gegen nicht-angenäherte Ergebnisse getestet.

3 Lösungsfindung

3.1 Theoretischer Teil

Nullstellen (Stützstellen) des $n+1$-ten Legendre Polynoms

$$L_3(x) = \frac{1}{8\cdot 6}\cdot\frac{d^3}{dx^3}\left(x^2-1\right)^3 = \frac{3}{48}\cdot\frac{d}{dx}\left(2\left(x^2-1\right)\cdot 2x\cdot 2x + 2\left(x^2-1\right)^2\right) = \frac{6}{48}\cdot x\left(20x^2-12\right)$$

$$\Rightarrow 20x^2-12=0 \quad \Rightarrow x^2 = \frac{6}{10} \qquad \Rightarrow x_{1,2} = \pm\sqrt{\frac{6}{10}},\ \ sowie\, x_3 = 0$$

Koeffizienten (Gewichte)

$$\alpha_{0\,2} = \int_{-1}^{1}\prod_{\substack{j=1\\ j\neq 0}}^{2}\frac{x-x_j}{x_i-x_j} = \int_{-1}^{1}\left(\frac{x-x_1}{x_0-x_1}\right)\cdot\left(\frac{x-x_2}{x_0-x_2}\right) = \int_{-1}^{1}\left(\frac{x+\sqrt{\frac{6}{10}}}{\sqrt{\frac{6}{10}}}\right)\cdot\left(\frac{x-\sqrt{\frac{6}{10}}}{-\sqrt{\frac{6}{10}}}\right) =$$

$$= \int_{-1}^{1}\frac{x^2-\frac{6}{10}}{-\frac{6}{10}} = \int_{-1}^{1}-\frac{10}{6}x^2+1 = \left[-\frac{10}{18}x^3+x\right]_{-1}^{1} = \left(-\frac{10}{18}+1\right)-\left(\frac{10}{18}-1\right) = -\frac{8}{9}$$

$$\alpha_{1\,2} = \int_{-1}^{1}\prod_{\substack{j=0\\ j\neq 1}}^{2}\frac{x-x_j}{x_i-x_j} = \int_{-1}^{1}\left(\frac{x-x_0}{x_1-x_0}\right)\cdot\left(\frac{x-x_2}{x_1-x_2}\right) = \int_{-1}^{1}\frac{x}{\sqrt{\frac{6}{10}}}\cdot\left(\frac{x+\sqrt{\frac{6}{10}}}{2\sqrt{\frac{6}{10}}}\right) =$$

$$= \int_{-1}^{1}\sqrt{\frac{6}{10}}x\cdot\left(\frac{x}{2\sqrt{\frac{6}{10}}}+\frac{1}{2}\right) = \int_{-1}^{1}\frac{x^2}{\frac{12}{10}}+\frac{1}{2}\sqrt{\frac{6}{10}}x = \int_{-1}^{1}\frac{10}{12}x^2+\frac{1}{2}\sqrt{\frac{10}{6}}\cdot x =$$

$$= \left[\frac{10}{36}x^3+\frac{1}{4}\sqrt{\frac{10}{6}}\cdot x^2\right]_{-1}^{1} = \frac{10}{36}+\frac{1}{4}\sqrt{\frac{10}{6}}-\left(-\frac{10}{36}+\frac{1}{4}\sqrt{\frac{10}{6}}\right) = \frac{5}{9}$$

$$Aufgrund\, von\, Symmetrieverhalten: \alpha_{2\,2} = \frac{5}{9}$$

Zusammenfassende Formel

$$I_2[p] = \int_{-1}^{1}p(x)\,dx = \sum_{i=0}^{2}\alpha_{i\,2}\cdot p(x_i) =$$
$$= \alpha_{0\,2}\cdot p(x_0)+\alpha_{1\,2}\cdot p(x_1)+\alpha_{2\,2}\cdot p(x_2) =$$
$$= \frac{8}{9}\cdot p(0)+\frac{5}{9}\cdot p\left(\sqrt{\frac{6}{10}}\right)+\frac{5}{9}\cdot p\left(-\sqrt{\frac{6}{10}}\right)$$

Hierbei ist die Funktion $p(x)$ eine beliebige Polynomfunktion.
Diese Funktion für $n=2$ soll nun in Assembler portiert und implementiert werden.

3.2 Praktischer Teil

Bei der Ausarbeitung gab es einige Dinge zu beachten, ein wichtiger Punkt war wie mit der übergebenen Bibliothek umzugehen ist. Da es sich um dynamische Bibliotheken handelt, die zur Laufzeit ins Programm geladen werden sollen, werden die Funktionen von "dlfcn.h" [2] zum bewältigen dieser Aufgabe genutzt. Diese ermöglichen es Bibliotheken zu laden und Funktionen, Variablen etc. zu extrahieren, insofern das Symbol (der Name) dieser bekannt sind. Außerdem ermöglicht es dlerror(void) [2] eine genauere Ausgabe eventuell auftretender Fehler beim Öffnen und Nutzen der übergebenen Datei.

Eine weitere wichtige Überlegung bestand im Umgang mit den Funktionspointern, also dem Inhalt der übergebenen Bibliothek. Die Funktionen können nicht direkt aufgerufen, sondern müssen erst aus der Bibliothek geladen werden. Die dabei entstehenden Funktionszeiger ermöglichen es diese Funktionen dann zu nutzen [4].

Außerdem musste geklärt werden, wie sich die Funktionspointer in der Assembler Implementierung nutzen lassen. Laut Jürgen Wolf besitzen diese Zeiger „schließlich ebenfalls eine Anfangsadresse im Speicher" [4], was in Assembler also eine Adresse für einen Branch-Befehl darstellt. Um die geladene Funktion im Assemblercode aufzurufen reicht es mit einem BLX [5] Befehl zum Beginn der Funktion, die als Parameter übergeben wird und in r0 liegt, zu springen. Der Eingabeparameter der Funktion, ein Double Wert, wird laut AAPCS §6.1.2.3 in dem niedrigsten, ausreichend großen Register, also d0, erwartet [1].

Ein weiteres Problem bestand darin die Calling Conventions von Fließkommazahlen korrekt zu befolgen. Laut AAPCS §6.1.2 (entspricht §5.1.2.1) müssen nur die Register s16-s31 (d8-d15) erhalten werden [1], daher werden diese in dieser Implementierung nicht verwendet. Des weiteren ist der Rückgabewert der Implementierung laut AAPCS §6.1.2.2 in den niedrigsten, dem Fließkommazahl-Typ entsprechend ausreichenden großen Registern abzulegen [1]. Der Rückgabewert der Implementierung ist vom Typ Double, wird somit also von dem in C geschriebenen Teil des Programms in d0 erwartet.

Um das Integral berechnen zu können müssen mehrere Fließkommazahlen, die Stützstellen und Gewichte, in dem Programm verwendet werden. Der erste Ansatz war diese in der Assembler Funktion zur Laufzeit zu berechnen. Das würde aber zu hohen Performance-Einbußen führen (insbesondere bei der Berechnung der Wurzel), weshalb die endgültige Implementierung einen anderen Ansatz nutzt.

Im nächsten Abschnitt werden einige Punkt noch genauer erläutert und weitere, speziellere Details bezüglich der Implementierung beschrieben, außerdem wird erklärt wie das Programm genutzt werden kann.

4 Dokumentation der Implementierung

4.1 Benutzerdokumentation

Das Programm akzeptiert grundsätzlich zwei Eingabeparameter, einen Pfad zur Bibliothek mit dem Polynom und optional eine Anzahl der Durchläufe, um die Laufzeit besser testen zu können. Kann die Bibliothek geöffnet und die Funktion extrahiert werden und sind die Durchläufe angegeben, werden beide Implementierungen entsprechend oft ausgeführt und die jeweils dafür benötigte Zeit gemessen. Anschließend wird das Ergebnis der Berechnung und der Zeitmessung ausgegeben und das Programm terminiert. Falls die Durchläufe nicht spezifiziert sind, dann kann entweder die C oder die ARM Implementierung durch Eingabe der entsprechenden Nummer gewählt werden. Daraufhin wird das Integral mit der gewählten Methode berechnet und ausgegeben, dann terminiert das Programm.

4.2 Entwicklerdokumentation

Das Programm ist in zwei Teile aufgeteilt, der C-Code kümmert sich dabei um Nutzereingaben und das Laden der Bibliothek, der Assembler-Code um die eigentliche Berechnung (es existiert aber auch eine C-Implementierung der Berechnung). Nach dem Start des Programms wird zuallererst versucht die als String übergebene Bibliothek zu öffnen. Dies geschieht mithilfe von dlopen(const char *filename, int **flag**) [2], als Flag wird RTLD_LAZY genutzt (das Laden der Funktionen findet erst statt wenn notwendig). Anschließend wird die Funktion f mithilfe von dlsym(void *handle, const char *symbol) [2] geladen. Falls Fehler auftreten, wird der Fehler mittels dlerror(void) [2] abgerufen und ausgegeben (diese Funktion wird mehrfach aufgerufen um bestehende Fehlermeldungen aus dem Buffer zu löschen). Ist das Laden von Bibliothek und Funktion erfolgreich, dann gibt es zwei Möglichkeiten: Ist nur der Bibliothekspfad angegeben wird der Nutzer aufgefordert eine Implementierung zu wählen. Die Eingabe wird dann durch scanf(const char ***format**, ...) [3] mittels einer Maske, die ausschließlich einen Char und die Eingabetaste akzeptiert, ausgelesen. Anschließend wird der stdin Buffer geleert um überschüssige Zeichen zu entfernen. Die gewählte Implementierung wird ausgeführt und ausgegeben. Alternativ (wenn eine Durchlaufzahl übergeben wurde) werden beide Funktionen entsprechend oft ausgeführt und der Start- und Endzeitpunkt jeweils gespeichert und anschließend verrechnet. Schlussendlich wird jeweils das Ergebnis sowie die benötigte Zeit ausgegeben.

Auch wenn die Nutzung von dynamischen Bibliotheken insbesondere für diese Art von Anwendungen sehr nützlich ist, birgt es einige Risiken. Einerseits bieten die Funktionen von "dlfcn.h" [2] keine Möglichkeit zu prüfen, ob die geladen Funktion die korrekte Signatur besitzt. Das kann dann zu unerwartetem Verhalten und Programmabstürzen führen. Des weiteren ist der Programmcode der Funktionen nicht bekannt, sondern nur die Funktionsnamen. Damit wäre es bspw. möglich, beliebigen (schadhaften) Code einzuschleusen. Wenn die Signatur der vom Programm erwarteten Funktion und deren

Name bekannt sind, werden die Funktionen von *dlsym(void *handle, const char *symbol) [2] akzeptiert. Wird dann die kontaminierte Funktion im Programm aufgerufen, könnte die manipulierte Bibliothek großen Schaden anrichten. Es ist daher ratsam nur Bibliotheken aus vertrauenswürdigen Quellen und Funktionen mit bekannten Signaturen zu nutzen.

Der Assembler Teil besteht grundsätzlich aus dem dreimaligen Aufrufen der übergebenen Funktion. Hierfür wird der entsprechende Eingabeparameter (Stützstelle) zunächst in das Register d0 geladen. Da die Speicheradresse des benötigten Parameters erst zur Laufzeit bekannt ist, muss zunächst die Adresse des Parameter-Labels in das IP-Register geladen werden. Dann erst wird der tatsächliche Wert aus dem Speicher in das Register übertragen. Am Ende der Berechnung wird das Ergebnis mit einem Faktor (Gewicht) multipliziert und zwischengespeichert. Nachdem alle Ergebnisse bestimmt und aufaddiert sind, wird das Endergebnis in d0 abgelegt und die Funktion terminiert.

In der Assembler Funktion wurden außerdem einige Optimierungen angewendet die die Laufzeit positiv beeinflussen sollen.

Wie in der Lösungsfindung beschrieben wurde, ist das Berechnen der benutzten Fließkommazahlen aufwendig und ineffizient. Da sich die Gewichte bzw. Stützstellen aber nicht ändern, sind die vorberechneten Hexadezimal-Repräsentationen der entsprechenden Double Werte daher im Datensegment des Assembler-Codes abgespeichert. Die benötigten Werte werden dann zur Laufzeit wie oben beschrieben mithilfe der Labels in die entsprechenden SIMD-Register geladen.

Eine weitere Optimierung bestand darin das zweite Gewicht nur einmal zu laden und zweimal zu nutzen. Außerdem werden jeweils die Addition der einzelnen Teile und die Multiplikation mit deren Gewicht mithilfe des Befehls VMLA [6] zu einer Instruktion zusammengefasst. Eine VSUB d0, d0, d0 Instruktion ermöglicht es den Eingabeparameter der ersten Berechnung schneller in ein Register zu speichern als dies mit dem Laden eines Labels möglich ist (da der Eingabeparameter Null ist).

5 Ergebnisse

5.1 Güte der Näherung

Nachdem nun die Quadratur-Funktion implementiert wurde, sind einige Auffälligkeiten bei der Güte der Genauigkeit als auch der Laufzeit von C und ARM-Implementierung festzustellen. Im folgenden wird zunächst über die Genauigkeit der Gauss-Formel erörtert und anschließend die Laufzeiten von C und ARM analysiert.

Zunächst sollte erwähnt werden, dass in der obigen Ausführung die Gauss-Formel mit vorgegebener Entwicklung $n = 2$ aufgestellt und evaluiert wurde. Es muss beachtet werden, dass bei der Summe von Null aus aufgezählt wurde, weshalb es am Schluss drei Stützstellen gibt (Ferner soll $n = 3$ die Anzahl der Stützstellen sein). Eine Recherche über die Gauss Quadratur ergab, dass diese nur bei Polynomen vom Grad maximal $2 \cdot n - 1$ ein exaktes Ergebnis liefern kann, wobei n die Anzahl der Stützstellen bzw. Summeniteration angibt.

Der Grund dafür ist, dass es laut Gleichung (4) insgesamt in der rechten Summe n frei wählbare Koeffizienten a_i und n frei wählbare Stützstellen x_i gibt. Zusammen gibt es also $2 \cdot n$ freie Variablen. Wenn man ein Polynom vom Grad $2 \cdot n - 1$ betrachtet ist zu erkennen, dass dieses Polynom ebenfalls $2 \cdot n$ frei wählbare Variablen als Koeffizienten besitzt. Aufgrund dieser Freiheitsgrade kann sich die Gauss-Summe mit n Stützstellen einem Polynom Grads $2 \cdot n - 1$ oder kleiner exakt annähern. Auch ist es bereits bewiesen, dass es kein Quadratur-Verfahren gibt welches exakte Werte für Polynome mit höheren Grad als 2n-1 liefert [7]. In der oben beschriebenen Implementation hat man folglich eine exakte Annäherung für Polynome von maximalem Grad 5 $(2 \cdot 3 - 1)$.

Außerdem kann man festgestellt, dass die Implementation trotzdem mit einer geringen Abweichung Ergebnisse liefern kann sollte die Annäherung von einem Polynom höheren Grades als $2 \cdot n - 1$ gefordert sein. Die hierfür geltende Bedingung ist jedoch dass der Graph wenige Höhepunkte bzw. Tiefpunkte zwischen -1 und 1 besitzen muss. Sollte diese Voraussetzung verletzt und insbesondere mehr als $2 \cdot n - 1$ Extrema im Intervall $[-1 : 1]$ vorhanden sein, so ist eine ansatzweise Annäherung der Polynomfunktion bzw. ein exaktes Ergebnis unmöglich. Derartige Schwankung im Bereich der Näherung sind ebenfalls durch den Freiheitsgrad zu begründen mit der Idee, dass ein Polynom vom Grad $2 \cdot n - 1$ sich einem Polynom höheren Grads gut annähern kann, solange diese eine ähnliche Gestalt hat (Vergleiche hierfür bspw. x^3 und x^7).

Diese These stützend ist in Abbildung 1 graphisch die Güte der Näherung dargestellt. Die Werte der Säulen geben an wie groß der berechnete Wert vom tatsächlichen Wert entfernt ist. Diese sind jeweils aufsteigend in Abhängigkeit der Polynomgrade sortiert, die zugehörigen Terme sind in Abbildung 2 beschrieben. Wie man erkennen kann, gibt es große Abweichung für $n = 6$ aufgrund der Wahl des 6-ten Legendre Polynoms, welches viele Extrema im Bereich $[-1 : 1]$ besitzt (Siehe hierfür Abbildung 3). Außerdem

ist anzumerken, dass kein bzw. nur ein sehr kleiner Fehler für $n = 7$ und $n = 100$ trotz der hohen Grade existiert. Das liegt an den wenigen Extrema im Bereich $[-1 : 1]$ (siehe Abbildung 4). Daher das Fazit: Ist der $2n - 1$ Grad überschritten, hängt die Güte der Näherung nicht mehr vom Grad des Polynoms ab, sondern dessen Gestalt.

$$I^{(n)}(f) = \sum_{i=0}^{n} a_i \cdot f(x_i) \tag{4}$$

5.2 Analyse von perf record

Mithilfe von perf record wurde die Ausführung der Implementierung für 10^7 Durchläufe aufgezeichnet und anschließend mit perf report ausgewertet. Wie sich in Abbildung 6 erkennen lässt, stammt der Großteil des Overheads von der Bibliothek, die Ausführung der beiden Implementierungen machen zusammen nur knapp ein Viertel der Laufzeit aus. Ebenfalls ist anzumerken dass der Overhead beider Versionen nahezu gleich groß ist.

5.3 Vergleich mit der disassemblierten C-Version

Die in C implementierte Version von Gauss-Legendre wurde mittels objdump disassembliert und soll nun mit der ARM-Version verglichen werden. Die Abbildung 7 zeigt, dass sich die C-Version von der im Rahmen des Projekts erstellte ARM-Implementierung in einigen Details durchaus unterscheidet. Die Konstanten werden nicht in .data abgelegt, daher können sie direkt, statt über den Umweg durch Labels geladen werden. Die übergebene Funktion wird auf dem Stack gespeichert und bei jedem Aufruf neu geladen, außerdem werden Multiplikation und Addition der Polynome hier separat ausgeführt.

5.4 Die Berechnungsdauer im Vergleich

Abschließend und mit den Erkenntnissen der vorangegangenen Auswertungen werden nun die Zeitmessungen analysiert. Wie man Abbildung 5 entnehmen kann ist die Berechnungsdauer für beide Implementierungen beinahe identisch, unabhängig davon wie viele Durchläufe gewählt wurden. Der Vorteil der Assembler-Version gegenüber der C-Version zeigt sich erst bei vielen Durchläufen und ist praktisch vernachlässigbar. Wenn man bedenkt wie ähnlich sich C und ARM Implementierung sind und dass der Großteil der Zeit während der Berechnung des Funktionswerts in der Bibliothek verstreicht, ist das auch nicht verwunderlich. Die Berechnungen innerhalb der Implementierung können nicht sinnvoll durch SIMD-Instruktionen ersetzt werden, da sich die Berechnung des Funktionswerts aufgrund der Nutzung von Bibliotheken nicht parallelisieren lässt. Eine Implementierung in Assembler hat also keinen großen Vorteil gegenüber einer Implementierung in C.

6 Zusammenfassung und Ausblick

Zusammenfassend lässt sich sagen, dass auch wenn die Implementierung vernachlässigbar schneller arbeitet als die native C-Version, sie dennoch qualitativ hochwertiger ist. Der Vergleich zur disassemblierten C-Version zeigt, dass die gewählten Optimierungen sinnvoll und gut sind und das Programm stabil und sicher arbeitet. Eine grundsätzlich andere Implementierung ist nicht denkbar. Es wäre jedoch vielleicht möglich gewesen den Vorteil gegenüber der C-Version weiter auszubauen, wenn die Konstanten direkt geladen werden würde, anstelle den leichten Umweg über Adresslabels zu nehmen.

Literatur

[1] ARM Procedure Call Standard for the ARM Architecture,
zuletzt aufgerufen am 23.01.2018
`infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf`

[2] "dlfcn.h" Linux man-page,
zuletzt aufgerufen am 24.01.2018
`https://linux.die.net/man/3/dlopen`

[3] scanf Linux man-page,
zuletzt aufgerufen am 24.01.2018
`https://linux.die.net/man/3/scanf`

[4] C von A bis Z, Jürgen Wolf, Rheinwerk Computing, Kapitel 12.10,
zuletzt aufgerufen am 22.01.2018
`http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/012_c_zeiger_010.htm`

[5] ARM Compiler toolchain Assembler Reference,
zuletzt aufgerufen am 22.01.2018
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489e/Cihfddaf.html`

[6] ARM Compiler toolchain Assembler Reference,
zuletzt aufgerufen am 22.01.2018
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/CIHCDBIG.html`

[7] Quadraturformeln und ihre Konvergenz, Jan Assion, Satz 5.2
zuletzt aufgerufen am 24.01.2018
`https://www.math.uni-bielefeld.de/~emmrich/studenten/jan.pdf`

Abbildungsverzeichnis

1 Abweichung der berechneten Werte zum tatsächlichen Wert 11
2 Verwendete Polynome und Abweichung zum tatsächlichen Wert 11
3 Darstellung des Polynoms 6-ten Grades 12
4 Darstellung des Polynoms 100-ten Grades 12
5 Graph der Ausführzeiten 13
6 Mit `perf record` aufgezeichneter Overhead 13
7 Disassemblierte C-Implementierung 14

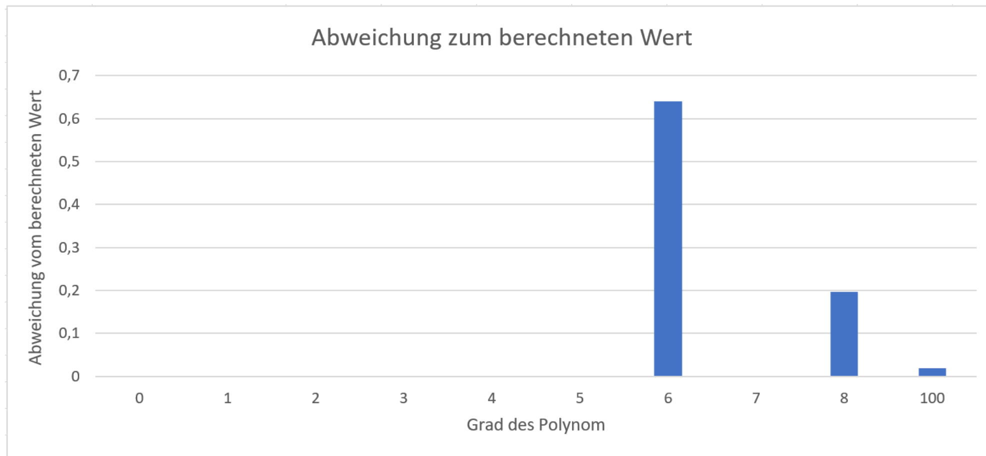


Abbildung 1: Abweichung der berechneten Werte zum tatsächlichen Wert

Polynomgrad	Verwendetes Polynom	Berechneter Wert	Tatsächlicher Wert
0	42	84	84
1	3x-1	-2	-2
2	x^2+1	2,666667	2,666667
3	x^3-25x	0	0
4	x^4-21x^3+52x^2+480x-512	-988,9333	-988,9333
5	x^5-21x^4+52x^3+480x^2-512x	311,6	311,6
6	231*16x^6-315*16x^4+105*16x^2-35*16x	-0,23999999	0,4
7	x^7+1	2	2
8	x^8-6x^6	-1,296	-1,492063492
100	x^100+1	2	2,0198

Abbildung 2: Verwendete Polynome und Abweichung zum tatsächlichen Wert

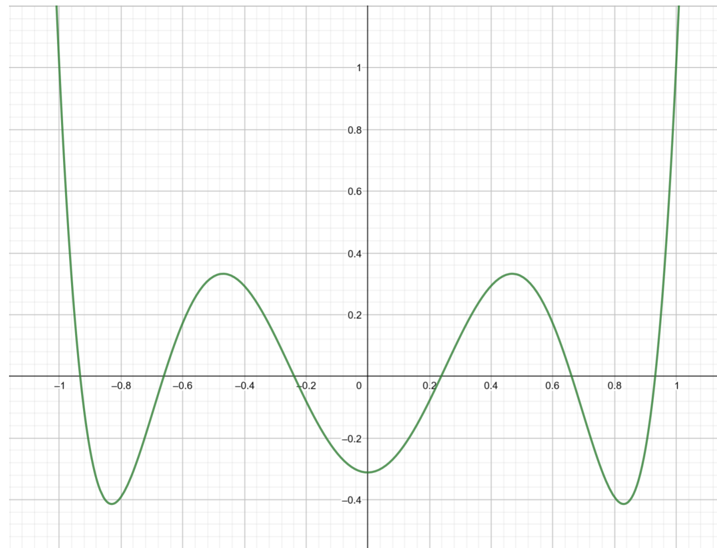


Abbildung 3: Darstellung des Polynoms 6-ten Grades

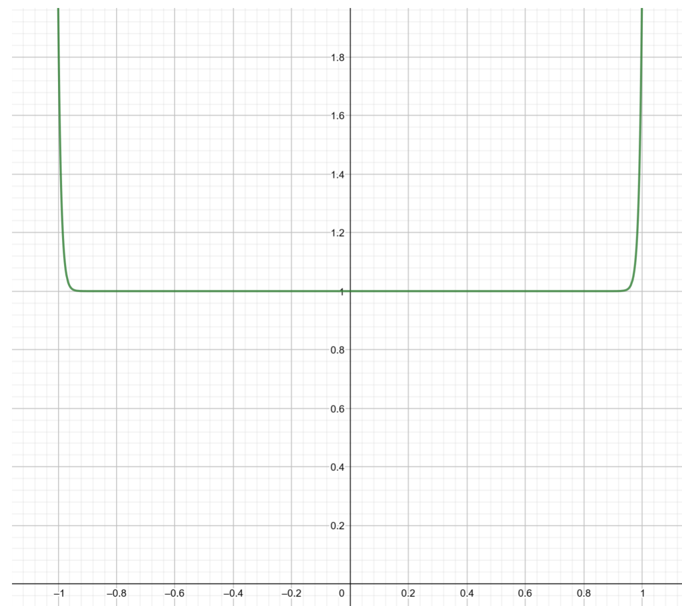
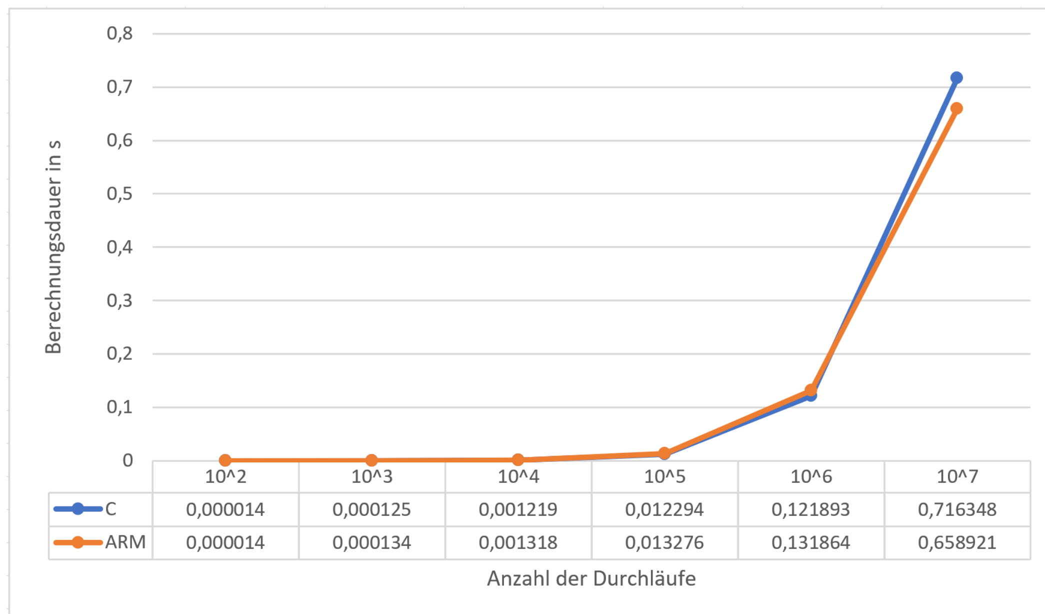


Abbildung 4: Darstellung des Polynoms 100-ten Grades



	10^2	10^3	10^4	10^5	10^6	10^7
C	0,000014	0,000125	0,001219	0,012294	0,121893	0,716348
ARM	0,000014	0,000134	0,001318	0,013276	0,131864	0,658921

Abbildung 5: Graph der Ausführzeiten

```
Samples: 1K of event 'cycles:u', Event count (approx.): 245304301
Overhead  Command  Shared Object                 Symbol
  69.19%  Gauss    x^4-21x^3+52x^2+480x-512.so   [.] f
  13.53%  Gauss    Gauss                         [.] gauss_legendre
  13.48%  Gauss    Gauss                         [.] gauss_legendre_2
   3.65%  Gauss    Gauss                         [.] main
```

Abbildung 6: Mit perf record aufgezeichneter Overhead

```
000105d8 <gauss_legendre>:
   105d8:       e92d4800        push    {fp, lr}
   105dc:       ed2d8b02        vpush   {d8}
   105e0:       e28db00c        add     fp, sp, #12
   105e4:       e24dd008        sub     sp, sp, #8
   105e8:       e50b0010        str     r0, [fp, #-16]
   105ec:       e51b3010        ldr     r3, [fp, #-16]
   105f0:       ed9f0b16        vldr    d0, [pc, #88]   ; 10650 <gauss_legendre+0x78>
   105f4:       e12fff33        blx     r3
   105f8:       eeb06b40        vmov.f64        d6, d0
   105fc:       ed9f7b15        vldr    d7, [pc, #84]   ; 10658 <gauss_legendre+0x80>
   10600:       ee268b07        vmul.f64        d8, d6, d7
   10604:       e51b3010        ldr     r3, [fp, #-16]
   10608:       ed9f0b14        vldr    d0, [pc, #80]   ; 10660 <gauss_legendre+0x88>
   1060c:       e12fff33        blx     r3
   10610:       eeb06b40        vmov.f64        d6, d0
   10614:       ed9f7b13        vldr    d7, [pc, #76]   ; 10668 <gauss_legendre+0x90>
   10618:       ee267b07        vmul.f64        d7, d6, d7
   1061c:       ee388b07        vadd.f64        d8, d8, d7
   10620:       e51b3010        ldr     r3, [fp, #-16]
   10624:       ed9f0b11        vldr    d0, [pc, #68]   ; 10670 <gauss_legendre+0x98>
   10628:       e12fff33        blx     r3
   1062c:       eeb06b40        vmov.f64        d6, d0
   10630:       ed9f7b0c        vldr    d7, [pc, #48]   ; 10668 <gauss_legendre+0x90>
   10634:       ee267b07        vmul.f64        d7, d6, d7
   10638:       ee387b07        vadd.f64        d7, d8, d7
   1063c:       eeb00b47        vmov.f64        d0, d7
   10640:       e24bd00c        sub     sp, fp, #12
   10644:       ecbd8b02        vpop    {d8}
   10648:       e8bd8800        pop     {fp, pc}
   1064c:       e320f000        nop     {0}
        ...
   10658:       1c71c71c        .word   0x1c71c71c
   1065c:       3fec71c7        .word   0x3fec71c7
   10660:       f43f7248        .word   0xf43f7248
   10664:       3fe8c97e        .word   0x3fe8c97e
   10668:       71c71c72        .word   0x71c71c72
   1066c:       3fe1c71c        .word   0x3fe1c71c
   10670:       f43f7248        .word   0xf43f7248
   10674:       bfe8c97e        .word   0xbfe8c97e
```

Abbildung 7: Disassemblierte C-Implementierung