Technische Universität München
Institut für Informatik
Prof. Dr. Hans-Joachim Bungartz
Dipl.-Inf. Christoph Riesinger
Dipl.-Inf. Dipl.-Math. Jürgen Bräckle

WS 2012/2013

Numerisches Programmieren

1. Programmieraufgabe: Zahlendarstellung, Gleitpunktarithmetik, schnellere aber nicht konforme mathematische Operationen

Motivation

Normierung von Vektoren

In der Mathematik und in der Informatik spielt die Normierung von Vektoren eine wichtige Rolle. Normierung bedeutet hier, dass ein Vektor beliebiger Länge zu einem Vektor der Länge 1 umgerechnet wird, der jedoch noch immer in die selbe Richtung zeigt, wie der Ursprungsvektor.
Besonders in der Computergrafik spielen Normierungen eine zentrale Rolle. Um

$$\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

$$s = \frac{1}{||\vec{v}||} = \frac{1}{\sqrt{\sum_{i=1}^{n} v_i^2}}$$

$$\vec{v}_{norm} = s \cdot \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

Abbildung 1: Beispiel für die (euklidische) Normierung eines Vektors.

eine dreidimensionale Oberfläche realistisch beleuchten zu können, sind die Normalenvektoren zu den Oberflächen nötig.

Schnelle Berechnung der Normalenvektoren

Zwei wesentliche Probleme treten bei der Berechnung von Normalenvektoren in den 3D-Engines von Computerspielen auf:

i) Je komplexer und detaillierter die Szene dargestellt werden soll, desto mehr Normalenvektoren müssen berechnet werden was logischerweise zu einem erhöhten Rechenaufwand führt.

ii) Die Normierung der Vektoren ist ein Vorgang, der viele Taktzyklen benötigt. Wie auf der vorherigen Seite zu sehen, ist für die Normierung die Berechnung einer Wurzel nötig, was je nach verwendeter Hardwarearchitektur und Programmiersprache ein Vielfaches an Taktzyklen (grober Richtwert: Faktor 20) erfordert, die für andere Berechnungen wie Addition, Multiplikation oder boolesche Operationen benötigt werden.

Daher ist man besonders an einer schnellen Berechnung von $\frac{1}{\sqrt{x}}$ interessiert.
Dabei kommt einem im Bereich der 3D-Computergrafik der Umstand zu Hilfe, dass die Präzision des Ergebnises von zweitrangiger Bedeutung ist. Ist das Ergebnis der Berechnung nicht exakt und entspricht nur in etwa dem tatsächlichen Ergebnis, so äußert sich das in der Regel höchstens in der Falschdarstellung von Farben um ein oder zwei Farbwerte, was vom menschlichen Auge kaum bis überhaupt nicht wahrgenommen wird.

Inhalt der Programmieraufgabe

In folgenden Abschnitt wird ein Algorithmus vorgestellt, der sich eben diesen Umstand zu Nutze macht: Er berechnet eine inverse Wurzel $\frac{1}{\sqrt{x}}$, ohne auf die Wurzeloperation zurück zu greifen. Darüber hinaus verzichtet er auf die Division, deren Berechnung ebenfalls zahlreiche Taktschritte dauert. Stattdessen kommen lediglich "schnelle Operationen" wie Additionen und Shift-Operationen zum Einsatz. Dafür entspricht das Ergebnis nicht dem IEEE-Standard. Diesen Kompromiss ist man jedoch gerne bereit, einzugehen.
Inhalt dieser Programmieraufgabe ist die Implementierung des Algorithmus "fast inverse square root" und die Analyse des Fehlers, der im Vergleich zur "exakten" Berechnung der inversen Wurzel gemacht wird. Der Algorithmus soll dabei nicht für eine Standard IEEE Gleitkommadarstellung (wie float oder double), sondern für die eigene Umsetzung einer Gleitpunktarithmetik entwickelt werden.

Der Algorithmus “schnelle inverse Wurzel”

Historisches

Die genaue Herkunft des hier vorgestellten Algorithmus mit dem Namen “fast inverse square root” ist nicht eindeutig. Erstmals publik wurde er durch den Quelltext des 3D-Multiplayershooters Quake 3 Arena, dessen 3D-Engine den Algorithmus verwendet. Spätestens durch die Veröffentlichung des Codes unter einer Open Source Lizenz war der Algorithmus für jedermann zugänglich. Es gilt jedoch als sicher, dass der Algorithmus nicht beim Spielehersteller id Software ersonnen wurde, sondern bereits deutlich älter ist und höchstwahrscheinlich bei SGI entwickelt wurde.

Der Algorithmus

Die genaue Herleitung des Algorithmus soll an dieser Stelle keine Rolle spielen. Auf den ersten Blick ist es verwunderlich, dass der Algorithmus das tut, was er soll. Auf den zweiten Blick jedoch ist dieser überraschend elegant. Stattdessen wird nun der Algorithmus in seiner ursprünglichen Version skizziert und später ein größerer Fokus auf Fehleruntersuchung und die ominöse “Magic Number” gelegt. Als Eingabe erhält der Algorithmus im Orginal eine IEEE 754 konforme 32-Bit Floating Point Zahl x. Von dieser berechnet er die inverse Wurzel $\frac{1}{\sqrt{x}}$.
Der Algorithmus “fast inverse square root” arbeitet wie folgt:

i) Die Bitfolge der `float`-Zahl wird als Bitfolge einer 32-Bit `int`-Zahl interpretiert. Auf die Darstellung von 32-Bit Floating Point Zahlen mit Vorzeichen, Exponent und Mantisse wurde bereits in der ersten Tutorübung intensiv eingegangen. **Achtung:** Es handelt sich hier **nicht** um einen klassischen Cast von `float` nach `int` sondern um eine explizite Neuinterpretation der Bitfolge.

ii) Diese eben erhaltene Integer-Zahl wird im Anschluss ganzzahlig durch 2 dividiert (abgerundet).

iii) Nun passiert der entscheidende Schritt: Das aktuelle Zwischenergebnis wird von einer bestimmten Konstante, einer sogenannten “Magic Number” abgezogen. Zu dieser Magic Number gibt es im Anschluss noch die ein oder andere Information.

iv) Zuletzt wird die nach obiger Subtraktion erhaltene Bitfolge, die in den Schritten 2 und 3 als `int`-Zahl interpretiert wurde, wieder als `float`-Zahl interpretiert und als Ergebnis zurück gegeben. Wieder handelt es sich **nicht** um einen Cast zwischen `int` und `float` sondern um eine Neuinterpretierung der 32 Bit.

Die “Magic Number”

Der Knackpunkt des Algorithmus liegt in der Magic Number. Der Begriff Magic Number wird in der Informatik immer dann verwendet, wenn eine scheinbar willkürlich gewählte Konstante zum Einsatz kommt, deren Herkunft unklar oder zumindest nicht auf Anhieb nachvollziehbar ist. Eine eben solche Magic Number ist essentieller Bestandteil des “fast inverse square root” Algorithmus.
Die Herleitung der Magic Number ist gut erforscht. Wir machen uns das Leben jedoch einfacher und werden die Magic Number “durch Ausprobieren” bestimmen.
Dabei ist zu beachten, dass es nicht “die” perfekte Magic Number gibt. Es gibt durchaus unterschiedliche Implementierungen, die unterschiedliche Magic Numbers verwenden, die je nach Wert von x unterschiedlich präzise Werte liefern.

Abbildung 2: Screenshot aus dem Spiel Team Arena, das die Quake 3 Engine verwendet. Durch die Veröffentlichung deren Quellcodes wurde der “fast inverse square root” Algorithmus der breiten Öffentlichkeit zugänglich.

1. Programmierteil: Gleitpunktarithmetik

Für den "fast inverse square root" Algorithmus brauchen wir zunächst eine geeignete Darstellung von Gleitpunktzahlen, die im folgenden beschrieben wird. Im ersten Teil dieser Programmieraufgabe soll diese Darstellung um benötigte Operationen erweitert werden.

Darstellung von Gleitkommazahlen

Reelle Zahlen werden in der Informatik im Allgemeinen als Gleitpunktzahlen repräsentiert. Dabei wird jede Zahl $r \in \mathbb{R}$ durch ein Vorzeichen v, eine sogenannte Mantisse (lateinisch: *Zugabe*) m, einen Exponenten e und eine Basis $b \in \mathbb{N} \setminus \{1\}$ dargestellt:

$$x = (-1)^v \cdot m \cdot b^e \tag{1}$$

Im Folgenden beschränken wir uns auf die Darstellung im Zweiersystem; setzen also $b = 2$. Man beachte, dass durch die Definition (1) Mantisse und Exponent nicht eindeutig festgelegt sind. Z.B. gilt:

$$\cdots = 0.25 \cdot 2^6 = 0.5 \cdot 2^5 = 1 \cdot 2^4 = \cdots$$

Um eine eindeutige Darstellung zu erhalten, legen wir eine Normierung fest: Der Exponent e soll stets so gewählt werden, dass die Mantisse m an ihrer höchstwertigsten Stelle eine Eins hat (siehe (2)). Ausgenommen ist die Null, da bei ihr überhaupt keine Stelle der Mantisse von Null verschieden ist. Dies ist nur eine Möglichkeit eine eindeutige Darstellung zu erhalten. Im IEEE-Standard 754 bzw. 854 wird eine andere Normierung verwendet
(vgl. `http://grouper.ieee.org/groups/754/reading.html`).

Eine reelle Zahl r im Binärsystem wird also dargestellt als

$$r = (-1)^v \cdot (r_{t-1}, r_{t-2}r_{t-3}\ldots r_1 r_0)_2 \cdot 2^{(e_{s-1}\ldots e_0)_2 - o}, \text{ mit } r_{t-1} \stackrel{!}{=} 1. \tag{2}$$

Hierbei bestimmt $v \in \{0, 1\}$ das Vorzeichen $(-1)^v$, die Zahlen $r_i \in \{0, 1\}, i = 0, \ldots, t-1$ den Wert der Mantisse m und $e_i \in \{0, 1\}, i = 0, \ldots, s-1$ mit dem Offset o den Wert des Exponenten e.

Beachten Sie, dass der Exponent analog zum IEEE-Standard mit einem Offset versehen ist, um auch negative Exponenten darstellen zu können. Bei einem Exponenten mit acht Bit ist der kleinste darstellbare Exponent Null (alle acht Bit 0) und der größte Exponent ist 255 (alle acht Bit 1). Um den eigentlichen Exponenten zu bekommen, wird in diesem Fall 127 abgezogen (allgemein $2^{i-1} - 1$). Für die Zahl $1 = (-1)^0 \cdot 1 \cdot 2^0$ wird also der Exponent $(01111111)_2$ abgespeichert.

Eine normalisierte Gleitpunktzahl hat eine feste Anzahl an Mantissenbits. Während der Berechnung von Operationen werden jedoch größere Mantissen

benötigt, um sicherzustellen, dass eine ideale Gleitpunktarithmetik implementiert wird. Der Begriff **ideale Gleitpunktarithmetik** bedeutet, dass das berechnete Ergebnis genau dem gerundeten exakten Ergebnis entspricht. Nach dem Berechnen muss das Ergebnis wieder normalisiert werden.

Rundung

Da nicht jede reelle Zahl als Maschinenzahl dargestellt werden kann, muss man sich Gedanken darüber machen, wie solche Zahlen dargestellt werden können. Wir benötigen also eine Vorschrift, nach der nicht darstellbare Zahlen auf darstellbare Zahlen abgebildet werden. Dies geschieht durch Runden der Mantisse. Dabei wird anhand der ersten abgeschnittenen Stelle entschieden, ob auf- oder abgerundet wird. Es gilt:

$$rd(r) = (-1)^v \cdot 2^e \cdot \begin{cases} r_{t-1}, r_{t-2} \dots r_1 r_0 & \text{für } r_{-1} = 0 \text{ (abrunden)} \\ r_{t-1}, r_{t-2} \dots r_1 r_0 + 2^{-t+1} & \text{für } r_{-1} = 1 \text{ (aufrunden)} \end{cases}$$

Um den relativen Abstand zwischen der Null und der kleinsten darstellbaren positiven Zahl 2^{-o} zu verringern, werden zusätzlich alle Zahlen aus dem Intervall

$$[0.5, 1) \cdot 2^{-o} = [1, 2) \cdot 2^{-o-1}$$

hin zu 2^{-o} aufgerundet.

Sonderfälle

Es gibt drei Sonderfälle, die bei Bearbeitung der Aufgabe beachtet werden müssen:

- Unendlich: Der Maximale Wert den das BitFeld für den Exponent annehmen kann ist bei Verwendung von bspw. 8 Bit 255. Der Wert des eigentlichen Exponenten ist somit $255 - \mathbf{o} = 255 - 127 = 128$. Alle größeren Exponenten können nicht mehr dargestellt werden. Da diese Zahlen dennoch mit ∞ repräsentiert werden sollten, verwenden wir den höchsten Exponenten für den speziellen Wert Unendlich. Die Mantisse wird komplett mit Nullen belegt. Das Vorzeichen legt fest, ob der Wert plus oder minus Unendlich ist.
- NaN: Dieser Wert entsteht z.B. beim Teilen von null durch null. Er erhält ebenfalls den höchsten Exponenten und eine beliebige Mantisse ungleich null.
- Null: Eine normalisierte Gleitpunktzahl kann nicht den Wert null annehmen, da das höchstwertigste Bit der Mantisse eine 1 ist. Zur Darstellung der Null werden Vorzeichen, Mantisse und Exponent auf null gesetzt.

Beschreibung des Programmgerüsts

Für den 1.Teil dieser Programmieraufgabe wird Ihnen ein Programmgerüst zur Darstellung von Gleitpunktzahlen zur Verfügung gestellt. Es besteht aus den beiden Klassen *Gleitpunktzahl* und *BitFeld*. Wie man in Gleichung (2) sieht, werden die Mantisse und der Exponent durch t bzw. s Bits dargestellt.

Die Klasse *BitFeld* dient zur Speicherung und Verarbeitung dieser Bitfelder. Es sind unter anderem bereits Methoden zum Schieben von Bitfeldern nach links bzw. rechts implementiert.

Die Klasse *Gleitpunktzahl* dient zur Speicherung und Verarbeitung von Gleitpunktzahlen. Sie enthält Variablen vom Typ *BitFeld* für Mantisse und Exponent und eine Variable vom Typ *boolean* für das Vorzeichen. Es sind unter anderem Methoden zum Normalisieren und Denormalisieren von Gleitpunktzahlen implementiert.

Vor dem Instanziieren des ersten Objekts der Klasse *Gleitpunktzahl* muss die Anzahl an Bits, die für die Speicherung von Mantisse und Exponent verwendet werden, festgelegt werden. Dazu werden die statischen Variablen *anzBitsMantisse* und *anzBitsExponent* mit den Methoden *setAnzBitsMantisse* und *setAnzBitsExponent* einmalig mit Werten belegt. Diese Werte dürfen nicht mehr geändert werden, damit sichergestellt ist, dass im Folgenden sämtliche Zahlen gleich aufgebaut sind.

Aufgabenstellung

Das gegebene Programmgerüst soll um die Addition und Subtraktion von Gleitpunktzahlen ergänzt werden.

Erläuterungen

Bei der Addition von Gleitpunktzahlen wird zunächst die betragsmäßig größere Zahl denormalisiert, damit beide Zahlen den gleichen Exponenten haben. Die Summe der beiden Gleitpunktzahlen $x = m_1 \cdot 2^{e_1}$ und $y = m_2 \cdot 2^{e_2}$ mit $x > y$ lautet:

$$\begin{aligned} x + y &= m_1 \cdot 2^{e_1} + m_2 \cdot 2^{e_2} = m_1 \cdot 2^{e_1 - e_2} \cdot 2^{e_2} + m_2 \cdot 2^{e_2} \\ &= (m_1 \cdot 2^{e_1 - e_2} + m_2) \cdot 2^{e_2} \end{aligned}$$

Analog behandelt man negative Vorzeichen bzw. die Subtraktion. Es muss darauf geachtet werden, dass das berechnete Ergebnis normalisiert wird. Das normalisierte Ergebnis muss genau dem gerundeten exakten Ergebnis entsprechen (optimale Gleitpunktarithmetik).

Die von Ihnen implementierten Methoden müssen dazu in der Lage sein, auf

beliebigen Gleitpunktzahlen inklusive der Null und Unendlich **(ohne den Sonderfall NaN) als Eingabe** zu arbeiten und ein korrektes Ergebnis aus der Menge der Gleitpunktzahlen **inklusive aller genannten Sonderfälle** zu berechnen.

Aufgaben

- Programmieren Sie in der Klasse *BitFeld* die gegebenen Methodenrümpfe *add* und *sub* entsprechend der Beschreibungen in den Kommentaren.
- Programmieren Sie in der Klasse *Gleitpunktzahl* die Methodenrümpfe *add* und *sub* entsprechend der Beschreibungen in den Kommentaren.
- Testen Sie Ihre Implementierung anhand selbstgewählter Beispiele. Als Anregung für eine Testumgebung können Sie das Beispielprogramm *Test_Gleitpunktzahl.java* verwenden. Hierin sind bereits einige Testfälle implementiert. Achten Sie beim Testen auch auf die Grenzen Ihrer Gleitpunktdarstellung. Im Falle von 2 Exponenten- und 4 Mantissenbits wäre 0.5 die betragsmäßig kleinste Zahl ungleich Null, 3.75 die betragsmäßig größte Zahl.

2.Programmierteil: Fast Inverse Square Root

Aufgabe

Ziel dieser Aufgabe ist es nicht, die Performanz der schnellen Wurzelberechnung zu untersuchen. Dafür sind die Uminterpretation und Operationen von Gleitpunktzahlen und BitFeldern in Java zu teuer, was das Benchmarkergebnis verfälscht. In anderen Programmiersprachen wie C/C++, die einen direkten Zugriff auf den Speicher ermöglichen, kann dies viel effizienter erfolgen. Vielmehr soll ein Gefühl für die Dimension des Fehlers vermittelt werden, den die schnelle Wurzelberechnung macht.
In dieser Programmieraufgabe sollen zwei Aufgaben bewältigt werden:

i) Zunächst soll der im zweiten Abschnitt skizzierte Algorithmus "fast inverse square root" in Java implementiert werden. Dazu stellen wir ein Programmgerüst zur Verfügung, das im nächsten Abschnitt näher erläutert wird. In diesem Schritt wird eine grobe Schätzung der Magic Number für eine Gleitpunktzahlarithmetik mit 3 Exponenten- und 9 Mantissenbits zur Verfügung gestellt, die jedoch **nicht** dem exakten/idealen Wert von ihr entspricht.

ii) Im Anschluss soll heuristisch ein besserer Wert der von uns vorgegebenen Magic Number bestimmt werden. Zu diesem Zweck soll der Fehler zwischen IEEE-konformer und durch "fast inverse square root" durchgeführten Berechnung betrachtet werden. Heuristisch bedeutet in diesem Fall "durch ausprobieren". Sie können also beispielsweise eine Reihe von denkbaren Magic Numbers durchlaufen und überprüfen, für welche Magic Numbers Ihr Programm minimale Fehler produziert. Bedenken Sie, dass unterschiedliche Fehlermetriken zur Bewertung herangezogen werden können. Sie müssen nicht angeben, wie Sie Ihre Magic Number gefunden haben.
Achten Sie dabei darauf, dass Sie sich bei der Abgabe auf eine konkrete Magic Number festlegen und diese **nicht** für jedes x oder beim Starten des Programms neu berechnen.

Erläuterung des Programmgerüsts

Das auf der Vorlesungsseite zur Verfügung gestellte Programmgerüst enthält die drei Klassen:

- `FastMath`:
  In dieser Klasse soll die Implementierung des Algorithmus zur schnellen Wurzelberechnung erfolgen. Dafür ist die Methode `invSqrt()` vorgesehen, deren Methodenrumpf zunächst leer ist. Darüber hinaus enthält die Klasse eine Konstante `MAGIC_NUMBER`, der anfänglich ein Testwert zugewiesen ist,

der jedoch im Laufe der Bearbeitung durch einen geeigneteren Wert für eine Gleitpunktzahlarithmetik mit 3 Exponenten- und 9 Mantissenbits ersetzt werden soll.

- `Plotter`:
  Diese Klasse implementiert einen einfach Plotter, der eine Menge von übergebenen zweidimensionalen Punkten in rot in ein logarithmisch skaliertes, kartesisches Koordinatensystem einträgt. Dafür müssen dem Konstruktor zwei gleich lange Arrays vom Typ `float` übergeben werden: `xData` und `yData`. Er wird beim aktuell zur Verfügung gestellten Programmgerüst automatisch gestartet. Darüber hinaus zeichnet der Plotter in grün automatisch die "exakte" Lösung `1.0 / Math.sqrt(x)` ein.
- `Test_FastInverse`:
  Diese Klasse enthält lediglich eine `main()`-Methode, die ein paar Testwerte berechnet und mit dem Plotter die absoluten Fehler zeichnet. Diese Anzeige basiert anfänglich auf einer 32bit Arithmetik mit der im Quake3-Code verwendeten Magic Number. Fühlen Sie sich frei, diese Klasse/Methode nach Ihren Wünschen zu erweitern und modifizieren um Ihre Implementierung des Algorithmus zu testen und eine möglichst optimale Magic Number zu bestimmen.

Tips zur Implementierung

- Aufgrund der Speicherung der führenden 1 funktioniert der Fast Inverse Square Root Algorithmus **nicht** bei direkter Uminterpretation der Gleitpunktzahl in ein BitFeld der Form $[Exponent|Mantisse]$. Für die Neuinterpretation in eine IEEE-ähnliche Bitfolge ohne führende 1 dienen die in der Klasse FastMath bereitgestellten Funktionen `gleitpunktzahlToIEEE` und `iEEEToGleitpunktzahl`. Für alle weiteren Schritte benötigen Sie nur schon vorhandene und von Ihnen implementierte Funktionalitäten der Klasse BitFeld.
- In der Literatur wird der Algorithmus zur schnellen Wurzelberechnung oft noch durch ein oder zwei Newtonschritte vervollständigt, die die Qualität der Berechnung erhöhen. Worum es sich bei diesen Newtonschritten genau handelt, wird im Laufe des Semesters noch genauer beleuchtet. Diese Newtonschritte müssen von Ihnen bei dieser Aufgabe weder beachtet noch implementiert werden.

Was wird von Seiten der Betreuer getestet/geprüft?

Ihr Programm durchläuft zur Bewertung eine Reihe von Testfällen, deren Ausgang die Grundlage der Bewertung der Programmieraufgabe liefert. Insgesamt werden bis zu 100 Punkte vergeben

Grundsätzlich gibt es drei Arten von Testfällen:

i) *Korrektheitstests*:
Bei diesen Tests wird überprüft, ob Ihre Implementierungen der Addition und Subtraktion korrekte Ergebnisse liefern und auf verschiedene Sonderfälle richtig reagieren.

ii) *Plausibilitätstests*:
Bei diesen Tests wird überprüft, wie Ihr Programm auf nicht zulässige Eingaben reagiert. Solche nicht zulässigen Eingaben wären für `invSqrt()` beispielsweise negative Zahlen, da für reelle negative Zahlen die Wurzelfunktion nicht definiert ist. In solchen Fällen soll ein spezieller Wert (`NaN`) zurückgegeben werden.

iii) *Qualitätstests*:
Diese Tests messen beispielsweise den größten und den kleinsten Fehler, den Ihre "fast inverse square root" Implementierung im Vergleich zur IEEE-konformen Implementierung macht und vergleicht diesen mit der Referenzimplementierung. Achten Sie dabei darauf, dass vor allem der Zahlenbereich getestet wird, der typischerweise bei der Normierung von Vektoren in der Computergrafik vorkommt.

Beachten Sie, dass nicht alle Testfälle in obiger Aufzählung erwähnt werden. Es existieren weitere Plausibilitäts- und Qualitätstests.

Selbstverständlich ist es nicht möglich eine Magic Number zu finden, die bezüglich sämtlicher in den Qualitätstests verwendeten Metriken optimal ist. Es ist daher durchaus möglich, die maximal erreichbare Punktzahl für diese Programmieraufgabe zu erhalten, auch wenn Ihre Lösung nicht in jedem Testfall ein optimales Ergebnis/das Ergebnis der Referenzimplementierung liefert.

Formalien und Hinweise

- Das Programmgerüst erhalten Sie auf den Webseiten zur Vorlesung.
- Ergänzen Sie das Programmgerüst **auf alle Fälle an den dafür vorgegebenen Stellen**! Darüber hinaus können Sie eigene Erweiterungen hinzufügen, falls Sie diese benötigen. **Wichtig** dabei ist, dass die Signaturen der Klassen `FastMath`, `BitFeld` und `Gleitpunktzahl` mit ihren Methoden

und Attributen erhalten bleiben, da sie den für die von uns durchgeführten Testfälle erforderlichen Code enthalten. Fügen Sie **keine** weiteren Dateien hinzu, da diese von unserem automatischen Korrektortool nicht verarbeitet werden.

- Beseitigen Sie vor Abgabe Ihres Programms alle Ausgaben an die Konsole, die Sie eventuell zu Debugging- oder Testzwecken eingefügt haben und reichen sie Ihre Lösung bis zum **19. November 2012, 12:00 Uhr** über Moodle ein.
- Reichen Sie nur die Dateien `BitFeld.java`, `Gleitpunktzahl.java` und `FastMath.java` mit den gleichnamigen Klassen ein.
- Bitte laden Sie Ihre java-Dateien als flaches tgz-Archiv hoch. Der Dateiname ist beliebig wählbar, bei der Erweiterung muss es sich jedoch um .tgz oder .tar.gz handeln.

  Ein solches Archiv können Sie beispielsweise mit dem Linux-Tool tar erstellen, indem Sie die laut Aufgabenstellung zu bearbeitenden java-Dateien in ein sonst leeres Verzeichnis legen und dort anschließend den Befehl

  ```
  > tar cvvzf numpro_aufg1.tgz *.java
  ```

  ausführen.
- Bei dieser Aufgabe kann es in Windows zu Problemen beim Testen von Rechenoperationen auf der Konsole kommen. Wir empfehlen daher, die Programme unter Linux (Rechnerhalle) zu testen.

Bitte beachten Sie: *Alle Abgaben, die nicht den formalen Kriterien genügen, werden grundsätzlich mit 0 Punkten bewertet!*