Kapitel 16
Numerische Anwendungen

16.1 Rundungsfehler

Mit der Darstellung von Zahlen unterschiedlicher Typen haben wir uns im letzten Kapitel ausführlich beschäftigt und auch bereits früher bei der Diskussion schwach besetzter Matrizen darauf hingewiesen, dass bei der Verwendung von Fließkommazahlen, die als Näherung für die in vielen mathematischen Gebieten verwendeten reellen Zahlen verwendet werden, eine exakte Darstellung von Werten nur in Ausnahmefällen möglich ist, selbst wenn gar keine reellen Zahlen, sondern nur rationale Zahlen in den Rechnungen eingesetzt werden. In diesem Kapitel wenden wir uns einer genaueren Betrachtung der Auswirkungen von Rundungen und der Kontrolle der Ergebnisse zu.

Die Ungenauigkeiten beginnen bereits bei der Eingabe von Zahlenwerten in eine Anwendung. Bei der dabei notwendigen Umwandlung nicht ganzzahliger Werte (*ganzzahlige Werte sind immer exakt wandelbar*) vom Dezimalformat in das dem Binärformat verwandte Oktalformat entstehen meist lange periodische Ziffernfolgen

$$0{,}345_{10} = 0{,}2\,\overline{605075341217270243 65}_8$$

Auch scheinbar „normale“ Zahlen können aufgrund der auftretenden Perioden in anderen Zahlensystemen nicht mehr exakt gespeichert werden, sondern müssen gerundet werden. Der gebräuchlichste Datentyp für Fließkommazahlen ist `double` mit 64 Bit Darstellungsbreite. Er erlaubt das Rechnen mit Zahlen in einem Zahlenraum von ca. $\pm 10^{\pm 300}$ mit einer relativen Genauigkeit von $\epsilon = 2{,}2 * 10^{-16}$. Das oben dargestellte Zahlenbeispiel ist also nur bis auf $\pm\epsilon/2$ exakt auf dem Rechner darstellbar.

Beginnen wir mit einer Spezifizierung der Fehlerbegriffe und einer Betrachtung der Konsequenzen der beschränkten Genauigkeit für Berechnungen. Ist x die im Rechner gespeicherte Zahl, X der wahre Wert, so hängen beide durch

$$x = X(1 + \delta)$$

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_16,

zusammen, wobei δ der relative Fehler ist.[1] Wir können nun die Fehlerfortpflanzung für die vier Grundrechenarten {+,–,*,/} prüfen, in dem wir in

$$z = x \circ y \; ; \quad \circ \in \{+, -, *, /\}$$

den Fehlerausdruck einsetzen, höhere Terme vernachlässigen und zum Ergebnis jeweils noch einen neuen elementaren Rundungsfehler hinzufügen, der durch das nochmalige Runden des Berechnungsergebnisses entsteht. Da normalerweise unbekannt ist, welches Vorzeichen die Abweichungen aufweisen, können wir als sinnvolles Ergebnis unserer Betrachtung nur eine Obergrenze für den Fehler angeben. Der echte Fehler wird in den meisten (*aber eben nicht allen*) Fällen kleiner sein. Zur Ermittlung der Grenzen arbeiten wir nicht mit den Fehlern selbst, sondern mit deren Beträgen. Der Fehler des Resultates kann dann nicht größer sein als die Summe der Beträge der Eingangsgrößen. Verifizieren Sie anhand dieser Ausführungen die folgenden Aussagen als Übung:

$$\begin{aligned}
&op = +: |\delta_z| \leq max\left(|\delta_x|, |\delta_y|\right) + \epsilon \\
&op \in \{*,/\}: |\delta_z| \leq |\delta_x| + |\delta_y| + \epsilon \\
&op = -: |\delta_z| \leq \left|\frac{X}{|X|-|Y|}\right| |\delta_x| + \left|\frac{Y}{|X|-|Y|}\right| |\delta_y| + \epsilon
\end{aligned}$$

Die ersten beiden Ausdrücke sind unkritisch, und auch im dritten wird in vielen Fällen wenig Ärger bereiten. Fatal ist die echte Subtraktion im letzten Ausdruck allerdings, sobald gleich große Zahlen im Spiel sind und deren Ergebnis im Verhältnis zu den Ausgangswerten nahe bei Null liegt. Da der kleinere Wert im Nenner der Bruches steht, ist der Faktor, mit dem die ursprünglichen Fehler multipliziert werden, größer als Eins. Der relative Fehler des Ergebnisses kann dann um Größenordnungen über den Fehlern der Eingangswerte liegen und das Ergebnis ist nicht mehr vertrauenswürdig. Wir demonstrieren dies an einem Beispiel, in dem wir nur die Reihenfolge der Operationen vertauschen:

```
r1=1.4142135623730951e+000
r2=1.4142135623730954e+000
r3=1.4142135623730953e-015
r1-r2-r3=1.6362581672981266e-015
r1-r3-r2=1.5543122344752192e-015

δ = 0.053  (das heißt 5.3% !)
```

Im ersten Algorithmus werden zunächst die gleich großen Zahlen voneinander abgezogen. Dabei bleibt nur eine Stelle übrig, die vom Rechenwerk wieder auf die volle Länge aufgefüllt wird (*mit Rundungsresten und Nullen, vergleiche die Zahlendarstellung im letzten Kapitel*). In der anschließenden Operation werden wiederum

[1] Die Differenz (x-X) ist der absolute Fehler, der Quotient (x-X)/X der relative Fehler bezogen auf X . Die kleinste aufgrund der Maschinengenauigkeit darstellbare Differenz zweier Größen bezeichnen wir als (*absoluten oder relativen*) Elementarfehler.

gleichgroße Zahlen miteinander verrechnet, so dass alle vorhandenen Ziffern berücksichtigt werden. Im zweiten Algorithmus wird zunächst eine sehr kleine von einer sehr großen Zahl abgezogen, wobei allenfalls die ersten Ziffern der kleineren Zahl überhaupt verwendet werden können. Im zweiten Schritt werden wieder gleich große Zahlen voneinander abgezogen. Die beiden Ergebnisse sind bereits ab der zweiten Stelle unterschiedlich.

Ergebnisse dieser Art dürfen nur dahingehend interpretiert werden, dass das Ergebnis (*in Bezug auf die Ausgangswerte*) Null ist (*oder sehr nahe daran liegt*). Keinem der beiden Rechenwege gebührt hier irgendein Vorrang; beide Zahlenwerte haben keine weitere Aussagekraft und sind zu verwerfen. Es lassen sich sogar Beispiele konstruieren, bei denen auf verschiedenen Rechenwegen unterschiedliche Vorzeichen entstehen.

Das Ergebnis kann eigentlich gar nicht genug gewürdigt werden! Zwei der ersten ehernen mathematischen Gesetze, die in der Schule vermittelt werden, sind nämlich das Assoziativgesetz und das Distributivgesetz:

$$a \circ (b \circ c) = (a \circ b) \circ c$$
$$a * (b + c) = a * b + a * c$$

Unser Resultat besagt im Grunde nicht weniger, als das fast die gesamte Mathematik, wie sie in den Lehrbüchern steht (*zumindest die, die den Körper der reellen Zahlen verwendet*), in dem Augenblick, in dem wir uns daran machen, Ergebnisse auf einem Rechner mit Hilfe von Fließkommazahlen zu berechnen, nicht mehr gilt! Wir müssen uns daher im Grunde immer fragen, was wir da wohl gerade berechnet haben, und die Nonchalance, mit der manche Informatiker über diese Tatsachen hinweggehen, weist nicht nur auf ein etwas gestörtes Verhältnis zu Mathematik hin, sondern auch zu den Eigenschaften der Maschinen, mit denen sie arbeiten.

Statistisch gesehen sind aber auch nur die Subtraktionen kritisch: Genügend große Mantissen vorausgesetzt, müssen schon sehr viele Additionen oder Multiplikationen durchgeführt werden, um den theoretischen Fehler in messbare Gefilde aufzuschaukeln. Betrachten wir als Beispiel eine 10 km lange Brücke, die auf eine Toleranz von $\pm$ 1 mm berechnet werden muss, so liegt die erforderliche Genauigkeit bei 10^{-7}, das heißt es dürfen etwa eine Milliarde `double`–Berechnungen vorgenommen werden, bis diese Grenze theoretisch erreicht wird.[2] Da die Fehler aber wohl kaum immer in die gleiche Richtung weisen, sondern sich gegenseitig kompensieren, ist es durchaus realistisch anzunehmen, dass der Fehler unterhalb 10^{-12} liegt.

Lassen sich solche Effekte bei Subtraktionen vermeiden? Die Antwort auf diese Frage ist leider wenig ermutigend: In wenigen bestimmten Fällen kann eine kritische Subtraktion zweier annähernd gleichgroßer Zahlen durch eine andere Operation ersetzt werden, die nur eine geringe Fehlerverstärkung aufweist. Dazu ist aber eine Untersuchung des Algorithmus und der Zahlenbereiche notwendig. Ist beispielsweise $1 - \cos(x)$ zu berechnen, so ist zu berücksichtigen, dass für $x \approx 0$

[2] In der Optik sind auch noch einige Zehnerpotenzen mehr gefordert.

der Kosinus fast Eins, so dass massive Fehlerverstärkung zu erwarten ist. Durch eine konjugierte Erweiterung ist das Desaster abzuwenden:

$$1 - \cos(x) = (1 - \cos(x)) * \frac{1 + \cos(x)}{1 + \cos(x)} = \frac{\sin(x)^2}{1 + \cos(x)}$$

Der rechte Term ist numerisch stabil, und eine Auswertung zeigt

```
x                    = 9.9999999999999995e-008
1-cos(x)             = 5.0000151588514008e-015
sin(x)^2/(1+cos(x)) = 4.9999999999999953e-015
```

Ein Algorithmus ohne Stabilisierung hätte nach Berechnung von $1 - \cos(x)$ nur noch Zahlenwerte mit einer Genauigkeit von fünf Stellen produziert – deutlich zu wenig für die meisten technischen Anwendungen. Allerdings ist die Stabilisierung nur für Argumente, die hinreichend klein sind, notwendig. In anderen Fällen kann eine Stabilisierung für ein bestimmte Argumentwerte sogar zu Instabilitäten gegenüber der ursprünglichen Formel an anderen Stellen führen, so dass programmintern eine Weichenstellung notwendig wird. Diese Form der Fehlervermeidung setzt daher eine Analyse der Algorithmen und eine Festlegung von Bereichen, in denen eine bestimmte Strategie verfolgt wird, voraus. Wir gehen nicht weiter auf dieses Thema ein und verweisen auf einschlägige Lehrbücher der numerischen Mathematik.[3]

Leider sind mathematische Stabilisierungen in vielen Fällen gar nicht möglich, so dass sich massive Fehlerverstärkungen nicht vermeiden lassen. Ein Beispiel ist das Skalarprodukt zweier Vektoren, das – nicht unbedingt als skalares Vektorprodukt deklariert – in einer Vielzahl von Algorithmen auftritt und aus theoretischen Gründen nur sehr selten zu vermeiden ist. Da man nie weiß, wann es zu echten Subtraktionen kommt und in welchem Verhältnis die Zahlen dann zueinander stehen, lässt sich hier theoretisch-algorithmisch nichts verbessern. Wir können lediglich versuchen, sie zu kontrollieren, das heißt zu einer Aussage zu kommen, wie weit einem Ergebnis zu trauen ist.

16.2 Kontrolle von Fehlern

Eine in fast allen Anwendungen auftretende Operation ist die erwähnte Bildung eines Skalarproduktes

$$c = \sum_{i=1}^{n} a_i * b_i$$

[3] Angemerkt sei noch, dass jeder Informatiker über die numerisch stabilisierte Form der Lösung einer quadratischen Gleichung (*sogenannte PQ-Formel*) verfügen sollte, bei deren Einsatz kein Nachdenken erforderlich ist. Echte algorithmische „Viechereien“ sind aber schon die Lösungen für die kubische Gleichung, in der fast schon die theoretisch mögliche Anzahl von Fallunterscheidungen benötigt wird.

mit irgendwelchen „Vektoren“ $\vec{a}$, $\vec{b}$. Die Summanden können wechselnd positiv oder negativ sein, ohne dass algorithmisch dagegen irgendetwas zu machen ist. Um es vorweg zu sagen: Eine Verringerung von Fehlern ist nicht erreichbar. Wenn man aber schon mit den Fehlern leben muss, sollte man wenigstens wissen, wie groß sie sind. Der Kunde wird als Ergebnis einer Rechnung eine Aussage wie 35,53 $\pm$ 0,02 erwarten und möglicherweise auch darauf bestehen, dass für die angegebenen Grenzen eine Garantie gegeben wird.

Um einen Vertrauensbereich für das Ergebnis garantieren zu können, sind zusätzlich Auswertungen notwendig. Wenn über $\vec{a}$, $\vec{ab}$ keine weiteren Aussagen über die Verteilung von Vorzeichen der Komponenten existieren, kann die Summierung beispielsweise in positive und negative Terme getrennt werden:

$$\begin{aligned} 0 \le i \le n\text{:} \quad & c = a_i * b_i \\ & c \ge 0\text{:} c_+ = c_+ + c, \\ & c < 0\text{:} c_- = c_1 + c \\ & c = c_+ + c_- \end{aligned}$$

Eine in dieser Form getrennte Summmierung führt jedoch dazu, dass die Kompensation der in unterschiedliche Richtungen weisenden Fehler geringer ist, das heißt das Ergebnis ist in den meisten Fällen noch schlechter als das schulmäßig ermittelte. Eine Implementierung hat daher ein anderes Aussehen:

```
for(i=0;i<n;i++){
    c=a[i]*b[i];
    sum+=c;
    if(c>0)
        sump+=c;
}//endfor
```

Anstelle eines Summenterms werden zwei ermittelt. Über das Verhältnis `sum/sump` kann der Fehler abgeschätzt werden. Bei $|c| \approx |c_+|$ muss man sich wenig Gedanken über den Fehler machen, während sich bei $|c| \ll |c_+|$ viele Summanden gegenseitig kompensiert haben und die Anzahl der exakten Stellen geringer ausfällt.

Sind über das Verhalten der Summenterme weitere Eigenschaften bekannt, so kann das Verfahren modifiziert werden. Insbesondere bei Reihenentwicklungen gilt häufig

$$s_0 < s_1 < \ldots < s_{max} > s_{max+1} > \ldots s_{n-1} \ge s_n \ge \ldots$$

das heißt die Summe wird zu Beginn größer und fällt dann monoton. Die Genauigkeit des Ergebnisses hängt dann vom Verhältnis des größten erreichten Zwischenwertes und dem Endwert ab. Wächst beispielsweise eine Summe bis in den Bereich $s_{max} \approx 10^8$ und fällt dann auf $s_\infty \approx 1$ ab, so sind acht Stellen an Genauigkeit verschwunden, da sie beim Subtrahieren durch irgendwelche Reste oder Nullen aus dem Rechenwerk aufgefüllt wurden. Zur Kontrolle wird in solchen Fällen das Maximum ermittelt:

```
for(i=0;i<n;i++){
    sum=sum+a[i]*b[i];
    sm=max(abs(sum),sm);
}//endfor
```

Aus dem Verhältnis `sm/sum` kann nun wieder ermittelt werden, wie viele Stellen des berechneten Wertes vertrauenswürdig sind.

Diese Art der Fehlerkontrollen setzen also ebenfalls Analysen der Algorithmen und Erweiterungen der Implementierungen voraus, denn man muss ja wissen, an welchen Positionen die zusätzlichen Messungen vorgenommen werden müssen und wo die Kontrollwerte benötigt werden. Anders ausgedrückt, eignen sich solche Kontrollen nicht für den Einsatz in Template-Klassen, da diese in der Regel viel zu allgemein gehalten sind.

Eine dritte, ebenfalls nur nach Analyse des Algorithmus durchführbare Kontrolle ist die Probe, die in der Schule meist eingeübt wird, sich bei der Implementierung von Algorithmen aber meist auf den grundsätzlichen Funktionstest beschränkt und bei der Problemlösung oft unterbleibt. Bei der Lösung von linearen Gleichungssystemen existieren beispielsweise zwei Stellen, an denen Proben durchgeführt werden können:

- Nach der Zerlegung einer Matrix in die linke und rechte Dreiecksmatrix kann untersucht werden, ob durch Multiplikation der Dreiecksmatrizen wieder die Ausgangsmatrix entsteht. Größere Abweichungen weisen darauf hin, dass einige Zeilen der Matrix weitgehen kolinear sind und eine Lösung grundsätzlich mit Vorsicht betrachtet werden muss.
- Nach der Ermittlung der Lösung kann diese in das Gleichungssystem eingesetzt werden. Treten größere Abweichungen zum Zielvektor auf, kann sogar das System nochmals für den Differenzvektor gelöst und so versucht werden, durch die hierdurch ermittelten Korrekturen die Lösung zu verbessern (*Nachiteration*).

Eine vierte, immer funktionierende Methode besteht in der Wiederholung der Rechnung mit einem Zahlentyp höherer Genauigkeit. Aufgrund des Aufwands wird man diesen Weg aber nur unter gewissen Voraussetzungen wählen.

(a) Durch Auswahl besonders kritischer Beispiele kann festgestellt werden, wie anfällig ein Algorithmus gegen numerische Instabilitäten ist. Das erfolgt einmalig beim Testen und Optimieren, so dass kaum Aufwandsprobleme entstehen.
(b) Für die Wiederholung im laufenden Betrieb müssen schon hinreichende Anzeichen auf eine mögliche Instabilität vorliegen. Die Messlatte für solche Anzeichen wird um so höher liegen, je größer der Aufwand für die erneute Rechnung ist. Steht beispielsweise kein hardwareunterstützter genauer Typ zur Verfügung, wird man den Einsatz länger hinauszögern als bei dessen Verfügbarkeit.

Als Mittelweg bleibt noch die Möglichkeit, die maximalen Fehler durch gezielte Einstellung der Rundungsrichtungen in jedem Rechenschritt zu ermitteln und die Fehler zu addieren. Hierdurch erhält man eine obere und eine untere Grenze um den nach normalen Regeln berechneten Wert, die der wahre Wert auch bei ungünstigster Kummulation von Fehlern nicht überschreiten kann. Der Vorteil liegt darin, dass solche Berechnungen mit schnellen Standarddatentypen durchgeführt werden können, so dass der Einsatz hochgenauer, aber langsamer Datentypen auf wirklich kritische Fälle begrenzt bleiben kann. Was sich zunächst einfach anhört, besitzt jedoch ebenfalls eine Reihe von Ecken und Kanten, wie die folgende ausführliche Untersuchung zeigt.

16.3 Arbeiten mit Polynomen

16.3.1 Eigenschaften von Operatoren

Einige Eigenschaften von Polynomen haben wir bereits diskutiert, andere sollten hier noch geklärt werden, um Ihnen eine korrekte Implementation zu ermöglichen. Eine charakteristische Eigenschaft, die immer bekannt sein sollte, ist der bereits erwähnte Grad des Polynoms. Hier ist eine Festlegung notwendig, wie der Grad im programmiertechnischen Sinn zu verstehen ist. Mathematisch wird ein Polynom durch

$$P(x) = \sum_{k=0}^{n} a_k * x^k$$

angegeben, wobei definitionsgemäß n der Grad ist, d.h. es gilt ($a_n \neq 0$). Programmierschleifen laufen in C/C++ aber nur bis (n-1). Bei einer Implementation gilt es also zu überlegen, ob eine Methode `polynom::grad()` auch für Schleifen verwendet werden soll, das heißt `for(i=0;i<=p.grad();++i)` als Schleifenkonstrukt verwendet wird oder beispielsweise eine weitere Methode `polynom::size()` definiert wird, die eine Nutzung in der herkömmlichen Art erlaubt (*ich empfehle die zweite Variante*).

Wenn der Grad eines Polynoms immer exakt bekannt sein soll, dürfen wir wie bei den schwach besetzten Matrizen kein Setzen der Koeffizienten mit dem `operator[](..)` zulassen. Vielmehr ist wie dort zu definieren (*das Innenleben dürfte dem Leser keine Probleme bereiten*).

```
const T& operator[](int i) const;
void set(int i, const T& t);
```

Bei der Addition (oder Subtraktion) gradgleicher Polynome kann der Grad des Ergebnisses kleiner sein als der Ausgangsgrad, wenn hohe Koeffizienten sich gegenseitig aufheben. Bei Datentypen, die ein exaktes Rechnen erlauben, ist es kein Problem, festzustellen, ob ein Koeffizient nach der Rechnung Null ist oder nicht;

anders sieht es bei rundungsfehlerbehafteten Datentypen aus, für die jeweils ein Vergleichswert benötigt wird. Wir haben uns mit diesem Thema bereits in Kap. 4 auseinander gesetzt. Muss nun Vergleichswert für jeden Koeffizienten gespeichert werden? Die folgenden Überlegungen zeigen, dass dies nicht der Fall ist.

Zur Speicherung des Referenzwertes bei kritischen Operationen (*Addition/Subtraktion sowie Division, weil dort im Rahmen des Algorithmus eine Subtraktion durchgeführt wird*) deklarieren wir in der Klasse ein zusätzliches Attribut, auf das zu Beginn einer arithmetischen Operation der höchste Koeffizient geschrieben wird. Die Gradprüfung ist am Ende jeder kritischen Aktion durchzuführen, so dass der Grad immer garantiert werden kann (*die Details überlasse ich Ihnen*).

Es ist einfach nachzuvollziehen, dass dieser eine Referenzwert für die Prüfung des kompletten Polynoms hinreichend ist. Beispielsweise ist in

$$P(x) = 3.2 + 1.2 * 10^{-15} * x + 1.03 * x^2$$

der Koeffizient bei x zwar nicht Null, aber im Grunde als Null zu betrachten (*die Prüfung niedriger Koeffizienten im normalen Rechenverlauf führen wir aus Effizienzgründen natürlich nicht durch*). Ziehen wir davon das Polynom $P(x) = 5.2 + 1.3 * 10^{-15} * x + 1.03 * x^2$ ab, so ist das Resultat nach dem oben angerissenen Prüfverfahren richtig ein Polynom nullten Grades. Hätten wir für die unteren Koeffizienten aber als Vergleichswert einen der beiden Werte herangezogen, so wäre das (*falsche*) Ergebnis ein Polynom ersten Grades mit sehr kleinen Koeffizienten.

Die Algorithmen für Addition/Subtraktion und Multiplikation sind aufgrund der Mathematik klar und bedürfen keiner weiteren Erläuterung:

$$P(x) + Q(x) = \sum_{k=0}^{max\ (grad(P),\ grad(Q))} (a_k + b_k) * x^k$$

$$P(x) * Q(x) = \sum_{k=0}^{grad(P)} \sum_{j=0}^{grad(Q)} a_k * b_j * x^{k+j}$$

Spezielles zur Division. Etwas komplexer ist lediglich die Division, die ähnlich einer Division ganzer Zahlen auf dem Papier durchgeführt wird. Allerdings ist hier eine Fallunterscheidung notwendig. Wir müssen nämlich beachten, ob das Polynom über einem Körper definiert ist oder über einem euklidischen Ring, oder verständlicher ausgedrückt, ob eine Division immer einen eindeutigen Quotienten erzeugt wie bei reellen Koeffizienten oder nur eine Division mit Rest möglich ist wie bei ganzzahligen Koeffizienten. Für das Ergebnis von

$$P(x) = S(x) * Q(x) + R(x)$$

gilt nämlich je nach Koeffizienteneigenschaft:

$$\begin{array}{ll} \textit{Körper}: & grad\,(R) < grad\,(Q) \\ \textit{Ring}: & grad(r) \leq grad(P) \end{array}$$

Haben wir einen Körper vor uns, wird die Division nach folgendem iterativen Algorithmus durchgeführt:

(a) Sei $n = grad(P), m = grad(Q)$. Für $n \geq m$ ist der nächste Koeffizient des Ergebnispolynoms

$$c_{n-m} = a_n / b_m$$

(b) Das Zwischenergebnis ist

$$P_{neu}(x) = P(x) - Q(x) * c_{n-m} * x^{n-m}$$

(c) Setze $P(x) = P_{neu}(x)$. Ist $n \geq m$, so wird bei (1) fortgefahren. Bei $n < m$ sind die Koeffizienten des Teilerpolynoms im Vektor $\vec{c}$ enthalten, das Polynom enthält den Divisionsrest, $Q(x)$ ist unverändert.

Nach einem Durchlauf ist der Grad des Dividenden um mindestens 1 gesunken. Bei Koeffizienten aus einem Ring geschieht das aber nur, wenn der Koeffizient im Dividenden ein Vielfaches des Koeffizienten des Divisors ist. Im anderen Fall – der Dividendenkoeffizient muss größer als der Divisorkoeffizient sein – ist nach der Division der Dividendenkoeffizient (*der spätere Rest bei der Division*) bei gleich bleibendem Grad lediglich kleiner als der Divisorkoeffizient und der Algorithmus bricht ab. Die Prüfung wäre also zu ergänzen durch

$$grad(P) \geq grad(Q) \wedge a_{n,P} \geq a_{m,Q}$$

Da solche Anwendungsfälle in der Praxis allerdings recht selten auftreten, verschieben wir die weitere Diskussion auf eine späteres Kapitel, in dem wir genau einen solchen Spezialfall betrachten.

> **Aufgabe.** Implementieren Sie einen Algorithmus für die Division mit Rest. Dieser kann immer verwendet werden, in dem je nach gesuchtem Ergebnis das Teilerpolynom (`operator%(..)`) oder das Restpolynom (`operator/(..)`) gelöscht wird.

Aufwandsbetrachtung. Die Multiplikation ist von quadratischer Ordnung bezüglich des Polynomgrades, das heißt Polynome vom Grad 1.000 benötigen das Einhundertfache der Rechenzeit von Polynomen des Grades 100. Ein anderer Multiplikationsalgorithmus mit geringerer Ordnung arbeitet mit einer Fouriertransformation der Koeffizienten und ist bei hohen Polynomgraden (>1.000) unter bestimmten Voraussetzungen einsetzbar (*siehe Kapitel „Zahlendarstellungen“*). Bei Polynomen mit derart hohen Graden ist man, von wenigen Ausnahmen abgesehen, meist weniger an der Berechnung von Funktionswerten als am Verhalten der Koeffizienten unter bestimmten Nebenbedingungen interessiert. Sie treten häufig im Zusammenhang mit langen ganzen Zahlen auf, und zusätzlich werden für den Algorithmus dann meist lange Fließkommazahlen benötigt. Wir verschieben

die Diskussion der Algorithmen deshalb auf das Kap. 12.1.2, in dem lange ganze Zahlen behandelt werden.

Schwach besetzte Polynome. In einigen Arbeitsbereichen wie der Zahlentheorie können Polynome mit sehr hohen Graden, aber nur wenigen von Null verschiedenen Koeffizienten auftreten. Hier bietet sich u.U. an, nicht alle Koeffizienten in einem Feld zu speichern, sondern nur die von Null verschiedenen zusammen mit ihren Indizes. Eine passende Speicherstruktur wäre beispielsweise

```
vector<map<int,T> > p;
```

Die Implementierung ist trivial, und ich überlasse das Weitere daher Ihnen. Neben einer Einsparung von Speicherplatz kann die Verwendung solcher Speicherstrategien auch zu einer Verbesserung des Laufzeitverhaltens führen, da nicht laufend sinnlose Rechnungen mit Null als Summand oder Faktor durchzuführen sind. Man muss dabei aber sicher sein, dass ein namhafter Teil der Koeffizienten dauerhaft Null ist und nicht während der Rechnung besetzt wird.

> **Aufgabe.** Konstruieren Sie einige Beispiels mit teilbesetzten Matrizen und führen Sie Laufzeitmessungen durch.

Für die **Berechnung von Funktionswerten** und Werten der Ableitung ist der Horner–Algorithmus einzusetzen

```
inline T operator()(const T& x)const{
    if(size()==0)
        return algebra<T>::null();
    T sum=dat->value[grad()]; int i;
    for(i=size()-2;i>=0;--i);
        sum=sum*x+dat->value[i];
    return sum;
};//end function
```

> **Aufgabe 16.3-1.** Entwerfen Sie einen Algorithmus für die Berechnung der Wertes der ersten Ableitung eines Polynoms für gegebenes x . Entwerfen Sie eine Algorithmus, der sowohl den Wert des Polynoms als auch den der ersten Ableitung errechnet.

16.3.2 Nullstellen I: Berechnen

Der in der letzten Aufgabe entworfene Algorithmus wird zur Berechnung von reellen Nullstellen von Polynomen benötigt. Ohne nun allzu tief in die Details einzudringen (*dies soll ja kein Lehrbuch über Numerische Mathematik werden*), möchte ich Ihnen hierzu einige weitere Übungsaufgaben geben. Nullstellen höherer Polynome lassen sich nicht analytisch, das heißt durch eine Formel ermitteln, sondern müssen durch ein Iterationsverfahren berechnet werden. „Höhere“ Polynome

sind solche ab dem Grad Vier, unter gewissen Nebenbedingungen auch ab den Grad Fünf, jedoch ist das bereits so aufwendig, dass man sich auf in der Praxis auf Nullstellen quadratischer Gleichungen beschränken und ab dem Grad Drei iterieren sollte. Für den Einsteiger häufig mit gedanklichen Problemen verbunden ist die Trennung des Nullstellenproblems in zwei verschiedene Unterprobleme: Die Bestimmung eines Intervalls, in dem mindestens eine Nullstelle liegt, und die Berechnung der Nullstelle selbst.

Ich beginne mit einigen Bemerkungen zum zweiten Teil. Gebräuchlich ist das Verfahren nach Newton, in dem aus einem Näherungswert x_k ein neuer und besserer Näherungswert durch

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

ermittelt wird. Die Iterationsformel resultiert aus der Überlegung, dass

$$f(x_2) \approx f(x_1) + f'(x_1) * (x_2 - x_1)$$

gilt, wenn x_1,x_2 nahe beieinander liegen. Betrachtet man x_2 als Nullstelle, also $f(x_2) = 0$, und löst nach x_2 auf, so erhält man die Iterationsformel, die allerdings nur dann gilt, wenn der Betrag der Ableitung in Richtung der Nullstelle nicht größer wird. Versuchen Sie, sich diese Beziehungen mittels eines Funktionsgraphen klar zu machen; weitere Details sind jedem Buch über numerische Mathematik zu entnehmen.

Ist die Nullstelle gefunden oder erreicht, so wird der Zähler im zweiten Term auf der rechten Seite Null und der neue Näherungswert ist gleich dem alten. Allerdings wird das wohl nur selten geschehen, wie unsere Betrachtungen über Rundungsfehler nahe legen.

> **Aufgabe.** Konstruieren Sie einige synthetische Polynome, deren Nullstellen Sie kennen, und versuchen Sie Kriterien zu entwickeln, wann eine Iteration, die von einem beliebigen Wert startet, abgebrochen werden kann.

Ihre Versuche haben Ihnen möglicherweise folgendes gezeigt:

(a) Die Rechnungen werden instabil, wenn es sich um höhere Nullstellen handelt. Das ist verständlich, denn ab einer zweifachen Nullstelle hat sowohl die Funktion als auch die Ableitung am Fixpunkt den Wert Null, so dass die Newton–Formel dort gar nicht auswertbar ist.
(b) Das Verfahren reagiert recht sensitiv auf den Startpunkt einer Iteration. Um eine bestimmte Nullstelle zu berechnen, muss der Startwert in der Nähe der Nullstelle liegen, und auch das reicht oft nicht: Wenn die Iteration auf der „falschen Seite“ der Nullstelle beginnt, wird anstelle der nahen Nullstelle möglicherweise eine andere, die in der gleichen Richtung vom Startpunkt aus liegt aber weiter entfernt ist, gefunden (*Sie können das Verhalten verstehen, wenn Sie den Graphen des Polynoms zeichnen und die Beziehungen der Newton–Formel in den*

Graphen eintragen. Hinweis: Die Newton–Formel ist eine Geradengleichung mit dem Schnittpunkt x_{k+1} mit der X–Achse).

Um das Problem des Nullstellengrades in den Griff zu bekommen, wird in eine Rechnung nicht das Originalpolynom, sondern

$$P(x) = \frac{f(x)}{f'(x)}$$

eingesetzt. Dieses Polynom besitzt nur noch einfache Nullstellen, was Sie sich klarmachen können, wenn Sie Summendarstellung eines Polynoms durch seine Nullstellendarstellung ersetzen

$$f(x) = \sum_{k=0}^{n} a_k x^k = a_n(x - x_0)^{e_0}(x - x_1)^{e_0} \ldots (x - x_m)^{e_m}$$

und $P(x)$ ausrechnen (*erweitern Sie Ihre Versuche mit dieser Methode*).

Der Algorithmus funktioniert auf jedem geeigneten Körper, also auch mit komplexen Zahlen, allerdings weiß man nur in diesem Körper, wie viele Nullstellen ein Polynom besitzt (*nämlich genauso viele, wie der Grad des Polynoms beträgt*). Im reellen Fall gilt dies nicht. Wenn der Algorithmus keine Nullstelle (mehr) findet, stellt sich die Frage „existieren eventuell noch mehr und wo sind sie zu suchen?“

16.3.3 Nullstellen II: Finden

Dieser Aufgabe nimmt sich das folgende Verfahren an:[4] Mittels einer Sturmschen Kette kann untersucht werden, wie viele (*reelle*) Nullstellen in einem bestimmten Intervall liegen. Eine Sturmsche Kette ist eine endliche Folge stetiger Funktionen $f_0, f_1, \ldots f_k$ mit den Eigenschaften

(a) $sing(f_k(x)) = const$
(b) $f_j(a) = 0 \Rightarrow sign(f_{j-1}(a)) = -sign(f_{j+1}(a)) \neq 0$
(c) $f_0(a) = 0 \Rightarrow sign(f_1(a)) \neq 0$

Bei einer Sturmschen Kette werden nur die Vorzeichen der Funktionen ausgewertet, genauer die Anzahl der Vorzeichenwechsel. Bedingung b) sagt aus, dass im Inneren der Kette bei stetiger Veränderung von x kein zusätzlicher Vorzeichenwechsel entstehen kann, da bei Nullwerden eines Gliedes die Nachbarglieder verschiedene Vorzeichen besitzen. Da das letzte Glied konstant ist und nur durch den Nulldurchgang von $f_0(x)$ die Anzahl der Vorzeichenwechsel in der Kette geändert werden

[4] Ich erkläre hier einige Details mehr als beim Newtonverfahren, da Sturmsche Ketten nicht in jedem Numerik-Buch gefunden werden können.

kann, gibt die Differenz der Vorzeichenwechsel an den Grenzen eines Intervalls die Anzahl der Nullstellen im Intervall an (*machen Sie sich dies an einer Skizze klar*). Obwohl eine solche Funktionenfolge vielleicht exotisch anmutet, wird sie für Polynome wird auf folgende verblüffend einfache Weise rekursiv gebildet:

$$f_0(x) = P(x), \qquad f_1(x) = P'(x)$$
$$f_k(x) = Q(x) * f_{k+1}(x) - f_{k+2}(x)$$

Die Kette bricht ab mit $f_r(x) = 0$, $Q(x)$ kann jeweils verworfen werden, da es nicht benötigt wird.

> **Aufgabe.** Berechnen Sie die Sturmsche Kette für ein vorgegebenes Polynom $P(x)$, das heißt die Folge der Divisionsreste. Definieren Sie dazu eine Klasse „SturmKette“ mit geeigneter Speicherstruktur. Der Algorithmus zu Berechnung der einzelnen Glieder der Kette entspricht bis auf das Vorzeichen (*Achtung!!*) dem Algorithmus für die Berechnung des „größten gemeinsamen Teilers“. Die Requisition für diese Aufgabe sind Ihnen somit alle bekannt.

Hinweis. $f_r(x) = 0$ bedeutet nicht automatisch $f_{r-1}(x) = const$. Wenn das Polynom mehrfache Nullstellen besitzt, ist das letzte Folgenglied nicht konstant und die Folge daher noch keine Sturmsche Kette. Um sicher eine Sturmsche Kette zu erhalten, ist daher die gesamte Folge durch das letzte Glied zu dividieren. Dieses wird dabei Eins, und aus $f_0(x)$ (*und allen Zwischengliedern*) werden die mehrfachen Nullstellen herausgekürzt. Die Nullstellen von $f_0(x)/f_{r-1}(x)$ sind jedoch immer noch die gleichen wie von $f_0(x)$

Zur Bestimmung der Anzahl der Nullstellen in einem Intervall $[a, b]$ wird an beiden Intervallgrenzen die Anzahl der Vorzeichenwechsel in der Kette berechnet. Der Betrag der Differenz beider Werte gibt die Anzahl der Nullstellen im Intervall an.

> **Aufgabe.** Komplettieren Sie durch Klasse `SturmKette` durch eine Auswertefunktion. Prüfen Sie wiederum mit synthetischen Polynomen. Die Kette muss nur einmal berechnet und kann anschließend beliebig oft mit unterschiedlichen Intervallen verwendet werden.

Mittels der Sturmschen Kette können nun die Bereiche genau festgelegt werden, in denen man den Newton–Algorithmus „von der Kette“ lässt, um die Nullstellen zu bestimmen. Im Wechselspiel beider Algorithmen kann sichergestellt werden, ein Intervall vollständig untersucht und alle Nullstellen gefunden zu haben. Weitere strategische Übungen überlasse ich Ihnen.

Anmerkung. Die Algorithmen lassen sich in dieser Kombination nur zur Suche reeller Nullstellen einsetzen. Für die Suche nach komplexen Nullstellen ist die Sturmsche Kette ungeeignet; die Methoden hierfür sind zwar auch nicht sonderlich kompliziert umsetzbar, verlangen aber zum Verständnis doch schon Kenntnisse der komplexen Analysis, weshalb ich hier nicht weiter auf dieses Thema eingehe. Bei Polynomen über anderen Körpern stellen sich weitere Probleme. Wegen der Bedeutung von Polynomen in vielen Anwendungsbereichen und der anschaulichen Übungsmöglichkeiten habe ich dem Thema trotzdem einigen Raum gegeben.

16.4 Intervallmathematik

16.4.1 Grundlagen

Die im letzten Kapitel vorgestellten Fehlerkontrollen und Algorithmenanpassungen sind aufwändig und in manchen Fällen nur schwierig durchführbar. Es besteht daher auch Bedarf an Methoden, die ohne aufwändige Vorarbeit an den Implementierungen der Algorithmen Vertrauensintervalle liefern,[5] wobei ebenfalls klar ist, dass diese Einsparung auf der Entwicklungsseite nur durch entsprechenden massiven Einsatz von Rechenleistung ausgeglichen werden kann. Die Lösung dieses Problems heißt „Intervallarithmetik“, wobei aber auch bei dieser Methode von vornherein anzumerken ist, dass man um reifliche Überlegungen, bei welchen Algorithmen man diese Methode anwendet, ebenfalls nicht herum kommt.

Anstatt wie bisher mit einem gerundeten Wert wird mit den Intervallen, in denen die wahren Werte liegen, gerechnet. Von der Zahl $\sqrt{2}$ können wir bei Rechnungen mit sechs Stellen Genauigkeit beispielsweise davon ausgehen, dass der korrekte Wert im Intervall

$$1{,}414213 < x < 1{,}414214$$

liegt. Addieren wir zu $\sqrt{2}$ einen zweiten Wert hinzu, von dem bekannt ist, dass er im Intervall

$$2{,}110000 \leq y \leq 2{,}110001$$

liegt, so kann schlimmstenfalls passieren, dass beide wahre Werte jeweils mit der linken oder rechten Intervallgrenze übereinstimmen. Der neue wahre Wert muss daher im Intervall

$$3{,}524213 \leq x + y \leq 3{,}524215$$

liegen. Anstelle der Addition zweier Werte (x,y) tritt daher nun zusätzlich die Addition von Intervallgrenzen, das heißt die Anzahl der Rechenschritte verdreifacht sich.

Hierbei haben wir zwar die Fehlergrößen δ berücksichtigt, aber noch nicht die Rundungsfehler ϵ, die in jedem Rechenschritt zusätzlich in Erscheinung treten. Besitzt das neue Ergebnis mehr Stellen als im Speicher vorhanden sind, so unterliegen auch die Intervallgrenzen selbst einer „Zwangsrundung“, die aber nun streng in einer Richtung durchgeführt wird, d.h. im unteren Intervall wird (*vorzeichenrichtig*) stets auf den nächstkleineren, im oberen Intervall auf den nächstgrößeren Wert gerundet:

---

[5] Vor allen Dingen, wenn bereits erprobte Algorithmen auf Problemfälle stoßen. Die Bereitschaft, für die 1–2% Problemfälle neue Implementationen zu finanzieren, ist natürlich begrenzt.

$$\begin{aligned}&[1{,}414213;\ 1{,}414214] + [0{,}08619231;\ 0{,}08619234]\\ &\quad = [1{,}50040531;\ 1{,}50040634]\\ &\quad = [1{,}500405;\ 1{,}500407]\end{aligned}$$

Bei diesem Intervall können wir nun sicher sein, dass auch nach Rundung des Rechenergebnisses der wahre Wert weiterhin im berechneten Intervall liegt. Allerdings nehmen wir für diese Garantie in Kauf, dass das Intervall ständig übermäßig verbreitert wird. Durch das in jedem Rechenschritt erfolgende Runden in bestimmte Richtungen entfernen sich die Intervallgrenzen naturgemäß bei längeren Rechnungen relativ weit voneinander. Ergänzt man die Angaben noch durch die Anzahl der Rechenoperationen, die zu einem Ergebnis geführt haben, beispielsweise

$$\begin{aligned}&result = 4.1234567, Anzahl\,Rechenschritte = 15\\ &garantierter\,Bereich: 4.1233300 \leq result \leq 4.1445332\end{aligned}$$

so erleichtert das eine Interpretation. Gilt mit der Bezeichnung $[a, x, b]$ für ein Intervall

$$x \approx (a+b)/2 \wedge |a-b| \approx n * \epsilon$$

so ist sicher nicht viel Aufregendes passiert und man kann dem Ergebnis vertrauen, ist $|a-b| \gg n * \epsilon$ und liegt x asymmtrisch im Intervall, so ist Vorsicht geboten.

Zur Durchführung der Rechnung können wir wahlweise nach Einlesen der Startwerte und Einstellung des Rechenwerkes auf eine bestimmte Rundungsart den Algorithmus dreimal hintereinander oder alternativ nur einmal, wobei jeder Rechenschritt simultan mit drei verschiedenen Rundungsarten durchgeführt wird, ausführen. Wie wir im folgenden Absatz sehen werden, scheidet der erste, programmiertechnisch einfachere Ansatz aus, da er Ergebnisse liefern kann, die sich der Interpretation entziehen. Wir müssen also einen besonderen Datentyp entwickeln, der an die Stelle des Datentyps `double` tritt und alle drei Rechnungen simultan ausführt.

16.4.2 Vergleiche gerundeter Zahlen

16.4.2.1 Vergleiche rundungsfehlerbehafteter Werte

Sollen zwei Variablen a und b dahingehend untersucht werden, ob ihre Inhalte als gleich zu betrachten sind, so kann bei Vorliegen von Rundungen nicht mehr auf die mathematische Identität $a = b$ geprüft werden, da dieser Fall vermutlich nie eintritt. Wir können nur definieren, dass die Zahlen als gleich betrachtet werden sollen, wenn ihre Differenz den absoluten Fehler nicht überschreitet. Den absoluten Fehler berechnen wir mit Hilfe von ϵ oder einem Vielfachen davon[6]

$$a = b \Leftrightarrow |a-b| \leq \Delta, \quad \Delta = a * \delta$$

[6] Der tatsächliche Fehler ist natürlich außer im ersten Schritt nicht bekannt, wird jedoch meist nicht größer als einige Hundert ϵ werden, sofern keine massive Auslöschungsverstärkung eintritt.

Dies ist eine Definition der Gleichheit von Rechenwerten und hat wenig damit zu tun, dass die wahren Werte ebenfalls tatsächlich gleich sind. Im Rahmen unserer Rechengenauigkeit sind die Zahlen nicht mehr unterscheidbar, wenn diese Relation erfüllt ist, aber schon bei Rechnungen mit höherer Genauigkeit muss es dabei nicht bleiben! Für den Vergleich wird eine „Führungsgröße“ zur Berechnung des Δ benötigt. Rein formal ist es nicht gleichgültig, welcher der Werte als Führungsgröße verwendet wird. Untersuchen wir den Ausdruck

$$a = b \quad \wedge \quad b \neq a$$

so erhalten wir für dessen Erfüllung die Ungleichung

$$b < |a - b| * \Delta^{-1} \leq a$$

die beispielsweise mit $\Delta = 0.1$, $b = 0.95$, $a = 1.05$ gilt. Neben dem Assoziativ- und dem Distributivgesetz haben wir damit auch noch das Kommutativgesetz auf Rechnern zu Fall gebracht. Allerdings sollte man diese Ausführungen für die Praxis nicht weiter ernst nehmen.

Die Notwendigkeit der Führungsgrößenvereinbarung hat unmittelbaren Einfluss auf das Design von Algorithmen. Ist der Vergleich $a > b$ zweier Größen bei genügendem Abstand auch ohne ein Δ noch relativ schlüssig durchzuführen, so ist für $a = b$ eine der Größen zur Berechnung von Δ heranzuziehen,[7] und für den Test $a = 0$[8] muss die Führungsgröße in der passenden Form extern bereitsgestellt werden. Gerade der letzte Testfall tritt häufig auf, wobei a in der Regel das Ergebnis einer längeren Berechnung. Als Bezugswert dient meist einer der Startwerte der Berechnung, und der Algorithmus muss für das Zwischenspeichern bis zum Vergleich Sorge tragen.[9]

Für die typunabhängige Formulierung von Algorithmen führen wir für die Vergleiche wieder eine Template-Klasse ein, die mit exakten Datentypen die Standardvergleichsoperatoren verwendet:

```
template <class T> struct compare {
    compare(){}
    void load(T const& t){}
    inline bool equal(const T& s, const T& t)
                      {return s==t;}
    inline bool less(const T& s, const T& t)
                      {return s<t;}
    inline bool zero(const T& s)
                 {return s==Constant<T>::null();}
```

[7] Man kann sich hier noch mit dem Gedanken trösten, dass ein Auftreten dieses Vergleichs nur auf einen schlecht aufgearbeiteten Algorithmus zurückzuführen ist.

[8] Leider taucht diese Prüfung relativ häufig auf und lässt sind kaum umgehen.

[9] Beispiel: Kontrolle des Grads von Polynomen nach Addition von Polynomen gleichen Grades. Zur Kontrolle sinnvollerweise wird der höchste Koeffizient eines der Polynome für einen rekursiven gliedweisen Vergleich verwendet.

```
    inline bool less_equal(const T& s, const T& t)
                {return less(s,t) || equal(s,t);
    }//end function

    inline bool greater(const T& s, const T& t){
        return !less_equal(s,t);
    }//end function

    bool greater_equal(const T& s, const T& t){
        return !less(s,t);
    }//end function
};//end struct
```

Für die rundungsfehlerbehafteten Größen sind Spezialisierungen dieser Klasse zu implementieren. Für den Datentyp `double` folgt beispielsweise

```
inline double& cmp_eps_double(){
    static double r=1.0e-13;
    return r;
}//endfunction

inline double& cmp_ref_double(){
    static double r=1.0;
    return r;
}//endfunction

template <> struct compare<double> {
    compare(){ load(cmp_ref_double());}

    inline void load(const double& s)
                { frexp(s,&ref);}

    inline bool equal(const double& s,
                      const double& t) const {
        return fabs(s-t)<cmp_eps_double();
    }//end function

    inline bool less(const double& s,
                     const double& t) const {
        return (s<t) && !equal(s,t);
    }//end function

    inline bool zero(const double& s) const {
        int e;
        if(s==0)
            return true;
        frexp(s,&e);
        return e<=ref-cmp_exp_double;
    }//end function

private:
```

```
    int ref;
};//end struct
```

Die Standardführungsgrößen sind in Form veränderbarer Singletons implementiert. Sie können durch individuelle Größen überschrieben werden. Um die Vergleiche korrekt einsetzen zu können, ist ein Objekt der Klasse `compare` mit hinreichendem Gültigkeitsbereich zu deklarieren. Es muss zu Beginn einer Rechnung den Vergleichswert aufnehmen und zwischenspeichern, weshalb wir auch nicht mit einer Template-Funktion an dieser Stelle auskommen, sondern ein Klasse mit einer davon abgeleiteten Instanz benötigen. Wie man aber unschwer erkennen kann, ist das im Fall exakter Größen nicht mit einem gegenüber den Standardvergleichsoperatoren erhöhten Aufwand verbunden, da für diese keine Zwischenspeicherung notwendig ist und der Optimierer den unnötigen Code entfernt. Für die Vergleichsoperationen von `double`-Werten werden die Exponenten verwendet, so daß ein `int`-Wert für die Speicherung genügt.

Aufgabe. Die Klasse `complex<>` ist eine Template-Klasse, die mit exakten oder rundungsbehafteten Werten betrieben werden kann. Für den korrekten Betrieb ist daher eine Spezialisierung

```
template <class T> struct compare<complex<T> >
```

notwendig. Entwerfen Sie diese.

Mit ein wenig Überlegung, an welcher Stelle welcher Wert als Referenz geladen werden muss und wie eine Spezialisierung im Falle eines komplizierteren Typs wie beispielsweise einem Polynom nun konkret aussieht, lassen sich mit Hilfe vom `compare` alle Algorithmen typunabhängig implementieren.

16.4.2.2 Vergleiche von Intervallzahlen

Wie sind Vergleiche durchzuführen, wenn wir Intervalle betrachten, konkret: wie ist beispielsweise

$$a < b, \quad a = [1.000,\ 1.500],\ b = [1.250,\ 1.750]$$

auszuwerten? Die Intervalle überlappen zwar in einer Weise, die eine positive Antwort nahe legen, jedoch können die wahren Werte trotzdem eine andere Relation erfüllen. Und wie ist der Vergleich

$$[a,b] = [c,d] \ \ mit \ \ a < c < d < b$$

auszuwerten? Je nach Führungsintervall können wir zu einem bestimmten oder völlig unbestimmten Ergebnis kommen.

Da wir die Standardrechenwerte ebenfalls vorliegen haben, werden Vergleiche daher mit diesen durchgeführt, aber da diese nicht exakt sind, können die Vergleichsentscheidungen auf der Grundlage dieser Werte bei überlappenden Intervallen in Bezug auf die wahren Werte auch falsch ausfallen. Theoretisch könnten

wir uns kritischen Vergleiche, also solche, die anders ausfallen würden, wenn wir sie mit den Grenzwerten ausführen würden, im Verlauf der Rechnung „merken“ und in wiederholten Programmläufen die anderen Rechenzweige untersuchen. Für die Praxis ist das allerdings aus mehreren Gründen uninteressant, angefangen damit, dass bei n aufeinander folgenden kritischen Vergleichen theoretisch $O(2^n)$ verschiedene Rechenwege existieren. Das lässt sich weder zeitmäßig verfolgen noch kann man voraussichtlich mit den Daten irgendetwas anfangen. Wir beschränken die Auswertung von Vergleichen also auf den Vergleich der Standardwerte. Damit ist auch die Entscheidung für die simultane Berechnung aller drei Werte gefallen. Bei getrennten Rechnungen mit unterschiedlichen, aber im Verlauf der Rechnung konstanten Rundungsmodellen können natürlich nur die jeweils vorhandenen Werte miteinander verglichen werden, d.h. hier können Vergleichsentscheidungen in den verschiedenen Durchläufen anders ausfallen, so dass die Ergebnisse der drei Rechnungen nicht miteinander in Beziehung zu setzen sind.

16.4.3 Zwischenbilanz

Halten wir als Ergebnis aus diesen Betrachtungen zunächst fest:

(a) Intervallrechnung ist sinnvollerweise immer im Simultanmodus durchzuführen, wobei alle Vergleiche mit den Standardwerten durchgeführt werden.
(b) Der Rechenaufwand wird hierdurch (*mindestens*) verdreifacht, da jeder Wert durch zwei Intervallgrenzen charakterisiert wird und bei Rechenoperationen beide Grenzen sowie der normale Wert neu berechnet werden müssen. Tatsächlich ist der Aufwand noch um Einiges höher, da die Umschaltbefehle für die Rundungsart und eine Reihe von Entscheidungen aufgrund der Vorzeichen der drei Zahlen hinzukommen, wie wir noch sehen werden.
(c) Das Ergebnis enthält den vollständigen Definitionsbereich des berechneten Wertes, was allerdings nicht bedeutet, dass dort auch der wahre Wert zu finden ist. Wird die Rechnung aufgrund einer sehr großen Intervallbreite mit einem Datentyp höherer Genauigkeit wiederholt, so ist es möglich, wenn auch in der Regel nicht wahrscheinlich, dass eine Verzweigungsentscheidung im Verlauf der Rechnung anders ausfällt und das Endergebnis außerhalb des zunächst ermittelten Intervalls liegt.
(d) Die Intervallrechnung „übertreibt“, da die Grenzen nach dem Prinzip des „schlechtesten denkbaren Falls“ angepasst werden. Je mehr Daten miteinander verrechnet werden, desto größer ist wird die Wahrscheinlichkeit, dass sich im Verlauf einer normalen Rechnung Rundungsfehler kompensieren und der berechnete Standardwert doch recht nahe am richtigen Ergebnis liegt, während die Grenzen bei jedem Schritt systematisch vom statistischen Mittel fortgedrückt werden.

Eine Intervallrechnung sollte daher nicht „blind“ in dem Sinne eingesetzt werden, dass man nur einmal sehen möchte, was passiert. Wegen (b) sind Langläufer, wegen (d) Algorithmen, in denen extrem viele Werte verdichtet werden, wenig geeignete

Kandidaten, und wie wir noch sehen werden, ist die Intervallrechnung für einige Algorithmen unbrauchbar. Möglich ist aber durchaus ein Mischeinsatz von Intervallrechnung und normaler Rechnung: ist man sich der numerischen Stabilität eines Teilalgorithmus sicher, so wird dieser Teil „normal“ berechnet, ggf.ergänzt durch die diskutierten Kontrollmaßnahmen. Lediglich dann noch verbleibenden kritische Teilalgorithmen werden mittels der Intervallrechnung durchgeführt.

16.4.4 Intervalltypen

Wir haben nun schon mehrfach erwähnt, dass nicht alle Algorithmen für den Einsatz mit Intervallzahlen geeignet sind. Um Entscheidungskriterien zu bekommen, für welche Arten von Algorithmen die Intervallrechnung geeignet ist, fassen das Konzept der Intervallrechnung nun zunächst in einer formalen Darstellung zusammen, wobei wir den Begriff „Intervall“ breiter fassen und dabei zu überaschenden Ergebnissen kommen, was das bisher entwickelte Bild der Intervallrechnung betrifft.

Intervalle werden durch ein Zahlenpaar (a,b) spezifiziert, wobei $a \leq b$, $a,b \in \mathbb{R}$ gilt (*eigentlich* (a,x,b) *mit* $a \leq x \leq b$. *Da aber im Weiteren klar ist, was gemeint ist, lassen wir den mittleren Wert x fort*). Bei komplexen Zahlen oder anderen mehrkomponentigen mathematischen Objekten wird jeder Komponente ein eigenes Intervall zugewiesen, zum Beispiel:

$$z = x + i * y, \;\; a \leq x \leq b, \;\; c \leq y \leq d$$

Den Intervallbegriff beschränken wir im Weiteren aber nicht auf den Einschlußfall $a \leq x \leq b$. Wir unterscheiden folgende Intervallklassen je nach möglicher Lage des wahren Wertes einer Größe x:

(a) Einschlussintervalle, $a \leq x \leq b$,	Symbol **I**	$[a, b]$
mit den Subtypen		
linksseitige Intervalle, $x \leq a$,	Symbol **L**	$[-\infty, b]$
oder		
rechtsseitige Intervalle, $b \leq x$,	Symbol **R**	$[a, \infty]$
offene Menge, $x \in \mathbb{R}$,	Symbol **O**	$[-\infty, \infty]$
(b) Ausschlussintervalle, $x \leq a \;\vee\; b \leq x$,	Symbol **A**	$\overline{[\![a, b]\!]}$

Unterstellen wir den unendlich-Symbolen folgende, wohl recht plausible Rechenregeln

$$\begin{aligned}
&\infty \pm |a| = \infty; \;\; -\infty - |a| = -\infty; \; a \in [R,\infty] \\
&\infty * (\pm\, a) = \pm\infty; \; a \in [R,\infty] \\
&|b| \,/(\pm\, \infty) = 0; \; b \neq \infty \\
&\infty - \infty \;\; , \;\; \infty/\infty \;\; : \; \textit{verboten}
\end{aligned}$$

so reicht es im Weiteren in der Regel, zwischen Einschluss- und Ausschlussintervallen zu unterscheiden und die Subtypen zu ignorieren. Auf dem Datentyp `double` vertreten die speziellen Bitmuster $\pm\#INF$ die unendlich-Symbole, die sich diesen Regeln entsprechend verhalten, so dass wir mit der Typeinschränkung auch praktisch richtig liegen.

Wird in einem vollständigen Modell der Intervallmathematik dieses Klassenschema tatsächlich benötigt? Damit die Klassifizierung einen Sinn erhält, müssen Übergänge zwischen den verschiedenen Intervallklassen im Laufe einer Rechnung stattfinden können, wobei Ausgangspunkt jeder Rechnung Einschlussintervalle sind, bei denen in den meisten Fällen alle drei Werte gleich sind. Im Rahmen der Berechnungen werden sie im Normalfall auch Einschlussintervalle bleiben, jedoch sind auch Übergänge zu anderen Klassen und folgenden Randbedingungen möglich. Im Rahmen der Addition kann einer der Fälle

$$(a < 0 < x < b) \ \vee \ (a < x < 0 < b)$$

eintreten, d.h. die Rechnung verläuft so ungünstig, dass sogar das Vorzeichen eines Zwischenergebnisses in Zweifel gezogen werden muss. Das ist zwar weiterhin ein Einschlussintervall, mit dem normal weitergerechnet werden kann, außer man trifft auf eine Division, bei dem diese Intervallzahl der Divisor ist. Wir zerlegen dazu die Division in eine Inversion und eine Multiplikation:

$$x/y \ \Leftrightarrow \ z = 1/y, \ x * z$$

Hat das (*Einschluss*)Intervall für die Variable y ein eindeutiges Vorzeichen,[10] so erhalten wir bei der Inversion $z = 1/y$ das Ergebnisintervall:[11]

$$y \in [a, b], sign(a) = sign(b) \neq 0 \ \Rightarrow \ z \in \left[\frac{1}{b}, \frac{1}{a}\right]$$

da mit a als kleinstem möglichen Wert für y das Inverse $1/a$ der größtmögliche Wert vonzentsteht. Das Ergebnis gehört der gleichen Intervallklasse an wie das Ausgangsintervall. Falls das Vorzeichen des Ausgangsintervall aber nicht eindeutig oder eine der Komponenten Null ist, erhalten wir:

(a) Wenn die Vorzeichen der Intervallgrenzen verschieden sind, so wird, wie man sich durch eine einfache Rechnung unmittelbar überzeugt, aus einem Einschlussintervall ein Ausschlussintervall

[10] Das Vorzeichen eines Intervalls ist eindeutig, wenn beide Grenzen des Intervalls das gleiche Vorzeichen aufweisen, das heißt $0 < a \ \vee \ b < 0$.

[11] Zur Beachtung: Im Gegensatz zum Normalwert, der beim Vergleich oder bei der Vorzeichenauswertung jeweils mit dem Rundungsfehlerwert korreliert werden muss, sind die Intervallgrenzen als fehlerfreie, direkt vergleichbare Wert zu betrachteten.

$$y \in [a,b], sign(a) = -sign(b) \neq 0 \;\Rightarrow$$
$$z \in [-\infty,\ 1/a] \;\vee\; z \in [1/b,\ \infty]$$

(b) Ist eine der Intervallgrenzen Null, so erhalten wir ein links- oder rechtsseitiges Intervall

$$y \in [a,\ b],\ a = 0,\ a < b \;\Rightarrow\; z \in [1/b,\ \infty]$$
$$y \in [a,\ b],\ a < b,\ b = 0 \;\Rightarrow\; z \in [-\infty,\ 1/a]$$

(c) Sind beide Intervallgrenzen Null ($a = b = 0$), so liegt eine echte Division durch Null vor und das Ergebnis ist nicht definiert. Den Fall einer Division durch Null beschränken wir auf dieses Intervall (*siehe a*). Technisch können wir das Ergebnis der Klasse **O** zuweisen.

Sie verifizieren sicher leicht die Symmetrie der Operationen (a)–(b): Ausschlussintervalle mit $a \leq 0 \leq b$ oder einseitige Intervalle mit eindeutigem Vorzeichen werden bei einer erneuten Inversion wieder zu Einschlussintervallen. Wir haben damit nachgewiesen, dass Intervalle aller Klassen in einer Rechnung aus Intervallen anderer Klassen entstehen können. Außer im Fall der Klasse **O** sind die Übergänge reversibel, und die Rechnung kann fortgesetzt werden.

Lohnt es sich überhaupt, in der Praxis die Intervallübergänge zuzulassen? Schließlich sind Aussagen wie „*das wahre Ergebnis ist größer als 5*“ oder „*das wahre Ergebnis liegt sicher nicht zwischen −200 und +125*“ in der Praxis wohl nur in extrem seltenen Fällen wirklich brauchbar. Trotzdem werden wir die Rechnung bei Verlassen eines Einschlussintervalls fortsetzen, denn der normale Rechenwert läuft ja weiterhin mit und kann ausgewertet werden, ohne dass noch einmal mit Programmierarbeiten zur Typumstellung von Intervall auf `double` begonnen werden muss. Verschaffen wir uns also eine Übersicht, welche Intervalle beim Aufeinandertreffen von Nicht-Einschlussintervallen bei einer Operation entstehen

16.4.4.1 Negierung

Bei der Negierung findet die Transformation

$$r = -r \;\Rightarrow\; [a,\ x,\ b] \rightarrow [-b, -x, -a]$$

statt. Formal finden dabei nur die Klassenübergänge $\boldsymbol{L} \rightarrow \boldsymbol{R}$ und $\boldsymbol{R} \rightarrow \boldsymbol{L}$ statt, die wir nicht weiter behandeln müssen, da es sich um Übergänge zwischen Einschlussintervallen handelt, die gemäß den Rechenregeln für unendlich-Symbole automatische gehandhabt werden.

16.4.4.2 Addition

Mit Hilfe der Negierung kann die Subtraktion auf die Addition zurückgeführt werden, so dass die Untersuchung der Addition bleibt. Im Falle sämtlicher Einschlussklassen, also $\boldsymbol{I}, \boldsymbol{R}, \boldsymbol{L}, \boldsymbol{O}$, gilt

$$[a, x, b] + [c, y, d] = [a + c, x + y, b + d]$$

Man leitet daraus unmittelbar ab, dass das Ergebnis weiterhin eine Einschlussklasse ist, aber dem ungünstigeren Typ beider Intervalle angehört und im schlimmsten Fall sogar den Übergang $\boldsymbol{R} + \boldsymbol{L} \rightarrow \boldsymbol{O}$ zur Folge hat. Auch diese Übergänge werden mittels der ∞-Rechenregeln automatisch abgewickelt.

Ist ein Ausschussintervall involviert, so verschiebt sich die Untergrenze des Ausschlussintervalls um die Obergrenze des anderen Inveralls, während die Obergrenze um die Untergrenze des anderen Intervalls verschoben wird:

$$\left[\overline{a, x, b}\right] + \left[c, y, d\right] = \left[\overline{a + d, x + y, b + c}\right]$$

Von $-\infty$ kommend kann x maximal den Wert a annehmen, zu dem im ungünstigsten Fall d addiert wird, womit man die linke Intervallgrenze erhält. Ähnlich argumentiert man bei der rechten Grenze. Mit der Ausnahme

$$\boldsymbol{A} + \boldsymbol{I} = \boldsymbol{A}\colon\ (a + d) < (b + c)$$

erhält man damit immer den Übergang $\boldsymbol{A} + \boldsymbol{X} \rightarrow \boldsymbol{O}$.

16.4.4.3 Inversion

Die Grundformel für die Inversion ist ähnlich simpel wie für die Negation.

$$[a, x, b] \rightarrow [1/b, 1/x, 1/a]$$

Wir haben einige Übergänge bereits oben diskutiert. Einschlussintervalle einschließlich einseitiger Intervalle, bei denen beide Grenzen gleiche Vorzeichen aufweisen, bleiben Einschlussintervalle, das offene Intervall bleibt ein offenes Intervall.

$$sign(a) = sign(b) \Rightarrow \boldsymbol{X} \rightarrow \boldsymbol{Y},\ \boldsymbol{X}, \boldsymbol{Y} \in \{\boldsymbol{I}, \boldsymbol{L}, \boldsymbol{R}\}$$
$$\boldsymbol{O} \rightarrow \boldsymbol{O}$$

Unterscheiden sich die Vorzeichen der Grenzen, entstehen Ausschlussintervalle. Ausschlussintervalle mit Grenzen unterschiedlicher Vorzeichen verhalten sich entgegengesetzt, dass heisst, es entstehen wieder Einschlussintervalle

$$sign(a) \neq sign(b) \Rightarrow \boldsymbol{X} \rightarrow \boldsymbol{A},\ \boldsymbol{A} \rightarrow \boldsymbol{X},\ \boldsymbol{X} \in \{\boldsymbol{I}, \boldsymbol{L}, \boldsymbol{R}\}$$

Für Ausschlussintervalle mit vorzeichengleichen Grenzen leitet man aus der Grundformel ab, dass sich der Charakter nicht ändert, das Intervall also ein Ausschlussintervall bleibt.

$$sign(a) = sign(b)\ \ \Rightarrow\ \ \boldsymbol{A} \rightarrow \boldsymbol{A}$$

16.4.4.4 Multiplikation

Wegen $a/b = a * (1/b)$ genügt die Untersuchung der Multiplikation; eine separate Untersuchung der Division erübrigt sich. Jedoch existiert bei der Multiplikation keine so einfach Grundformel wie bei den anderen Operationen, sondern es sind mehrere Fallunterscheidungen notwendig. Behandeln wir zunächst die Einschlussintervalle $[a, b] * [c, d]$. Hier sind bereits mehrere Fälle zu unterscheiden:

(1) $0 \leq sign(a) \ \wedge \ 0 \leq sign(c) \ \Rightarrow \ [a * c, b * d]$
(2) $sign(b) \leq 0 \ \wedge \ sign(d) \leq 0 \ \Rightarrow \ [b * d, a * c]$
(3) $sign(a) \neq sign(b) \ \wedge \ sign(c) \neq sign(d) \ \Rightarrow$
$[min(a * d, b * c), max(a * c, d * d)]$
(4) $0 \leq sign(a) \ \wedge \ sign(c) \neq sign(d) \ \Rightarrow \ [b * c, b * d]$
(5) $sign(b) \leq 0 \ \wedge \ sign(c) \neq sign(d) \ \Rightarrow \ [a * d, a * c]$

Zu diesen Regel gelangt man, wenn man nach den kleinsten und größten Termen unter den Produkten

$$a * c, a * d, b * c, b * d$$

unter Berücksichtigung des Vorzeichens sucht. Im Fall (1) ist das trivial, im Fall (2) ist zu berücksichtigen, dass sich die Vorzeichen umkehren, im Fall (3) sind jeweils sogar zwei Produkte gleichen Vorzeichens zu untersuchen, und in den Fällen (4) und (5) wirkt aufgrund der verschiedenen Vorzeichen des Ergebnisses nur eine Grenze des einen Intervalls mit.

Wie bei der Addition ist das Ergebnisintervall jeweils mindestens vom ungünstigeren Typ der beiden Eingangsintervalle. Eine weitere Klassifizierung ist in der Praxis allerdings nicht notwendig.

Beim Aufeinandertreffen von Ausschlussintervallen können ebenfalls zwei Fälle eintreten

(6) Die Multiplikation zweier Ausschlussintervalle, von denen die Lücke mindestens eines der beiden die Null nicht überschreitet, führt zu einem offenen Intervall:

$$0 \leq a \ \vee \ v \leq 0 \quad \Rightarrow \overline{[a, x, b]} * \overline{[c, y, d]} = \left[-\infty, x * y, \infty\right]$$

Ist die Lücke zum Beispiel positiv, so wird die gesamte negative Halbachse abgedeckt. Gleichgültig, wie die Lücke im zweiten Intervall beschaffen ist, es lassen sich immer Produkte bilden, die innerhalb der ursprünglich ausgeschlossenen Bereiche liegen

(7) Schließen beide Intervalle die Null ein, tritt ein zu (3) ähnlicher Fall ein. Werte im Bereich der Null sind in diesem Fall nicht erreichbar, so dass das Ergebnis weiterhin ein Ausschlussintervall ist.

$$\begin{array}{c} a < 0 \ \wedge \ 0 < b \ \wedge c < 0 \ \wedge \ 0 < d \quad \Rightarrow \\ \overline{[a,b]} * \overline{[c,d]} = \overline{[max\,(a*d, b, *c), min\,(a*c, b*d)]} \end{array}$$

Zu beachten ist gegenüber (3), dass Minimum und Maximum hier die Rollen tauschen.

Abschließend bleibt noch zu untersuchen, was beim Aufeinandertreffen von Ausschluss- und Einschlussintervallen geschieht. Bei der Multiplikation $[a,b]*\overline{[c,d]}$ kann das Ergebnis nur wieder ein Ausschlussintervall oder ein offenes Intervall sein. Die Grenzen des Ausschlussintervalls werden durch die Multiplikation jeweils zu

$$\overline{[c,d]} \rightarrow \overline{[max\,(c*a, c*b)\,, min\,(d*a, d*b)]}$$

verschoben, d.h. es entsteht bei

$$max\,(c*a,\ c*b) \geq min\,(d*a,\ d*b)$$

ein offenes Intervall. Man kann zwar nun folgern, dass ein Ausschlussintervall, das die Null ausschließt, und ein Einschlussintervall, welches die Null nicht einschließt, auf jeden Fall zu einem Ausschlussintervall führt, und damit weitere Regeln erzeugen, jedoch ist hier die direkte Prüfung der vier Produkte sinnvoller.

Damit sind nun alle Regeln komplett, und wie sich zeigt, können zumindest der Theorie nach alle Intervallklassen während einer Rechnung tatsächlich erzeugt werden. Zwar ist in den meisten Fällen beim Übergang der reinen Einschlussklasse in eine andere nach einigen weiteren Schritten ein offenes Intervall zu erwarten, das dann im weiteren Verlauf der Rechnung auch nicht mehr verlassen werden kann, jedoch lässt sich beispielsweise für den folgenden Ausdruck mit einem Ausschlussintervall

$$\frac{r_1}{\frac{1}{r_2} + r_3 - r_4},\ r_1..r_4 \in I \quad , \quad r_2 = [a < 0\,, 0 < b]$$

ohne weiteres ein Zahlenbeispiel konstruieren, das als Ergebnis wieder ein nicht unbedingt unbrauchbares Einschlussintervall aufweist.

Aufgabe. Prüfen Sie dies nach und konstruieren Sie ein Beispiel!

16.4.5 Implementierung einer Intervallklasse

Um die Implementation der Intervallklasse möglichst performant zu gestalten, nuzten wir zwei Konzepte:

- Die Rechenregeln für ∞-Symbole werden auch von den hardwaregestützten Datentypen eingehalten, die verbotenen Fälle treten bei korrekter Programmierung nicht auf. Wir können uns also darauf beschränken, zwischen Ein- und Ausschlussintervallen zu unterscheiden und müssen die Subtypen nicht weiter berücksichtigen.
- Die Intervallrechnung macht im Grunde nur dann Sinn, wenn am Schluss der Rechnung echte Einschlussintervalle vorliegen. Wir optimieren in dieser Richtung und führen die Rechenoperationen für Einschlussintervalle als inline-Funktionen aus. Bei offenen und halboffenen Intervallen werden dabei einige Rechnungen zu viel ausgeführt, jedoch wäre der Aufwand für die zusätzlichen Vergleiche und Klassifizierungen im korrekten Einsatz nicht vertretbar.
- Um die Größe der inline-Funktionen zu begrenzen, werden die Operationen für Ausschlussintervalle in normale Funktionen verlagert.

Die Intervallklasse wird daher zunächst mit folgendem Datengerüst ausgestattet:

```
class Intervall {
public:
    ...
    typedef Intervall      value_type;
    typedef Intervall*     pointer;
    typedef Intervall&     reference;
private:
    double lower,upper,standard;
    int count;
    bool inclusiv;
};//end class
```

In `count` wird die Anzahl der Operationen mitgezählt, die an der Berechnung des Intervalls beteiligt waren, `inclusiv` unterscheidet zwischen Einschluss- (`true`) und Ausschlussintervallen (`false`).

Die Implementation der Rechenoperationen führen wir exemplarisch am Beispiel der Multiplikation vor. Berücksichtigt werden bei Einschlussintervallen die Fälle

```
(01)     [+,+]*[+,+] :   [l1*l2,u1*u2]
(02)     [-,-]*[-,-] :   [u1*u2,l1*l2]
(03)     [-,-]*[+,+] :   [l1*u2,l2*u1]
(04)     [+,+]*[-,-] :   [l2*u1,l1*u2]
(05)     [-,+]*[-,-] :   [u1*l2,l1*l2]
(06)     [-,+]*[-,+] :
           [min(u1*l2,l1*u2),max(l1*l2,u1*u2)]
(07)     [-,+]*[+,+] :   [l1*u2,u1*u2]
(08)     [-,-]*[-,+] :   [l1*u2,l1*l2]
(09)     [+,+]*[-,+] :   [u1*l2,u1*u2]
```

Die Zuordnung der Werte zu den jeweiligen Typen erfolgt durch die inline-Funktionen

```
void Intervall::c1(double const& l1,
                   double const& l2,
                   double const& u1,
                   double const& u2){
    double dl,du;
    fesetround(FE_DOWNWARD);
    dl=l1*l2;
    fesetround(FE_UPWARD);
    du=u1*u2;
    lower=dl;
    upper=du;
}//end function

Intervall& Intervall::operator*=(
                        Intervall const& di){
    standard*=di.standard;
    count+=di.count;
    if(!inclusiv || !di.inclusiv){
        mul_excl(di);
        return *this;
    }//endif        standard*=di.standard;
    if(0<=lower){
        if(0<=di.lower){        // (01)
            c1(lower,di.lower,upper,di.upper);
        }else if(di.upper<=0){  // (04)
            c1(upper,di.lower,lower,di.upper);
        }else{                  // (09)
            c1(upper,di.lower,upper,di.upper);
        }//endif
    }else if(0<=upper){
        if(0<=di.lower){        // (07)
            c1(lower,di.upper,upper,di.upper);
        }else if(di.upper<=0){  // (05)
            c1(upper,di.lower,lower,di.lower);
        }else{                  // (06)
            double d1,d2;
            fesetround(FE_DOWNWARD);
            d1=min(upper*di.lower,
                   lower*di.upper);
            fesetround(FE_UPWARD);
            d2=max(lower*di.lower,
                   upper*di.upper);
            lower=d1;
            upper=d2;
        }//endif
    }else{
```

```
        if(0<=di.lower){        // (03)
            c1(lower,di.upper,upper,di.lower);
        }else if(di.upper<=0){  // (02)
            c1(upper,di.upper,lower,di.lower);
        }else{                  // (08)
            c1(lower,di.upper,lower,di.lower);
        }//endif
    }//endif
    fesetround(FE_TONEAREST);
    return *this;
}//end function
```

Die jeweilige Rundungsmethode wird mittels der Systemfunktion `fesetround` eingestellt. Bei Verlassen der Methode ist die normale Rundungsmethode eingestellt, so dass Intervallzahlen mit Standard-`double`-Rechnungen gemischt werden können. Im Durchschnitt werden 3 Vergleiche (2-4) und 3 Rechenschritte benötigt. Die an der Entstehung eines Intervalls beteiligten Rechenschritte ergeben sich aus der Summe der Rechenschritte der beiden beteiligten Intervallzahlen. Die Initialisierung erfolgt mit dem Wert Eins, die Verrechnung mit einer normalen `double`-Zahl erhöht die Anzahl der Rechenschritte ebenfalls um Eins.

Die Berücksichtigung der Ausschlussintervalle erfolgt in der externen Funktion

```
double const INFINIT = 1.0/0.0 ;
void Intervall::mul_excl(Intervall const& di){
    double r1,r2;
    if(inclusiv){
        fesetround(FE_UPWARD);
        r1=max(lower*di.lower,upper*di.lower);
        fesetround(FE_DOWNWARD);
        r2=min(lower*di.upper,upper*di.upper);
    }else if(di.inclusiv){
        fesetround(FE_UPWARD);
        r1=max(lower*di.lower,lower*di.upper);
        fesetround(FE_DOWNWARD);
        r2=min(upper*di.lower,upper*di.upper);
    }else{
        fesetround(FE_UPWARD);
        r1=max(lower*di.upper,upper*di.lower);
        fesetround(FE_DOWNWARD);
        r2=min(lower*di.lower,upper*di.upper);
    }//endif
    if(r1<r2){
        lower=r1;
        upper=r2;
        inclusiv=false;
```

```
    }else{
        lower=-INFINIT;
        upper=INFINIT;
        inclusiv=true;
    }//endif
    fesetround(FE_TONEAREST);
}//end function
```

Hierbei haben wir die Regeln für die Multiplikation zwischen Ein- und Ausschlussintervallen und zwischen zwei Ausschlussintervallen zusammengefasst.

Ähnlich können wir die Division als inline-Funktion implementieren, wobei hier die Definition einer externen Funktion nicht notwendig wird:

```
Intervall& Intervall::operator/=(
                        Intervall const& i){
    bool diff_vz;
    Intervall d(1.0);
    d.standard/=i.standard;
    diff_vz=i.lower<=0 && 0<=i.upper;
    fesetround(FE_DOWNWARD);
    if(diff_vz){
        d.lower/=i.lower;
        fesetround(FE_UPWARD);
        d.upper/=i.upper;
    }else{
        d.lower/=i.upper;
        fesetround(FE_UPWARD);
        d.upper/=i.lower;
    }//endif
    fesetround(FE_TONEAREST);
    d.count=i.count;
    if(!isfinite(i.lower) && !isfinite(i.upper)){
        d.lower=-INFINIT;
        d.upper=INFINIT;
        d.inclusiv=true;
    }else{
        d.inclusiv=i.inclusiv ^ diff_vz;
    }//endif
    return *this*=d;
}//end function
```

Bei der Berechnung der neuen Grenzen sind die Vorzeichenunterschiede der alten zu beachten. Die Vorzeichenunterschiede können aufgrund der Symmetrie bei der Umwandlung von Ausschluss- in Einschlussintervalle und umgekehrt ebenfalls recht einfach durch ein exklusives Oder genutzt werden.

Aufgabe. Prüfen Sie die Umsetzung der Rechenregeln in den beiden Operationen. Implementieren Sie anschließend die weiteren Standardoperatoren für Addition, Subtraktion und Negierung gemäß den ermittelten Regeln. Hierbei werden die Rechenoperationen mit Ausschlussintervallen zweckmäßigerweise wieder in externe Funktionen ausgelagert. Implementieren Sie zusätzlich Methoden, die das Mischen von Intervallzahlen mit normalen `double`-Werten erlauben.

Eine `to_string`-Funktion gibt die Intervallzahl komplett mit beiden Grenzen und der Anzahl der Rechenschritte aus, so dass eine Interpretation des Ergebnisses möglich ist.

```
template <>
string to_string(Intervall const& a){
    string s;
    s=string("(") + to_string(a.low()) + ","
                  + to_string(a.std())+ ","
                  + to_string(a.high())+ ",["
                  + to_string(a.ops()) + ","
                  + to_string(a.include()) + "])" ;
    return s;
}//end function
```

Aufgabe. Ergänzen Sie die Intervallklasse um die notwendigen Methoden und implementieren Sie auch eine Methode `from_string(..)`. Je nach Aufbau Ihrer Implementation müssen Sie gegebenenfalls mehrfach zwischen verschiedenen Namensbereichen wechseln, falls der Compiler bei der Übersetzung nicht alles auflösen kann.

Für Debug-Zwecke beim Test der Klasse und für spätere einfache Typunterscheidungen ist eine Methode für die genaue Intervalltypfeststellung hilfreich:

```
enum IType { incl=1,
             excl=2,
             left=4,
             right=8,
             open=16,
             error=32 };

Intervall::IType  Intervall::itype() const{
    if(!inclusiv){
        if(!isfinite(lower) && !isfinite(upper))
            return error;
        if(!isfinite(lower) && standard<upper)
            return error;
        if(lower<standard && !isfinite(upper))
            return error;
```

```
        if(lower<standard && standard<upper)
             return error;
        return excl;
    }//endif
    if(!isfinite(lower) && !isfinite(upper))
             return open;
    if(!isfinite(lower))
        if(standard<=upper) return left;
        else                return error;
    if(!isfinite(upper))
        if(lower<=standard) return right;
        else                return error;
    if(lower<=standard && standard<=upper)
              return incl;
    else      return error;
}//end function
```

Recht einfach fällt auch die Implementation der Vergleichsklasse `compare` aus, da wir uns vereinbarungsgemäß auf den Vergleich der Standardwerte beschränken.

```
template <> struct compare<Intervall> {
    compare():cmp(){}

    inline void load(const Intervall& s)
                { cmp.load(s.std());}

    inline bool equal(const Intervall& s,
                      const Intervall& t) const {
        return cmp.equal(s.std(),t.std());
    }//end function

    inline bool less(const Intervall& s,
                     const Intervall& t) const {
        return (s.std()<t.std()) && !equal(s,t);
    }//end function

    inline bool zero(const Intervall& s) const {
        return cmp.zero(s.std());
    }//end function

private:
    compare<double> cmp;
};//end struct
```

Damit ist trotz der sehr aufwändigen Voruntersuchung die Implementation der Intervallklasse bereits abgeschlossen (*gegebenenfalls weitere Spezialisierungen allgemeiner Template-Klassen können Sie nach Bedarf vornehmen*).

Aufgabe. Vergessen Sie nicht, ein Testprogramm zu schreiben, dass alle Klassenkombinationen und Operationen abdeckt und das Ergebnis prüft. Die Prüfung kann mit der Methode `itype()` erfolgen, die extra aus diesem Grund auch den Zustand `error` besitzt, die Prüffälle können mit ein wenig Überlegung auch mittels Zufallzahlengeneratoren erzeugt werden, was aufgrund der Vielzahl der Fälle Schreibarbeit und Fehler vermeidet.

16.4.6 Einsatz der Intervallrechnung

Ein Einsatz der Intervallrechnung ist unter zwei Randbedingungen kritisch:

(a) Rekursionen auf einem Wert mit hohen Wiederholungsanzahlen. Hierbei kann es passieren, dass die Anzahl der Operationen und damit die Anzahl der systematischen Rundungen so hoch wird, dass das resultierende Intervall in keinem Bezug mehr zum wahrscheinlichen Fehler steht.
(b) Algorithmen mit Iterationswerten in der Gegend von Null. In solchen Algorithmen kommt es fast automatisch zu Einschlussintervallen, die die Null einschließen, und ist anschließend eine Division enthalten, haben wir es mit offenen Intervallen zu tun. Im weiteren ist es dann meist nur noch ein kurzer Schritt bis zu einem offenen Intervall.

Ein unkritischer Algorithmus sollte unter diesen Randbedingungen beispielsweise die Berechnung von quadratischen Wurzeln reeller Zahlen sein, der folgender Rekursion genügt:

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{c}{x_k}\right)$$

Der Zielwert schließt die Null nicht ein, und der Algorithmus konvergiert bekanntermaßen quadratisch, so dass wenige Schritte genügen. In der Tat finden wir im Testlauf die Werte

```
sqrt(2)=(1.4142135623730911e+000,
         1.4142135623730947e+000,
         1.4142135623730994e+000,[485,true])
x^2=(1.9999999999999889e+000,
     1.9999999999999991e+000,
     2.0000000000000124e+000,[970,true])
2-x^2=(-1.1102230246251565e-014,
       -8.8817841970012523e-016,
        1.2434497875801753e-014,[971,true])
```

Die Werte bestätigen einige Vermutungen, die man über den Verlauf einer Intervallrechnung anstellen kann:

- Die Intervallzahl nimmt zweimal an jedem Durchgang teil, so dass sich der Zahl der formal durchgeführten Operationen in jedem Schritt verdoppelt. Trotz der schnellen Konvergenz ist die Zahl der Operationen mit 485 recht hoch.
- Der Algorithmus ist numerisch stabil. Die Intervallbreite sollte daher bei $2 * n * \epsilon$ liegen, was durch das dritte Werttripel vollauf bestätigt wird.
- Bei einer neu generierten Intervallzahl sollte statistisch gesehen jeder Wert im Intervall gleich wahrscheinlich sein (*Rechteckverteilung*). Die fortwährende Überlagerung von Größen, die einer Rechteckverteilung folgen, führt zu einer Gaussverteilung des resultierenden Wertes. Der normal berechnet Wert sollte also etwa in der Mitte des Intervalls liegen (*was die Daten bestätigen*), der wahre Wert relativ nahe daran.

Eine Summarion von 1.000.000 Werten im Intervall $(-1,1)$ führt zu dem Ergebnis

```
s=(8.7800805669181784e+001,
   8.7800805688651892e+001,
   8.7800805708141908e+001,[1000001,true])

high-low        =3.89601e-008
std-low         =1.94701e-008
(high-low)/std  =4.43733e-010
```

Auch hier liegt der normal berechnet Wert ziemlich genau im Zentrum des Ergebnisintervalls, von dem immerhin noch 9 Stellen vertrauenswürdig sind. Allerdings ist es mit dem Wert „88“ auch recht weit von der Null entfernt. Läge es näher daran, wären entsprechend viele Stellen von fraglicher Qualität.

Als Beispiel dafür, dass eine Intervallrechnung für die Interpretation eines Ergebnisses nicht ausreicht, sondern auch die anderen in den ersten Teilkapiteln untersuchten Kontrollmöglichkeiten berücksichtigt werden müssen, betrachten wir die Lösung eines linearen Gleichungssystems mit zwei fast kolinearen Zeilenvektoren[12] mit der Lösung

```
x=(-2.6309076534946140e+001,
   -2.6309076534393647e+001,
   -2.6309076533841399e+001,[5038,true])
  ( 2.0437174133303878e+002,
    2.0437174133322557e+002,
    2.0437174133341250e+002,[2526,true])
  (-8.9820879195042806e+001,
   -8.9820879195001183e+001,
   -8.9820879194959517e+001,[1285,true])
  (-1.4164949978303792e+002,
```

[12] Zufallzahlen im Intervall (–1,1) in einer 5×5-Matrix. Die 2. und 3. Zeile der Matrix sind identisch bis auf ein Element, dass sich um den Faktor 1,1 vom korrespondierenden der anderen Zeile unterscheidet.

```
   -1.4164949978299600e+002,
   -1.4164949978295408e+002,[707,true])
  (-3.2272995055171748e+001,
   -3.2272995055161381e+001,
   -3.2272995055150979e+001,[514,true])
```

Rein formal scheinen die Werte durchaus vertrauenswürdig zu sein, da auch im ungünstigsten Fall von 9–10 Stellen exakt sind. Wenn wir jedoch eine Probe machen und das Residuum, also $\vec{b} - A * \vec{x}$ berechnen, erhalten wir

```
Residuum=(-2.1823809426280150e-010,
           8.6597395920762210e-015,
           2.1823409745991285e-010,[10077,true])
         (-2.9937891055098476e-010,
           5.4456439357863928e-014,
           2.9934571488254846e-010,[10077,true])
         ( 1.5104182008557636e+000,
           1.5104182011558045e+000,
           1.5104182014556999e+000,[10078,true])
         (-1.5104182013795673e+000,
          -1.5104182011556999e+000,
          -1.5104182009319820e+000,[10077,true])
         (-5.1449644544732109e-010,
          -5.9952043329758453e-014,
           5.1453641347620760e-010,[10077,true])
```

Während die Komponenten 1, 2 und 5 durch die Lösung korrekt wiedergegeben werden, ist bei den Komponenten 3 und 4 nicht eine Stelle, das Vorzeichen eingeschlossen, mit der Vorgabe identisch. Woran es nun im Einzelnen liegt, dass die Intervallrechnung hier mehr oder weniger versagt hat (*grundsätzliche Nichteignung des Algorithmus, falsche Konstruktion, oder ...*), wollen wir hier nicht weiter ergründen.

Betrachten wir als abschließendes Beispiel die Nullstellenberechnung eines Polynoms. Als Polynom wählen wir

$$P(x) = (x - a)^3(x - b)^2$$

mit

```
b = 1.5497299111911375e-001
a = 1.4848170415356914e+000
```

Der Algorithmus besteht aus der Division des Polynoms durch den ggT des Polynoms mit seiner Ableitung (*hierdurch wird aus einer mehrfachen Nullstelle eine*

einfache Nullstelle gemacht und der Algorithmus dadurch numerisch stabilisiert) und einer anschließenden Newton-Iteration. Als Ergebnis erhält man

```
1. NS:(1.5497299111769644e-001,
       1.5497299111911436e-001,
       1.5497299112053109e-001,[316796,true])
2. NS:(1.4848170415307300e+000,
       1.4848170415356965e+000,
       1.4848170415406661e+000,[1270908,true])
```

Trotz der großen Anzahl an Rechenoperationen besitzt das Ergebnis noch hinreichend viele genaue Stellen, und auch die Probe fällt in diesem Fall günstig aus:

```
P(x1)=(-2.7598149460184018e-013,
       -5.2041704279304213e-018,
        2.7597802515488823e-013,[1584150,true])
P(x2)=(-1.1616469331593438e-010,
        1.1449174941446927e-015,
        1.1616942217212989e-010,[6354710,true])
```

Allerdings fällt das Ergebnis bei anderer Wahl der Nullstellen deutlich anders aus:

```
b =        5.0887539292580954e+001
a =        7.0667958616901146e+001
x1 =     ( 5.0887537085767917e+001,
           5.0887539091667612e+001,
           5.0887541097570960e+001,[1270908,true])
x2 =     ( 7.0667946164411447e+001,
           7.0667958616928061e+001,
           7.0667971069437868e+001,[5087356,true])
P(x1) = (-3.0700235736370087e+002,
         -5.9604644775390625e-007,
          3.0700236940383911e+002,[6354710,true])
P(x2) = (-3.1610643427371979e+003,
         -9.5367431640625000e-007,
          3.1610648975372314e+003,[25436950,true])
```

Woran liegt das und wie kann man an ein brauchbares Ergebnis kommen? Aufgrund des hohen Polynomgrades unterscheiden sich die Koeffizienten des Polynoms sehr stark und schwanken im Bereich $(1..10^8)$. Eine Rechnung mit den gleichen Einstellungen wie im ersten Fall liefert Ergebnisse mit offenen Intervallen, da der ggT des Polynoms mit seiner Ableitung nicht mehr ermittelt werden kann (*er wird als numerische Konstante bestimmt*) und die nachfolgende Iteration der Nullstellen drei- bzw. zweifache Nullstellen iteriert, was numerisch instabil ist. Zumindest diese numerische Instabilität zeigt die Intervallrechnung einwandfrei an.

Um überhaupt numerisch stabil rechnen zu können, muss der freie Verfahrensparameter `cmp_eps_double()` von 10^{-12} auf 10^{-8} erniedrigt werden. Statt den Algorithmus akribisch zu untersuchen und irgendwann das Versagen des ggT-Algorithmus festzustellen, kann man auch einfach an freien Parametern eines Verfahrens „drehen“ und am Ergebnis der Intervallrechnung ablesen, ob eine Verbesserung eintritt. Das ist hier der Fall und führt zunächst dazu, dass bei den Nullstellen nunmehr 6 Stellen wirklich vertrauenswürdig sind. Macht man nun eine Probe, so liegt der zentrale Wert zwar immer noch recht nahe an der Null, allerdings liegen die Grenzen extrem weit auseinander, weil der Term x^5 mit $x \approx 60$ schon bei geringen Änderungen sehr empfindlich reagiert.

Damit möchte ich die Anzahl der Beispiele beschließen. Halten wir fest: Numerische Instabilitäten von Algorithmen können dazu führen, dass von einem berechneten Ergebnis nicht eine einzige von 16 Stellen beim Datentyp `double` mit dem wahren Wert übereinstimmt. Eine Kontrolle ist also in vielen Fällen unumgänglich. Für Kontrollzwecke stehen eine Reihe verschiedener Methoden zur Verfügung, von denen aber in der Regel keine alleine als Universalwerkzeug genügt; statt dessen muss man im Allgemeinen algorithmusabhängig verschiedene Methoden kombinieren, um die Problemfälle zu ermitteln. Sofern freie Parameter in den Algorithmen vorhanden sind – im Normalfall sind das die Entscheidungsgrenzen für Gleichheit oder Ungleichheit – kann man auch durch deren Manipulation versuchen, das Ergebnis zu verbessern, ohne sofort eine sehr tiefschürfende Analyse durchzuführen. Hat man Problemfälle erkannt, empfiehlt sich natürlich die Suche nach Hilfsalgorithmen, die rechtzeitig Aufklärung verschaffen, ob und mit welchen Parametern ein Algorithmus für die Ermittlung einer Lösung eingesetzt werden kann. Mathematische Programmierung bleibt also ein sehr anstrengendes Geschäft.