Algorithmen, Rechnen und Computer

Horst Hollatz
horst@hollatz.de

Urania-Sonntagsvorlesung der TU Magdeburg im Mai 1988

Zusammenfassung

Es werden einige Probleme dargestellt, die bei der Nutzung von Rechnern zum Lösen von Aufgaben auftreten. Unter diesen gibt es zunächst solche, die auf keinem Rechner lösbar sind. Die Nichtlösbarkeit resultiert aus der Nicht-Algorithmierbarkeit der Aufgabe. Die meisten den Menschen und von ihnen gestellten Aufgaben sind nicht algorithmierbar. Andererseits gibt es auch solche Aufgaben, deren berechnete Lösung mit der wahren Lösung nicht einmal näherungsweise übereinstimmt. An Beispielen wird den Ursachen für das Rechner-Versagen bzw. für das fehlerhafte Ergebnis nachgegangen.

Jeder, der schon einmal mit einem Rechner zu tun hatte, weiß, dass man dem Rechner einen Algorithmus übergeben muss, damit er eine Aufgabe lösen kann. Algorithmen sind grundlegend für das Lösen von Aufgaben mittels Rechnern; ohne Algorithmen keine Rechnernutzung. Aber ohne Rechner gibt es unzählige Algorithmen, die wir fast unbewusst ständig anwenden: Es gibt z.B. Algorithmen, um Kuchen zu backen (nämlich in Form von Rezepten), um Sonaten zu spielen (in Form von Noten), um Autos zu bauen, um Handschuhe zu stricken usw. Diese Algorithmen unterscheiden sich aber grundlegend von jenen, die wir auf dem Rechner benutzen. Die Lebensalgorithmen arbeiten mit unscharfen Eingabedaten und die Ergebnisse, die als Lösungen der zugrunde liegenden Aufgabe akzeptiert werden, streuen stark. So werden nach dem gleichen Rezept viele verschiedene Kuchen gebacken, Sonaten werden verschieden interpretiert, obwohl die gleichen Noten vorliegen und es werden Autos gebaut, die von manchen Leuten nicht als solche angesehen sind. Genau genommen, führt jeder neue Ablauf eines solchen Algorithmus zu einer bisher noch nicht aufgetretenen Lösung; insbesondere auch darum, weil sich die Eingaben bei jeder Wiederholung ändern und die Randbedingungen, unter denen der Algorithmus abläuft, bei jeder Wiederholung etwas anders sind. So ist ein Musiker für jedes Konzert anders aufgelegt, hat gegebenenfallls sein Instrument etwas anders gestimmt, das Publikum wird bei jedem Auftritt auch anders (vielleicht nur ein wenig anders) reagieren und so die Interpretation des Musikstückes mehr oder weniger beeinflussen.
Dass die Lösungen von aus dem Leben gegriffenen Aufgaben stark streuen, hat noch andere Gründe. Obwohl die Lebensalgorithmen hinreichend genaue Vorschriften darstellen, müssen doch die in ihnen auftretenden Formulierungen von den ausführenden Menschen interpretiert, in eine Folge für sie ausführbarer Anweisungen übersetzt werden. Die Satzbedeutungen schwanken aber von Mensch zu Mensch und von Land zu Land, weil die

Weltbilder der Menschen entsprechend den zeitlichen, örtlichen, wirtschaftlichen und sozialen Gegebenheiten stark differieren; sogar momentane psychische oder physische Gegebenheiten haben einen oft zeitlich begrenzten Einfluss. Wir sind sicher froh darüber, dass die Menschen verschieden und damit die Lebensaufgaben, die verwendeten Algorithmen zu ihrem Lösen und die Interpretation der Algorithmen hinreichend unscharf bzw. verschieden sind. Die Stabilität einer Gesellschaft hängt wohl entscheidend davon ab, wie es ihr gelingt, die Verschiedenheit ihrer Landeskinder, die sich über ihre Weltbilder insbesondere in der Verschiedenheit ihrer sozialen und wirtschaftlichen Ziele äußert, einerseits zu fördern und andererseits zum Wohle des Ganzen zu nutzen.

Die Aufgaben, die wir den Rechnern zum Lösen anvertrauen, dürfen nach unserem Verständnis keine solche Unschärfe in der Formulierung haben, denn wir streben i. a. eine solche Formulierung an, dass die Aufgabe eindeutig lösbar wird und die verwendeten Algorithmen die Lösung der Aufgabe zu finden haben. Dies unterstellt einen viel schärferen Lösungsbegriff als im normalen Leben. Wir wissen, dass Rechner Ampeln und Stahlwerke steuern, sie reservieren Flüge, steuern Roboter, fertigen Lohnabrechnungen an u.a.m. Aus unserer Schulzeit erinnern wir uns an Algorithmen für die Addition, Subtraktion, Multiplikation und Division von Zahlen, oder an das Wurzelziehen. Bei der ungeheuren Vielfalt von Algorithmen erhebt sich schon fast unberechtigt die Frage: Gibt es eine Aufgabe, die ein noch so großer oder noch so raffiniert konstruierter Rechner nicht erfüllen kann? Anders gefragt: Gibt es überhaupt irgendeine Aufgabe, für die man im Augenblick oder vielleicht sogar niemals einen Algorithmus angeben kann, der diese Aufgabe löst? Die Antwort auf diese Frage lautet überraschenderweise: Ja! Es gibt zahllose Dinge, die ein Rechner nicht kann und niemals können wird. Die Zahl der Aufgabenstellungen, die nicht mit dem Rechner gelöst werden können, ist unendlich viel größer als die mit dem Rechner lösbaren. Rechner können die meisten Dinge nicht. Und das liegt daran: Die meisten Aufgaben sind nicht algorithmierbar! Dabei brauchen wir nicht einmal solche Aufgaben in unsere Überlegungen einzubeziehen, die mit Gefühlen oder anderen subjektiven Aspekten von Menschen zusammenhängen. Die Grenzen der Leistungsfähigkeit von Rechnern liegen in ihren eigenen Wurzeln, so wie wir es bei jeder Einzelwissenschaft beobachten können. Die Idee, einen Algorithmus oder ein Rezept zum Lösen irgendeiner Aufgabe zu haben, existiert schon seit tausenden von Jahren. Man glaubte stets: Wenn irgendeine Aufgabe genau formuliert ist, dann findet man am Ende – hinreichend langes Bemühen und intensive Arbeit vorausgesetzt – auch stets eine Lösung oder es kann bewiesen werden, dass keine Lösung existiert. Anders gesagt, man glaubte: Zu jeder Aufgabe gibt es auch einen Algorithmus, der diese Aufgabe löst. Selbst einer der größten Mathematiker des 20. Jahrhunderts – David HILBERT (1862-1943) – war ein großer Anhänger dieses Glaubens. Hilberts Ziel war es, ein mathematisches System zu ersinnen, in dem alle Aufgaben präzise als Aussagen formulierbar sind und algorithmisch entschieden werden kann, ob sie wahr oder falsch sind. Das erste berühmte Resultat hierzu erschien im Jahre 1931, als der damals 25-jährige Kurt GÖDEL zeigte, dass es keinen Algorithmus gibt, der für **jede** Aussage über natürlichen Zahlen herausfindet, ob sie wahr ist oder nicht. Danach fand man weitere Aufgaben, die nicht algorithmisch lösbar sind. Natürlich hatte man den Verdacht, dass dieses Dilemma an unserer Vorstellung von einem Algorithmus liegt; doch es zeigte sich bald, dass alle neuen Definitionen eines Algorithmus äquivalent sind. Die Ergebnisse über Aufgaben, die nicht mit dem Rechner lösbar sind, wurden bereits gefunden, bevor

die ersten Rechner im heutigen Sinne gebaut waren!
Wir wollen nun eine einfache Aufgabe betrachten, die nicht algorithmisch lösbar ist. In der ganzen Welt weiß man, dass Programmierer beim Programmschreiben Fehler machen, insbesonder solche, die ein Programmende verhindern; man sagt, das Programm schleift endlos. Es wäre schön, das Vorhandensein von Endlosschleifen festzustellen, bevor die Ausführung eines fehlerhaften Programms Rechenzeit verschwendet. Unsere Aufgabe lautet damit: Ist für ein beliebiges Programm bestimmbar, ob es eine Endlosschleife enthält? Oder anders: Gibt es einen Algorithmus, der zu jedem gegebenen Programm und seinen Eingabedaten uns mitteilt, ob es jemals hält, falls es mit den Eingabedaten ausgeführt wird? Nehmen wir für einen Augenblick an, es gibt einen solchen Algorithmus; wir nennen ihn STOPP–T; er hätte dann zwei Eingaben P,E und gibt die Antwort 'ja' aus, falls Programm P mit E als Eingabe halten würde, sonst 'nein'. Da STOPP–T die Beendigung eines Programms für alle Daten prüft, dürfen wir es selbst als Eingabe verwenden und erhalten so den Algorithmus STOPP–T–neu mit der Eingabe P. Daraus bauen wir einen neuen Algorithmus SPAßIG, der nur die Eingabe P hat:

SPAßIG(P):
Falls STOPP–T–neu(P) 'nein' ausgibt, so stoppe, sonst schleife endlos.

Was passiert nun, wenn man SPAßIG(SPAßIG) ausführt? Wenn SPAßIG(SPAßIG) stoppt, bildet der Algorithmus eine Endlosschleife; wenn aber SPAßIG(SPAßIG) eine Endlosschleife bildet, so stoppt er. Die Ausführung von SPAßIG(SPAßIG) kann weder stoppen noch endlos schleifen. Dieser Widerspruch kann nur dadurch aufgelöst werden, dass wir die einzige Voraussetzung zur Herleitung von SPAßIG fallenlassen, d. h. die Existenz von STOPP–T. Die Nichtexistenz eines solchen Algorithmus bedeutet aber nicht, dass man für eingeschränkte Fälle keinen solchen Algorithmus finden kann, im Gegenteil, bei solchen Aufgaben ist das Problemlösen mit menschlicher Kreativität verbunden.
Grundlegendes Resultat, aus dem man viele algorithmisch unlösbaren Aufgaben schließen kann, ist folgender Satz von RICE.
Es sei eine Aufgabe A gegeben, die vom Algorithmus P gelöst wird, vom Algorithmus Q aber nicht. Dann existiert kein Algorithmus, der feststellt, ob ein beliebiger Algorithmus die Aufgabe A löst oder nicht.
Im folgenden wollen wir uns nur noch mit Algorithmen und den daraus resultierenden Rechenprogrammen beschäftigen. Es ist bedauerlich, dass die meisten existierenden Rechenprogramme Fehler enthalten, auch solche Programme, die schon viele Jahre tadellos gelaufen sind. Hauptarbeit vieler Programmierer in der ganzen Welt ist es, Fehler in ihren oder anderen Programmen zu suchen, zu finden und zu beseitigen. Der Entwurf von Algorithmen ist oft eine kreative Aufgabe. Die menschliche Unzulänglichkeit beim Verstehen von Algorithmen – die großen und kleinen Missverständnisse – verursachen die meisten Fehler in Rechenprogrammen. Wir haben zwei Möglichkeiten, ein Rechenprogramm auf richtiges Arbeiten hin zu untersuchen:
1. Das Programm wird mit einer bestimmten Menge von Eingaben ausgeführt um festzustellen, welche Ergebnisse es erzeugt.
Damit ist die Wirkung des Programms ausschließlich für die gewählten Eingabedaten aufgezeigt. In nicht untersuchten Fällen können Fehler auftreten, wie wir z. B. von Programmsystemen wissen, die Überwachungsfunktionen auszuführen haben. Es wird niemals

einen Algorithmus geben, der für ein beliebiges Programm solche Eingabedaten erzeugt, dass alle möglichen Programmdurchläufe untersucht werden können, denn diese Aufgabe ist nicht algorithmierbar.
2. Die Korrektheit eines Programms wird für alle zugelassenen Eingabedaten mathematisch bewiesen.
Auch in diesem Falle hat man keine volle Sicherheit, dass keine Fehler vorhanden sind, weil ja der Beweis fehlerhaft sein kann. Die Erfahrung zeigt aber, dass mathematische Beweise in erheblichem Maße das Vertrauen in die Korrektheit eines Programms stärken können. Einen extrem formalen mathematischen Beweis für die korrekte Funktion eines Programms zu erstellen, ist ein aufwendiges Unterfangen und praktisch unmöglich bei großen Programmsystemen; in der Praxis werden Kompromisse geschlossen.
Größte Korrektheitsanforderungen bei Algorithmensystemen gibt es wohl in der Raumfahrt. Mit großem finanziellem und materiellem Aufwand werden die einzelnen Komponenten auf einwandfreie Funktion hin untersucht. Trotzdem verfehlte vor Jahren eine Venussonde der USA um über 50000 km ihr Ziel; viele Millionen Dollar waren verloren, da die erhofften Resultate nicht eintraten. Wie sich später herausstellte, war nicht eine Komponente des Systems ausgefallen oder hatte gar fehlerhaft gearbeitet; der Fehler lag im Zusammenspiel von Komponenten. In einem Programm wurde angenommen, dass eine gewisse Zahl mit 16 Stellen Genauigkeit eingegeben wird, während in Wahrheit nur 7 Stellen übergeben wurden! Diese Kleinigkeit verursachte den Misserfolg des gesamten Unternehmens.
Werden einem Rechner Überwachungsfunktionen übertragen, so wird man hohe Sicherheitsanforderungen sowohl an das Gerät 'Rechner' als auch an die verwendeten Rechenprogramme stellen. Besteht ein solches Überwachungssystem aus z. B. 100 Einzelprogrammen – was sicherlich wenig ist –, so müssten für die Korrektheit des Systems (bei korrekten Einzelprogrammen) im Falle einer fehlerhaften Arbeit, des Ausfalls oder des bewussten Abschaltens einzelner Elemente mindestens 2 hoch 100 Fälle untersucht werden. Wenn man für das Durchspielen nur eines Falles den millionsten Teil einer Sekunde benötigt, so würde das Durchspielen aller Fälle Jahrhunderte dauern. Diese Überlegungen haben weitreichende Auswirkungen auf die globalen Probleme der Menschheit. So wird durch sie z. B. die Notwendigkeit einer umfassenden, allseitigen Abrüstung bewiesen. Die riesigen Algorithmensysteme, wie sie in der Rüstung eingesetzt werden, sind prinzipiell nicht und niemals auf Korrektheit überprüfbar. Diese Tatsache ist auch jenen Wissenschaftlern bekannt, die an der Entwicklung solcher Systeme arbeiten.
Fast noch schlimmer bzw. folgenschwerer als algorithmisch unlösbare Aufgaben und fehlerhafte Rechenprogramme sind solche algorithmisch lösbaren, die, obwohl man den Algorithmus mathematisch exakt in ein Rechenprogramm umgesetzt hat, auf dem Rechner nicht die gewünschten Resultate liefern. Ein erstes **Beispiel** soll diese Behauptung stärken. Wir wollen

$$w = 9x^4 - y^4 + 2y^2$$

für $x = 10864.0$ und $y = 18817.0$ berechnen. Wenn wir die Berechnung nach den folgenden 4 mathematisch gleichwertigen Formeln

$$w_1 = 9 * x * x * x * x - y^4 + 2 * y * y$$

$$w_2 = (3 * x * x - y * y) * (3 * x * x + y * y) + 2 * y * y$$

$$w_3 = 9 * x^4 + (2 * y^2 - y * y * y * y)$$

$$w_4 = (9 * x^4 + 2 * y^2) - y^4$$

ausführen, erhalten wir bei 7-stelliger Rechnung die Werte

$$w_1 = 236052992.0, w_2 = 708158976.0, w_3 = 0.0, w_4 = 0.0$$

und bei 16-stelliger Rechnung die Werte

$$w_1 = -320.0, w_2 = 1.0, w_3 = 160.0, w_4 = -160.0$$

bzw. andere, aber auch stark differierende Werte in Abhängigkeit vom verwendeten Rechner. Wenn wir x und y in der Eingabe vertauschen, liefern alle 4 Formeln sowohl bei 7- als auch bei 16-stelliger Rechnung das gleiche Ergebnis, nämlich

$$w = 1.114420E + 19.$$

Welches Programm hat das richtige Ergebnis berechnet?
Das Beispiel lehrt uns zunächst, dass mathematisch gleichwertige Formeln im allgemeinen in einem Rechenprogramm verschiedene Resultate haben. Vergleichen wir die Berechnung von w_1 und w_4 miteinander, so müssen wir berechtigt vermuten, dass bekannte Rechengesetze für reelle Zahlen, wie etwa die Unabhängigkeit der Addition von der Reihenfolge

$$(a + b) + c = a + (b + c)$$

auf einem Rechner nicht mehr gelten. Um die verschiedenen Ergebnisse bei der Berechnung der gleichen mathematischen Formel erklären zu können, müssen wir genauer untersuchen, was unsere Vorstellung bezüglich der Ausführung eines numerischen Algorithmus von der wirklichen Ausführung im Rahmen eines Rechenprogramms auf einem konkreten Rechner unterscheidet. Beantworten wir zunächst die Frage nach dem Weg, auf dem man von der Formulierung einer Aufgabe zu einem Maschinenergebnis gelangt. Aus einer praktischen Aufgabe wird ein Modell für die Aufgabe entworfen. In das Modell gehen Daten ein, die wir aus unterschiedlichen Gründen nicht exakt kennen. Wegen dieser Datenunsicherheit wird die Aufgabe viele mögliche Ergebnisse haben. Aus diesem Modell muss man durch eine geeignete Idealisierung – Anwendung von Theorien und Gesetzmäßigkeiten, Vernachlässigung von Einflüssen und Abhängigkeiten, die man als unwesentlich ansieht – eine mathematische Aufgabenstellung herausarbeiten. Eine mathematische Aufgabe, die einer numerischen Behandlung zugängig ist, besteht i. a. aus Gleichungen, Ungleichungen oder ähnliche Beziehungen zwischen bekannten Größen und Funktionen, den sog. Daten und unbekannten Größen bzw. Funktionen. Durch die numerische Rechnung kann lediglich über eine endliche Folge von Operationen aus den gegebenen Größen eine Reihe von Zahlen gewonnen werden, die das Ergebnis des mathematischen Problems wiedergeben; sie sind entweder Nähungswerte für die gesuchte Lösung der Aufgabe oder sie bestimmen als Parameter eine Näherungsfunktion, die gesuchte Näherungslösung. Die Spezifikation der endlichen Folge von Rechenschritten, die aus den Daten diese Ergebniswerte erzeugen,

nennt man numerischen Algorithmus. Damit auf einem konkreten Rechnersystem aus konkret vorgegebenen Daten nach einem spezifizierten numerischen Algorithmus Zahlenwerte für die Ergebnisgrößen erzeugt werden können, muss der Algorithmus in ein Rechenprogramm umgesetzt werden. Dies geschieht i. a. durch Formulierung des Algorithmus in einer höheren Programmiersprache. Das Rechenprogramm wird sodann durch Komponenten des Betriebssystems in eine Folge von Anweisungen übersetzt, die der Rechner unmittelbar verstehen und ausführen kann, das sog. Maschinenprogramm. Die Umwandlung eines numerischen Algorithmus in ein ausführbares Rechenprogramm bezeichnet man als Implementierung des numerischen Algorithmus. Das Rechenprogramm operiert dabei stets nur mit solchen Zahlen, die auch auf dem Rechner dargestellt werden können. Damit wird klar, dass das vom Rechner präsentierte Maschinenergebnis praktisch nie mit dem gesuchten Ergebnis der urprünglichen Aufgabe übereinstimmen kann. Die Abweichung eines nach einer bestimmten Vorschrift sich ergebenden Resultates von dem gewünschten nennt man in der numerischen Mathematik Fehler. Die drei wesentlichen Fehlertypen sind folgende:

- Rechenfehler: Diese entstehen als Folge der Implementierung des numerischen Algorithmus; anstelle des gewünschten Resultates eines numerischen Algorithmus berechnet das Rechenprogamm ein Maschinenresultat. Die arithmetischen Operationen können im Rechner nur mit Zahlen ausgeführt werden, die auch im Rechner darstellbar sind. Ist das Ergebnis einer Operation keine Rechnerzahl, so muss es auf eine solche 'gerundet' werden.

- Verfahrensfehler: Nur bei wenigen mathematischen Aufgaben kann ein numerischer Algorithmus angegeben werden, dessen Ergebnis (unter gewissen Voraussetzungen) mit der Lösung der mathematischen Aufgabe übereinstimmt. Fast stets sind unendliche Abläufe durch endliche in geeigneter Weise zu ersetzen. Für die gleichen Daten weicht daher das Ergebnis eines numerischen Algorithmus vom wahren Ergebnis des dargestellten mathematischen Probems ab. Diese Abweichung nennt man Verfahrensfehler.

- Eingabefehler: Auf Grund der idealisierten Annahmen im mathematischen Modell sind einige Daten i. a. mit beträchtlichen Unsicherheiten behaftet. Solche Unsicherheiten bewirken natürlich auch Unsicherheiten im Ergebnis der mathematischen Aufgabe. Mit den Eingabedaten des Problems ändern sich auch die Ergebnisse. Diese Fehlerart hat einen anderen Charakter als Rechen– und Verfahrensverfehler, weil man hier gar nicht genau sagen kann, was eigentlich eine Lösung der Aufgabe sein soll. Die Auswirkung dieser Unsicherheit auf das Ergebnis des mathematischen Modells begrenzt unmittelbar die Genauigkeit, mit der die numerische Lösung des mathematischen Problems sinnvollerweise anzugeben ist. Der Eingabefehler begrenzt damit auch die Genauigkeit, die bei dem Maschinenergebnis anzustreben ist. Andererseits kann der Eingabefehler auch so schwerwiegend sein, dass die im Rechner vorhandene Aufgabe bei jedem numerischen Algorithmus eine Lösung hat, die mit der wahren nichts zu tun hat.

Diese theoretischen Überlegungen wollen wir nun durch einige Beispiele illustrieren.

Beispiel 2: Wir wollen folgendes lineare Gleichungssystem in 4 Unbekannten lösen:

$$x + \frac{1}{2}y + \frac{1}{3}u + \frac{1}{4}v = 1$$

$$\frac{1}{2}x + \frac{1}{3}y + \frac{1}{4}u + \frac{1}{5}v = 1$$

$$\frac{1}{3}x + \frac{1}{4}y + \frac{1}{5}u + \frac{1}{6}v = 1$$

$$\frac{1}{4}x + \frac{1}{5}y + \frac{1}{6}u + \frac{1}{7}v = 1$$

Die exakte Lösung lautet, wie man sich durch Einsetzen überzeugen kann:

$$x = -4, y = 60, u = -180, v = 140.$$

Die Zahlen 1/3, 1/6 und 1/7 sind nicht exakt im Rechner darstellbar; aus diesem Grunde müssen ihnen bei der Eingabe Maschinenzahlen zugeordnet werden. Wenn wir diese Zahlen der Reihe nach mit 4, 5, 6 und 8 Stellen eingeben, erhalten wird folgende Maschinenergebnisse:

	x	y	u	v
4:	-5.8999	80.5437	-228.5033	171.1528%
5:	-4.1814	61.9951	-184.7562	143.0748%
6:	-4.0262	60.2963	-180.7181	140.4694%
8:	-4.0003	60.0033	-180.0080	140.0052%

Wir erkennen insbesondere, dass erst ab einer Eingabegenauigkeit von 5 Ziffern die Maschinenlösung sich der wahren Lösung nähert. Außerdem bemerken wir, dass die Lösung empfindlich auf eine Änderung der Eingabedaten reagiert. Jedoch sind die gefundenen Lösungen in dem Sinne exakt, dass sie Lösungen jener Aufgaben darstellen, die sich im Rechner befinden; dies wird insbesondere durch die Nachiteration angezeigt. Die sensible Änderung der Lösung bei Änderung der Eingabedaten ist nicht etwa Folge des verwendeten Algorithmus, sondern eine Eigenschaft der Aufgabe selbst. Man nennt eine numerische Aufgabe stabil, wenn der Fehler in der Lösung in der Größenordnung des Eingabefehlers liegt. In diesem Sinne ist die obige Aufgabe instabil. Die Verstärkung des Eingabefehlers in der Lösung der Aufgabe kann kein noch so ausgefeilter Algorithmus verhindern, er ist unvermeidlich. Instabile Aufgaben verlangen zu ihrer numerischen Behandlung solche Algorithmen, die den Eingabefehler möglichst nicht noch zusätzlich verstärken. Instabile Aufgaben treten in Anwendungen oft auf, so z. B. in der Raumfahrt. Beim Wiedereintritt eines Raumschiffes in die Erdathmosphäre spielen der Luftwiderstand, die Anfluggeschwindikeit und der Eintrittswinkel in die Erdathmosphäre eine entscheidende Rolle. Besonders empfindlich reagiert das Rückkehrmanöver auf Änderungen der Luftkräfte, von denen Anfluggeschwindigkeit und Eintrittswinkel abhängig gemacht werden müssen, um ein Verglühen bzw. ein Zurückschleudern in den Weltraum zu vermeiden. Daraus resultiert die Gefährlichkeit des Landemanövers. Nur eine genaue Kenntnis der auftretenden

Luftkräfte ermöglicht es, die Bahnparameter so zu bestimmen, dass das Landemanöver erfolgreich abgeschlossen werden kann, d. h. dass gerade solche Maschinenlösungen gefunden werden, die eine sichere Landung des Raumschiffes garantieren.
Beispiel 3: Wir wollen die Quadratwurzel x aus einer positiven Zahl a ziehen:

$$x^2 = a$$

d. h. gesucht ist eine Zahl x mit $x = \frac{a}{x}$. Dies ist eine Operation, die sich auf dem Rechner nicht exakt ausführen lässt. Daher muss man eine Näherungsmethode anwenden. Wählt man x willkürlich, so wird eine Seite der Gleichung größer als die andere sein, so dass der Mittelwert eine bessere Näherung ist:

$$x_{neu} = \frac{1}{2}(x + \frac{a}{x}).$$

Dieses Vorgehen lässt sich wiederholt anwenden. In der folgenden Tabelle sind die berechneten Näherungen nach 4 Wiederholungen für einige Werte von a den exakten gegenübergestellt. Dabei wurde als Startwert der jeweilige Wert für a genommen.

Wert	Wurzel	berechnet	Fehler
0.000001	0.001000	0.062505	6150.53 %
0.000010	0.003162	0.062553	1878.10 %
0.000100	0.010000	0.063030	530.31 %
0.001000	0.031623	0.067725	114.66 %
0.010000	0.100000	0.108404	8.40 %
0.100000	0.316228	0.316246	0.0056 %
1.000000	1.000000	1.000000	0.0000 %
10.000000	3.162278	3.162456	0.0056 %
100.000000	10.000000	10.840435	8.40 %
1000.000000	31.622777	67.725327	114.66 %
10000.000000	100.000000	630.303596	530.31 %
100000.000000	316.227766	6255.311608	1878.10 %

Eine oberflächliche Betrachtung dieser Tabelle zeigt uns folgendes: Nach 4 Iterationen ist der Verfahrensfehler für Zahlen außerhalb des Intervalls (0.1,10) so groß, dass die Methode unbrauchbar erscheint, falls man nicht eine erheblich höhere Anzahl von Schritten verwendet. Andererseits sollte uns aber die besonders einfache Iterationsformel ermutigen, darüber nachzudenken, wie man sie trotzdem anwenden kann. In der Tat: Wir benötigen für das Wurzelziehen nur eine Iterationsformel für Zahlen aus dem Intervall (0.1, 10)! Jede Zahl a lässt sich als Produkt einer geradzahligen Potenz von 10 und einer Zahl aus dem Intervall (0.1,10) darstellen:

$$a = 10^{2n} * b, \qquad 0.1 \leq b \leq 10$$

Für den ersten Faktor lässt sich die Wurzel sofort angeben, während die Wurzel des 2. Faktors nach der obigen Methode berechnet werden kann. Die folgende Tabelle zeigt die Ergebnisse nach dieser Änderung des Verfahrens, wobei nun als Startwert stets die Zahl 1 genommen wurde.

Wert	Wurzel	berechnet	Fehler
0.000001	0.001000	0.001000	0.0000 %
0.000010	0.003162	0.003162	0.0056 %
0.000100	0.010000	0.010000	0.0000 %
0.001000	0.031623	0.031625	0.0056 %
0.010000	0.100000	0.100000	0.0000 %
0.100000	0.316228	0.316246	0.0056 %
1.000000	1.000000	1.000000	0.0000 %
10.000000	3.162278	3.162456	0.0056 %
100.000000	10.000000	10.000000	0.0000 %
1000.000000	31.622777	31.624556	0.0056 %
10000.000000	100.000000	100.000000	0.0000 %
100000.000000	316.227766	316.245562	0.0056 %

Wir erkennen, dass bei Ausnutzung einer einfachen Tatsache, sich der Verfahrensfehler entscheidend reduziert und die Ergebnisse brauchbar werden. Um Methoden zu finden, die den Verfahrensfehler reduzieren, müssen oft mathematische Theorien bemüht werden. Meist ist es auch schwierig oder unmöglich, realistische Abschätzungen für den Verfahrensfehler zu gewinnen. Viele Mathematiker beschäftigen sich weltweit mit dieser Frage. Ähnlich wichtig ist die Aufgabe, für einen numerischen Algorithmus jene Bereiche für die Eingabedaten zu finden, innerhalb derer der Verfahrensfehler in akzeptierten Grenzen bleibt. Es gibt aber auch Verfahren, bei denen jede Mühe vergebens ist, weil der Rechenfehler in entscheidender Weise dominiert, wie das folgende Beispiel zeigt.

Beispiel 4: Für die Funktion

$$y = x^n * e^{(1-x)}$$

ist das bestimmte Integral über dem Intervall (0, 1) zu bestimmen. Dazu sei $I(n)$ der Wert dieses Integrals. Eine einfache Überlegung zeigt uns, dass folgender Zusammenhang gilt:

$$I(0) = e - 1 = 1.7182818284590\ldots$$

$$I(n) = -1 + n * I(n-1), n = 1, 2, 3, \ldots$$

$$0 < I(n+1) < I(n)$$

und $I(n)$ wird für große n beliebig klein. Die folgende Tabelle zeigt uns die Werte von $I(n)$ für $n = 0, 1, \ldots, 22$ bei Rechnung mit 7 Stellen und 16 Stellen.

n	$I_7(n)$	$I_{16}(n)$
0	1.71828	1.71828
2	0.43656	0.43556
4	0.23876	0.23876
6	0.16294	0.16292
8	0.12480	0.12332
10	0.23215	0.09911
12	17.64343	0.08281
14	3196.10474	0.07110
16	767048.12500	0.06506
18		0.90685
20		323.60412
22		149482.10144

In Übereinstimmung mit unserer mathematischen Erwartung werden die Integralwerte anfangs mit jedem Schritt kleiner. Doch urplötzlich wachsen sie von Schritt zu Schritt immer rascher. Haben wir vielleicht einen Fehler gemacht? Um dies festzustellen, wenden wir die Formel rückwärts an, indem wir für ein großes N annehmen, dass bereits $I(N) = 0$ gilt. Die Rückwärtsformel lautet

$$I(n-1) = \frac{1}{n}(1 + I(n)), n = N, N-1, \ldots, 1.$$

Die folgende Tabelle zeigt die erhaltenen Werte.

n	$I_7(n)$	$I_{16}(n)$
0	1.71828	1.71828
1	0.71828	0.71828
2	0.43656	0.43656
3	0.30369	0.30369
4	0.23876	0.23876
5	0.19382	0.19382
6	0.16290	0.16290
7	0.14028	0.14028
8	0.12222	0.12222
9	0.10000	0.10000
10	0.00000	0.00000

Wir erkennen, dass unsere mathematischen Voraussetzungen an die Methode richtig waren. Als Ursache für das äußerst fehlerhafte Maschinenergebnis müssen die im Rechenprogramm sich schnell aufschaukelnden Rechenfehler angesehen werden. Die obige Vorwärtsformel zeigt uns, dass der Rechenfehler aus dem Schritt $n-1$ im Schritt n um den Faktor n verstärkt wird, also wird der Eingabefehler im Schritt n um den Faktor $1 * 2 * 3 * ... * n$ verstärkt, so dass schon der unvermeidbare Eingabefehler zum Versagen der Methode führen muss. Um diese Aussage noch zu stärken, betrachten wir die Ergebnisse von Beispiel 4 für den Fall, dass der Startwert $I(0)$ nur mit 4 Stellen eingegeben wird.

n	$I_7(n)$	$I_{16}(n)$
0	1.71800	1.71800
1	0.71800	0.71800
2	0.43600	0.43600
3	0.30800	0.30800
4	0.23200	0.23200
5	0.16001	0.16000
6	-0.03999	-0.40000
7	-1.27973	-1.28000
8	-11.23781	-11.24000
9	-102.14027	-102.16000
10	-1022.40277	-1022.60000
11	-11247.43066	-11249.60000
12		-134996.20000

Im Gegensatz zum obigen Fall strebt hier die Folge der berechneten Integralwerte sogar gegen minus unendlich! Die Rechnung zeigt uns außerdem, wie sich ein kleiner Rechenfehler von Schritt 4 nach Schritt 5 mit jedem weiteren Schritt eine Stelle höher schiebt. Insbesondere zeigt uns dieses Beispiel, dass es numerische Methoden gibt, bei denen ein winziger Eingangsfehler zu katastrophalen Fehlern im Maschinenergebnis führt. Eine numerische Methode nennt man stabil, wenn der Ausgangsfehler in der Größenordnung des Eingabefehlers liegt. Die obige Methode ist offenbar nicht stabil. Die Suche nach numerisch stabilen Algorithmen für wichtige praktische Aufgaben ist eine aktuelle und wichtige Forschungsaufgabe der numerischen Mathematik.
Was heißt nun 'richtiges numerisches Rechnen' auf dem Computer?
Die hier dargelegten Gedanken führen uns auf folgende Thesen.

- Will man eine numerische Aufgabe mit einem Rechner lösen, so untersuche man zunächst, ob es sich um eine stabile Aufgabe handelt oder nicht. Diese Untersuchung sollte zu einer realistischen Abschätzung des zu erwartenden Fehlers in der Lösung in Abhängigkeit vom Fehler in den Eingabedaten führen.

- Zur Lösung der Aufgabe gibt es oft mehrere numerische Algorithmen. Bei der Auswahl des numerischen Algorithmus für die zu lösende Aufgabe berücksichtige man den optimalen Arbeitsbereich des Algorithmus, d. h. jenen Bereich, in dem der Verfahrensfehler möglichst gering ist; man verwende Methoden, die die Eingabedaten so transformieren, dass sie im optimalen Arbeitsbereich liegen,

- Man benutze stabile Algorithmen insbesondere dann, wenn eine instabile Aufgabe vorliegt.

- Bei der Implementierung eines numerischen Algorithmus kommt es darauf an, den Rechenfehler so klein wie möglich zu halten.

Die Verstärkung von Rechenfehlern hat in den meisten Fällen zwei Ursachen: Aus vielen kleinen Zahlen werden große erzeugt. Meist geschieht dies über die Aufsummierung vieler

Zahlen mit unterschiedlichen Exponenten. Bei der Addition einer großen und einer kleinen Zahl gehen vormals richtige Stellen verloren, so ist z. B. bei 4-stelliger Rechnung

$$1000. + 0.1000 = 1000.$$

Selbst wenn wir 10000mal die Zahl 0.1 addieren würden, es ändert sich nichts, obwohl eigentlich 2000 herauskommen müsste. Besonders gefährlich ist die Subtraktion zweier Zahlen, die die gleiche Größenordnung haben. Bei 4-stelliger Rechnung ist

$$0.2439 - 0.2438 = 0.0001.$$

Diese Operation wird exakt ausgeführt, da das Ergebnis wieder eine Rechnerzahl (hier 4-stellige Zahl) ist. Ist jedoch die letzte Stelle eines Operanden fehlerbehaftet, so geraten hier diese Fehler in führende Stellen, sie werden verstärkt, da die richtigen Stellen verschwunden sind. Diesen Effekt nennt man 'Auslöschung'. Instabilitäten eines numerischen Algorithmus sind meist mit dem Auftreten von Auslöschung verbunden. Darum bedeutet richtiges numerisches Rechnen auf dem Computer vor allem, numerische Algorithmen so zu formulieren und zu implementieren, dass für die zu lösenden Aufgaben Auslöschung während der Rechnung vermieden wird. Hierfür gibt es kein allgemeingültiges Rezept, da diese Aufgabe nicht algorithmierbar ist.

```
c  main svor
      program svor
      implicit real*8(a-h,o-z)
      parameter (r1=10864.0d0,
     f           r2=18817.0d0,
     f           a4=0.3333d0,
     f           b4=0.1667d0,
     f           c4=0.1429d0,
     f           a5=0.33333d0,
     f           b5=0.16667d0,
     f           c5=0.14286d0,
     f           a6=0.333333d0,
     f           b6=0.166667d0,
     f           c6=0.142857d0,
     f           a8=0.33333333d0,
     f           b8=0.16666667d0,
     f           c8=0.14285714d0)
      real*8 x
      call beis1(r1,r2)
      pause
      call beis1(r2,r1)
      pause
      call beis2(4,a4,b4,c4)
      pause
      call beis2(5,a5,b5,c5)
      pause
      call beis2(6,a6,b6,c6)
      pause
      call beis28(8,a8,b8,c8)
      pause
      call beis3
      pause
      call beis3r
      pause
      x=dexp(1.d0)-1.d0
      call beis4(16,x)
      pause
      call  beis4r(10)
      pause
      call beis4(4,1.718d0)
      pause
      end
c
      subroutine beis1(x,y)
      real*8 x,y,w1,w2,w3,w4
    1 format (//,10x,
     f'eine einfache Gleichung nach ',
     f'4 Methoden berechnet',//,10x,
     f'Rechnung in 7- und 16-stelliger ',
     f'Genauigkeit',//,10x,
     f'w1 = 9*x*x*x*x - y**4 + 2*y*y',/,10x,
     f'w2 = (3*x*x - y*y) * (3*x*x + y*y)',
     f' + 2*y*y',/,10x,
     f'w3 = 9*x**4 + (2*y**2 - y*y*y*y)',
```

```
     f/,10x,
     f'w4 = (9*x**4 + 2*y**2) - y**4',
     f//,10x,'an der Stelle x=',f8.1,
     f' y=',f8.1,//,10x,
     f'Methode 1',8x,'Methode 2',8x,
     f'Methode 3',8x,'Methode 4',/)
    2 format (10x,4(f13.1,2x),/)
    3 format (10x,4(e15.7,2x),/)
c
      w1=(9*x*x*x*x - y**4) + 2*y*y
      w2=(3*x*x-y*y)*(3*x*x+y*y)+2*y*y
      w3=9*x**4 + (2*y**2 - y*y*y*y)
      w4=(9*x**4 + 2*y**2) - y**4
      write (*,1) x,y
      u=x
      v=y
      v1=9*u*u*u*u - v**4 + 2*u*u
      v2=(3*u*u-v*v)*(3*u*u+v*v)+2*v*v
      v3=9*u**4 + (2*v**2 - v*v*v*v)
      v4=(9*u**4 + 2*v**2) - v**4
      if (abs(v1+v2+v3+v4).le.1.e13) then
        write (*,2) v1,v2,v3,v4
        write (*,2) w1,w2,w3,w4
      else
        write (*,3) v1,v2,v3,v4
        write (*,3) w1,w2,w3,w4
      endif
      return
      end
c
      subroutine beis2(l,a1,a2,a3)
      real*8 a1,a2,a3
      real lu
      dimension a(4,4),x(4),b(4),lu(4,4),
     f          c(4),d(4)
    1 format (////,10x,
     f'Loesung eines Gleichungssystems '
     f'mit 4 Unbekannten',//,10x,
     f'und der Koeffizientenmatrix:',
     f' a(i,j) = 1/(i+j-1) '
     f'(Hilbert-Matrix)',//,10x,
     f' die exakte Loesung lautet:',
     f//,13x,'-4   6   -180   140')
    2 format (/,10x,
     f'berechnete Loesung bei',i3,
     f       ' genauen Eingabestellen:')
    3 format (/,10x,4(f10.4,1x),/)
    4 format (10x,'nachiterierte Loesung',
     f       //,10x,4(f10.4,1x),/)
c
      n=4
      do i=1,n
        b(i)=1.0
        do j=1,n
```

```
          a(i,j)=1.0/(i+j-1.0)
        enddo
      enddo
      a(1,3)=a1
      a(3,1)=a(1,3)
      a(2,2)=a(1,3)
      a(3,4)=a2
      a(4,3)=a(3,4)
      a(4,4)=a3
      write (*,1)
      call lup(a,lu,n)
      call forw(lu,b,c,n)
      call backw(lu,c,x,n)
      write (*,2) l
      write (*,3) x
      call nachit(a,lu,b,x,c,d,n,1)
      write (*,4) x
      return
      end
c
      subroutine beis28(l,a1,a2,a3)
      real*8 a,b,x,lu,c,d,a1,a2,a3
      dimension a(4,4),x(4),b(4),lu(4,4),
     f          c(4),d(4)
    1 format (////,10x,
     f'Loesung eines Gleichungssystems '
     f'mit 4 Unbekannten',//,10x,
     f'und der Koeffizientenmatrix:',
     f' a(i,j) = 1/(i+j-1) '
     f'(Hilbert-Matrix)',//,10x,
     f' die exakte Loesung lautet:',
     f//,13x,'-4   6   -180   140')
    2 format (/,10x,
     f'berechnete Loesung bei',i3,
     f       ' genauen Eingabestellen:')
    3 format (/,10x,4(f10.4,1x),/)
    4 format (10x,'nachiterierte Loesung',
     f       //,10x,4(f10.4,1x),/)
c
      n=4
      do i=1,n
        b(i)=1.d0
        do j=1,n
          a(i,j)=1.d0/(i+j-1.d0)
        enddo
      enddo
      a(1,3)=a1
      a(3,1)=a(1,3)
      a(2,2)=a(1,3)
      a(3,4)=a2
      a(4,3)=a(3,4)
      a(4,4) = a3
      write (*,1)
      call lup8(a,lu,n)
```

```
      call forw8(lu,b,c,n)
      call backw8(lu,c,x,n)
      write (*,2) l
      write (*,3) x
      call nachi8(a,lu,b,x,c,d,n,1)
      write (*,4) x
      return
      end
c
      subroutine beis3
      real*8 x,y,z,xx
    1 format (//,10x,
     f'iterative Berechnung der '
     f'Quadratwurzel',
     f//,10x,'mit 4 Iterationen, '
     f'Startwert = wert',
     f//,10x,'Wert',13x,'Wurzel',
     f11x,'berechnet',
     f8x,'prozentualer Fehler',/)
    2 format(5x,3(f15.6,2x),f12.4,' %')
c
      x=0.1d-6
      write (*,1)
      do j=1,12
        x=10*x
        z=dsqrt(x)
        xx=x
        do i=1,4
          xx=(xx+(x/xx))/2
        enddo
        y=100*dabs(z-xx)/z
        write (*,2) x,z,xx,y
      enddo
      return
      end
c
      subroutine beis3r
      real*8 x,y,z,xx
    1 format (//,10x,
     f'richtige iterative Berechnung der '
     f'Quadratwurzel',
     f//,10x,'mit 4 Iterationen und '
     f'Startwert = 1',
     f//,10x,'Wert',13x,'Wurzel',11x,
     f'berechnet',
     f8x,'prozentualer Fehler',/)
    2 format(5x,3(f15.6,2x),f12.4,' %')
c
      x=0.1d-6
      write (*,1)
      do j=1,12
        x=10*x
        z=dsqrt(x)
        e=expo(x,ie)
```

```
        a=1.0
        if (iabs((ie/2)*2).lt.iabs(ie)) a=10.0
        y=(x/e)*a
        if (ie.lt.0) ie=ie-1
        e=10.0**(ie/2)
        xx=1.d0
        do i=1,4
          xx=(xx+(y/xx))/2
        enddo
        xx=xx*e
        y=100*dabs(z-xx)/z
        write (*,2) x,z,xx,y
      enddo
      return
      end
c
      subroutine beis4(l,ss)
      real*8 s,se,ss
      dimension t(50),s(50)
    1 format (///,10x,
     f'iterative Vorwaertsberechnung des',
     f' Integrals ueber',//,10x,
     f' die Funktion '
     f'x hoch n mal e hoch(1-x) ueber ',
     f//,10x,'dem Intervall (0,1) '
     f'fuer n=0,1,2,... bei ',i2,
     f' Eingabestellen',/)
    2 format(25x,f15.5,2x,i3)
    3 format (10x,2f15.5,2x,i3)
c
      se=1000000.d0
      write(*,1) l
      t(1)=ss
      s(1)=ss
      k=1
      do while (k.le.50.or.dabs(s(k)).le.se)
        t(k+1)=t(k)*k-1.0
        s(k+1)=s(k)*k-1.d0
        k=k+1
      enddo
      i=1
      do while(dabs(s(i)).le.se)
        if (abs(t(i)).gt.se) then
          write (*,2) s(i),i-1
        else
          write (*,3) t(i),s(i),i-1
          if (l.gt.14) i=i+1
        endif
        i=i+1
      enddo
      return
      end
c
      subroutine beis4r(k)
```

```
      real*8 s
      dimension s(50),t(30)
    1 format (///,10x,
     f'iterative Rueckwaertsberechnung des',
     f' Integrals ueber',//,10x,
     f'die Funktion  '
     f'x hoch n mal e hoch (1-x) ueber ',
     f//,10x,'dem Intervall (0,1) '
     f'fuer n=10,9,8,...0',/)
    2 format (10x,2f15.5,2x,i3)
c
      write (*,1)
      if (k.gt.30) return
      t(k+1)=0.0
      s(k+1)=0.d0
      ke=k-1
      do i=1,k
        ii=k-i+1
        t(ii)=(1.0+t(ii+1))/ii
        s(ii)=(1.d0+s(ii+1))/ii
      enddo
      kk=k+1
      do i=1,kk
        ii=i-1
        write (*,2) t(i),s(i),ii
      enddo
      return
      end
c
c     lu-zerlegung
c
      subroutine lup8(a,lu,n)
      real*8 a,lu,z
      dimension a(n,n),lu(n,n)
      do i=1,n
        do k=1,n
          lu(i,k)=a(i,k)
        enddo
      enddo
      do k=1,n-1
        do i=k+1,n
          z=lu(i,k)/lu(k,k)
          do j=k+1,n
            lu(i,j)=lu(i,j)-z*lu(k,j)
          enddo
          lu(i,k)=z
        enddo
      enddo
      return
      end
c
c     vorwaertseinsetzen
c
      subroutine forw8(a,b,x,n)
```

```
      real*8 a,b,x,z,y
      dimension a(n,n),b(n),x(n)
      do i=1,n
        z=0.d0
        do k=1,i-1
          z=z+a(i,k)*x(k)
        enddo
        x(i)=b(i)-z
      enddo
      return
      end
c
c     rueckwaertseinsetzen
c
      subroutine backw8(a,b,x,n)
      real*8 a,b,x,z
      dimension a(n,n),b(n),x(n)
      do k=n,1,-1
        z=0.d0
        do i=k+1,n
          z=z+a(k,i)*x(i)
        enddo
        x(k)=(b(k)-z)/a(k,k)
      enddo
      return
      end
c
c     nachiteration
c
      subroutine nachi8(a,lu,b,x,y,z,n,it)
      real*8 a,lu,b,x,y,scalp8,z
      dimension a(n,n),lu(n,n),x(n),y(n),
     f          z(n),b(n)
      do k=1,it
        do i=1,n
          z(i)=scalp8(-b(i),a(i,1),n,x,n)
        enddo
        call forw8(lu,z,y,n)
        call backw8(lu,y,z,n)
        do i=1,n
          x(i)=x(i)-z(i)
        enddo
      enddo
      return
      end
c
c     skalarprodukt
c
      real*8 function scalp8(s,a,ja,b,n)
      real*8 a,b,s,t,u,tr
      dimension a(n),b(n)
      t=0.d0
      tr=0.d0
      ii=1
```

```
      do i=1,n
        u=a(ii)*b(i)
        tr=tr-(((t+u)-t)-u)
        t=t+u
        ii=ii+ja
      enddo
      scalp8=s+(t+tr)
      return
      end
c
c     lu-zerlegung
c
      subroutine lup(a,lu,n)
      real lu
      dimension a(n,n),lu(n,n)
      do i=1,n
        do k=1,n
          lu(i,k)=a(i,k)
        enddo
      enddo
      do k=1,n-1
        do i=k+1,n
          z=lu(i,k)/lu(k,k)
          do j=k+1,n
            lu(i,j)=lu(i,j)-z*lu(k,j)
          enddo
          lu(i,k)=z
        enddo
      enddo
      return
      end
c
c     vorwaertseinsetzen
c
      subroutine forw(a,b,x,n)
      dimension a(n,n),b(n),x(n)
      do i=1,n
      z=0.d0
        do k=1,i-1
          z=z+a(i,k)*x(k)
        enddo
        x(i)=b(i)-z
      enddo
      return
      end
c
c     rueckwaertseinsetzen
c
      subroutine backw(a,b,x,n)
      dimension a(n,n),b(n),x(n)
      do k=n,1,-1
        z=0.d0
        do i=k+1,n
          z=z+a(k,i)*x(i)
```

```
        enddo
        x(k)=(b(k)-z)/a(k,k)
      enddo
      return
      end
c
c     nachiteration
c
      subroutine nachit(a,lu,b,x,y,z,n,it)
      real lu
      dimension a(n,n),lu(n,n),x(n),y(n),
     f          z(n),b(n)
      do k=1,it
        do i=1,n
          z(i)=scalp(-b(i),a(i,1),n,x,n)
        enddo
        call forw(lu,z,y,n)
        call backw(lu,y,z,n)
        do i=1,n
          x(i)=x(i)-z(i)
        enddo
      enddo
      return
      end
c
c     skalarprodukt
c
      real function scalp(s,a,ja,b,n)
      real*8 t,u,tr
      dimension a(n),b(n)
      t=0.d0
      tr=0.d0
      ii=1
      do i=1,n
        u=a(ii)*b(i)
        tr=tr-(((t+u)-t)-u)
        t=t+u
        ii=ii+ja
      enddo
      scalp=s+(t+tr)
      return
      end
c
c     exponentenbestimmung
c
      real function expo(x,i)
      real*8 x
      i=0
      z=dabs(x)
      y=1.0
      if (z.lt.0.1d0) then
        do while (z.lt.0.1d0)
          z=z*10.d0
          y=0.1d0*y
```

```
    i=i-1
  enddo
else
  if (z.ge.1.d0) then
    do while (z.ge.1.d0)
      z=z*0.1d0
      y=y*10.d0
      i=i+1
    enddo
  endif
endif
expo=y
return
end
```