1 Nichtlineare Gleichungen in einer Unbekannten

1.1 Ein kurzer Rundgang im Garten der Gleichungen

Als Einstieg in die Numerische Mathematik behandeln wir numerische Lösungsverfahren für Gleichungen in einer Unbekannten. *Linear* sind solche Gleichungen, wenn sie sich in der Form

$$kx = d, \quad k, d \text{ gegeben}, \quad x \text{ gesucht}$$

schreiben lassen. Offensichtlich gibt es, falls $k \neq 0$, eine eindeutige Lösung. Dieses Thema ist also vorläufig abgehakt, wir kümmern uns nun um *nichtlineare* Gleichungen. (Die linearen Gleichungen werden uns erst dann intensiver beschäftigen, wenn sie in Massen, als Systemen mit *mehreren* Unbekannten auftreten.)

Analytische oder numerische Lösung Wenn sich durch algebraische Umformungen die Lösung einer Gleichung explizit, also in der Form $x = \ldots$, anschreiben lässt (im obigen Beispiel: $x = d/k$, allgemein ein Term, in dem nur die üblichen Standard-Rechenoperationen und -Funktionen auftreten), spricht man von einer *analytischen* Lösung

Analytisch lösbar sind beispielsweise *quadratische* Gleichungen, also solche, die sich als

$$x^2 + px + q = 0 \quad p, q \text{ gegeben}, \quad x \text{ gesucht}$$

schreiben lassen. Sie kennen sicherlich die Lösungsformel

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

Es reicht aber nicht, eine Lösungsformel hinschreiben zu können, sie muss auch verlässlich genaue Ergebnisse liefern. Die scheinbar triviale Lösung einer quadratischen Gleichung nach obiger Formel kann recht ungenau werden. Lassen Sie Ihren Taschenrechner damit die kleinere Lösung der quadratischen Gleichung

$$x^2 - 12345678x + 9 = 0$$

berechnen. Der (sechzehnstellig) genaue Wert ist $x_1 = 7{,}290\,000\,597\,780\,479 \times 10^{-7}$. Obwohl übliche Rechner zehn- bis vierzehnstellig genau rechnen, liefern sie nur die ersten paar Stellen richtig. Die numerisch genauere Methode berechnet zuerst die *betragsmäßig größere* Lösung x_1 mit der klassischen Formel und findet dann die *betragsmäßig kleinere* Lösung x_2 mit der alternativen Lösungsformel

$$x_2 = \frac{q}{x_1}\,.$$

Algebraische und transzendente Gleichungen Lineare, quadratische und kubische Gleichungen sind die einfachsten Beispiele *polynomialer* Gleichungen. Ein Polynom in einer Variablen x ist eine Summe von x-Potenzen, multipliziert mit Koeffizienten, also ein Ausdruck der Form

$$a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0.$$

Die höchste auftretende Potenz heißt die *Ordnung* des Polynoms oder der Gleichung.

Kubische Gleichungen und Gleichungen vierter Ordnung sind im Prinzip analytisch lösbar, aber die Formeln (Cardanische Formeln, G. Cardano[1], N. Tartaglia[2], L. Ferrari[3], um 1540) sind so unhandlich, dass sie praktisch kaum verwendet werden. Numerische Verfahren sind in diesen Fällen meist sinnvoller. Sie liefern Näherungen, die schrittweise, mit immer besserer Genauigkeit, die Lösungen anstreben. Ab Polynomgrad fünf gibt es ohnehin keine allgemeinen Lösungsformeln mehr.

> Der junge norwegische Mathematiker Niels Henrik Abel führt 1826 den „Beweis der Unmöglichkeit, algebraische Gleichungen von höheren Graden als dem vierten allgemein aufzulösen“. Ab dem fünften Grad lassen sich Gleichungen also (im Allgemeinen) nicht durch eine *endliche Zahl elementarer Rechenoperationen* (Addition, Subtraktion, Multiplikation, Division, Wurzelziehen) lösen.

Um die Vorstellung der verschiedenen Gleichungstypen zum Abschluss zu bringen: Gleichungen, in denen auch noch Bruchterme, Wurzeln oder rationale Exponenten vorkommen, lassen sich (möglicher Weise nur mit hohem Aufwand und komplizierten Umformungen) auf Systeme polynomialer (man sagt auch: algebraischer) Gleichungen zurückführen. Terme oder Funktionen, die sich nicht mittels endlich vieler elementarer Rechenoperationen formulieren lassen, sind etwas, das die Kräfte der Algebra übersteigt (*„quod vires algebrae transcendit“*, sagte Leibnitz) und heißen deswegen *transzendent*.

Beispielsweise sind die trigonometrischen Funktionen, die Exponentialfunktion und die entsprechenden Umkehrfunktionen transzendente Funktionen. Treten solche Funktionen in Gleichungen auf, ist normaler Weise nur numerische Lösung möglich.

> Explizite Lösungsformeln gibt es nur für polynomiale Gleichungen niedrigen Grades und die allereinfachsten transzendenten Glaichungen. In allen anderen Fällen können nur numerische Methoden eine Lösung finden.

1.2 Begriffe, Probleme, Lösungen

Hier behandelte Aufgabentypen:

$$\begin{array}{rcll} g(x) & = & h(x), & \text{Finden einer } \textit{Lösung} \text{ einer Gleichung} \\ f(x) & = & 0, & \text{Finden einer } \textit{Nullstelle} \text{ der Funktion } f \\ x & = & \phi(x), & \text{Finden eines } \textit{Fixpunktes} \text{ der Funktion } \phi \end{array}$$

> Eine Lösung der Gleichung $f(x) = 0$ heißt *Nullstelle* der Funktion f.
> Eine Lösung der Gleichung $x = \phi(x)$ heißt *Fixpunkt* der Funktion ϕ.

> Eine Fixpunkt-Gleichung $x = \phi(x)$ lässt sich natürlich sofort umformen auf $\phi(x) - x = 0$. Jeder Fixpunkt von ϕ ist also zugleich Nullstelle von $f(x) = \phi(x) - x$.
> Selbstverständlich muss der Funktionsterm in einer Nullstellen-Aufgabe nicht automatisch f heißen, ebensowenig wie in Fixpunkt-Gleichungen die Funktion mit ϕ bezeichnet sein muss. Die Vorlesungsunterlagen schreiben aber in der Regel $x = \phi(x)$, wenn diese Gleichung durch Umformen aus $f(x) = 0$ entstanden ist.

[1] auch bekannt durch Kardanwelle und kardanische Aufhängung, die er ebenfalls nicht erfunden hat
[2] Niccolò Fontana Tartaglia verriet Cardano die Lösung unter dem Siegel der Verschwiegenheit; war stinksauer, als der sie trotzdem veröffentlichte.
[3] Das Rennen um die Lösung für Gleichungen vierten Grades, sozusagen die Formel Vier, wurde damals von Ferrari gewonnen.

Nullstellen von Polynomen nennt man auch *Wurzeln*.[4]

Eine *analytische Lösung* ist ein expliziter Ausdruck, in dem nur bekannte Größen und Funktionen vorkommen.

> Welche Funktionen dabei als „bekannt“ vorausgesetzt werden, ist nicht exakt festgelegt. Letztlich lassen sich auch von so geläufigen Funktionen wie Sinus oder Cosinus Werte nur durch numerische Verfahren berechnen – auch wenn Ihnen der Taschenrechner diese Arbeit abnimmt.

Demgegenüber steht die *numerische Lösung*, eine Rechenvorschrift, die eine schon irgendwie bekannte Näherung schrittweise verbessert.

Mehrfache Nullstellen: Eine Funktion f hat an der Stelle x eine genau n-fache Nullstelle, wenn zugleich $f(x) = 0, f'(x) = 0, f''(x) = 0, \ldots, f^{(n-1)}(x) = 0$ und $f^{(n)}(x) \neq 0$. (Dabei setzen wir die Existenz stetiger Ableitungen mindestens bis zur n-ten Ordnung voraus.)

Die auftretenden Funktionen $f, g, \ldots$ und Variablen $x, y, \ldots$ bezeichnen in dieser Vorlesung in der Regel *reelle* Größen. Die *komplexen* Zahlen sind an sich der natürliche Lebensraum für Polynome und Funktionen (unter anderem deswegen, weil Polynome n-ten Grades dort immer genau n Nullstellen haben, Fundamentalsatz der Algebra). Die meisten Definitionen und Verfahren lassen sich leicht für komplexe Variable und komplexwertige Funktionen verallgemeinern. Trotzdem beschränken wir uns (abgesehen von gelegentlichen Hinweisen) auf Rechenverfahren in den reellen Zahlen.

Checkliste zum Lösen nichtlinearer Gleichungen

Gleichzeitig Inhaltsangabe und Stoffübersicht der folgenden Abschnitte.

- Vorarbeiten
  - Überblicken Sie den Verlauf der Funktionen (Wertetabelle, graphische Darstellung).
  - Definitionsbereich? Wo können die Lösung liegen? Wie viele Lösungen gibt es?
  - Lassen sich günstige Umformungen finden?
- Trivialmethoden für Computer oder Taschenrechner
  - Systematisches Einsetzen in Wertetabelle
  - Hineinzoomen im Funktionsgraph
- Klassische Lösungsverfahren
  - Intervallhalbierung
  - Sekantenmethode und Regula Falsi
  - Newton-Verfahren (heißt auch Newton-Raphson-Verfahren)
  - Fixpunkt-Iteration

1.3 Beispiele zum Aufwärmen

In den Übungen und in der Vorlesung diskutieren wir Beispiele der folgenden Art. Auch die folgenden Abschnitte 1.5 und 1.6 bringen weitere Erklärungen.

[4] Allerdings klingt „Wurzel“ statt „Lösung“ oder „Nullstelle“ im heutigen Fachdeutsch eher veraltet; im Englischen ist *root of a polynomial* der gängige Fachausdruck, und auch *root of a function or an equation* ist neben *zero of a function or solution of an equation* durchaus üblich.

Aus der Finanzmathematik

Ein Kredit von 100.000 € soll in 180 Monatsraten zu je 900 € zurückgezahlt werden. Was ist der Zinssatz bei diesen Konditionen?

Die Rentenformel für nachschüssige Zahlung liefert für den (monatlichen) Aufzinsungsfaktor q die Gleichung

$$900 = 100\,000\frac{q-1}{1-q^{-180}}. \tag{1}$$

Zustandsgleichung eines realen Gases

Wie groß ist das Molvolumen von Stickstoff bei 20 C und 1 bar $= 10^5$ Pa nach der Van der Waals-Gleichung?

Die Zustandsgleichung

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT$$

beschreibt den Zusammenhang zwischen Druck p, Molvolumen V_{mol} und Temperatur T. Die Konstanten a und b haben für Stickstoff die Werte

$$a = 0{,}129\,\mathrm{Pa\,m^6/mol^2}, \quad b = 38{,}6 \times 10^{-6}\,\mathrm{m^3/mol}.$$

Die molare Gaskonstante ist $R = 8{,}3145\,\mathrm{J/molK}$. Nach Einsetzen der Zahlenwerte verbleibt als Gleichung für V_{mol}:

$$\left(100\,000 + \frac{0{,}129}{V_{mol}^2}\right)(V_{mol} - 0{,}000\,038\,6) = 2437{,}4 \tag{2}$$

Widerstände in Rohrleitungen

Die Rohrreibungszahl λ hängt von der Reynoldszahl Re ab. Bei laminarer Strömung gilt einfach $\lambda = 64/\mathrm{Re}$. Im turbulenten Bereich, ab etwa $\mathrm{Re} > 2000$, listen technische Handbücher verschiedene, teilweise empirische Formeln für λ. Auf theoretischem Weg hat Prandtl für ein glattes Rohr die Beziehung

$$\lambda = \frac{1}{(2\log_{10}(\mathrm{Re}\sqrt{\lambda}) - 0{,}8)^2} \tag{3}$$

abgeleitet, die bis $\mathrm{Re} = 3{,}4 \times 10^6$ mit Versuchen übereinstimmt. Wie groß ist λ bei $\mathrm{Re} = 1 \times 10^6$?

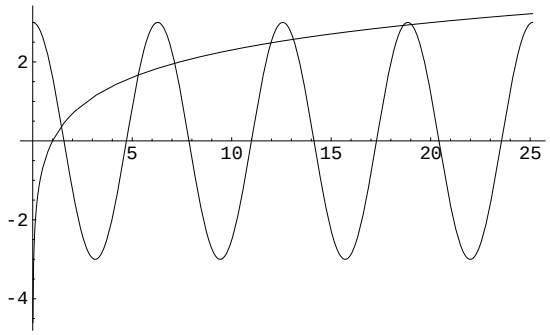


Abbildung 1: Schaubild zur Gleichung $3\cos x = \log x$. Den x-Werten der Schnittpunkte der Funktionsgraphen entsprechen die Lösungen der Gleichung.

Ohne tiefere Bedeutung

Es ist gut, wenn die bisherigen Beispiele den Eindruck einer gewissen Praxisnähe vermittelt haben. Der technischen Hintergrund und damit verbundene Verständnisschwierigkeiten verstellen aber den Blick auf die mathematischen Inhalte. Sie lernen hier nicht Physik, sondern numerische Verfahren, und die lassen sich leichter an einfachen Musterbeispielen illustrieren. Deswegen:

Finden Sie die Lösungen der Gleichung

$$3\cos x = \log x \tag{4}$$

Wichtiger Hinweis: hier meint log natürlich den natürlichen Logarithmus[5]. Argumente in Winkelfunktionen sind immer im Bogenmaß einzusetzen!

1.4 Graphische Lösung: Ein Bild sagt mehr als tausend Formeln

Entsprechend der Checkliste aus Kapitel 1.2 verschaffen wir uns am Beispiel von Gleichung 4 einen ersten Überblick. Diese Gleichung läßt nicht unmittelbar erkennen, ob, wo und wieviele Lösungen sie hat. Da sowohl Cosinus als auch Logarithmus geläufige Funktionen sind, bietet sich eine graphische Darstellung an. (Abbildung 1). Aus dem Schaubild lässt sich die Anzahl und ungefähre Lage der Lösungen erkennen. Rechenprogramme, die Wertetabellen berechnen oder in einen Funktionsgraphen hineinzoomen können, liefern rasch brauchbare Werte (die Checkliste nennt diese Vorgangsweisen „Trivialmethoden“).

1.5 Passende Umformungen: Nullstellen und Fixpunkte

Die Lösungen der Gleichung $3\cos x = \log x$ sind genau die Nullstellen der Funktion $f(x) = 3\cos x - \log x$. Ein Vergleich von Abbildung 1 mit Abbildung 2 stellt diesen Sachverhalt klar

[5] Für den dekadischen Logarithmus sprechen außer der evolutionsbedingten Zufälligkeit, dass Menschen zehn Finger haben, keine Argumente. Für Leute, die nicht bis drei zählen können, ist die Basis $e = 2{,}718\,281\,8\ldots$ ohnedies natürlicher.

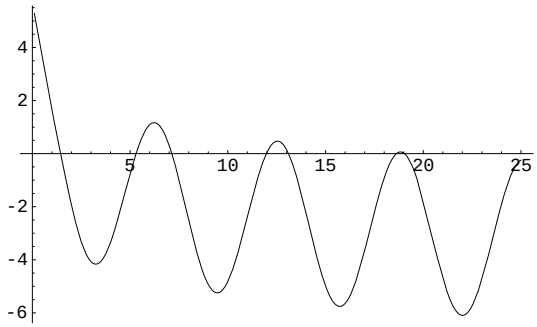


Abbildung 2: Schaubild zur Funktion $f(x) = 3\cos x - \log x$. Die Nullstellen von f entsprechen den x-Werten der Schnittpunkte in der Abbildung 1

und zeigt zum Beispiel: In der Nähe von $x = 5$, jedenfalls im Bereich $4 < x < 6$, muss eine der Nullstellen von f liegen.

Welche Form der graphischen Darstellung man günstigerweise wählt, hängt von der gegebenen Gleichung ab. In diesem Beispiel lassen sich cos und log als bekannte Funktionen leicht skizzieren, deswegen ist die Darstellung der Lösung durch die (x-Werte der) Schnittpunkte zweier Kurven übersichtlich. Andererseits lässt die Darstellung von $f(x) = 3\cos x - \log x$ die Nullstellen unmittelbar erkennen. Die klassischen Methoden zum Finden von Nullstellen ab Kapitel 1.7 erfordern ohnedies eine solche Umformung der Gleichung.

Die Gleichung $3\cos x = \log x$ lässt sich aber auch beispielsweise umformen zu

$$x = \arccos \frac{\log x}{3} \quad . \tag{5}$$

In dieser Form liegt eine Fixpunkt-Aufgabe $x = \phi(x)$ vor, mit $\phi(x) = \arccos((\log x)/3)$.

Fixpunkt-Iteration

Was passiert, wenn man auf der rechten Seite von Gleichung 5 einen Wert für x einsetzt, den Ausdruck ausrechnet und das Ergebnis wieder in der rechten Seite einsetzt? Beginnend etwa mit $x = 1$ liefert dieses Verfahren die Folge

$$1; \quad 1{,}5708; \quad 1{,}419\,69; \quad 1{,}453\,72; \quad 1{,}445\,76; \quad 1{,}447\,61; \quad 1{,}447\,18\ldots$$

Die Folge konvergiert gegen $\xi = 1{,}447\,258\,6$, das ist die kleinste Lösung der gegebenen Gleichung und gleichzeitig der einzige Fixpunkt der Funktion

$$\phi(x) = \arccos \frac{\log x}{3}.$$

Sie sehen hier ein Beispiel einer *Fixpunkt-Iteration*.

Fixpunkt-Iteration
Gegeben eine Gleichung $x = \phi(x)$.

> Beginne mit einem Startwert
> Setze Wert auf rechter Seite der Formel ein und werte aus
> Setze das Ergebnis wieder und wieder rechts in die Formel ein, bis sich die Resultate nicht mehr ändern

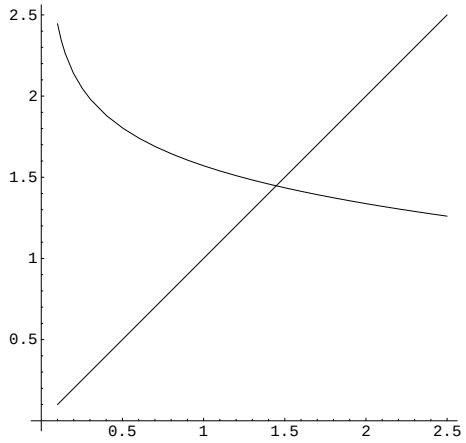


Abbildung 3: Schaubild zur Fixpunktaufgabe mit der Funktion $\phi(x) = \arccos((\log x)/3)$. Der Fixpunkt von ϕ entspricht der Nullstelle von f in der Nähe von 1,4. Weitere Fixpunkte von ϕ gibt es nicht. Durch die Umformulierung sind alle anderen Lösungen der ursprünglichen Gleichung verlorengegangen!

Weitere Beispiele von Fixpunkt-Iterationen:

- Geben Sie eine Zahl in den Taschenrechner ein und drücken Sie wiederholt auf die Wurzeltaste. Die Ergebnisse konvergieren gegen 1 (Fixpunkt von $f(x) = \sqrt{x}$).
- Geben Sie eine Zahl < 20 in den Taschenrechner ein und drücken Sie abwechselnd wiederholt auf die Tasten exp und $1/x$. Die Ergebnisse (nach dem $1/x$-Schritt) konvergieren gegen 0,567 14 (Fixpunkt von $f(x) = 1/\exp x$).
- Berechnen von Quadratwurzeln war schon in der griechischen Antike ein wichtiges Problem und (für rationale Zahlen) gelöst. Die Wurzel aus a ist definiert als Lösung von $x^2 = a$; eine für $x \neq 0$ äquivalente Umformung dieser Gleichung ist

  $$x = \frac{1}{2}\left(x + \frac{a}{x}\right) \ .$$

  Schon den Babyloniern soll die oft als Heron-Verfahren bezeichnete Iteration

  $$x^{(0)} = a; \quad x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \text{ für } k = 0, 1, 2, \ldots$$

  bekannt gewesen sein.
- Gleichung 3 ist eine Fixpunkt-Gleichung. Mit dem Startwert 0,05 liefern wenige Fixpunkt-Iterationen eine genaue Lösung.

Aber es funktioniert nicht immer: Eine andere mögliche Fixpunkt-Form von Gleichung 4 lautet

$$x = \exp(3\cos x) \ .$$

Wenn Sie hier $x = 1$ rechts einsetzen und das für die Ergebnisse jeweils wiederholen, erhalten Sie die Folge

$$1; \quad 5{,}057\,68; \quad 2{,}760\,46; \quad 0{,}061\,745\,5; \quad 19{,}971; \quad 3{,}6805\ldots$$

Ihre Werte wechseln unregelmäßig und konvergieren nicht.

Zusammenfassung

Nicht jede Fixpunkt-Iteration konvergiert. Passende Umformungen sind nicht immer leicht zu finden. Anderseits sind viele numerische Verfahren vom Typ einer Fixpunkt-Iteration. Das rechtfertigt eine ausführliche theoretische Untersuchung solcher Verfahren im Kapitel 1.12.

1.6 Diskussion der Beispiele: Wichtige und unwichtige Terme

Hier werden die in Kapitel 1.1 vorgestellten Beispiele ausführlich besprochen.

1.6.1 Eine fast lineare Gleichung

Die am Anfang von Kapitel 1.1 erwähnte Gleichung

$$x^2 - 12345678x + 9 = 0$$

ist, wenn es um die betragskleinere der beiden Lösungen geht, eigentlich keine quadratische Gleichung! Begründung: Die gesuchte Lösung ist von der Größenordnung 10^{-6} bis 10^{-7}; der Term x^2 in der Gleichung ist also gegenüber dem linearen Term $12345678x$ um mehr als zehn Größenordnungen kleiner. Für alle praktischen Zwecke ist eine solche Gleichung linear mit einem kleinen quadratischen Korrekturterm. Lösen Sie daher nach dem linearen Term auf:

$$x = \frac{1}{12345678}(x^2 + 9) \; .$$

Der Startwert $x^{(0)} = 0$ liefert selbst auf den billigsten Taschenrechnern ohne Wurzeltaste bereits ein bessere Näherung $x^{(1)} = 7{,}290\,000\,597\,78 \times 10^{-7}$ als die meisten Rechner durch Anwendung der Standard-Lösungsformel erreichen können.

> Locker formuliert: Viele Gleichungen enthalten Terme, in denen die Unbekannte zwar auftritt, aber im Vergleich zu anderen Termen wenig Einfluss hat. Wenn eine Gleichung dadurch leichter lösbar wird, lassen sich solche Terme in erster Näherung vernachlässigen. In weiteren Schritten korrigiert man das Ergebnis, indem man Näherungswerte in den anfangs vernachlässigten Termen einsetzt.

1.6.2 Van der Waals-Gleichung

Die Gleichung (2) lässt sich zu einer kubischen Gleichung umformen,

$$-4.9794 \cdot 10^{-6} + 0.129 V_{mol} - 2441.3 V_{mol}^2 + 100000 V_{mol}^3 = 0 \quad , \tag{6}$$

und wäre damit im Prinzip analytisch lösbar. Tun Sie 's nicht! Ein wenig Einsicht in den physikalischen Hintergrund dieser Gleichung legt eine andere Vorgangsweise nahe: Bei Zimmertemperatur ist Stickstoff nahezu ein ideales Gas, das der Gleichung

$$pV_{mol} = RT$$

gehorcht. In der Van der Waals-Gleichung

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT \tag{7}$$

ist der Term a/V_{mol}^2 eine Korrektur der idealen Gasgleichung und für die im Beispiel gegebenen Parameter gegenüber p vernachlässigbar klein. Dem umgeformten Polynom (6) sieht man es

nicht an, aber die ursprüngliche Gleichung (7) entspricht – im Bereich der gegebenen Daten – nicht einer „richtigen“ kubischen Gleichung, sondern vielmehr einer lineare Gleichung in V_{mol} plus einem kleinen Korrekturterm a/V_{mol}^2.

Daher lässt sich diese Gleichung auflösen, wenn man „unwichtige“ Terme der Unbekannten auf der rechten Seite stehen lässt. Hier formen wir um zu

$$V_{mol} = \frac{RT}{p + a/V_{mol}^2} + b = \frac{2437{,}4}{100000 + 0{,}129/V_{mol}^2} + 0{,}000\,038\,6$$

und ignorieren wir erst einmal den Korrekturterm a/V_{mol}^2. Das liefert eine nullte Näherung für das Molvolumen,

$$V_0 = \frac{2437{,}4}{100000} + 0{,}000\,038\,6 = 0{,}024\,413\ .$$

Der Trick ist nun, diese Näherung für V_{mol} in der rechten Seite der Gleichung einzusetzen und daraus eine verbesserte Näherung

$$V_1 = \frac{2437{,}4}{100000 + 0{,}129/0{,}024\,413^2} + 0{,}000\,038\,6 = 0{,}024\,360$$

zu berechnen. Wiederholtes Einsetzen liefert keine weitere Verbesserung:

$$V_2 = \frac{2437{,}4}{100000 + 0{,}129/0{,}024\,360^2} + 0{,}000\,038\,6 = 0{,}024\,360\ .$$

Damit haben wir (jedenfalls auf fünf Dezimalstellen genau) den Wert $V_{mol} = 0{,}024\,360\,\mathrm{m}^3$ bestimmt.

> Bußübung für die Fastenzeit: Schlagen Sie in Wikipedia die Cardanischen Formeln nach und lösen Sie die Aufgabe damit. Vergleichen Sie den Zeitaufwand mit der obigen Methode.

1.6.3 Finanzmathematik

In Gleichung 1 erwarten wir für den Aufzinsungsfaktor q einen Wert knapp über 1. Den Term q^{-180} im Nenner wird vermutlich $\ll 1$ und nicht so wichtig sein. Das motiviert, die Gleichung nach dem q im Zähler aufzulösen.

$$q = 1 + \frac{900}{100000}(1 - q^{-180})$$

Ignoriert man q^{-180} auf der rechten Seite, dann folgt als nullte Näherung

$$q_0 = 1 + \frac{900}{100000} = 1{,}009$$

Auch hier funktioniert der Trick, q_0 in der rechten Seite einzusetzen und daraus eine verbesserte Näherung

$$q_1 = 1 + \frac{900}{100000}(1 - 1{,}009^{-180}) = 1{,}007\,206$$

zu berechnen. Wiederholtes Einsetzen liefert

$$q_2 = 1{,}006\,529 \quad q_3 = 1{,}006\,210 \quad q_4 = 1{,}006\,047\ldots$$

Es braucht aber hier insgesamt 14 Iterationen, bis sich die Werte bei $q = 1{,}005\,851$ stabilisieren.

Bemerkungen zum Abschluss

Ist eine Gleichung in der Form $f(x) = g(x)$ gegeben (Beispiel: Gleichung 4), lässt sich nicht unmittelbar erkennen, welche Terme „wichtig“ oder „unwichtig“ sind. Regel: man löse nach jener Seite der Gleichung auf, welcher den *steileren* Funktionsgraph im Schnittpunkt hat.

Passende Umformungen für Fixpunkt-Iterationen erfordern oft ein tieferes Verständnis der einzelnen Terme in einer Gleichung. Es gibt zum Glück Lösungsverfahren, die mehr nach „Schema F“ ablaufen. Eines davon stellt das nächste Kapitel vor.

1.7 Intervallhalbierung

Kennen Sie die Geschichte von den zwei Möglichkeiten? Sie beginnt mit dem Zwischenwertsatz.

Zwischenwertsatz

Eine Funktion f, die auf einem abgeschlossenen Intervall $[a, b]$ stetig ist, nimmt in diesem Intervall auch jeden Wert zwischen $f(a)$ und $f(b)$ an.

Ist f insbesondere für $x = a$ negativ und für $x = b$ positiv (oder umgekehrt), dann garantiert der Zwischenwertsatz: f hat mindestens eine Nullstelle in diesem Intervall.

Es gibt immer zwei Möglichkeiten...

Angenommen, wir suchen eine Nullstelle einer im Bereich $a \leq x \leq b$ stetigen Funktion. Es lässt sich rechnerisch sofort prüfen, ob $f(a)$ und $f(b)$ unterschiedliches Vorzeichen haben. Wenn ja, dann garantiert der Zwischenwertsatz die Existenz eine Nullstelle im Bereich $a \leq x \leq b$, aber wir wissen nicht, wo sie liegt. Nun gibt es zwei Möglichkeiten: Entweder ist $b - a$ klein, dann ist es gut: Wir können sowohl a als auch b als Näherung für eine Nullstelle von f auffassen. Andernfalls berechnen wir den Mittelpunkt c des Intervalls, $c = (a + b)/2$. Nun gibt es wieder zwei Möglichkeiten. Ist $f(c) = 0$, so ist es gut: es liegt dort eine Nullstelle vor. Anderenfalls hat f an den Enden eines der Teilintervalle $a \leq x \leq c$ oder $c \leq x \leq b$ verschiedene Vorzeichen (klar? Das ist der springende Punkt!). In einem der beiden Intervalle muss also eine Nullstelle liegen. Betrachten wir dieses Intervall und nennen wir der Einfachheit die neuen Intervallgrenzen wieder a und b.

Nun gibt es zwei Möglichkeiten: Entweder ist $b-a$ klein, dann ist es gut: Wir können sowohl a als auch b als Näherung für eine Nullstelle von f auffassen. Andernfalls bilden wir $c = (a+b)/2$. Nun gibt es wieder zwei Möglichkeiten...

Sie können nun die Geschichte selber fortsetzen. Beachten Sie aber, dass die Intervalllänge in jedem Erzählschritt halbiert wird. Für jede beliebig klein vorgegebene Genauigkeitsschranke $\epsilon > 0$ erreichen Sie nach einer endlichen Anzahl von Schritten ein Intervall mit Länge $b-a < \epsilon$. Damit endet die Geschichte wie im wirklichen Leben: Es gibt immer zwei Möglichkeiten, aber jede Entscheidung schränkt den Freiraum für weitere Aktionen ein. Irgendwann sind die Alternativen dann doch ausgeschöpft.

Formalisiert angeschrieben, lautet dieses Verfahren

Intervallhalbierung (Bisektionsverfahren)
Gegeben eine Funktion f, zwei Werte a und b mit $f(a) \cdot f(b) < 0$, eine Fehlerschranke $\epsilon > 0$. Ist f im Intervall $a \leq x \leq b$ stetig, dann findet dieser Algorithmus die Näherung c an eine Nullstelle ξ von f mit Fehler $|c - \xi| < \epsilon$.

Wiederhole
  setze $c \leftarrow (a + b)/2$
  falls $f(a) \cdot f(c) < 0$
    setze $b \leftarrow c$
  sonst
    setze $a \leftarrow c$
bis $|b - a| < \epsilon$ oder $f(c) = 0$

Lineare Konvergenz

Die beste Schätzung für den Wert der Nullstelle ist der Mittelpunkt des Intervalls. Der maximale Fehlerbetrag ist dann durch $\epsilon_0 \leq |b-a|/2$ beschränkt; größer als die halbe Intervallbreite kann er nicht sein. Intervallhalbierung reduziert diese Fehlerschranke pro Schritt um den Faktor $1/2$ oder, da

$$\left(\frac{1}{2}\right)^{3{,}3} \approx \frac{1}{10} \quad ,$$

um einen Faktor $1/10$ pro (durchschnittlich) 3,3 Schritten. Man kann sagen: Intervallhalbierung produziert eine korrekte Dezimalstelle pro 3,3 Iterationen. Der maximale Fehler nach dem i-ten Schritt, ϵ_i, ist höchstens halb so groß wie der vorherige maximale Fehler ϵ_{i-1}. Es gilt also

$$\epsilon_i \leq C\epsilon_{i-1} \quad \text{mit } C = \frac{1}{2} \; .$$

Allgemein: Wenn bei einem Verfahren für die Fehlerschranken aufeinanderfolgender Iterationsschritte gilt

$$\epsilon_i \leq C\epsilon_{i-1} \quad \text{mit } C < 1 \; .$$

spricht man von *linearer* Konvergenz.

Vor- und Nachteile

Vorteile der Intervallhalbierung: einfach zu verstehen, leicht zu programmieren. Wenn die Voraussetzungen erfüllt sind, konvergiert es mit Sicherheit. Es ist ein *Einschlussverfahren*, das heißt, es liefert nicht nur einen Näherungswert, sondern grenzt die Lösung von beiden Seiten her ein.

Nachteile: Man braucht Startwerte–aber das ist ein Problem jedes numerischen Verfahrens. Intervallhalbierung ist langsam; nur lineare Konvergenz – die dafür aber sicher.

1.8 Regula Falsi (lineares Eingabeln)

Funktionen, die in der Umgebung der Nullstelle glatt verlaufen, lassen sich dort durch eine Gerade annähern. Statt, wie bei der Intervallhalbierung, den Wert c genau in der Mitte zwischen a und b anzunehmen, wählen wir c als Nullstelle der Gerade durch $(a, f(a))$ und $(b, f(b))$, siehe Abbildung 4.

$$c = a - f(a)\frac{a-b}{f(a)-f(b)} = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

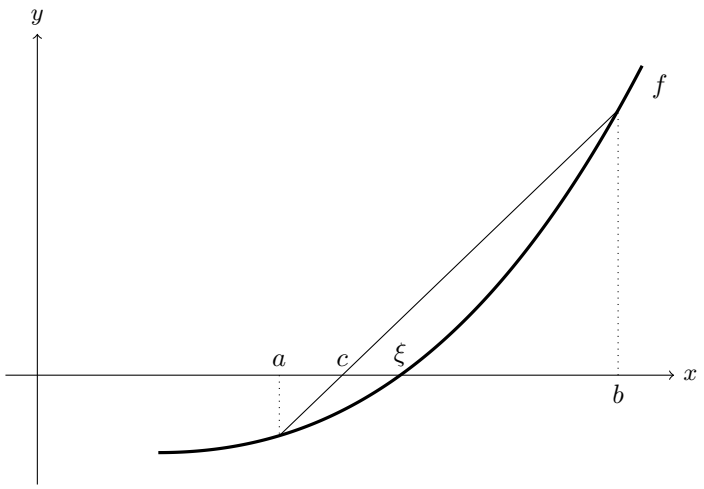


Abbildung 4: Die Regula Falsi berechnet c, die Nullstelle der Verbindungsgeraden, als Näherungswert an die Nullstelle ξ der Funktion f.

Regula Falsi (lineares Eingabeln)
Gegeben eine Funktion f, zwei Werte a und b mit $f(a) \cdot f(b) < 0$ und eine Genauigkeitsschranke $\epsilon > 0$. Ist $f(x)$ im Intervall $a \leq x \leq b$ stetig, dann findet dieser Algorithmus[a] eine Näherung c an eine Nullstelle ξ von f mit Genauigkeit $|c - \xi| < \epsilon$.

Wiederhole
  setze $c \leftarrow a - f(a)\dfrac{a-b}{f(a)-f(b)}$
  falls $f(b) \cdot f(c) < 0$
    setze $a \leftarrow b$
  sonst
    (klassische Version) nix
    (Illinois-Variante) reduziere $f(a)$ auf $\frac{1}{2}f(a)$
    (Pegasus-Variante) reduziere $f(a)$ auf $\dfrac{f(a)f(b)}{f(b)+f(c)}$
  setze $b \leftarrow c$
bis $|b-a| < \epsilon$ oder $f(c) = 0$

[a] mit dem hier gegebenen Abbruchkriterium allerdings nur die beiden Varianten

Allerdings bringt die Regula Falsi in der Standard-Version im Vergleich zur Intervallhalbierung kein wesentlich besseres Konvergenzverhalten. Typischer Weise bleibt nach einigen Iterationen die Intervallgrenze a fix, die andere Grenze b konvergiert zwar zur Nullstelle, aber die Abbruchbedingung $|b - a| < \epsilon$ wird nicht erreicht. Sorgfältige Programmierer würden im obigen Algorithmus jedenfalls noch eine Notbremse einbauen: zähle die Anzahl der Iterationen mit und brich ab, wenn eine Maximalzahl überschritten wird.

Die Illinois- oder die Pegasus-Variante verbessern das Konvergenzverhalten im Vergleich zur Intervallhalbierung deutlich; mutige Programmierer würden in diesem Fall auf die Abfrage

nach einer maximalen Iterationszahl verzichten.

Intervallhalbierung und die verschiedenen Regula-Falsi-Versionen haben gemeinsam, daß sie die Nullstelle von beiden Seiten her „eingabeln“ — sie sind Einschlussverfahren, das ist gut. Nachteilig ist, dass man zu Beginn des Verfahrens zwei Näherungswerte braucht, und zwar je einen auf jeder Seite der Nullstelle. Das kann sehr schwer zu erreichen sein, wenn man zwei nahe beisammen liegende Nullstellen hat, da dann eine der ursprünglichen Näherungen dazwischen liegen muß. Mehrfache Nullstellen gerader Ordnung können diese Verfahren überhaupt nicht finden.

> Was ist „falsch“ an der Regula Falsi? Natürlich nicht die Regel selbst, sondern die angenommenen Startwerte a und b. Aus diesen beiden „falschen Lösungen“ berechnet die Regel eine bessere Näherungslösung.
> Die Methode ist uralt, die Grundidee war schon Jahrhunderte vor Chr. weltweit bekannt: Babyloniern, Ägypter, Inder und Chinesen lösten damit lineare Gleichungen. Aus arabischen Quellen nach Europa bringt sie um 1200 Leonardo von Pisa, genannt Fibonacci. Er beschreibt mehrere Varianten, darunter die *regula duarum falsarum positionum*, die „Methode vom doppelten falschen Ansatz“. So sollte sie auch richtiger Weise heißen, aber es hat sich schlampig verkürzt „Regula Falsi“ durchgesetzt.
> Fibonacci löste damit nur lineare Probleme; da berechnet die Regel aus zwei falschen Startwerten sofort die richtige Lösung. Die Anwendung als iteratives Verfahren für Nullstellen nichtlinearer Funktionen ist dann doch nicht so alt. Mitte des vorigen Jahrhunderts fand man kleine, aber nicht unwesentliche Verbesserungen der Rechenregel (Pegasus-, Illinois-Variante). Sogar noch kürzlich, 2020, veröffentlichten Oliveira und Takahashi eine weitere Verbesserung (`https://en.wikipedia.org/wiki/ITP_method`).

1.9 Sekantenmethode

Die Sekantenmethode berechnet gleich wie die Regula Falsi eine neue Näherung durch lineare Interpolation, verlangt aber nicht, dass die Werte a und b die Nullstelle einschließen, siehe Abbildung 5.

Die formale Beschreibung des Verfahrens bezeichnet hier die Startwerte a und b mit $x^{(0)}$ und $x^{(1)}$ und die weiteren iterativ berechneten Näherungswerte mit $x^{(k)}, x^{(k+1)}, \ldots$

Sekantenmethode
Gegeben eine Funktion f, zwei Werte $x^{(0)}$ und $x^{(1)}$, eine Genauigkeitsschranke $\epsilon > 0$ und eine maximale Iterationsanzahl k_{max}. Für hinreichend gute Startwerte $x^{(0)}$ und $x^{(1)}$ findet dieser Algorithmus die Näherung $x^{(k)}$ an eine Nullstelle ξ von f mit Genauigkeit $|x^{(k)} - \xi| \approx \epsilon$ oder bricht nach einer Maximalzahl von k_{max} Schritten ab.

setze $k = 1$
Wiederhole
  setze $x^{(k+1)} = x^{(k)} - f(x^{(k)}) \dfrac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})}$
  erhöhe $k = k + 1$
bis $|x^{(k+1)} - x^{(k)}| < \epsilon$ oder $k \geq k_{max}$

Superlineare Konvergenz

Die Sekantenmethode zeigt *superlineare* Konvergenz. (Notwendige technische Details: f zweimal stetig differenzierbar, keine mehrfache Nullstelle.) Das heißt, für die Fehlerschranken $|x^{(k+1)} - \xi|$ und $|x^{(k)} - \xi|$ aufeinanderfolgender Schritte gilt, sofern $|x^{(k)} - \xi|$ schon hinreichend klein ist:

$$|x^{(k+1)} - \xi| \leq C|x^{(k)} - \xi|^p \quad \text{mit } p > 1 \ .$$

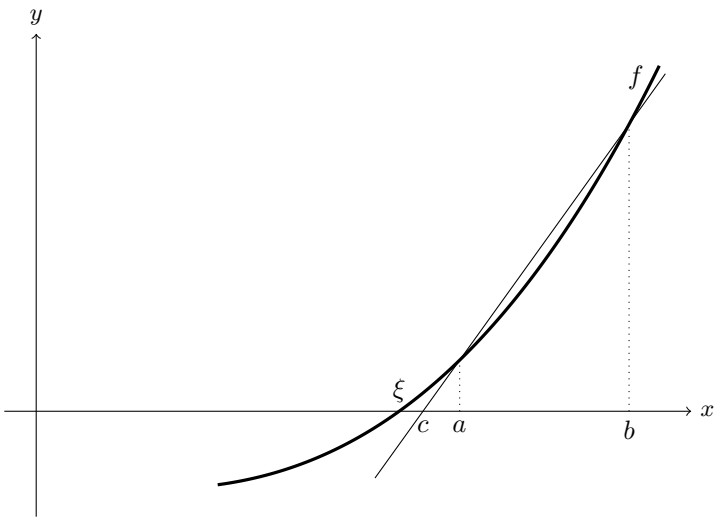


Abbildung 5: Die Sekantenmethode berechnet den nächsten Näherungswert c mittels einer Schnittgeraden (Sekante) durch zwei Punkte des Funktionsgraphen. Die beiden Werte a und b schließen die Nullstelle ξ jedoch nicht unbedingt ein.

Der Fehler reduziert sich also nicht bloß um einen Faktor C, sondern zusätzlich noch mit der Potenz p. Für die Sekantenmethode lässt sich zeigen

$$p = \frac{1+\sqrt{5}}{2} \approx 1{,}618 \ .$$

Angenommen, es ist $|x^{(k)} - \xi| = 0{,}01$. Überlegen Sie sich, was den Fehler stärker verringert: Multiplikation mit einem Faktor $C = 1/2$, oder Potenzieren mit $p = 1{,}6$!

1.10 Newton-Verfahren

Heißt auch Newton-Raphson-Verfahren, aber erst einige Jahrzehnte nach Isaac Newton und Joseph Raphson formuliert Thomas Simpson das Verfahren so, wie wir es heute kennen.

Gesucht sei eine Nullstelle der Funktion f. Gegeben sei ein Startwert $x^{(0)}$ in der Nähe der Nullstelle. Das Newton-Verfahren versucht, ähnlich der Sekantenmethode, die Funktion f durch eine lineare Funktion anzunähern und verwendet dazu die Tangente an f im Punkt $(x^{(0)}, f(x^{(0)}))$. Der Schnittpunkt der Tangente mit der x-Achse ist der nächste Näherungswert, siehe Abbildung 6.

Herleitung aus der Taylorentwicklung von f um den Punkt $x^{(0)}$. Ist f genügend oft differenzierbar, dann gilt:

$$f(x) = f(x^{(0)}) + (x - x^{(0)})f'(x^{(0)}) + \frac{(x-x^{(0)})^2}{2!}f''(x^{(0)}) + \dots$$

Es soll gelten $f(x) = 0$. Vernachlässigen von Gliedern höherer Ordnung liefert die Gleichung

$$0 = f(x^{(0)}) + (x - x^{(0)})f'(x^{(0)}) \ ,$$

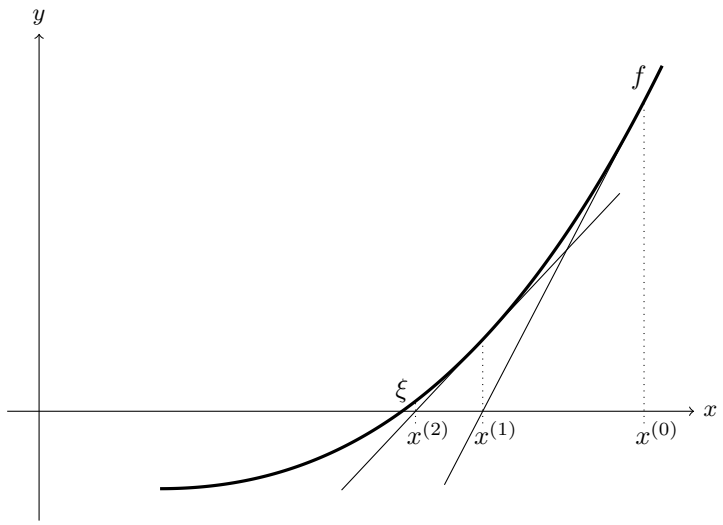


Abbildung 6: Graphische Deutung des Newton-Verfahrens: Die Tangente an f im Punkt $(x^{(0)}, f(x^{(0)}))$ schneidet die x-Achse in $x^{(1)}$. Der Wert $x^{(2)}$ im nächsten Schritt liegt schon nahe an der Nullstelle ξ.

aus der sich x ausdrücken lässt:

$$x = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} \ .$$

Newton-Verfahren
Gegeben eine differenzierbare Funktion f und ein Startwert $x^{(0)}$.
Gesucht eine Nullstelle von f.

Iterationsvorschrift

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \qquad \text{für } k = 0, 1, 2 \ldots$$

Quadratische Konvergenz

Das Newton-Verfahren zeigt *quadratische* Konvergenz. Das heißt, für die Fehlerschranken $\epsilon_{k+1} = |x^{(k+1)} - x|$ und $\epsilon_k = |x^{(k)} - x|$ aufeinanderfolgender Schritte gilt, sofern ϵ_k schon hinreichend klein ist:

$$\epsilon_{k+1} \leq C\epsilon_k^2$$

Der neue Fehler ist also um einen Faktor C kleiner als das *Quadrat* des alten Fehlers. Der genaue Wert von C ist dabei nicht so wichtig.

Angenommen, es ist $\epsilon_k = 10^{-4}$. Das heißt, der Fehler beträgt eine Einheit in der vierten Nachkommastelle. Dann gilt bei quadratischer Konvergenz $\epsilon_{k+1} = C \cdot 10^{-8}$. Der Fehler beträgt also C Einheiten in der achten Nachkommastelle. Wenn C größenordnungsmäßig im Bereich 1 ist, hat sich die Anzahl der korrekten Stellen ungefähr verdoppelt.

Quadratische Konvergenz: Neuer Fehler $\sim$ Quadrat des alten Fehlers.

Faustregel: Sofern schon einige signifikante Stellen exakt sind, sind im nächsten Näherungswert etwa doppelt so viele signifikante Stellen korrekt.

1.11 Abbruchbedingungen

Rechner haben nur eine fixe Zahl von Binärstellen zur Verfügung, um Gleitkommazahlen zu speichern. Möglicherweise erreicht $f(x)$ für kein Gleitkomma-Argument x exakt den Wert Null. Wenn die Nullstelle ξ in der Gegend von 1 liegt, können Sie leicht eine Näherung x mit absolutem Fehler $|x - \xi| < 10^{-6}$ finden. Liegt die Nullstelle um $\xi \approx 10^{22}$, werden Sie einen absoluten Fehler dieser Güte nicht erreichen können. Eine übliche Wahl der Abbruchschranke ϵ ist $\varepsilon_m(|a| + |b|)/2$, wenn ε_m die Maschinengenauigkeit und a, b die ursprünglichen Intervallgrenzen sind. Wenn a, b und die Nullstelle selber nahe bei Null liegen, ist Vorsicht bei dieser Formel geboten. Die Abbruchschranke darf jedenfalls nicht kleiner als die kleinste positive Maschinenzahl sein (typischerweise um 10^{-38} für 4-Byte-Datentypen, 10^{-308} für 8-Byte-Datentypen).

Maschinengenauigkeit

Die Maschinengenauigkeit ε_m ist die kleinste positive Gleitkommazahl, die, zur Gleitkommazahl 1,0 addiert, eine von 1,0 verschiedene Summe ergibt (typischerweise um 10^{-7} für 4-Byte-Datentypen, 10^{-16} für 8-Byte-Datentypen).

1.12 Fixpunkt-Iteration

Im Abschnitt 1.5 haben wir bereits Fixpunkte von Funktionen durch wiederholtes Einsetzen bestimmt. Viele numerische Verfahren lassen sich als Spezialfälle einer Fixpunkt-Iteration betrachten. Aussagen über die Konvergenz von Fixpunkt-Iterationen sind deswegen von allgemeiner Bedeutung.

Fixpunkt-Iteration
Gegeben eine Funktion ϕ und ein Startwert $x^{(0)}$.
Gesucht ein Fixpunkt ξ von ϕ.

Iterationsvorschrift
$x^{(k+1)} = \phi(x^{(k)})$ für $k = 0, 1, 2 \ldots$

Fixpunkt-Iteration konvergiert für kontrahierende Abbildungen
Die Funktion ϕ besitze einen Fixpunkt ξ Sei ferner I ein offenes Intervall der Form $(\xi - r, \xi + r)$ um den Fixpunkt ξ, in dem ϕ als *kontrahierende Abbildung* wirkt, d. h.

$$|\phi(x) - \phi(y)| \leq C|x - y| \text{ gilt mit } C < 1 \text{ für alle } x, y \in I \ .$$

Dann konvergiert für alle $x^{(0)} \in I$ die Fixpunkt-Iteration $x^{(k+1)} = \phi(x^{(k)})$ mindestens linear gegen ξ.

Beweis: Zuerst zeigt man durch Induktion: $x^{(k)} \in I$ für alle $k = 0, 1, 2 \ldots$ Die Aussage ist laut Voraussetzung richtig für $k = 0$. Angenommen, es liegt bereits $x^{(k)} \in I$, also weniger als r von ξ entfernt: $|x^{(k)} - \xi| < r$. Dann können wir die Kontraktionsbedingung und Fixpunkt-Eigenschaft für $x^{(k)}$ und ξ anwenden und erhalten

$$|x^{(k+1)} - \xi| = |\phi(x^{(k)}) - \phi(\xi)| \leq C|x^{(k)} - \xi| < Cr \; .$$

Da $C < 1$, ist also auch

$$|x^{(k+1)} - \xi| < r \quad \text{und somit} \quad x^{(k+1)} \in I$$

Aus diesen Überlegungen folgt auch unmittelbar für die Fehler $\epsilon^{(k)} = |x^{(k)} - \xi|$ und $\epsilon^{(k+1)} = |x^{(k+1)} - \xi|$:

$$\epsilon^{(k+1)} \leq C\epsilon^{(k)} \leq C^k \epsilon_0, \text{ somit } \epsilon^{(k+1)} \to 0 \text{ für } k \to \infty \; .$$

So wie der Satz hier formuliert ist, setzt er die Existenz eines Fixpunktes voraus. Dadurch wird der Konvergenz-Beweis kurz und schmerzlos. Eine etwas allgemeinere Formulierung und ein technisch aufwändigerer Beweis zeigen, dass aus der Kontraktions-Eigenschaft auch schon die Existenz und Eindeutigkeit eines Fixpunktes folgen. Das ist der berühmte Fixpunktsatz von Banach.

Zusammenhang kontrahierende Abbildung–Steigung der Funktion

Die Eigenschaft $|\phi(x) - \phi(y)| \leq C|x - y|$ bedeutet für $C < 1$ anschaulich: Funktionswerte unterscheiden sich weniger als die Eingabewerte. Wie stark sich Funktionswerte im Verhältnis zu Eingabewerten ändern, ist (im Grenzwert für kleine Änderungen) durch die Steigung der Funktion bestimmt.

Ist ϕ in einer Umgebung von ξ stetig differenzierbar und $|\phi'(\xi)| < 1$, so ist in einer Umgebung von ξ die Kontraktionseigenschaft erfüllt: Wegen der Stetigkeit von ϕ' gibt es ein offenes Intervall I um ξ, in dem $|\phi'| \leq C < 1$ gilt. Für $x, y \in I$ gilt nach dem Mittelwertsatz der Differentialrechnung

$$\phi(x) - \phi(y) = (x - y)\phi'(\eta) \quad \text{für ein } \eta \in I \; .$$

Damit ist auch

$$|\phi(x) - \phi(y)| \leq C|x - y| \; , \quad C < 1$$

Eine Kurzfassung dieser Aussage:

Abbildung 7 illustriert das Konvergenzverhalten der Fixpunkt-Iteration für verschiedene ϕ.

1.13 Konvergenzordnung

Wir haben lineare, superlineare und quadratische Konvergenz bereits erwähnt. Hier fassen wir den Begriff der Konvergenzordnung genauer.

Konvergenzordnung
Sei ξ Fixpunkt von ϕ, und es gelte für alle Startwerte aus einem Intervall um ξ und die zugehörige Folge $\{x^{(k)}\}$ aus der Vorschrift $x^{(k+1)} = \phi(x^{(k)}), k = 0, 1, 2 \ldots$

$$|x^{(k+1)} - \xi| \leq C|x^{(k)} - \xi|^p$$

mit $p \geq 1$ und $C < 1$, falls $p = 1$.
Das Iterationsverfahren heißt dann ein Verfahren von mindestens p-ter Ordnung

Für das lokale Konvergenzverhalten einer Fixpunkt-Iteration ist der Wert der ersten Ableitung am Fixpunkt maßgeblich. Für $|\phi'(\xi)| < 1$ ist lineare Konvergenz gesichert; je kleiner der Betrag der Ableitung, desto schneller konvergiert das Verfahren, wobei $C \approx |\phi'(\xi)|$. Ganz besonders rasche, nämlich superlineare Konvergenz tritt auf, wenn $|\phi'(\xi)| = 0$.

Mit Hilfe der Taylorentwicklung lässt sich zeigen: Ist $\phi(x)$ in einer Umgebung von ξ genügend oft differenzierbar und

$$\phi'(\xi) = 0,\ \phi''(\xi) = 0, \ldots, \phi^{(p-1)}(\xi) = 0, \text{ und } \phi^{(p)}(\xi) \neq 0\ ,$$

dann liegt für $p = 2, 3, \ldots$ ein Verfahren p-ter Ordnung vor. Ein Verfahren erster Ordnung liegt vor, wenn zu $p = 1$ gilt: $|\phi'(\xi)| < 1$.

1.14 Konvergenz des Newton-Verfahrens

Das Newtonverfahren, angewandt auf die Funktion f, entspricht einem Fixpunkt-Verfahren für die Funktion ϕ,

$$\phi(x) = x - \frac{f(x)}{f'(x)}$$

Nun ist

$$\phi'(x) = \frac{f''(x)f(x)}{(f'(x))^2}\ ,$$

und da an einer einfachen Nullstelle $f(x) = 0, f'(x) \neq 0$ gilt, verschwindet $\phi'(x)$ dort. Man überzeugt sich leicht, dass $\phi''(x) \neq 0$ gilt, sofern $f''(x) \neq 0$. Daraus folgt die quadratische Konvergenz des Newtonverfahrens bei einfachen Nullstellen. Bei mehrfachen Nullstellen lässt sich lineare Konvergenz nachweisen.

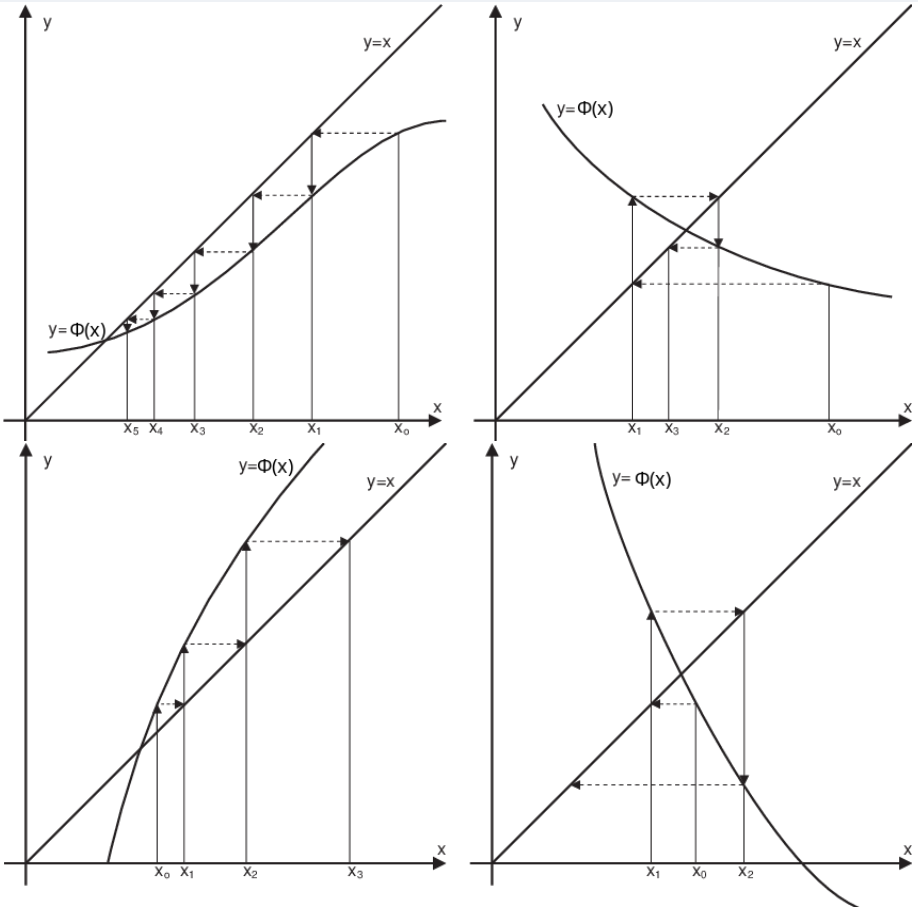


Abbildung 7: Fixpunkt-Iteration in graphischer Darstellung für verschiedene Funktionen ϕ. Mögliche Fälle: Einseitige Annäherung an den Fixpunkt, falls in einer Umgebung des Fixpunktes $0 < \phi' < 1$; alternierende Konvergenz, falls $-1 < \phi' < 0$, Divergenz falls $\phi' > 1$ oder $\phi' < -1$.

2 Systeme nichtlinearer Gleichungen

Abschnitt 1.2 definiert die Begriffe *Lösung, Nullstelle, Fixpunkt* für skalare Funktionen $\mathbb{R} \to \mathbb{R}$. Diese Begriffe lassen sich problemlos auf vektorwertige Funktionen $\mathbb{R}^n \to \mathbb{R}^n$ übertragen. Auch hier lassen sich Gleichungen auf verschiedene Weise formulieren.

Schreibweise für Vektoren und vektorwertige Funktionen: Fettdruck

Reellwertige Funktionen, Skalare: $f : \mathbb{R} \to \mathbb{R}$, $y = f(x)$
Vektorwertige Funktionen, Vektoren: $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, $\mathbf{y} = \mathbf{f}(\mathbf{x})$

Komponenten eines Vektors $\mathbf{x} \in \mathbb{R}^n$:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{oder } \mathbf{x}^T = [x_1, x_2, \ldots, x_n]$$

Normalerweise ist mit $\mathbf{x}$ ein Spalten-, mit $\mathbf{x}^T$ ein Zeilenvektor gemeint.

Iterationsindizes werden hier (um sie von Vektorkomponenten zu unterscheiden) hochgestellt und in Klammern gesetzt: $\mathbf{x}^{(k)}, k = 0, 1, 2 \ldots$

2.1 Lösung, Nullstelle und Fixpunkt: mehrdimensionaler Fall

Aufgabentypen im $\mathbb{R}^n$

Es seien $\mathbf{f}, \mathbf{g}, \mathbf{h}, \mathbf{\Phi}$ Funktionen $\mathbb{R}^n \to \mathbb{R}^n$ und $\mathbf{x} \in \mathbb{R}^n$

Problemstellung: gesucht ist ein $\mathbf{x}$, für das gilt...

$\mathbf{g}(\mathbf{x}) = \mathbf{h}(\mathbf{x})$, (Finden einer *Lösung* des Gleichungssystems)
$\mathbf{f}(\mathbf{x}) = 0$, (Finden einer *Nullstelle* der Funktion $\mathbf{f}$)
$\mathbf{x} = \mathbf{\Phi}(\mathbf{x})$, (Finden eines *Fixpunktes* der Funktion $\mathbf{\Phi}$)

Im Vergleich zu den Definitionen von Abschnitt 1.2 hat fast nichts geändert außer der Schreibweise.

Beispiel: ein *nichtlineares Gleichungssystem* mit zwei Unbekannten

$$\begin{aligned} 4x_1 - x_2 + x_1x_2 &= 1 \\ -x_1 + 6x_2 &= 2 - \log(x_1x_2) \end{aligned}$$

hat die Form $\mathbf{g}(\mathbf{x}) = \mathbf{h}(\mathbf{x})$ mit

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(x_1, x_2) \\ g_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 4x_1 - x_2 + x_1x_2 \\ -x_1 + 6x_2 \end{bmatrix}, \qquad \mathbf{h}(\mathbf{x}) = \begin{bmatrix} h_1(x_1, x_2) \\ h_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 - \log(x_1x_2) \end{bmatrix}$$

Das Gleichungssystem lässt sich umformulieren:

$$\begin{aligned} 4x_1 - x_2 + x_1x_2 - 1 &= 0 \\ -x_1 + 6x_2 + \log(x_1x_2) - 2 &= 0 \end{aligned}$$

In dieser Form lautet die Aufgabe: gesucht sind *Nullstellen der vektorwertigen Funktion* $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$, also Lösungen von $\mathbf{f}(\mathbf{x}) = 0$ mit

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 4x_1 - x_2 + x_1x_2 - 1 \\ -x_1 + 6x_2 + \log(x_1x_2) - 2 \end{bmatrix}$$

Eine andere, äquivalente Umformung liefert

$$\begin{aligned} x_1 &= \frac{1}{4}(x_2 - x_1x_2 + 1) \\ x_2 &= \frac{1}{6}(x_1 - \log(x_1x_2) + 2) \end{aligned}$$

Hier sind *Fixpunkte der vektorwertigen Funktion* $\mathbf{\Phi} : \mathbb{R}^2 \to \mathbb{R}^2$ gesucht, also Lösungen von $\mathbf{x} = \mathbf{\Phi}(\mathbf{x})$ mit

$$\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} \phi_1(x_1, x_2) \\ \phi_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} \frac{1}{4}(x_2 - x_1x_2 + 1) \\ \frac{1}{6}(x_1 - \log(x_1x_2) + 2) \end{bmatrix}$$

Noch ein Hinweis zur Schreibweise: Wenn wir einen speziellen Fixpunkt gefunden haben, dann bezeichnen wir den im Folgenden mit $\boldsymbol{\xi}$, um ihn von anderen, allgemeinen Werten $\mathbf{x}$ zu unterscheiden.

2.2 Mehrdimensionale Fixpunkt-Iteration

Fixpunkt-Iterationen sind auch im mehrdimensionalen Fall möglich. Ein Fixpunkt einer Abbildung $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$ ist – völlig analog zur eindimensionalen Definition – ein Wert $\boldsymbol{\xi} \in \mathbb{R}^n$, für den gilt:

$$\boldsymbol{\xi} = \mathbf{\Phi}(\boldsymbol{\xi}) \, .$$

Genauso wie im eindimensionalen Fall findet Fixpunkt-Iteration (falls sie konvergiert) einen Fixpunkt. Noch einmal: Wir setzen hier Vektoren aus dem $\mathbb{R}^n$ und vektorwertige Funktionen in fetter Schrift ($\mathbf{\Phi}, \boldsymbol{\xi}, \mathbf{x} \ldots$), zum Unterschied von Variablen und reellwertigen Funktionen ($\phi, \xi, x, \ldots$). Sonst ändert sich nichts am Schema der Fixpunkt-Iteration.

Fixpunkt-Iteration, mehrdimensional
Gegeben sei eine Abbildung $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^n, \quad \mathbf{x} \to \mathbf{\Phi}(\mathbf{x})$.
Gesucht ist ein Fixpunkt $\boldsymbol{\xi}$ von $\mathbf{\Phi}$.

$\mathbf{x}^{(0)}$ als Startwert gegeben.
Iterationsvorschrift
$\mathbf{x}^{(k+1)} = \mathbf{\Phi}(\mathbf{x}^{(k)})$ für $k = 0, 1, 2 \ldots$

Beachte die Konvergenzbedingungen (Abschnitt 2.4)!

Beispiel: Fixpunkt-Iteration für ein System zweier nichtlinearer Gleichungen

Gegeben sei das nichtlineare Gleichungssystem (log ist natürlich der natürliche Logarithmus)

$$\begin{aligned} 4x - y + xy - 1 &= 0 \\ -x + 6y + \log(xy) - 2 &= 0 \end{aligned}$$

Ausgehend von der Näherungslösung $x_0 = 1$ und $y_0 = 1$ bestimme man durch geeignete Fixpunkt-Iteration verbesserten Näherungen.

In der Nähe des Startwertes hängt die erste Gleichung am stärksten vom Term $4x$ ab; die zweite Gleichung von $6y$. Vorgangsweise: löse die beiden Gleichungen jeweils nach diesen Termen auf.

$$\begin{aligned} x &= \frac{1}{4}(y - xy + 1) \\ y &= \frac{1}{6}(x - \log(xy) + 2) \end{aligned}$$

Die Funktion $\mathbf{\Phi}$ ist hier ein Vektor aus zwei reellwertigen Funktionen ϕ und ψ, der Vektor $\mathbf{x}$ hat zwei Komponenten x und y.

$$\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} \phi(x,y) \\ \psi(x,y) \end{bmatrix} = \begin{bmatrix} \frac{1}{4}(y - xy + 1) \\ \frac{1}{6}(x - \log(xy) + 2) \end{bmatrix}$$

Iteration liefert die Folge $(1;1)$, $(1/4;1/2)$, $(0{,}343\,75; 0{,}721\,574)$, $(0{,}368\,383; 0{,}622\,985)$, …, die gegen den Fixpunkt $(0{,}353\,443\,88; 0{,}639\,968\,47)$ konvergiert.

2.3 Normen

Exakte Lösung, Näherungslösung und Fehler sind bei Gleichungssystemen jeweils Vektoren im $\mathbb{R}^n$. Wir brauchen ein Maß für die Größe oder Länge des Fehlervektors, oder für den Abstand der Näherung von der exakten Lösung. Im eindimensionalen Fall messen wir die „Größe“ von x mit dem Absolutbetrag $|x|$, und den Abstand zweier Werte x und y auf der reellen Achse durch $|y - x|$.

Während es aber in $\mathbb{R}$ nur eine sinnvolle Definition für den Absolutbetrag gibt, stehen im $\mathbb{R}^n$ mehrere Möglichkeiten offen. Da ist zunächst einmal die „übliche“ Definition für die Länge eines Vektors, auch *euklidische* Länge oder *2-Norm* genannt. Oft lässt sich aber mit anderen Normen einfacher arbeiten. Wir verwenden noch die *1-Norm* und die *∞-Norm*.

Normen im $\mathbb{R}^n$ für einen Vektor $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \quad , \quad \text{Einsnorm, Summennorm}$$

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} (x_i)^2} \quad , \quad \text{Zweinorm, euklidische Norm}$$

$$\|\mathbf{x}\|_\infty = \max_i |x_i| \quad , \quad \text{Unendlich-Norm, Maximums-Norm}$$

Erinnern Sie sich an die Definition einer Norm aus Mathematik 2?

Eine Norm im $\mathbb{R}^n$ ist eine Funktion, die jedem Vektor $\mathbf{x} \in \mathbb{R}^n$ eine nichtnegative reelle Zahl $\|\mathbf{x}\| \in \mathbb{R}_0^+$ zuordnet, wobei $\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\forall \alpha \in \mathbb{R}$ drei Bedingungen gelten müssen:

- Nur der Nullvektor hat Norm 0

$$\|\mathbf{x}\| = 0 \quad \Rightarrow \quad \mathbf{x} = 0$$

- Skalar α läst sich als Betrag herausheben

$$\|\alpha \cdot \mathbf{x}\| = |\alpha| \cdot \|\mathbf{x}\|$$

- Die Dreiecksungleichung gilt

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$$

Norm und Distanz

Eine Norm kann auch die *Distanz* zwischen zwei Punkten $\mathbf{x}$ und $\mathbf{y}$ messen:

$$\text{dist}(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|$$

- Taxis in Manhattan messen Strecken in der 1-Norm.

  Deswegen heißt die 1-Norm auch Taxi- oder Manhattan-Norm

- Abstand in der Luftlinie entspricht der 2-Norm.
- Größter Unterschied in den Komponenten: ∞-Norm.

Matrixnormen

- Der Hauptberuf von Matrizen ist, Vektoren zu multiplizieren.
- Das Ergebnis einer Matrix-Vektor-Multiplikation ist wieder ein Vektor; der ist gewöhnlich länger oder kürzer und verdreht gegenüber Ausgangsvektor.
- Eine *Matrixnorm* misst als „Größe“ einer Matrix, wie „stark“ sie auf Vektoren wirkt.
- Eine gegebene Matrix kann Vektoren nicht beliebig stark verlängern. Es gibt für jede Matrix einen „Maximal-Verlängerungs-Faktor“

Der „Maximal-Verlängerungs-Faktor“ ist eine Matrixnorm

Verschiedene Matrixnormen

Die 1-, 2- und ∞-Normen lassen sich von den entsprechenden Vektornormen ableiten: Sie geben für die Rechenoperation $\mathbf{y} = A \cdot \mathbf{x}$ an, um wieviel $\mathbf{y}$ gegenüber $\mathbf{x}$ *maximal vergrößert* wird. Eins- und ∞-Norm lassen sich aus den Matrixelementen einfach berechnen:

$\|A\|_1$	*Einsnorm* : maximale Spaltenbetragssumme
$\|A\|_\infty$	*Unendlich-Norm* : maximale Zeilenbetragssumme

Für die Matrix-Zweinorm lässt sich keine so einfache Rechenvorschrift angeben, obwohl gerade sie häufig verwendet wird.

MATLAB kann alle Normen leicht berechnen: $\|A\|_1$ = `norm(A,1)`, $\|A\|_2$ = `norm(A)`, $\|A\|_\infty$ = `norm(A,Inf)`.

Matrixnorm, allgemeine Definition

Matrizen lassen sich addieren und mit Skalaren multiplizieren. In diesem Sinn verhalten sie sich genauso wie Vektoren des $\mathbb{R}^n$. Alles, was sich wie ein Vektor verhält, können wir als „Vektor“ interpretieren: Die $m \times n$-Matrizen bilden einen *Vektorraum*. Der Begriff „Norm“ wird genau so definiert wie die Norm von Vektoren des $\mathbb{R}^n$. Vergleichen Sie die Definition einer Norm im $\mathbb{R}^n$ auf Seite 23 – sie wird hier nahezu wörtlich übernommen.

Eine *Norm* im $\mathbb{R}^m \times \mathbb{R}^n$ ist eine Funktion, die jeder $m \times n$-Matrix A eine nichtnegative reelle Zahl $\|A\| \in \mathbb{R}_0^+$ zuordnet, wobei $\forall A, B \in \mathbb{R}^m \times \mathbb{R}^n$, $\forall \alpha \in \mathbb{R}$ drei Bedingungen gelten müssen:

- Nur die Nullmatrix hat Norm 0:

$$\|A\| = 0 \quad \Rightarrow \quad A = 0$$

- Skalar α läst sich als Betrag herausheben:

$$\|\alpha \cdot A\| = |\alpha| \cdot \|A\|$$

- Die Dreiecksungleichung gilt:

$$\|A + B\| \leq \|A\| + \|B\|$$

Diese drei Grundregeln muss jede Norm erfüllen. Aber es gibt für die 1-, 2- oder ∞-Norm noch Bonus-Features. Für diese Matrix-Normen gelten nämlich noch folgende Rechenregeln:

$$\|A \cdot B\| \leq \|A\| \cdot \|B\| \tag{8}$$

$$\|A \cdot \mathbf{x}\| \leq \|A\| \cdot \|\mathbf{x}\| \tag{9}$$

Vergleiche Absolutbetrag: $|a \cdot b| = |a| \cdot |b|$

Frobeniusnorm:

Noch eine weitere Norm; Die Frobenius-Norm $\|A\|_F$ wird so ähnlich berechnet wie die Vektor-Zweinorm: *Quadrieren, summieren, Wurzel ziehen*

Frobenius-Norm: $\qquad \|A\|_F = \sqrt{\sum a_{ij}^2}$

Die Frobeniusnorm lässt sich leichter berechnen als die Matrix-Zweinorm und dient zu deren Abschätzung:

$$\|A\|_2 \leq \|A\|_F$$

Auch für $\|A\|_F$ gelten neben den Norm-Axiome noch die Rechenregeln

$$\|A \cdot B\|_F \leq \|A\|_F \cdot \|B\|_F \quad , \quad \|A \cdot \mathbf{x}\|_2 \leq \|A\|_F \|\mathbf{x}\|_2$$

MATLAB: $\|A\|_F =$ `norm(A,'fro')`.

Matrixnormen – das Kleingedruckte[6]

[6] Was hier dasteht, ist nicht wichtig,
wenn 's nicht dastünd', wär's nicht richtig.

Die lockere Erklärung *„Matrixnorm ist maximaler Verlängerungsfaktor“* ist mathematisch korrekt für 1-, 2- und ∞-Norm, wenn Vektorlängen in den jeweiligen Normen gemessen werden. Die Frobeniusnorm überschätzt aber gewöhnlich den maximal auftretenden Verlängerungsfaktor, wenn Vektorlängen in der 2-Norm gemessen werden. Immerhin liefert sie eine obere Schranke für den Verlängerungsfaktor.
Auch die Vorschrift $\|A\| = \max_{i,j} |a_{ij}|$ erfüllt die drei Bedingungen einer Norm, ist aber nicht immer eine obere Schranke für den Verlängerungsfaktor.

2.4 Konvergenz

Die Konvergenz der mehrdimensionalen Fixpunkt-Iteration hängt wie im eindimensionalen Fall mit dem Begriff der kontrahierenden Abbildung zusammen.

Konvergenz der Fixpunkt-Iteration im $\mathbb{R}^n$
Die Funktion $\mathbf{\Phi}(x)$ besitze einen Fixpunkt $\boldsymbol{\xi}$: $\mathbf{\Phi}(\boldsymbol{\xi}) = \boldsymbol{\xi}$. Sei ferner B eine offene Umgebung um den Fixpunkt in der Form $B = \{\mathbf{x} : \|\boldsymbol{\xi} - \mathbf{x}\| < r\}, \quad r > 0$. Wenn $\mathbf{\Phi}$ in B eine *kontrahierende Abbildung* in (irgend-) einer Norm $\|\cdot\|$ ist, d. h.,

$$\|\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y})\| \leq C\|\mathbf{x} - \mathbf{y}\| \,, \quad C < 1 \text{ für alle } \mathbf{x}, \mathbf{y} \in B \,,$$

dann konvergiert die Fixpunkt-Iteration $\mathbf{x}^{(k+1)} = \mathbf{\Phi}(\mathbf{x}^{(k)})$ mindestens linear gegen $\boldsymbol{\xi}$ für alle $\mathbf{x}^{(0)} \in B$.

Der Beweis erfolgt analog zu der eindimensionalen Form des Konvergenzsatzes. Auch der Begriff der Konvergenzordnung läßt sich unter Verwendung von Normen geradewegs auf den mehrdimensionalen Fall übertragen.

Kontraktion und Jacobi-Matrix

Das Konvergenzkriterium $|\phi'(\xi)| < 1$ im eindimensionalen Fall (vergleiche Seite 17) lässt sich auf den mehrdimensionalen Fall übertragen. Dazu fasst man die partiellen Ableitungen von $\mathbf{\Phi}$ in einer Matrix D_ϕ, genannt die *Jacobi-Matrix*, zusammen.

Jacobi-Matrix D_ϕ einer Funktion $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$

$$D_\phi = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \frac{\partial \phi_1}{\partial x_2} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \frac{\partial \phi_2}{\partial x_1} & \frac{\partial \phi_2}{\partial x_2} & \cdots & \frac{\partial \phi_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \frac{\partial \phi_n}{\partial x_2} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}$$

Dann lässt sich ganz ähnlich wie im eindimensionalen Fall aussagen:

Das Fixpunktverfahren konvergiert lokal,
falls in der 1-,2-, Frobenius- oder ∞-Matrixnorm gilt

$$\|D_\phi\| < 1$$

2.5 Newton-Verfahren für Systeme

Gegeben sei eine vektorwertige Funktion $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. Gesucht sei eine Nullstelle von $\mathbf{f}$. Das ist ein Vektor $\mathbf{x} \in \mathbb{R}^n$ als Lösung von

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

Dies ist die allgemeine Formulierung eines Systems von n linearen oder nichtlinearen Gleichungen in n Unbekannten. Und noch einmal sei darauf hingewiesen: wir setzen Vektoren aus dem $\mathbb{R}^n$ und vektorwertige Funktionen in fetter Schrift $(\mathbf{x}, \mathbf{f}(\mathbf{x}), \ldots)$, zum Unterschied von Variablen und reellwertigen Funktionen $(x, f(x), \ldots)$.

Komponentenweise ausgeschrieben mit

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \quad \text{und} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{lautet das System} \quad \begin{array}{l} f_1(x_1, x_2, \ \ldots \ , x_n) = 0 \\ f_2(x_1, x_2, \ \ldots \ , x_n) = 0 \\ \qquad \ldots \\ f_n(x_1, x_2, \ \ldots \ , x_n) = 0 \end{array} .$$

Das Newton-Verfahren für Systeme führt die Lösung eines nichtlinearen Systems auf die Lösung einer Folge von linearen Gleichungssystemen zurück. Die Lösung von Systemen linearer Gleichungen ist vergleichsweise einfach gegenüber nichtlinearen Gleichungssystemen. Wir behandeln lineare Gleichungssysteme später noch ausführlich, aber einstweilen nehmen wir an, dass Sie aus der Mittelschule damit hinreichend vertraut sind.

Sofern die entsprechenden partiellen Ableitungen existieren, definieren wir die *Jacobi-Matrix* D_f von $\mathbf{f}$ durch

$$D_f = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} & \cdots & \dfrac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \dfrac{\partial f_n}{\partial x_2} & \cdots & \dfrac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Angenommen, ein Punkt $\mathbf{x}^{(0)}$ ist als Startwert in der Nähe einer Nullstelle gegeben. Dann lässt sich $\mathbf{f}$ in der Umgebung von $\mathbf{x}^{(0)}$ in linearisierter Näherung schreiben als (Taylorscher Lehrsatz für Funktionen mehrerer Veränderlicher)

$$\mathbf{f}(\mathbf{x}^{(0)} + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}^{(0)}) + D_f(\mathbf{x}^{(0)}) \cdot \Delta\mathbf{x} + \mathbf{R}$$

mit einem Restglied $\mathbf{R}$, das im Limes $\Delta\mathbf{x} \to 0$ mit höherer Ordnung verschwindet. Wir vernachlässigen das Restglied und fordern $\mathbf{f}(\mathbf{x}^{(0)} + \Delta\mathbf{x}) = 0$. Aus der daraus entstandenen Gleichung

$$0 = \mathbf{f}(\mathbf{x}^{(0)}) + D_f(\mathbf{x}^{(0)}) \cdot \Delta\mathbf{x}$$

lässt sich der Korrekturvektor $\Delta\mathbf{x}$ bestimmen und damit eine verbesserte Näherung $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}$.

Das Newton-Verfahrens für Systeme lässt sich so formulieren:

Newton-Verfahren für Systeme
Gegeben eine differenzierbare vektorwertige Funktion $\mathbf{f}$ und ein Startwert $\mathbf{x}^{(0)}$. Gesucht eine Nullstelle von $\mathbf{f}$.

Iterationsvorschrift

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$

mit $\Delta\mathbf{x}^{(k)}$ als Lösung von $D_f(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$

Auch dieses Verfahren ist ein Fixpunktverfahren, und zwar für die Funktion

$$\mathbf{\Phi}(\mathbf{x}) = \mathbf{x} - D_f^{-1}(\mathbf{x})\mathbf{f}(\mathbf{x}).$$

Notwendig für die Durchführbarkeit ist, dass D_f^{-1} existiert.

Man kann zeigen: Sofern D_f^{-1} an der Nullstelle existiert, konvergiert das Verfahren für genügend genaue Startwerte quadratisch.

Da es oft sehr mühsam ist, immer alle Elemente von D_f an jedem Punkt $\mathbf{x}^{(k)}$ zu berechnen, geht man manchmal so vor, daß man D_f an einem einzigen Punkt $\mathbf{x}^{(0)}$ berechnet und für den weiteren Verlauf des Verfahrens fix lässt. Dieses Verfahren heißt vereinfachtes Newton-Verfahren. Dafür muss $x^{(0)}$ bereits eine brauchbare Näherung sein. Das vereinfachte Newton-Verfahren konvergiert allerdings nur linear.

Das Newton-Verfahren für Systeme erfordert also in jedem Schritt die Lösung eines linearen Gleichungssystems. Das nächste Kapitel bringt die systematische Behandlung linearer Gleichungssysteme.

Beispiel: nichtlineares Gleichungssystem aus Abschnitt 2.2

Die Funktion $\mathbf{f}$ und ihre Jacobi-Matrix D_f sind hier

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 4x - y + xy - 1 \\ -x + 6y + \log(xy) - 2 \end{bmatrix}, \qquad D_f = \begin{bmatrix} 4+y & -1+x \\ -1+\frac{1}{x} & 6+\frac{1}{y} \end{bmatrix}.$$

Startwert $(1;1)$ eingesetzt liefert

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \qquad D_f = \begin{bmatrix} 5 & 0 \\ 0 & 7 \end{bmatrix}.$$

Zu lösen ist also das Gleichungssystem

$$\begin{bmatrix} 5 & 0 \\ 0 & 7 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

Es liefert den Korrekturvektor und die verbesserte Lösung

$$\Delta\mathbf{x}^{(0)} = \begin{bmatrix} -0,6 \\ -0,428571 \end{bmatrix}, \quad \mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}^{(0)} = \begin{bmatrix} 0,4 \\ 0,571429 \end{bmatrix}.$$

Der nächste Schritt wertet zuerst $\mathbf{f}$ und D_f für die neuen Werte von $\mathbf{x}$, löst das Gleichungssystem für den Korrekturterm $\Delta\mathbf{x}^{(1)}$ und errechnet daraus die verbesserte Näherung $\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \Delta\mathbf{x}^{(1)}$. Die Matrix D_f hat aber hier nicht mehr so „schöne“ Einträge; das Gleichungssystem ist deswegen nicht so unmittelbar lösbar wie im ersten Schritt. Das vereinfachte Newtonverfahren würde zwar $\mathbf{f}$ neu auswerten, die Matrix D_f des ersten Schrittes beibehalten. Einfacherer Rechengang, aber langsamere (nur lineare statt quadratischer) Konvergenz!

3 Lineare Gleichungssysteme, direkte Verfahren

Wir verwenden die Standard-Notation für ein System linearer Gleichungen:

$$A\mathbf{x} = \mathbf{b} \,,$$

worin A die Koeffizientenmatrix, $\mathbf{b}$ die rechte Seite und $\mathbf{x}$ den Lösungsvektor bezeichnet. Wenn das System aus n Gleichungen und Unbekannten besteht, dann ist A eine $n \times n$-Matrix.

Lösungsverfahren für lineare Gleichungssysteme lassen sich in zwei Hauptgruppen unterteilen: direkte und iterative Verfahren.

- Direkte Verfahren berechnen (rundungsfehlerfreie Rechnung vorausgesetzt) eine exakte Lösung. Eliminations- und Substitutionsverfahren sowie die Cramersche Regel fallen in diese Kategorie. Wenn Sie mit Papier und Stift Systeme mit zwei oder drei Unbekannten lösen wollen, sind solche direkte Verfahren die Methoden der Wahl. Computer können heutzutage problemlos für mehrere zehntausend Gleichungen und Unbekannten direkte Lösungsmethoden verwenden.
- Iterative Verfahren berechnen schrittweise immer bessere Näherungslösungen. Diese Methoden eigenen sich aber nur für Gleichungssysteme mit spezieller Matrix-Struktur. Computer lösen damit riesig große Gleichungssysteme (mehrere Millionen Unbekannte), wie sie zum Beispiel bei numerischer Strömungssimulation oder Festigkeitsberechnungen auftreten.

Dieses Kapitel behandelt direkte Verfahren und wiederholt (was aus Mathematik 1 bekannt sein sollte) theoretische Aussagen zur Existenz und Eindeutigkeit der Lösung; iterative Methoden behandelt Kapitel 4.

Software in anerkannt hoher Qualität ist in der Programmbibliothek LAPACK (`http://www.netlib.org/lapack/`) frei verfügbar. Sie werden auch in kommerziell angebotenen Paketen nichts Besseres finden. Auch MATLAB enthält die LAPACK-Algorithmen. (Übrigens ist MATLAB ursprünglich als einfache Benutzerschnittstelle zu diesem Programmpaket entstanden).

3.1 Dreiecksmatrizen

Wenn A eine untere oder obere *Dreiecksmatrix* ist, kann man das Gleichungssystem $A\mathbf{x} = \mathbf{b}$ einfach durch schrittweises Einsetzen lösen (Vorwärts- oder Rückwärtssubstitution).

Andernfalls transformiert man das Gleichungssystem auf Dreiecksgestalt, wie es Abschnitt 3.2 beschreibt. Andere Möglichkeit: man *faktorisiert* A zuerst in ein Produkt von Dreiecksmatrizen. Das entsprechende Verfahren beschreibt Abschnitt 3.5.

Wir bezeichnen Dreiecksmatrizen mit L und R. In der üblichen Notation sind in L nur Einträge im linken unteren Dreieck von Null verschieden und alle Einträge der Hauptdiagonale gleich eins. In R sind nur Einträge im rechten oberen Dreieck einschließlich der Hauptdiagonale ungleich Null. Beispiel für $n = 4$:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix}, \qquad R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

Wenn die Elemente einer Dreiecksmatrix L auf einem Feld `a[i][j]` und die rechte Seite $\mathbf{b}$ auf einem Vektor `b[i]` gespeichert sind, löst das folgende Java-Programmsegment das Gleichungssystem $L\mathbf{x} = \mathbf{b}$ schrittweise durch *Vorwärts-Substitution*.

Achtung, Java zählt Feldindizes von 0 bis $n-1$; konventionelle Mathematik-Schreibweise zählt Zeilen und Spalten von 1 bis n.)

```
for (int i=0; i<n; i++) {
   x[i] = b[i];
   for (int j=0; j<i; j++) {
      x[i] -= a[i][j] * x[j];
   }
}
```

Ähnlich kompakt lässt sich die Lösung eines Gleichungssystems mit rechter oberer Dreiecksmatrixformulieren. Unterschiede zu vorhin: Die *Rückwärts-Substitution* beginnt in der letzten Zeile und schreitet von unten nach oben fort; durch das Hauptdiagonalelement wird dividiert; weiters ist hier die rechte Seite anfangs bereits auf `x[ ]` gespeichert, der Algorithmus überschreibt `x[ ]` mit dem Lösungsvektor.

```
for (int i=n-1; i>-1; i--) {
   for (int j=i+1; j<n; j++) {
      x[i] -= a[i][j] * x[j];
   }
   x[i] /= a[i][i];
}
```

Der Rechenaufwand, gemessen an der Zahl der Punktoperationen (Multiplikationen und Divisionen) beträgt für die Vorwärts-Substitution $n^2/2 - n/2$. Die Rückwärts-Substitution braucht $n^2/2 + n/2$ Punktoperationen. Für große n ist allein der quadratische Term ausschlaggebend. Wir sagen daher

Rechenaufwand bei Vorwärts- oder Rückwärts-Substitution
Lösen eines $n \times n$-Dreieckssystem erfordert $O(n^2)$ Punktoperationen.

Der Rechenaufwand wächst also quadratisch mit der Anzahl der Gleichungen.

Frage: Was tut man, wenn ein Gleichungssystem nicht in Dreiecksform vorliegt? Antwort: man formt es in ein solches um (natürlich so, dass Originalsystem und umgeformtes System äquivalent sind, also genau die gleichen Lösungen haben).

Darum geht es im nächsten Abschnitt.

3.2 Gauß-Elimination

Die einfache Gauß-Elimination läßt sich so formulieren:

Einfache Gauß-Elimination
Gegeben eine $n \times n$-Matrix A und rechte Seite $\mathbf{b}$. Sofern keines der a_{kk} zu einer Division durch Null führt, transformiert dieses Verfahren das System $A\mathbf{x} = \mathbf{b}$ auf ein äquivalentes System in oberer Dreiecksform $R\mathbf{x} = \mathbf{c}$.

Für alle Spalten $k = 1, \ldots n-1$
in Spalte k eliminiere alle Einträge unterhalb des Diagonalelements

Das Eliminieren in Spalte k läuft dabei folgendermaßen ab:

Für alle Zeilen $i = k+1, \ldots, n$ unterhalb der Diagonale
setze $p = a_{ik}/a_{kk}$
subtrahiere das p-fache der k-ten Zeile von Zeile i

Die Subtraktion wird durch folgende Schleife bewerkstelligt:

Für alle Spalten $j = k, \ldots n$
$a_{ij} = a_{ij} - pa_{kj}$
Für rechte Seite: $b_i = b_i - pb_k$

Diese Rechenvorschrift *überschreibt* Einträge in A und $\mathbf{b}$ mit den jeweiligen Zwischenresultaten und letztlich mit den Einträgen von R und $\mathbf{c}$. Sie verzichtet darauf, Einträge unterhalb der Diagonale (die eigentlich gleich Null sein sollen) zu löschen. Es wird sich später, in Abschnitt 3.5, herausstellen, dass diese Einträge eine wichtige Bedeutung haben.

Der Rechenaufwand beträgt $n^3/3 - n/3$ Punktoperationen für die Transformation der Matrix und $n^2/2 - n/2$ Punktoperationen für die Transformation der rechten Seite.

Als JAVA Code sieht Gauß-Elimination verblüffend einfach aus. Zu Beginn muss in `x[ ]` die rechte Seite gespeichert sein. In drei geschachtelte Schleifen wird schließlich die Lösung auf `x[ ]` geschrieben.

```
for (int k=0; k<n; k++) {
   for (int i=k+1; i<n; i++) {
      double p = a[i][k] / a[k][k];
      for (int j=k+1; j<n; j++) {
         a[i][j] -= p * a[k][j];
      }
      x[i] -= p * x[k];
   }
}
```

Das Programm spart es sich, im k-ten Schritt die Elemente unterhalb der Hauptdiagonale in der k-ten Spalte zu berechnen, weil ohnehin 0 herauskommen muss. Es verzichtet auch darauf, diese Nullen explizit in die Matrix hineinzuschreiben, sondern lässt dort die Zwischenresultate einfach stehen.

Das schrittweise Rückwärts-Einsetzen läßt sich mit $n^2/2 + O(n)$ Punktoperationen genäß dem Programmsegment aus dem vorigen Abschnitt erledigen. (Diese Doppelschleife verwendet nur das obere Dreieck von A; die Zwischenresultate $\neq 0$ unterhalb der Diagonale von vorhin stören daher nicht.)

Kombiniert liefern diese beiden Codesegmente einen einfachen Gleichungslöser.

Rechenaufwand bei einfacher Gauss-Elimination wächst kubisch mit der Anzahl der Gleichungen.
Gauß-Elimination löst ein $n \times n$-System $A\mathbf{x} = \mathbf{b}$ mit $O(n^3)$ Rechenoperationen.

(Genau nachgezählt sind es $n^3/3 + n^2 - n/3 = n^3/3 + O(n^2)$ Punktoperationen.)

3.3 Pivotsuche

Die einfache Gauß-Elimination hat einen Haken: Der Rechenschritt $p = a_{ik}/a_{kk}$ kann zu einer Division durch Null führen. Für eine Matrix zufällig gewählter reeller Zahlen ist das extrem unwahrscheinlich, aber Murphy's Gesetz besagt: *If anything can go wrong, it will.* Und tatsächlich versagt das Verfahren bei so simplen Systemen wie

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

weil gleich als erste Rechenoperation durch Null dividiert wird. Das ist definitiv nicht Schuld des Gleichungssystems, es hat nämlich die eindeutige Lösung $x_1 = 1, x_2 = 1$. Vertauscht man andererseits erste und zweite Gleichung, dann läuft das Verfahren problemlos.

Auch aus Gründen der Rechengenauigkeit kann es günstig sein, Gleichungen oder Unbekannte systematisch zu vertauschen. Man nennt diese Vorgehensweise *Pivotierung*.

Gauß-Elimination mit vollständiger Pivotsuche
Gegeben eine $n \times n$-Matrix A und rechte Seite $\mathbf{b}$. Dieses Verfahren transformiert das System $A\mathbf{x} = \mathbf{b}$ auf obere Dreiecksform $R\mathbf{x} = \mathbf{c}$. Die Matrix A wird dabei durch R und der Vektor $\mathbf{b}$ durch $\mathbf{c}$ überschrieben.

```
Für alle Spalten k = 1,...,n-1
    suche das betragsgrößte Element in der
    quadratischen Untermatrix aus den Zeilen
    und Spalten k bis n.

    Bringe dieses Element durch geeignetes Vertauschen
    von Gleichungen und Unbekannten an die Stelle a_kk.

    Wenn a_kk = 0
        Stopp.
    sonst
        in Spalte k eliminiere alle
        Einträge unterhalb des Diagonalelementes
        wie in der simplen Gauß-Elimination.
```

Pivot bedeutet „Drehachse, Angelpunkt". Bei der Elimination dreht sich alles darum, welche Gleichung jeweils benützt werden soll, um die entsprechende Unbekannte in den verbleibenden Gleichungen zu eliminieren. Angelpunkt ist das Element a_{kk}, mit dessen Hilfe der Faktor $p = a_{ik}/a_{kk}$ berechnet wird. Es heißt deswegen das *Pivotelement*. Ein günstiges Pivotelement durch geeignete Vertauschungen zu finden heißt *Pivotsuche* oder *-wahl*.

Die notwendigen Zeilen- und Spaltenvertauschungen komplizieren ein Rechenprogramm in Vergleich zur Dreifachschleife der einfachen Gauß-Elimination erheblich. Der Gewinn an zusätzlicher Einsicht in das Verfahren steht dazu in keinem annehmbaren Verhältnis. Deswegen ist hier kein Programm zur vollständigen Pivotsuche abgedruckt.

Üblicherweise führen Gleichungslöser keine vollständige, sondern nur *Zeilen-Pivotsuche* durch. Das heißt, sie beschränken sich der Einfachheit halber auf Vertauschen von Zeilen (=Gleichungen)[7]. Die Empfindlichkeit gegenüber Rundungsfehlern ist bei Zeilen-Pivotsuche etwas höher als bei vollständiger Pivot-Suche.

3.4 Lösbarkeit linearer Gleichungssysteme

Die Faustregel „Bei genausoviel Gleichungen wie Unbekannten gibt es immer eine Lösung“ *ist falsch*. Ich wiederhole (weil ich es bei Prüfungen immer wieder so höre): *ist falsch*!

Bei den drei Gleichungssystemen

$$\begin{aligned} x+y&=2 \\ 2x+2y&=4 \end{aligned} \qquad \begin{aligned} x+y&=2 \\ x+2y&=3 \end{aligned} \qquad \begin{aligned} x+y&=2 \\ 2x+2y&=3 \end{aligned}$$

sehen Sie hoffentlich mit freiem Auge: Beim ersten und beim zweiten ist $x = 1, y = 1$ eine Lösung. Das dritte System ist unlösbar. Beim ersten System gibt es allerdings noch unendlich viele weitere Lösungen. Diese beispiele illustrieren den allgemeinen Fall.

Lösbarkeit linearer Systeme
Für ein lineares Gleichungssystem gilt genau eine von drei Aussagen: Es gibt

- unendlich viele Lösungen;
- eine eindeutige Lösung;
- keine Lösung.

(Sie haben das in der Mathematik-Grundvorlesung gelernt!)

> Dieser Abschnitt behandelt nur Systeme mit genausoviel Gleichungen wie Unbekannten, aber die obige Aussage gilt auch für lineare Systeme mit mehr Gleichungen als Unbekannten. Bei weniger Gleichungen als Unbekannten sind nur die zwei Fälle möglich: keine Lösung oder unendlich viele Lösungen.

Das Gaußsche Eliminationsverfahren kann entscheiden, welcher Fall vorliegt und mögliche Lösungen berechnen.

3.4.1 Eliminationsverfahren

Gauß-Elimination mit vollständiger oder Zeilen-Pivotsuche transformiert die Originalmatrix A und rechte Seite $\mathbf{b}$ auf ein System in *Stufenform*: Von jeder Zeile zur nächsten nimmt (von links her gesehen) die Zahl der führenden Nullen um mindestens eins zu.

Mögliche Fälle nach Abschluss des Eliminationsverfahrens
Das System ist auf Stufenform transformiert.

- Es treten Nullzeilen in A auf und alle entsprechenden Einträge in b sind ebenfals Null: *unendlich viele Lösungen.*
- Es treten Nullzeilen in A auf, aber zumindest ein entsprechender Eintrag in b ist nicht Null: *keine Lösung.*
- Es treten keine Nullzeilen in A auf: *eine eindeutige Lösung.*

[7] Sie finden in Wikipedia unter dem Stichwort *Gaußsches Eliminationsverfahren* Pseudocode in mehreren Varianten.

Der MATLAB-Befehl `rref([A,b])` (rref steht für *reduced row echelon form*, reduzierte Stufenform) führt eine Variante des Eliminationsverfahrens durch (Gauß-Jordan Verfahren), allerdings nur mit Spalten-Pivotsuche. Für die Ergebnismatrix in der rref-Form gelten die gleichen Aussagen wie oben.

Wenn ein Computer die Elimination in Gleitkomma-Arithmetik durchführt, lässt sich nicht so leicht überprüfen, ob Einträge exakt gleich Null sind. Durch Rundungsfehler in den Eingabedaten und während der Rechnung werden Matrixelemente oft nicht exakt Null, sondern nur extrem klein. Es ist überhaupt nicht trivial, eine Schranke anzugeben, ab der Einträge als Null anzusehen sind. Dubiose Grenzfälle, in denen das Eliminationsverfahren gerade noch eine Lösung findet, obwohl Matrixzeilen schon fast null sind, heißen *numerisch singulär*.

3.4.2 Rang der Matrix und der erweiterten Matrix

Der *Rang einer $m \times n$-Matrix* ist die Anzahl ihrer linear unabhängigen Zeilen- oder Spaltenvektoren.

> Auch wenn die Matrix unterschiedlich viele Zeilen und Spalten hat, gilt: es gibt immer genau so viele linear unabhängige Zeilen wie Spalten; kurz: Zeilenrang = Spaltenrang.

Der MATLAB-Befehl `rank(A)` bestimmt den Rang der $n \times n$-Matrix A, und `rank([A,b])` den Rang der *erweiterten Koeffizientenmatrix* (der Matrix des Gleichungssystems mit rechter Seite als letzte Spalte angefügt).

Rang und Lösbarkeit: Fallunterscheidungen

- `rank(A)` = `rank([A,b])` $< n$: *unendlich viele Lösungen*
- `rank(A)` $< n$ und `rank(A)` $\neq$ `rank([A,b])`: *keine Lösung*
- `rank(A)` $= n$: *eindeutige Lösung*

Es gibt verschiedene Methoden, den Rang einer Matrix zu berechnen, zum Beispiel Transformation auf Stufenform durch Gauß-Elimination: Der Rang ist die Anzahl der von Null verschiedenen Zeilen. Es gibt dabei aber kein einfaches Entscheidungskriterium, ob ein Wert bloß infolge Rundungsfehlern oder „echt“ ungleich Null ist. MATLAB verwendet ein sehr rechenaufwendiges Verfahren (Singulärwertzerlegung), das aber die verläßlichsten Aussagen liefert.

Falls ein Gleichungssystem unendlich viele Lösungen hat, lässt sich die allgemeine Lösung entweder aus dem `rref([A,b])`-Ergebnis ablesen oder durch folgende Befehle finden:

`pinv(A)*b` liefert eine spezielle Lösung
`null(A)` liefert den *Nullraum* von A: eine Liste von linear unabhängigen Lösungen des *homogenen Systems* $A\mathbf{x} = 0$.

Die allgemeine Lösung ist die Summe aus der speziellen Lösung und einer beliebigen Linearkombination aus dem Nullraum.

> `null(A,'r')` liefert ebenfalls eine Liste von linear unabhängigen Lösungen des homogenen Systems, aber mit „schöneren“ Zahlen bei einfachen Beispielmatrizen. Allerdings rechnet diese Variante numerisch weniger zuverlässig. (Lassen Sie sich nie von äußerer Schönheit blenden, wenn dahinter Falschheit lauert!)

Am Beispiel des Systems $A\mathbf{x} = \mathbf{b}$ mit

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 6 \\ -1 & -2 & -2 & -2 \\ 3 & 6 & 8 & 10 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \quad \texttt{rref([A,b])} = \begin{bmatrix} 1 & 2 & 0 & -2 & -2 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Aus dem `rref([A,b])`-Ergebnis lässt sich ablesen: $x_2 = \lambda$ und $x_4 = \mu$ sind frei wählbare Parameter, $x_3 = 1 - 2\mu, x_1 = -2 - 2\lambda + 2\mu$. In Vektorform:

$$\mathbf{x} = \begin{bmatrix} -2 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

Bei so einer einfachen Matrix kann MATLAB eine partikuläre Lösung auch in der Form `x=A\b` und den Nullraum mittels `null(A,'r')` berechnen. Dieses Vorgehen ist bei praktischen Problemen mit realen Daten nicht zu empfehlen, aber hier liefert es (abgesehen von einer Warnmeldung) das „schöne“ Ergebnis

$$\mathbf{x} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

`pinv(A)*b` und `null(A)`liefern die aus numerischer Sicht vorteilhafteste Darstellung,

$$\mathbf{x} = \begin{bmatrix} -0.2069 \\ -0.41379 \\ 0.034483 \\ 0.48276 \end{bmatrix} + \lambda \begin{bmatrix} -0.77069 \\ 0.10421 \\ 0.56227 \\ -0.28113 \end{bmatrix} + \mu \begin{bmatrix} -0.48335 \\ 0.54725 \\ -0.61115 \\ 0.30558 \end{bmatrix}$$

MATLAB berechnet dieses Resultat mit aufwändigen und zuverlässigen numerischen Verfahren. Aber anders als bei den vorigen Darstellungen der Lösung ergibt probeweises Einsetzen in den Ausdruck $A\mathbf{x} - \mathbf{b}$ nicht exakt Null, sondern aufgund der Rundungsfehler Werte im Bereich 1×10^{-15}. (Schönheit hängt oft doch auch mit Wahrheit zusammen).

3.4.3 Determinante

Die Determinante determiniert, ob ein Gleichungssystem eindeutig lösbar ist.

> Gleichungssysteme $A\mathbf{x} = \mathbf{b}$ mit $\det A \neq 0$ sind eindeutig lösbar.
>
> Allerdings ist diese Regel für das numerische Rechnen unbrauchbar.

Die folgende symmetrische 8×8-Matrix lässt sich in MATLAB durch `A=rosser` erzeugen.

$$A = \begin{bmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{bmatrix}$$

Für sie berechnet MATLAB derzeit[8] $\det A = -9480{,}580$ also deutlich $\det A \neq 0$. Damit wären Gleichungssysteme mit A eindeutig lösbar. Als Rang berechnet MATLAB aber (korrekt) `rank(A)`=7, und weil $7 < 8$ ist, kann es keine eindeutigen Lösungen geben. MATLABs Wert für die Determinante ist ziemlich falsch.

Für die 6×6-Matrix H, eine sogenannte Hilbert-Matrix,

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{bmatrix}$$

berechnet MATLAB $\det A = 5{,}3673 \times 10^{-18}$, und das könnte man mit voller Berechtigung gerundet für $\det A = 0$ durchgehen lassen. Als Rang berechnet MATLAB aber (korrekt) `rank(H)`=6. Gleichungssysteme mit H sind also eindeutig lösbar (allerdings extrem anfällig gegenüber Rundungsfehlern).

Diese Beispiele illustrieren:

> Der numerisch berechnete Wert einer Determinante sagt nichts über die Lösbarkeit eines Gleichungssystems aus.

3.5 LR-Zerlegung

Die einfache Gauß-Elimination liefert (wenn sie nicht abbricht) mehr als die Transformation auf Dreiecksgestalt. Sie kann gleichzeitig auch eine Zerlegung

$$A = LR$$

errechnen, wobei L eine untere Dreiecksmatrix mit Einsen in der Hauptdiagonale und R eine obere Dreiecksmatrix ist.

> **LR-Zerlegung**
> Das Gaußsche Eliminationsverfahren ohne Pivotwahl faktorisiert (wenn es nicht abbricht) eine Matrix A in ein Produkt $A = LR$ aus einer linken unteren Dreiecksmatrix L und einer rechten oberen Dreiecksmatrix R.

Nach Durchführen einer Gauß-Elimination mit Pivot-Suche enthalten oberes und unteres Dreieck der Ergebnismatrix die LR-Zerlegung einer Matrix mit - im Vergleich zur Ausgangsmatrix - entsprechend vertauschten Zeilen und Spalten.

Die Elemente von L sind 1 entlang der Hauptdiagonale, und darunter gleich den Multiplikatoren $p = a_{ik}/a_{kk}$ an den entsprechenden Stellen (i, k). Die Elemente von R sind genau jene, die das Eliminationsverfahren in das obere rechte Dreieck schreibt.

Die einzige Änderung im Programm auf Seite 30 zum Eliminationsverfahren ist, sich die Zwischenresultate `p` zu merken. Praktischerweise lässt sich jedes `p` auf dem entsprechenden Feldelement `a[i][k]` speichern; das Verfahren eliminiert nämlich gerade diesen Eintrag, erzeugt also an dieser Stelle eine Null. Die Null muss man sich nicht merken, aber dafür kann man an der dadurch freigewordenen Stelle das Zwischenresultat `p` speichern.

[8] mit Version 2022b. Der Wert mit Version 2021b war $\det A = -10\,611$. Ältere Versionen so um 2018 lieferten $\det A = -9478{,}9$; die 2015-Version bringt $-9448{,}8$; vor 2013 war $\det A = -13\,017$. Es sollte Sie beunruhigen, dass ein Rechenergebnis je nach Programmversion so unterschiedlich ausfällt!

Computerprogramme formulieren das Verfahren in aller Regel so, dass die Originalmatrix A durch R und L überschrieben wird. Das obere Dreieck von A enthält nach erfolgreichem Ablauf die Nichtnull-Einträge von R; die Einsen in der Hauptdiagonale von L verstehen sich von selbst, man braucht sie nicht zu speichern; die restlichen Nichtnull-Elemente von L finden unterhalb der Hauptdiagonale von A Platz. Das Elegante an dieser Speicherung ist, dass sie sich im Verlauf des Verfahrens quasi von selbst ergibt.

Siehe Übungsunterlagen für weitere Informationen!

Für die LR-Zerlegung braucht man keine rechte Seite. Die kommt erst später ins Spiel. Zur Lösung des Systems $A\mathbf{x} = \mathbf{b}$, wenn $A = LR$ bereits gegeben ist, formt man nämlich um:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} && \\ (LR)\mathbf{x} &= \mathbf{b} && \\ L(R\mathbf{x}) &= \mathbf{b} && \text{setze } \mathbf{y} = R\mathbf{x} \\ L\mathbf{y} &= \mathbf{b} && \text{löse durch Vorwärts-Substitution nach } \mathbf{y} \\ R\mathbf{x} &= \mathbf{y} && \text{löse durch Rückwärts-Substitution nach } \mathbf{x} \end{aligned}$$

Der Rechengang und der Rechenaufwand sind der einfachen Gauß-Elimination völlig äquivalent. Der Vorteil der LR-Zerlegung zeigt sich aber, wenn mehrere Systeme mit der selben Matrix A und unterschiedlichen rechten Seiten $\mathbf{b}_1, \mathbf{b}_2, \ldots$ gelöst werden sollen. Die LR-Zerlegung ist der arbeitsintensive Teil ($\sim n^3/3$ Punktoperationen) und muss nur einmal durchgeführt werden. Die einzelnen Lösungen kosten dann nur mehr $\sim n^2$ Punktoperationen pro rechter Seite.

Für symmetrische Matrizen lassen sich spezielle Varianten der Gauß-Elimination durchführen. Ausnützen der Symmetrie spart Rechenzeit und Speicherplatz. Eine mögliche Zerlegung ist

$$A = LDL^T \,,$$

mit einer Diagonalmatrix D. Die *Cholesky-Zerlegung*

$$A = LL^T$$

ist für symmetrisch positiv definiter Matrizen das Standardverfahren.

3.6 Ein Rechenbeispiel zu Gauß-Elimination und LR-Zerlegung

Das Gaußsche Eliminationsverfahren ist eine Rechenvorschrift zur systematischen Elimination von Unbekannten. Bei Gleichungssystemen mit „einfachen“ Zahlen weicht man oft vom systematischen Weg ab und versucht, den Rechengang abzukürzen (z. B. schon vorhandene Nullen auszunützen). Dabei besteht immer die Gefahr, „im Kreis herum“ zu rechnen, Gleichungen nicht oder doppelt zu verwenden. Es ist daher wichtig, eine allgemein gültige Rechenvorschrift angeben zu können.
Gegeben sei das Gleichungssystem $A \cdot \mathbf{x} = \mathbf{b}$ mit

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$

Man rechnet man mit der *erweiterten Koeffizientenmatrix*

$$[A\,\mathbf{b}] = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 10 & 20 & 23 & 6 \\ 15 & 50 & 67 & 14 \end{bmatrix}$$

Elimination in der ersten Spalte

(Vergleichen Sie mit dem Algorithmus auf Seite 30.)

$n = 3$; in der Spalte $k = 1$ sollen alle Einträge unterhalb des Diagonalelementes eliminiert werden, das sind die Elemente in Zeilen $i = 2, 3$

$k = 1, i = 2$: Elimination in erster Spalte, zweite Zeile

Elimination von $a_{ik} = a_{21} = 10$ mittels des Diagonalelements $a_{kk} = 5$ von Zeile $k = 1$. Multipliziere erste Zeile mit $a_{ik}/a_{kk} = 2$ und subtrahiere:

$$\begin{array}{cccc|} 10 & 20 & 23 & 6 & \\ 10 & 12 & 14 & 12 & |- \\ \hline 0 & 8 & 9 & -6 & \end{array}$$

$k = 1, i = 3$: Elimination in erster Spalte, dritte Zeile

Elimination von $a_{ik} = a_{31} = 15$ mittels des Diagonalelements $a_{kk} = 5$ von Zeile $k = 1$. Multipliziere erste Zeile mit $a_{ik}/a_{kk} = 3$ und subtrahiere:

$$\begin{array}{cccc|} 15 & 50 & 67 & 14 & \\ 15 & 18 & 21 & 18 & |- \\ \hline 0 & 32 & 46 & -4 & \end{array}$$

Transformierte erweiterte Koeffizientenmatrix

nach Elimination in der ersten Spalte:

$$[A\,\mathbf{b}]^{(1)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 32 & 46 & -4 \end{bmatrix}$$

Elimination in der zweiten Spalte

In der Spalte $k = 2$ sollen alle Einträge unterhalb des Diagonalelementes eliminiert werden, das ist nur mehr das Element in Zeile $i = 3$

$k = 2, i = 3$: Elimination in zweiter Spalte, dritte Zeile

Elimination von $a_{ik} = a_{32} = 32$ mittels des Diagonalelements $a_{kk} = 8$ von Zeile $k = 2$. Multipliziere zweite Zeile mit $a_{ik}/a_{kk} = 4$ und subtrahiere:

$$\begin{array}{cccc|} 0 & 32 & 46 & -4 & \\ 0 & 32 & 36 & -24 & |- \\ \hline 0 & 0 & 10 & 20 & \end{array}$$

Transformierte erweiterte Koeffizientenmatrix

Nach Elimination in der zweiten Spalte ist das Eliminationsverfahren beendet.

$$[A\,\mathbf{b}]^{(2)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

Rücksubstitution

Aus der dritten Zeile:

$$\begin{aligned} 10x_3 &= 20 \\ x_3 &= 2 \end{aligned}$$

Einsetzen für x_3 in zweiter Zeile

$$\begin{aligned} 8x_2 + 9x_3 &= -6 \\ 8x_2 + 18 &= -6 \\ 8x_2 &= -24 \\ x_2 &= -3 \end{aligned}$$

Einsetzen für x_2 und x_3 in erster Zeile

$$\begin{aligned} 5x_1 + 6x_2 + 7x_3 &= 6 \\ 5x_1 - 18 + 14 &= 6 \\ 5x_1 &= 10 \\ x_1 &= 2 \end{aligned}$$

Der Matlab-Befehl `x = A\b` arbeitet im Prinzip nach diesem Verfahren, allerdings in Form der LR-Zerlegung mit Spaltenpivotsuche.

Mehrere rechte Seiten

Sind zur selben Matrix A Lösungen für mehrere rechte Seiten zu berechnen, fügt man diese der erweiterten Koeffizientenmatrix als weitere Spalten an und rechnet in gleicher Weise.

Pivotwahl

Bei diesem Beispiel war es nicht notwendig, Gleichungen zu tauschen, um Divisionen durch Null zu vermeiden. Bei Spaltenpivotwahl hätte man vor dem ersten Schritt die dritte Gleichung zur ersten gemacht (weil sie den betragsgrößten Eintrag in der ersten Spalte hat).

LR-Zerlegung

Die auf Dreiecksform transformierte Matrix und die entsprechenden Pivot-Faktoren liefern auch die LR-Zerlegung. Eine rechte Seite ist für die LR-Zerlegung nicht notwendig.

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 0 & 8 & 9 \\ 0 & 0 & 10 \end{bmatrix}$$

Der MATLAB-Befehl `[L, U]=lu(A)` verwendet im Prinzip das Gaußsche Eliminationsverfahren, aber liefert zu diesem Zahlenbeispiel eine andere „quasi“-LR-Zerlegung. Die U- bzw. R-Matrix[9] ist eine echte obere Dreiecksmatrix, L ist eine „durcheinandergeratene“ untere Dreiecksmatrix. Begründung: Spaltenpivotsuche vertauscht Zeilen in der Matrix und ändert im Rechengang Reihenfolge und Zahlenwerte. Sie verringert dadurch die Rundungsfehler.

[9] Dem deutschen Fachbegriff Links-Rechts-Zerlegung entspricht auf Englisch *lower–upper (LU) decomposition* oder *factorization*. Deswegen heißt der entsprechende MATLAB-Befehl `lu` und nicht `lr`.

Zerlegung mit dem Befehl `[L, U]=lu(A)` liefert

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1/3 & 4/5 & 1 \\ 2/3 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 15 & 50 & 67 \\ 0 & -40/3 & -65/3 \\ 0 & 0 & 2 \end{bmatrix}$$

3.7 Weitere Anwendungen der LR-Zerlegung

3.7.1 Determinante

Determinante
Wenn eine Faktorisierung $A = LR$ gegeben ist, ist $\det A$ das Produkt der Hauptdiagonalelemente von R.

Begründung: Die Determinante einer Dreiecksmatrix ist das Produkt der Hauptdiagonalelemente. In L stehen dort lauter Einsen, somit ist $\det L = 1$. Nach den Rechenregeln für Determinanten ist dann

$$\det A = \det(LR) = (\det L)(\det R) = \det R \; .$$

Rechenaufwand für eine $n \times n$-Matrix: $n^3/3 + 2n/3 - 1$ Punktoperationen.

Vergleiche dazu das klassische Verfahren, Entwickeln nach Zeilen oder Spalten: Für eine $n \times n$-Matrix sei dafür $w(n)$ der Rechenaufwand an Punktoperationen. Im allgemeinen Fall sind n Unterdeterminanten von $(n-1) \times (n-1)$-Matrizen zu berechnen und noch mit den jeweiligen Elementen zu multiplizieren. Für den Rechenaufwand gilt also

$$w(n) = nw(n-1) + n = n(w(n-1) + 1) \; .$$

Die Funktion $w(n)$ wächst rasch, stärker als die Faktorielle $n!$. Für die Determinante einer 10×10-Matrix braucht ein schneller PC (10^7 Multiplikationen/sec) eine knappe Sekunde, schon bei einer 13×13-Matrix geht sich eine Viertelstunde Kaffeepause aus, auf die Determinante einer 15×15-Matrix warten Sie zweieinhalb Tage, dreizehn Jahrtausende bei einer 20×20-Matrix, und eine 25×25-Matrix wäre nach 80 Milliarden Jahren noch nicht fertig.
Das rasante Wachsen von $w(n)$ und den vergleichsweise geringen Aufwand der LR-Zerlegung illustriert die folgende Tabelle. Sie soll eindringlich auf die Bedeutung rechengünstiger Algorithmen und den Unterschied zwischen polynomialer und exponentieller Laufzeit hinweisen.

n	$w(n)$	$n^3/3 + 2n/3 - 1$
2	2	3
3	9	10
4	40	23
5	205	44
6	1 236	75
7	8 659	118
8	69 280	175
9	623 529	248
10	6 235 300	339
15	2 246 953 104 075	1 134
20	4 180 411 311 071 440 000	2 679
25	26 652 630 354 867 072 870 693 625	5 224

3.7.2 Inverse

Die zu einer (nichtsingulären) Matrix A inverse Matrix A^{-1} braucht kaum ein Rechenverfahren in expliziter Form. Ist etwa der Vektor $\mathbf{x} = A^{-1}\mathbf{y}$ gefragt, so erhält man $\mathbf{x}$ auch genauso gut als Lösung des Gleichungssystems $A\mathbf{x} = \mathbf{y}$, und das ist von Rechenaufwand und -genauigkeit her günstiger.

Warnung 1: Bevor Sie eine Inverse berechnen, fragen Sie dreimal nach, ob Sie diese wirklich brauchen.

Warnung 2: Falls Sie sich noch an die aus der linearen Algebra bekannte Formel (die mit den Determinanten der Kofaktoren) erinnern: vergessen Sie diese. Sie ist von theoretischer Bedeutung, weil sie die Existenz von Inversen nichtsingulärer Matrizen zeigt. Berechnen sollten Sie die Inverse so nicht (überlegen Sie: Rechenaufwand $O(n^5)$, wenn Sie die einzelnen Unterdeterminanten alle mittels LR-Zerlegung rechnen; exponentieller Rechenaufwand, wenn Sie die Determinanten entwickeln).

Wenn es denn sein muss, gehen Sie so vor: Nennen Sie die erste Spalte der Inversen $\mathbf{x}_1$. Die erste Spalte der Einheitsmatrix I ist der Einheitsvektor $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$. Laut Definition gilt

$$AA^{-1} = I \ .$$

Daher gilt insbesondere

$$A\mathbf{x}_1 = \mathbf{e}_1 \ .$$

Das heißt: die erste Spalte der Inversen erhalten Sie als Lösung eines Gleichungssystems mit dem ersten Einheitsvektor als rechter Seite.

In offenkundiger Verallgemeinerung:

Inverse
Die i-te Spalte von A^{-1} ist Lösung des Gleichungssystems

$$A\mathbf{x}_i = \mathbf{e}_i \ .$$

Hier haben Sie also Gleichungssysteme mit derselben Matrix A und mehreren rechten Seiten zu lösen. Vorgangsweise:

Zerlegung $A = LR$; (Aufwand $(n^3 - n)/3$).
Für $i = 1, \ldots, n$
    Lösung $LR\mathbf{x}_i = \mathbf{e}_i$; (Aufwand jeweils n^2).

Rechenaufwand gesamt $(4n^3 - n)/3$.

Der *Gauß-Jordan-Algorithmus* ist eine speziell günstig organisierte Form des Eliminationsverfahrens; er eignet sich gut zur Berechnung mit Stift und Papier. (Sie kennen dieses Verfahren aus Mathematik 1.)

3.7.3 Symmetrisch positiv definite Matrizen

Einfache Argumente der linearen Algebra zeigen: Für symmetrisch positiv definite Matrizen bricht die einfache Gauß-Elimination nie wegen $a_{kk} = 0$ ab. Pivot-Suche ist daher unnötig. Umgekehrt kann man symmetrisch positiv definite Matrizen dadurch charakterisieren, dass die im k-ten Schritt der LR-Zerlegung auftretenden a_{kk} alle > 0 sind.

Allerdings führt man bei symmetrisch positiv definiten Matrizen (wie bereits auf Seite 36 erwähnt) eher Zerlegungen der Form $A = LL^T$ (Cholesky-Zerlegung) oder $A = LDL^T$ durch. Vorteil: bei geschickter Programmierung weniger Rechenzeit und Speicherplatz erforderlich.

3.7.4 Unvollständige Zerlegung

Auch wenn in einer Matrix A die meisten Elemente null sind, können die Faktoren L und R deutlich mehr Nichtnull-Einträge enthalten. Durch Gauß-Elimination werden zusätzliche *Fällterme* erzeugt (also Elemente $\neq 0$ an Stellen, wo die Ausgangsmatrix Elemente $= 0$ hatte). Wenn man alle (oder kleine) Fällterme einfach vernachlässigt, reduzieren sich der Rechenaufwand und benötigte Speicherplatz einer solchen *unvollständigen Zerlegung* drastisch. Natürlich gilt dann nicht mehr $LR = A$, sondern $LR = \tilde{A}$ für eine Näherung $\tilde{A}$ an A. Unvollständigen Zerlegungen sind leistungsfähige Präkonditionierer für iterative Gleichungslöser.

Das Prinzip lässt sich durch eine geringfügige Änderung im Beispielprogramm zur LR-Zerlegung illustrieren (mehr Informationen im Übungsteil): Man ersetze dort in der innersten Schleife

```
For j = k + 1 To n
   a(i, j) = a(i, j) - p * a(k, j)
Next
```

durch

```
For j = k + 1 To n
   if a(i, j) <> 0 then
      a(i, j) = a(i, j) - p * a(k, j)
Next
```

Damit ist aus der LR-Zerlegung eine unvollständige Zerlegung ohne zusätzliche Fällterme geworden. In dieser Form spart das Programm allerdings keinen Speicherplatz und nicht wirklich Rechenzeit. Dazu wären Datenstrukturen notwendig, die gezielt nur die Nichtnull-Elemente speichern, auf diese zugreifen und damit manipulieren (Speicherformate für spärlich besetzte Matrizen).

3.8 Fehlerempfindlichkeit

Rundungsfehler und Fehler in den Eingabedaten verfälschen eine Matrix A zu $A + \delta A$ und die rechte Seite $\mathbf{b}$ zu $\mathbf{b} + \delta\mathbf{b}$. Die Lösung dieses leicht veränderten Systems wird um ein (hoffentlich nicht zu großes) $\delta\mathbf{x}$ von der Lösung des unverfälschten Systems abweichen:

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \quad .$$

Wie hängt $\delta\mathbf{x}$ von δA und $\delta\mathbf{b}$ ab?

Konditionszahl
Die *Konditionszahl* $\kappa(A)$ misst, wie empfindlich in einem System $A\mathbf{x} = \mathbf{b}$ der relative Fehler von $\mathbf{x}$ von kleinen relativen Änderungen in A und $\mathbf{b}$ abhängt.

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \right)$$

In der obigen Ungleichung bezeichnet $\|\cdot\|$ sowohl eine Vektornorm (zum Beispiel 1-, 2- oder ∞-Norm) als auch die entsprechende Matrixnorm. Aus den Rechenregeln für Vektor- und Matrixnormen (vergleiche Abschnitt 2.3) lässt sich herleiten:

$$\kappa(A) = \|A\|\|A^{-1}\|$$

Beweisskizze: Beginne mit dem gestörten System

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}\,,$$

löse Klammern auf

$$A\mathbf{x} + A \cdot \delta\mathbf{x} + \delta A \cdot \mathbf{x} + \delta A \cdot \delta\mathbf{x} = \mathbf{b} + \delta\mathbf{b}\,;$$

weil $A\mathbf{x} = \mathbf{b}$, können wir links $A\mathbf{x}$ und rechts $\mathbf{b}$ streichen. Für kleine $\delta\mathbf{b}$ und δA ist das Produkt $\delta A \cdot \delta\mathbf{x}$ von höherer Ordnung klein; wir vernachlässigen ihn hier. Somit bleibt

$$A \cdot \delta\mathbf{x} + \delta A \cdot \mathbf{x} = \delta\mathbf{b}\ .$$

Daraus lässt sich $\delta\mathbf{x}$ ausdrücken:

$$\delta\mathbf{x} = A^{-1}\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\ ,$$

wende auf beiden Seiten eine Vektornorm an,

$$\|\delta\mathbf{x}\| = \|A^{-1}\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\|\,,$$

nütze eine Eigenschaft der Matrixnorm, Ungleichung (9),

$$\|\delta\mathbf{x}\| \le \|A^{-1}\| \cdot \|\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\|\,,$$

verwende die Dreiecksungleichung,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\left(\|\delta\mathbf{b}\| + \|\delta A \cdot \mathbf{x}\|\right)\ ,$$

erweitere die Terme in der Klammer,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\left(\frac{\|A\mathbf{x}\|}{\|\mathbf{b}\|}\|\delta\mathbf{b}\| + \frac{\|A\|}{\|A\|}\|\delta A \cdot \mathbf{x}\|\right)\ ,$$

vewende noch einmal eine Ungleichung für die Matrixnormen und hebe $\|A\|$ heraus,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\| \cdot \|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}\|\mathbf{x}\| + \frac{\|\delta A\|}{\|A\|}\|\mathbf{x}\|\right)\ ,$$

eine letzte Division durch $\|\mathbf{x}\|$, und wir sind fertig:

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \|A^{-1}\| \cdot \|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\delta A\|}{\|A\|}\right)\ .$$

Der relative Fehler in $\mathbf{x}$ kann also $\kappa(A)$-mal größer als der relative Fehler in A und $\mathbf{b}$ sein. Ein Gleichungssystem, dessen Matrix eine große Konditionszahl hat, wird sehr empfindlich auf Fehler in den Eingabedaten reagieren. Ein solches System heißt *schlecht konditioniert*. Geometrische Veranschaulichung: schleifender Schnitt zweier Geraden.

Die Berechnung der Konditionszahl direkt gemäß der Definition würde die Berechnung der Inversen erfordern und wäre unsinnig aufwendig. Viele Gleichungslöser liefern Schätzungen von $\kappa(A)$ als Nebenprodukt. Es gilt zum Beispiel

$$\kappa(A) \ge \frac{\max|\lambda|}{\min|\lambda|}$$

(Verhältnis vongrößtem zu kleinsten Eigenwert-Bertag; Eigenwerte werden in Abschnitt **??** behandelt).

4 Iterative Verfahren für lineare Gleichungssysteme

Gegeben sei ein lineares Gleichungssystem in n Gleichungen und Unbekannten.

$$\begin{array}{lcl} a_{11}x_1 + a_{12}x_2 + & \ldots & + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + & \ldots & + a_{2n}x_n = b_2 \\ \vdots & & \vdots \quad\; \vdots \\ a_{n1}x_1 + a_{n2}x_2 + & \ldots & + a_{nn}x_n = b_n \end{array} \tag{10}$$

In Matrixschreibweise

$$A\mathbf{x} = \mathbf{b}\,. \tag{11}$$

Das klassische Lösungsverfahren dafür, jedenfalls für Systeme von zwei bis einigen hundert Gleichungen, ist Gauß-Elimination. Sie wird in Kapitel 3 systematisch behandelt. Aus vielen Anwendungen (Strömungssimulation, Seismik, Computertomographie, Festigkeitsrechnungen mit finiten Elementen...) resultieren Gleichungssysteme mit vielen tausend oder sogar Millionen Unbekannten. Solche Systeme werden meist iterativ gelöst. Sie lernen hier nur einige Basis-Verfahren kennen, auf denen die leistungsfähigeren Methoden aufbauen.

4.1 Einfache iterative Verfahren: Jacobi, Gauß-Seidel, SOR

Angenommen, die Diagonalelemente a_{ii} einer $n \times n$-Matrix A sind alle ungleich null. Dann wäre folgendes Rezept zur Lösung von $A\mathbf{x} = \mathbf{b}$ möglich (ein Fixpunkt-Verfahren):

Jacobi-Verfahren für $A\mathrm{x} = \mathrm{b}$, einfach formuliert
Löse jede Gleichung nach ihrem Diagonal-Term auf, setze Startwerte ein, iteriere.

Ausführlicher in der komponentenweisen Schreibweise (10) formuliert: Bringen Sie jeweils in der i-ten Zeile alle Terme bis auf den i-ten auf die rechte Seite und lösen Sie nach x_i auf. Ein entsprechend umgeformtes 3×3-System sieht dann so aus:

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned}$$

Angenommen, $\mathbf{x}^{(k)}$ ist ein näherungsweiser Lösungsvektor. Das Jacobi-Verfahren erzeugt eine neue Näherung durch

$$\begin{aligned} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{aligned}$$

Vielleicht finden Sie Matrix-Schreibweise übersichtlicher. Dazu definieren wir eine Matrix $D = [d_{ij}]$ mit den gleichen Diagonalelementen wie A und null in allen Nichtdiagonalelementen. Die restlichen Elemente von A schreiben wir in eine Matrix E:

$$A = D + E \text{ mit } D = [d_{ij}], \quad d_{ij} = \begin{cases} a_{ii} & \text{falls } i = j, \\ 0 & \text{sonst.} \end{cases} \qquad E = A - D\,. \tag{12}$$

Das Gleichungssystem (11) lässt sich dann äquivalent umformen zu

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (D+E)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= \mathbf{b} - E\mathbf{x} \\ \mathbf{x} &= D^{-1}(\mathbf{b} - E\mathbf{x}) \ . \end{aligned}$$

Die letzte Gleichung ist eine Fixpunktgleichung. Die entsprechende Fixpunkt-Iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{13}$$

heißt *Jacobi-Verfahren* .is called *Jacobi method* .

Iterationsschritt des Jacobi-Verfahrens
In Matrix-Schreibweise für Zerlegung $A = D + E$:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Komponentenweise geschrieben: für $i = 1, \ldots, n$

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

Das Jacobi-Verfahren nützt nicht die aktuellste Information zur Berechnung von $x_i^{(k+1)}$. Beispielsweise verwendet es $x_1^{(k)}$ zur Berechnung von $x_2^{(k+1)}$, obwohl $x_1^{(k+1)}$ bereits verfügbar ist. Wenn wir das Verfahren so formulieren, dass es immer die aktuellsten Näherungswerte an die x_i verwendet, erhalten wir das *Gauß-Seidel-Verfahren* .

Für die Matrix-Schreibweise des Gauß-Seidel-Verfahrens definieren wir eine Matrix $C = [c_{ij}]$ mit den gleichen Elementen wie A in und unterhalb der Hauptdiagonale und Null oberhalb der Hauptdiagonale. Die restlichen Elemente von A schreiben wir in eine Matrix E:

$$A = C + E \text{ mit } C = [c_{ij}], \quad c_{ij} = \begin{cases} a_{ij} & \text{falls } i \geq j, \\ 0 & \text{sonst.} \end{cases} \qquad E = A - C \ . \tag{14}$$

Dieselben Schritte, die für das Jacobi-Verfahren zur Fixpunkt-Gleichung 13 geführt haben, können wir mit der Matrix C statt D wiederholen und erhalten die Iterationsvorschrift für das Gauß-Seidel-Verfahren:

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{15}$$

Iterationsschritt des Gauß-Seidel-Verfahrens
einfach formuliert
Löse der Reihe nach jede Gleichung nach ihrem Diagonal-Term auf, setze Startwerte ein, iteriere, verwende jeweils neuerste Näherungswerte.

Matrix-Schreibweise für Zerlegung $A = C + E$

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Komponenten-Schreibweise
für $i = 1, \ldots, n$

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

Das Gauß-Seidel-Verfahren lässt sich oft deutlich beschleunigen, wenn man den aus der Iterationsformel erhaltenen Wert nicht direkt verwendet, sondern die Änderung von $x_i^{(k)}$ zu $x_i^{(k+1)}$ noch um einen Faktor $\omega > 1$ vergrößert. Dieses iterative Verfahren heißt *SOR-Verfahren* (SOR steht für *successive overrelaxation*)

Ein geeigneten Wert für ω lässt sich aber nicht so einfach angeben. Die Theorie sagt: $1 \leq \omega < 2$, mit Werten eher in der Nähe von 2. Für $\omega = 1$ reduziert sich SOR auf Gauß-Seidel.

Iterationsschritt des SOR-Verfahrens
einfach formuliert
Jeweils neuer Näherungswert zuerst als Zwischenresultat $y_i^{(k+1)}$ aus Gauß-Seidel-Schritt; endgültiger Näherungswert durch Extrapolation (Überrelaxation) aus alter Näherung und Zwischenresultat: $x_i^{(k+1)} = \omega y_i^{(k+1)} + (1 - \omega) x_i^{(k)}$

Die Komponenten-Schreibweise sieht hier bereits etwas unübersichtlich aus.

für $i = 1, \ldots, n$

$$y_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

$$x_i^{(k+1)} = \omega y_i^{(k+1)} + (1 - \omega) x_i^{(k)}$$

Auch dieses Verfahren lässt sich mit einer Zerlegung $A = B + E$ ähnlich wie die Gleichungen 13 und 15 anschreiben:

$$\mathbf{x}^{(k+1)} = B^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{16}$$

Darin ist B eine Kombination der Matrizen C und D von vorhin (12, 14), und zwar

$$B = C + \left(\frac{1}{\omega} - 1 \right) D$$

4.2 Konvergenz des Jacobi- und des Gauß-Seidel-Verfahrens

Nicht für jede beliebige Matrix A konvergieren die drei oben vorgestellten Verfahren. Die Konvergenz des Jacobi-Verfahrens lässt sich zeigen, indem man nachweist, dass es sich bei der Fixpunkt-Iteration um eine kontrahierende Abbildung handelt. Dazu definieren wir:

Eine $n \times n$-Matrix $A = [a_{ij}]$ heißt *stark diagonaldominant*, wenn

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{für} \quad i = 1, 2, \ldots, n$$

Es muss also in jeder Zeile die Summe der Beträge der Nichtdiagonalelemente kleiner sein als der Betrag des Diagonalelementes.

Konvergenz des Jacobi-Verfahrens
Das Jacobi-Verfahren konvergiert bei Gleichungssystemen mit stark diagonaldominanter Matrix für beliebige Startwerte zur eindeutigen Lösung.

Beweis: Wir zeigen, dass die zur Iterationsvorschrift (13) gehörige Funktion $\mathbf{\Phi}(\mathbf{x}) = D^{-1}(\mathbf{b} - E\mathbf{x})$ für beliebige $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ eine kontrahierende Abbildung in der Maximumsnorm ist. Laut Abschnitt 2.4 ist damit Konvergenz gewährleistet. Wir vereinfachen erst einmal

$$\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y}) = D^{-1}(\mathbf{b} - E\mathbf{x}) - D^{-1}(\mathbf{b} - E\mathbf{y}) = D^{-1}E(\mathbf{y} - \mathbf{x})$$

Die i-te Zeile der Matrix $D^{-1}E$ lautet

$$\frac{a_{i1}}{a_{ii}} \quad \frac{a_{i2}}{a_{ii}} \quad \ldots \quad \frac{a_{i,i-1}}{a_{ii}} \quad 0 \quad \frac{a_{i,i+1}}{a_{ii}} \quad \ldots \quad \frac{a_{in}}{a_{ii}}$$

Die Summe der Elementbeträge in dieser Zeile ist < 1 (Begründung: Aus der Summe $1/|a_{ii}|$ herausheben, Diagonaldominanz ausnützen). Damit ist auch für die Zeilensummen- oder Unendlich-Norm der Matrix $D^{-1}E$ gezeigt:

$$\|D^{-1}E\|_\infty < 1 \ .$$

Aus einer Eigenschaft der Matrixnorm, Ungleichung (9), folgt sofort die Kontraktionseigenschaft

$$\|\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y})\|_\infty = \|D^{-1}E(\mathbf{y} - \mathbf{x})\|_\infty \leq \|D^{-1}E\|_\infty \cdot \|\mathbf{y} - \mathbf{x}\|_\infty \leq C\|\mathbf{y} - \mathbf{x}\|_\infty$$

mit $C = \|D^{-1}E\|_\infty < 1$.

Mit beträchtlich mehr Aufwand lässt sich zeigen, dass auch für eine größere Klasse von Matrizen, nämlich *schwach diagonaldominante, irreduzible* Matrizen, das Jacobi-Verfahren konvergiert. Diese Aussage ist wichtig, weil viele Aufgaben aus der Praxis (numerische Lösung partieller Differentialgleichungen) genau solche Matrizen liefern. Der Vollständigkeit halber hier die Definitionen:
Eine $n \times n$-Matrix $A = [a_{ij}]$ ist *schwach diagonaldominant*, wenn

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{für} \quad i = 1, 2, \ldots, n$$

und zumindest für ein i echte Ungleichheit gilt. Um Irreduzibilität zu untersuchen, zeichnen Sie für jedes i einen Punkt. Für jedes Matrixelement $a_{ij} \neq 0$ in A verbinden Sie die Punkte i und j durch einen Pfeil in Richtung $i \to j$. Die Zeichnung stellt einen *gerichteten Graph* dar. Wenn man von jedem beliebigen Punkt zu jedem anderen gelangen kann, indem man den Pfeilen folgt, dann heißt dieser Graph *zusammenhängend* und die Matrix A ist *irreduzibel*.

In der Regel konvergiert das Gauß-Seidel-Verfahren rascher als das Jacobi-Verfahren. Es braucht für die gleiche Genauigkeit typischerweise nur halb so viele Iterationen. Das SOR-Verfahren mit optimal gewähltem Relaxationsparameter ω braucht größenordnungsmäßig $\sqrt{N}$ Iterationen, wo das Jacobi-Verfahren N Iterationen braucht.

Es gibt aber auch Matrizen, für die ein Verfahren konvergiert, das andere aber nicht.

Wir zitieren hier ohne Beweis zwei weitere Sätze, die Konvergenzbedingungen formulieren.

> Wenn A positive Elemente in der Hauptdiagonale hat und alle andern Elemente ≤ 0 sind, dann konvergiert das Gauß-Seidel-Verfahren genau dann, wenn das Jacobi-Verfahren konvergiert. Wenn beide Verfahren konvergieren, dann ist das Gauß-Seidel-Verfahren asymptotisch schneller (*Satz von Stein und Rosenberg*).
>
> Ist A symmetrisch positiv definit, dann konvergiert das Gauß-Seidel-Verfahren.

4.3 Moderne iterative Gleichungslöser

Gleichungssysteme aus dem Bereich der Strömungssimulation, der Festigkeitsanalyse, der Finanzmathematik und vieler weiterer Anwendungsgebiete erreichen leicht eine Größe von mehreren Millionen Unbekannten. Dafür sind aber pro Matrixzeile nur wenige Elemente von Null verschieden (So eine Matrix heißt *schwach besetzt*). Für die Lösung solcher Systeme werden heute fast ausschließlich iterative Verfahren eingesetzt. Die klassischen Methoden (Jacobi, Gauß-Seidel) konvergieren aber zu langsam und erfordern daher zuviel Rechenaufwand.

Diese Unterlagen können nur eine einführende Übersicht auf einige Prinzipien geben, die moderne iterative Gleichungslöser verwenden.

4.3.1 Splittings, Präkonditionierung

Angenommen, Sie sollen das System

$$A\mathbf{x} = \mathbf{b}$$

lösen.

Günstige Taktik: Sie ersetzen die Matrix A in dieser Aufgabe durch eine andere Matrix $\tilde{A}$, für die Sie Gleichungssysteme viel leichter lösen können. Sie können es sich dabei einfach machen und für $\tilde{A}$ die Einheitsmatrix I wählen, oder den diagonalen Anteil von A, oder gezielt nur bestimmte Matrixelemente aus A herausstreichen.

Schreiben Sie $A = \tilde{A} + E$. Eine solche Aufspaltung heißt ein *Splitting* von A in eine Näherung (auch: *Präkonditionierer*) und einen Restanteil E. Das ursprüngliche Gleichungssystem formulieren Sie dann als Fixpunkt-Aufgabe um.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (\tilde{A} + E)\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x} + E\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x} &= \mathbf{b} - E\mathbf{x} \\ \mathbf{x} &= \tilde{A}^{-1}(\mathbf{b} - E\mathbf{x}) \end{aligned}$$

Das Jacobi-Verfahren benützt diese Idee mit $\tilde{A} = D$, dem diagonalen Anteil. Das Gauß-Seidel-Verfahren lässt sich ebenfalls in dieser Form darstellen, indem man $\tilde{A}$ aus A durch Streichen sämtlicher Elemente oberhalb der Hauptdiagonale gewinnt.

Allgemein gilt, je besser $\tilde{A}$ die Original-Matrix approximiert, um so rascher konvergiert ein solches iterative Verfahren. Besonders gute Splittings entstehen aus *unvollständiger LR-Zerlegung*. Diese Methoden werden bei den Eliminationsverfahren in Abschnitt 3.7.4 kurz behandelt.

In der oben angegebenen Fixpunkt-Form wird das Verfahren aber nicht implementiert, da man die Matrix $\tilde{A}^{-1}$ nur in einfachsten Fällen (wie etwa $\tilde{A} = D$) explizit bilden sollte. Eine algebraisch gleichwertige, aber für Rechner geeignete Form lautet

Iterativer Gleichungslöser, Grundschema für $A = \tilde{A} + E$
Für ein geeignetes Splitting $A = \tilde{A} + E$, einen beliebigen Startvektor $\mathbf{x}^{(0)}$ und eine vorgegebene Genauigkeitsschranke $\epsilon > 0$ findet dieser Algorithmus eine Näherungslösung von $A\mathbf{x} = \mathbf{b}$.

Beginne mit Startvektor $\mathbf{x}^{(0)}$
setze $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
iteriere für $k = 0, 1, 2, \ldots$
  löse $\tilde{A}\mathbf{d}^{(k+1)} = \mathbf{r}^{(k)}$
  setze $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{d}^{(k+1)}$
  setze $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - A\mathbf{d}^{(k+1)}$
bis $\|\mathbf{r}^{(k+1)}\| < \epsilon$
Ergebnis: Näherungslösung $\mathbf{x}^{(k+1)}$

Bei Iterationen dieser Form nennt man $\tilde{A}$ auch die *Präkonditionierungsmatrix*.

Für einen Vektor $\mathbf{x}$ und gegebenes A und $\mathbf{b}$ bezeichnet man den Ausdruck $\mathbf{b} - A\mathbf{x}$ als *Residuum*. Die Aufgabe, ein Gleichungssystem $A\mathbf{x} = \mathbf{b}$ zu lösen, kann man also gleichwertig umformulieren in die Aufgabe, ein $\mathbf{x}$ mit verschwindendem Residuum zu finden. Man kann leicht nachprüfen, dass die Vektoren $\mathbf{r}^{(k)}$ im obigen Grundschema tatsächlich die jeweiligen Residuen sind: $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$. Das Abbruchkriterium des Verfahrens fordert also ein Residuum, das betragsmäßig kleiner als eine vorgegebene Schranke ist.

Vorsicht! Ein kleines Residuum bedeutet nicht automatisch, dass auch der Fehler $\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}$ zwischen exakter und genäherter Lösung klein ist. Es gilt beispielsweise, falls A symmetrisch ist, die Abschätzung

$$\frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\max}|} \leq \|\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}\|_2 \leq \frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\min}|}$$

wobei $\lambda_{\max}$ und $\lambda_{\min}$ den betragsgrößten bzw. -kleinsten Eigenwert von A bezeichnen. Wenn also $\lambda_{\min}$ nahe Null liegt, sagt ein kleines Residuum noch nicht viel über die Größe des Fehlers aus.

Ebensowenig kann man aus der Kleinheit der Korrekturen $\mathbf{d}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ unmittelbar auf die Kleinheit des Fehlers schließen. Moderne iterative Verfahren können allerdings praktischerweise Näherungen für die Eigenwerte mit geringem Zusatzaufwand mitrechnen und somit verlässliche Schranken für den Fehler liefern.

4.3.2 Konvergenzbeschleunigung durch Minimieren des Residuums

Häufig beobachtet man beim obigen Grundschema, dass sich die Vektoren $\mathbf{d}^{(k)}$, um die sich die Näherungsvektoren pro Iterationsschritt ändern, zwar in eine günstige Richtung zeigen, aber nicht die passende Länge haben. Anstatt den Näherungsvektor $\mathbf{x}^{(k)}$ pro Iterationsschritt nur um den Vektor $\mathbf{d}^{(k+1)}$ zu korrigieren, kann man daher versuchen, gleich ein Vielfaches ω dieser Korrektur anzubringen. (Eine ähnliche Idee verwendet auch schon das SOR-Verfahren.)

Wenn sich der Näherungsvektor beim Schritt von k nach $k+1$ um $\omega\mathbf{d}^{(k+1)}$ ändert, dann lässt sich leicht nachrechnen, dass sich der Restvektor um $-\omega A\mathbf{d}^{(k+1)}$ ändert. Man ändert also das Grundschema, setzt

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \omega\mathbf{d}^{(k+1)}\\ \mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}\end{aligned}$$

und wählt ω in jedem Schritt so, dass der Betrag $\|\mathbf{r}^{(k+1)}\|$ dadurch möglichst klein wird. Wie geht das? Mit der üblichen Vorgehensweise, wenn man einen Extremwert sucht: Differenzieren und Nullsetzen der Ableitung. (Es bedeutet hier$\|\cdot\|$ immer die 2-Norm, die euklidische Länge eines Vektors.)

Wir arbeiten der Einfachheit halber mit dem Betragsquadrat von $\mathbf{r}^{(k+1)}$. Es lässt sich als Funktion von ω schreiben:

$$\begin{aligned}\|\mathbf{r}^{(k+1)}\|^2 &= (\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)})\\ &= \Big((\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}) \cdot (\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)})\Big)\\ &= \Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big)\end{aligned}$$

Die einzelnen inneren Produkte sind hier konstante skalare Größen. Differenzieren nach ω und Nullsetzen der Ableitung liefert

$$\begin{aligned}0 &= \frac{d}{d\omega}\|\mathbf{r}^{(k+1)}\|^2\\ &= \frac{d}{d\omega}\Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big)\\ &= -2(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + 2\omega(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}) \quad \text{, daraus}\\ \omega &= \frac{\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}}{A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}}\end{aligned}$$

4.3.3 Konvergenzbeschleunigung durch orthogonale Suchrichtungen

Wir setzen nun also

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}$$

wobei ω so optimal gewählt ist, dass der Betrag von $\mathbf{r}^{(k+1)}$ minimal wird. Das heißt, jede weitere Korrektur des Residuums in Richtung $A\mathbf{d}^{(k+1)}$ kann nur eine Verschlechterung bringen. Falls im nächsten Iterationsschritt

$$\mathbf{r}^{(k+2)} = \mathbf{r}^{(k+1)} - \omega A\mathbf{d}^{(k+2)}$$

die Korrektur $A\mathbf{d}^{(k+2)}$ eine Komponente in Richtung $A\mathbf{d}^{(k+1)}$ enthält, tritt aber genau das ein.

Daher: Ist das Residuum entlang einer Richtung bereits minimiert, dann darf es entlang dieser Richtung nicht mehr korrigiert werden. Wir brauchen also ein Verfahren, das aus der Residuums-Korrektur $A\mathbf{d}^{(k+2)}$ die unerwünschte Komponente in Richtung $A\mathbf{d}^{(k+1)}$ herausnimmt. Das läßt sich durch *Orthogonalisierung* erreichen.

Seien $\mathbf{p}$ und $\mathbf{q}$ zwei Vektoren $\neq 0$. Die Komponente von $\mathbf{p}$ in Richtung von $\mathbf{q}$ ist gegeben durch

$$\left(\frac{\mathbf{p}\cdot\mathbf{q}}{\mathbf{q}\cdot\mathbf{q}}\right)\mathbf{q}$$

Der Vektor

$$\mathbf{p}-\left(\frac{\mathbf{p}\cdot\mathbf{q}}{\mathbf{q}\cdot\mathbf{q}}\right)\mathbf{q}$$

enthält also keine Komponente mehr in Richtung $\mathbf{q}$, steht also orthogonal auf $\mathbf{q}$. (Sonderfall: wenn $\mathbf{p}$ ein skalares Vielfaches von $\mathbf{q}$ ist, liefert diese Rechnung den Nullvektor.)

In dieser Weise lassen sich aus einem Vektor $\mathbf{p}$ auch sukzessive alle Komponenten in Bezug auf ein System von Vektoren herausnehmen.

Orthogonalisieren
Gegeben m von 0 verschiedene Vektoren $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ und ein Vektor $\mathbf{p} \in \mathbb{R}^n$. Dieser Algorithmus entfernt aus $\mathbf{p}$ alle Komponenten in Richtung $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m$.

Für $i = 1 \ldots m$
  rechne inneres Produkt $r_i = \mathbf{p}\cdot\mathbf{q}_i / \mathbf{q}_i\cdot\mathbf{q}_i$
  subtrahiere Komponente: ersetze $\mathbf{p} = \mathbf{p} - r_i\mathbf{q}_i$

P. K. W. Vinsome, damals bei einer Erdölgesellschaft angestellt, veröffentlichte 1976 ein Verfahren, ORTHOMIN, welches alle diese Ideen (Präkonditionierun, Minimierung, Orthogonalisierung) verwendet. Inzwischen wurde eine Fülle von Verfahren entwickelt, die auf ähnlichen Prinzipien beruhen und heute alle als *Krylov-Unterraum-Verfahren* bezeichnet werden.

Für symmetrisch positiv definite Matrizen wurde aus diesen Ideen schon früher ein besonders elegantes und effizientes Verfahren entwickelt, die Methode der konjugierten Gradienten (Hestenes und Stiefel, 1952). Sie ist das Standardverfahren zur iterativen Lösung schwach besetzter, symmetrisch positiv definiter Systeme.

Für unsymmetrische Matrizen sind GMRES (für *generalized minimal residual method*), BiCG (für *biconjugate gradients*) oder CGS (für *conjugate gradient squared*) gängige Verfahren.

5 Überbestimmte Systeme

Ein lineares Gleichungssystem $A\mathbf{x} = \mathbf{b}$ mit mehr Gleichungen als Unbekannten heißt *überbestimmt*.

In so einem Fall ist A eine rechteckige $n \times m$-Matrix mit $n > m$, hat also mehr Zeilen als Spalten. In der Regel hat ein solches System keine exakte Lösung, aber eine eindeutig bestimmte „am wenigsten falsche“ *Ausgleichslösung nach der Methode der kleinsten Quadrate*.

Regelfall: m linear unabhängige Spalten in A, $m+1$ linear unabhängige Spalten in der erweiterten Koeffizientenmatrix $[A, \mathbf{b}]$.
Sonderfälle:

- $\operatorname{rank} A = \operatorname{rank}[A, \mathbf{b}] = m \rightarrow$ eindeutige exakte Lösung.

Systeme ohne vollen Spaltenrang; die nachfolgend beschriebenen Normalengleichungen sind nicht eindeutig lösbar.

- $\operatorname{rank} A = \operatorname{rank}[A, \mathbf{b}] < m \rightarrow$ unendlich viele exakte Lösungen;
- $\operatorname{rank} A < \operatorname{rank}[A, \mathbf{b}] < m \rightarrow$ keine exakte Lösung, unendlich viele gleichberechtigte Ausgleichslösungen.

Überbestimmte Systeme entstehen beispielsweise bei der Auswertung von Messergebnissen, wenn mehr Messungen vorliegen als Parameter gesucht sind.

5.1 Normalengleichungen

Überbestimmte Systeme
Lösung nach der *Methode der kleinsten Quadrate*: suche jenes $\mathbf{x}$, für das die euklidische Norm des Residuenvektors

$$\mathbf{r} = \mathbf{b} - A\mathbf{x}$$

minimal wird. Führt auf die *Normalengleichungen*

$$A^T A\mathbf{x} = A^T \mathbf{b}$$

Herleitung 1: Differenzieren von $(\|\mathbf{r}\|_2)^2 = \mathbf{r}^T \cdot \mathbf{r}$ nach den einzelnen Komponenten von $\mathbf{x}$; Nullsetzen der Ableitung.

Herleitung 2: Nehmen wir an, $\hat{\mathbf{x}}$ sei ein Vektor, der $A^T(\mathbf{b} - A\hat{\mathbf{x}}) = 0$ (also die Normalengleichungen) erfüllt. Wir zeigen nun: für jeden anderen Vektor $\mathbf{x} \neq \hat{\mathbf{x}}$ gilt

$$\|\mathbf{b} - A\mathbf{x}\|_2 \geq \|\mathbf{b} - A\hat{\mathbf{x}}\|_2,$$

das heißt, für keinen anderer Vektor gibt es einen kleineres Residuum. In diesem Sinn ist $\hat{\mathbf{x}}$ eine optimale Ausgleichslösung des Systems $A\mathbf{x} = \mathbf{b}$.

Im Regelfall sind alle Spaltenvektoren in A linear unabhängig. Dann gilt sogar strikt $\|\mathbf{b} - A\mathbf{x}\|_2 > \|\mathbf{b} - A\hat{\mathbf{x}}\|_2$ und $\hat{\mathbf{x}}$ ist die eindeutig bestimmte beste Ausgleichslösung.

Beweis: Man setze $\hat{\mathbf{r}} = \mathbf{b} - A\hat{\mathbf{x}}$ und $\mathbf{r} = \mathbf{b} - A\mathbf{x}$. Dann können wir schreiben

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = (\mathbf{b} - A\hat{\mathbf{x}}) + A(\hat{\mathbf{x}} - \mathbf{x}) = \hat{\mathbf{r}} + A(\hat{\mathbf{x}} - \mathbf{x})$$

Damit gilt für den Zusammenhang zwischen $\mathbf{r}^T \cdot \mathbf{r}$ und $\hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}}$

$$\begin{aligned}\mathbf{r}^T \cdot \mathbf{r} &= (\hat{\mathbf{r}} + A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (\hat{\mathbf{r}} + A(\hat{\mathbf{x}} - \mathbf{x})) \\ &= \hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}} + \hat{\mathbf{r}}^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x})) + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot \hat{\mathbf{r}} + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x})) \\ &= \hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}} + (\hat{\mathbf{r}}^T A) \cdot (\hat{\mathbf{x}} - \mathbf{x})) + (\hat{\mathbf{x}} - \mathbf{x})^T \cdot (A^T \hat{\mathbf{r}}) + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x}))\end{aligned}$$

Weil $\hat{\mathbf{x}}$ laut Voraussetzung die Normalengleichungen erfüllt, ist $A^T \hat{\mathbf{r}} = 0$, ebenso gilt $\hat{\mathbf{r}}^T A = 0$. Es bleibt also

$$\mathbf{r}^T \cdot \mathbf{r} = \hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}} + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x})) \; .$$

Der letzte Term ist (als inneres Produkt eines Vektors mit sich selbst) immer ≥ 0. Sind alle Spaltenvektoren in A linear unabhängig, dann ist mit $\mathbf{x} \neq \hat{\mathbf{x}}$ auch $A(\hat{\mathbf{x}} - \mathbf{x}) \neq 0$; der letzte Term ist dann echt > 0. Damit ist die Aussage bewiesen.

Herleitung 3: Geometrische Veranschaulichung im Fall $\mathbf{x} \in \mathbb{R}^2$ und A eine 3×2-Matrix. Die Vektoren aus der Menge$\{A\mathbf{x} : \mathbf{x} \in \mathbb{R}^2\}$ spannen (wenn die Spalten von A linear unabhängig sind) eine Ebene im Raum auf. Es soll also $A\hat{\mathbf{x}}$ jener Punkt (Ortsvektor) auf der Ebene sein, der von $\mathbf{b}$ minimalen Abstand hat. Der kleinstmögliche Abstand ist der Normalabstand. Der Residuumsvektor $\mathbf{b} - A\hat{\mathbf{x}}$ steht somit normal auf die Ebene, also auf alle Vektoren der Form $A\mathbf{x}$. Aus

$$(A\mathbf{x})^T \cdot (\mathbf{b} - A\hat{\mathbf{x}}) = \mathbf{x}^T \cdot (A^T(\mathbf{b} - A\hat{\mathbf{x}})) = 0$$

folgt $A^T(\mathbf{b} - A\hat{\mathbf{x}}) = 0$, weil nur der Nullvektor auf alle Vektoren $\mathbf{x} \in \mathbb{R}^2$ orthogonal sein kann.

5.2 Weitere Verfahren

Die Lösung überbestimmeter Systeme über die Normalengleichungen ist das klassische Standardverfahren, aber deswegen nicht unbedingt der günstigste Algorithmus. Es ist bloß das einzige, das sich bei kleinen Beispielen mit vertrauten Methoden (Matrixmultiplikation, Elimination) händisch durchrechnen lässt. Modernere Programmpakete und Rechenumgebungen (wie MATLAB) verwenden die QR-Zerlegung. Bei Systemen *ohne vollen Spaltenrang* (seien sie überbestimmt oder nicht) ist die Singulärwertzerlegung (*singular value decomposition, SVD*) vorteilhaft. In diesem Fall gibt es nämlich unendlich viele Lösungen (exakt oder im Sinn der kleinsten Quadrate), und SVD liefert automatisch die betragskleinste. Aus der QR-Zerlegung lassen sich hingegen die Lösungen mit den meisten Null-Komponenten ablesen.

5.3 Beispiele zu überbestimmten Systemen

Gegeben sind drei Gleichungen in zwei Unbekannten,

$$\begin{array}{rcrcl} 2x & + & y & = & 19 \\ -4x & + & 4y & = & 13 \\ 4x & - & y & = & 17 \end{array}$$

Das Gleichungssystem in Matrixschreibweise lautet

$$A\mathbf{x} = \mathbf{b} \text{ mit } A = \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 19 \\ 13 \\ 17 \end{bmatrix}$$

Methode der Normalengleichungen

Bilde $A^T \cdot A$ und $A^T\mathbf{b}$:

$$A^T \cdot A = \begin{bmatrix} 2 & -4 & 4 \\ 1 & 4 & -1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix} = \begin{bmatrix} 36 & -18 \\ -18 & 18 \end{bmatrix} = 18 \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

$$A^T\mathbf{b} = \begin{bmatrix} 2 & -4 & 4 \\ 1 & 4 & -1 \end{bmatrix} \cdot \begin{bmatrix} 19 \\ 13 \\ 17 \end{bmatrix} = \begin{bmatrix} 54 \\ 54 \end{bmatrix} = 18 \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

Die Normalengleichungen lauten daher (schon durch 18 gekürzt):

$$\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$, oder in ausgeschriebener Form $$\begin{array}{rcrcl} 2x & - & y & = & 3 \\ -x & + & y & = & 3 \end{array}$$

Addition der beiden Gleichungen liefert sofort $x = 6$, und daraus durch Einsetzen $y = 9$.

Minimaler Fehler in verschiedenen Normen

Die Normalengleichungen liefern $\mathbf{x} = [6; 9]$. Einsetzen in die ursprünglichen Gleichungen zeigt aber: diese „Lösung“ erfüllt keine der drei Gleichungen exakt. Der Fehlervektor $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ lautet

$$\begin{bmatrix} 19 \\ 13 \\ 17 \end{bmatrix} - \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} 6 \\ 9 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \\ 2 \end{bmatrix}$$

Der Fehlervektor hat Länge 3, und das ist die kleinstmögliche Länge. Gemeint ist hier die euklidische Länge, also die Zweinorm.

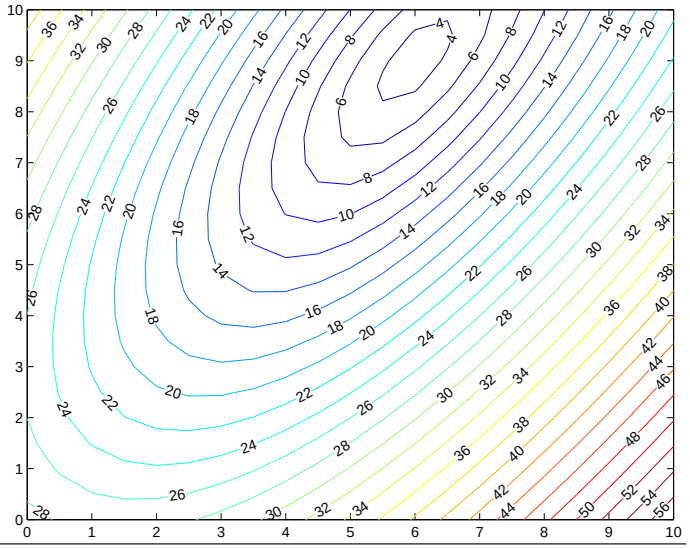
Ändert man beispielsweise den Vektor $\mathbf{x}$ geringfügig auf

$$\mathbf{x} = \begin{bmatrix} 6 \\ 8.8 \end{bmatrix} = \begin{bmatrix} 6 \\ 44/5 \end{bmatrix}$$

so lautet der Fehlervektor

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = \begin{bmatrix} -9/5 \\ 9/5 \\ 9/5 \end{bmatrix} = \begin{bmatrix} -1.8 \\ 1.8 \\ 1.8 \end{bmatrix}$$

mit euklidischer Länge $\|\mathbf{r}\|_2 = \frac{9}{5}\sqrt{3} = 3.1177$, also deutlich über dem Optimum.



Höhenschichtlinien der Länge des Fehlervektors in der Zweinorm. Das Fehlerminimum bei $[6; 9]$ ist deutlich zu erkennen.

In der Unendlichnorm (Maximum der Komponentenbeträge) ist der Fehler nun allerdings geringer: 1.8 statt 2.

Versuchen wir noch einen anderen Vektor $\mathbf{x}$:

$$\mathbf{x} = \begin{bmatrix} 6.15 \\ 9.4 \end{bmatrix}, \quad \text{zugehöriger Fehlervektor } \mathbf{r} = \mathbf{b} - A\mathbf{x} = \begin{bmatrix} -2.7 \\ 0 \\ 1.8 \end{bmatrix}$$

$$\|\mathbf{r}\|_2 = 3.2450, \quad \|\mathbf{r}\|_\infty = 2.7$$

Dieser Fehlervektor liegt also sowohl in der 2- als auch der ∞-Norm über den beiden vorherigen. Misst man aber die Summe der Absolutbeträge (die Einsnorm), so ergibt sich hier der Wert 4.5. Die beiden vorigen Fehlervektoren hatten Einsnormen von 5 bzw. 5.4. In der Einsnorm ist also die hier gewählte Lösung optimal.

Lösung durch QR-Zerlegung von A

$$A = Q \cdot R\,, \quad \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix} = \begin{bmatrix} -1/3 & 2/3 & -2/3 \\ 2/3 & 2/3 & 1/3 \\ -2/3 & 1/3 & 2/3 \end{bmatrix} \cdot \begin{bmatrix} -6 & 3 \\ 0 & 3 \\ 0 & 0 \end{bmatrix}$$

Das transformierte Gleichungssystem $R\mathbf{x} = Q^T\mathbf{b}$ ist ebenfalls überbestimmt und lautet ausgeschrieben

$$\begin{array}{rcrcl} -6x & + & 3y & = & -9 \\ & & 3y & = & 27 \\ & & 0 & = & 3 \end{array}$$

Wenn man die ersten beiden Gleichungen exakt löst (wegen Dreiecksform einfach durch Rücksubstitution), liefern sie keinen Beitrag zum Residuum. Die letzte Gleichung hängt nicht von $\mathbf{x}$ ab. Keine Wahl von $\mathbf{x}$ kann den Beitrag dieser Gleichungen zum Residuum ändern.

Daher ist die aus den ersten beiden Gleichungen bestimmte Lösung optimal für das transformierte System. Weil die Transformation die Norm des Fehlervektors nicht beeinflusst (wegen Orthogonalität von Q), ist diese Lösung auch optimale Lösung von $A \cdot \mathbf{x} = \mathbf{b}$.

Weiteres Beispiel: Lösung mit Singulärwertzerlegung

Ein anderes überbestimmtes System lautet

$$A\mathbf{x} = \mathbf{b} \text{ mit } A = \begin{bmatrix} 14 & -2 \\ -4 & 22 \\ 16 & -13 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -80 \\ 40 \\ -145 \end{bmatrix}$$

Die Singulärwertzerlegung von A ist

$$A = U \cdot S \cdot V^T, \begin{bmatrix} 14 & -2 \\ -4 & 22 \\ 16 & -13 \end{bmatrix} = \begin{bmatrix} -1/3 & -2/3 & -2/3 \\ 2/3 & -2/3 & 1/3 \\ -2/3 & -1/3 & 2/3 \end{bmatrix} \cdot \begin{bmatrix} 30 & 0 \\ 0 & 15 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} -3/5 & -4/5 \\ 4/5 & -3/5 \end{bmatrix}^T$$

Das transformierte System lautet in diesem Fall

$$S \cdot \mathbf{y} = U^T \cdot \mathbf{b}, \begin{bmatrix} 30 & 0 \\ 0 & 15 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 150 \\ 75 \\ -30 \end{bmatrix}$$

Wegen der Diagonalgestalt von S ist die Optimallösung direkt ablesbar: $\mathbf{y} = [5; 5]$. Die Lösung des Originalsystems erhält man über die Beziehung

$$\mathbf{x} = V \cdot \mathbf{y} = \begin{bmatrix} -3/5 & -4/5 \\ 4/5 & -3/5 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 5 \end{bmatrix} = \begin{bmatrix} -7 \\ 1 \end{bmatrix}$$

Wenn QR-Zerlegung oder SVD bereits gegeben sind, ist die Lösung von (überbestimmten) Gleichungssystemen relativ einfach. Der eigentliche Arbeitsaufwand steckt im Berechnen der Zerlegungen. Auf die dabei verwendeten Verfahren kann im Rahmen der Vorlesung nicht eingegangen werden.

MATLAB-Befehle

QR-Zerlegung einer Matrix A:	`[Q R]=qr(A)`
SVD, Singulärwertzerlegung $A = U \cdot S \cdot V^T$	`[U S V]=svd(A)`
Lösung eines überbestimmten Gleichungssystems	
... durch QR-Zerlegung	`A\b`
... durch SVD	`pinv(A)*b`

5.4 Anpassen eines linearen Modells (einer Ausgleichsebene)

Dieser Abschnitt erläutert anhand eines weiteren Beispiels die Lösung überbestimmter Systeme. Vom Thema her überschneidet er sich mit Kapitel 6; dort werden weitere Methoden zur Approximation von Daten behandelt. Was das Beispiel hier illustrieren soll: die unterschiedlichen Größenordnungen bei den Zwischenergebnissen und der daraus resultierene Einfluss der Rundungsfehler im Vergleich zwischen der Methode der Normalengleichung und der QR-Zerlegung.

Für die Blies (einen Nebenfluss der Saar) sollen die Hochwasserstände am Pegel Neunkirchen aus den Wasserständen des Pegels Ottweiler und des Pegels Hangard vorhergesagt werden. Es liegen die Daten der Scheitelwasserstände von 12 Winterhochwässern aus den Jahren 1963–1971 vor:

Wasserstand in cm												
Neunkirchen y	172	309	302	283	443	298	319	419	361	267	337	230
Ottweiler x_1	93	193	187	174	291	184	205	260	212	169	216	144
Hangard x_2	120	258	255	238	317	246	265	304	292	242	272	191

Daten-Quelle: U. Maniak, Hydrologie und Wasserwirtschaft, Springer, 1988

Wir unterstellen den Daten das lineare Modell $a_0 + a_1x_1 + a_2x_2 = y$. In diesem Ansatz sind a_0, a_1 und a_2 unbekannte Koeffizienten, die aus den zwölf gegebenen Werte-Tripeln möglichst gut bestimmt werden sollen. Geometrisch lassen sich die Tripel $(x_1|x_2|y)$ als Punkte im Raum interpretieren. Das lineare Modell entspricht dann einer Ebene, die möglichst gut an die Datenpunkte angepasst werden soll.

Einsetzen der Daten in den Ansatz liefert das Gleichungssystem

$$\begin{array}{rcrcrcr} a_0 & + & 93a_1 & + & 120a_2 & = & 172 \\ a_0 & + & 193a_1 & + & 258a_2 & = & 309 \\ a_0 & + & 144a_1 & + & 191a_2 & = & 230 \\ \vdots & & & & & & \vdots \\ a_0 & + & 187a_1 & + & 255a_2 & = & 302 \end{array}$$

Es liegt somit ein überbestimmtes Gleichungssystem $A\mathbf{x} = \mathbf{b}$ vor, mit 12×3-Matrix A und rechter Seite $\mathbf{b}$,

$$A = \begin{bmatrix} 1 & 93 & 120 \\ 1 & 193 & 258 \\ 1 & 187 & 255 \\ \vdots & \vdots & \vdots \\ 1 & 144 & 191 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 172 \\ 309 \\ 302 \\ \vdots \\ 230 \end{bmatrix}$$

Der klassische Weg zur Ausgleichslösung nach der Methode der kleinsten Quadrate führt über die Normalengleichungen $A^T \cdot A\mathbf{x} = A^T\mathbf{b}$, in diesem Beispiel

$$\begin{bmatrix} 12 & 2328 & 3000 \\ 2328 & 480202 & 609985 \\ 3000 & 609985 & 780572 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 3740 \\ 766996 \\ 975996 \end{bmatrix}$$

Es treten recht unterschiedlich große Zahlen in diesem System auf (Grössenordnung 10^1 bis 10^6). Das ist typisch für Normalengleichungen und ein Indiz dafür, dass dieses System empfindlich gegenüber Rundungsfehlern ist. Numerisch günstiger, aber praktisch nur rechnergestützt durchführbar ist die QR-Zerlegung. Das transformierte System $R\mathbf{x} = Q^T\mathbf{b}$ lautet hier

$$\begin{bmatrix} -3.4641 & -672.0357 & -866.0254 \\ 0 & -169.0266 & -165.5656 \\ 0 & 0 & -56.2141 \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} -1079.6 \\ -245.14 \\ -7.2659 \\ -2.4029 \\ \vdots \\ -11.003 \end{bmatrix}$$

Die ersten drei Gleichungen liegen als System in oberer Dreiecksform vor und sind durch Rücksubstitution exakt auflösbar. Die restlichen neun Gleichungen von der Form $0 = -2.4029, \ldots$ sind klarerweise unlösbar. Sie liefern den Beitrag zum Restvektor.

locker gesagt: QR-Zerlegung transformiert ein überbestimmtes System in ein exakt lösbares System in Dreiecksform und einen unlösbaren Rest. Ignorieren der unlösbaren Gleichungen liefert die bestmögliche Lösung im Sinn der kleinsten Quadrate

Anmerkung: Es ist zwar unmittelbar einsichtig, dass obiges Rezept die Lösung mit betragsmäßig minimalem Restvektor für das *transformierte* System liefert. Allerdings hat das *Originalsystem* einen anderen Restvektor. Die eigentliche Pointe des Verfahrens ist, dass die Transformation von dem einen zum anderen Restvektor durch Multiplikation mit einer *orthogonalen* Matrix passiert. Multiplikation mit einer orthogonalen Matrix lässt den Betrag eines Vektors unverändert. Daher hat auch der Restvektor des Originalsystems minimalen Betrag.

In MATLAB verwendet der Standard-Gleichungslöser-Befehl `A\b` im Falle eines überbestimmten Systems automatisch das QR-Verfahren und liefert hier

```
>> A\b
ans =
   22.5505
    1.3237
    0.1293
```

(Die aufwändigere Lösung mit Singulärwertzerlegung (MATLAB: `pinv(A)*b`) liefert hier das gleiche Resultat.)

Das Modell zur Vorhersage der Hochwasser-Scheitelwerte lautet somit

$$y = 22.5505 + 1.3237x_1 + 0.1293x_2$$

Man erkennt, dass die x_2-Daten gut zehnfach weniger Einfluss auf das Vorhersagemodell haben (weil der entsprechende Koeffizient im linearen Modell nur 0.1293 im Vergleich zu 1.3237 beträgt). Eine mögliche Interpretation wäre, dass der Wasserstand in Neunkirchen hauptsächlich vom Pegel Ottweiler und nicht wirklich vom Pegel Hangard abhängt.

Wie vertrauenswürdig ist dieses Modell? Einsetzen der Datenpunkte liefert den Restvektor.

Messwert y	172	309	302	283	443	298	319	419	361	267	337	230
Modellvorhersage	161	311	303	284	449	298	328	406	341	278	344	238
Residuum	11	-2	-1	-1	-6	0	-9	13	20	-11	-7	-8

Der mittlere absolute Fehler liegt bei 7,3 cm, der maximale Fehler allerdings bei 20 cm.

Eine genauere Beurteilung des Modells im Hinblick auf

- die Richtigkeit des angenommenen linearen Zusammenhangs,
- Einfluss der einzelnen Modellgrößen
- Wahrscheinlichkeit, dass der Prognosewert mit gewisser Genauigkeit zutrifft
- mögliche Ausreißer

erfordert Methoden der multivariaten Statistik und der Regressionsanalyse (und wohl auch eine größere Datenmenge).

6 Approximation von Daten, Ausgleichsrechnung

6.1 Lineare Datenmodelle

Ein Beispiel dazu hat schon Kapitel 5.4 gebracht. Die Vorlesungsfolien und die Übungen bringen weitere Beispiele dazu. Das best-angepasste Modell ergibt sich dabei immer aus der kleinste-Quadrate-Näherung an ein überbestimmtes Gleichungssystem.

Aber nicht immer liefert die Methode der kleinsten Fehlerquadrate eine plausible Anpassung. Einige wenige grob falsche Werte in den Daten („Ausreißer“) können das Ergebnis gewaltig verzerren. Im Kapitel 6.6 wird eine robuste Methode vorgestellt. MATLAB bietet in seinen Toolboxen verschiedene Methoden zur robusten Anpassung (*robust fit*) an.

6.2 Polynomiale Regression

Hier handelt es sich um einen wichtigen Spezialfall der linearen Datenmodelle, für den sich die allgemeinen Formeln etwas vereinfachen.

Eine typische Aufgabe zu diesem Abschnitt könnte lauten: Gegeben sind Messergebnisse für einen Satz von Temperaturwerten T und die entsprechenden Widerstandswerte R eines Temperaturfühlers. Der Zusammenhang zwischen R und T lässt sich näherungsweise in der Form $R = a + bT + cT^2$ beschreiben. Wenn für genau drei (verschiedene) T-Werte Daten vorliegen, lassen sich die drei Parameter a, b und c eindeutig bestimmen. Es ist aber sinnvoll, mehr Messungen durchführen, damit Messfehler der Einzelmessungen nicht so stark ins Gewicht fallen. Die Parameter a, b und c werden dann durch *Ausgleichsrechnung* (man sagt auch *Regression*) bestimmt.

Polynomiale Regression (Ausgleich durch ein Polynom)
Gegeben: $m + 1$ Wertepaare $(x_i, y_i), i = 0, \ldots, m$
Gesucht: $p(x)$, ein Polynom n-ten Grades, $n < m$, so dass die Summe der Fehlerquadrate

$$\sum_{i=0}^{m} (p(x_i) - y_i)^2$$

minimal wird. Locker formuliert: $y = p(x)$ approximiert möglichst gut die Datenpunkte.

Nicht immer ist ein Polynomansatz ein geeignetes Modell, aber oft lässt sich ein scheinbar komplizierteres Modell auf ein polynomiales Modell zurückführen (Beispiele in den Übungen).

Direkter Lösungsweg: Ansatz des Polynoms mit unbestimmten Koeffizienten,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x_n \ .$$

Einsetzen der gegebenen Wertepaare führt auf ein System von $m + 1$ linearen Gleichungen in den $n + 1$ unbekannten Koeffizienten $a_0, a_1, \ldots, a_n$.

Sofern $n < m$, liegt ein *überbestimmtes System* vor.

- klassische Lösung: Näherung nach der Methode der Normalengleichungen: lässt sich bei kleinen Beispielen mit Papier und Stift rechnen; bei großen Datenmengen Gefahr von Rundungsfehlern.

  Die Normalengleichungen sind aber nur dann eindeutig lösbar, wenn in den insgesamt $m + 1$ Wertepaaren mindestens $n + 1$ der x-Werte verschieden sind.

- moderner Lösungsweg: QR-Zerlegung. Praktisch nur am Rechner durchführbar. Bessere Konditionszahl, weniger anfällig für Rundungsfehler.

Dieser Lösungsweg (Ansatz mit unbestimmten Koeffizienten, Aufstellen des überbestimmten Systems, Bilden der Normalengleichungen) lässt sich für die polynomiale Regression etwas abkürzen. Setzt man

$$s_0 = m + 1, \quad t_0 = \sum_{i=0}^{m} y_i$$

und

$$s_k = \sum_{i=0}^{m} x_i^k, \quad t_k = \sum_{i=0}^{m} x_i^k y_i \quad \text{für } k > 0,$$

so lassen sich, wie man leicht herleiten kann, die Normalengleichungen in folgender Gestalt schreiben:

$$\begin{array}{lllllll}
s_0 a_0 & + & s_1 a_1 & + & \ldots\ldots & + & s_n a_n \quad = t_0 \\
s_1 a_0 & + & s_2 a_1 & + & \ldots\ldots & + & s_{n+1} a_n \quad = t_1 \\
 & & & & \ldots\ldots & & \\
s_n a_0 & + & s_{n+1} a_1 & + & \ldots\ldots & + & s_{2n} a_n \quad = t_n
\end{array}$$

Die Normalengleichungen können für größere n ziemlich schlecht konditioniert sein. Bessere Resultate lassen sich in solchen Fällen durch Entwicklung nach orthogonalen Polynomen erzielen.
Verwendet man beispielsweise als Datensatz die Punkte

$$(x, \exp(x)) \quad \text{für } x = 0;\, 0,01;\, 0,02; \ldots; 3,99; 4$$

und approximiert die Daten durch Regressionspolynome verschieden hohen Grades (Rechnung mit vierzehnstelliger Genauigkeit), so wird die Güte der Approximation vorerst mit steigendem Grad besser. Ab dem dreizehnten Grad aber wachsen die Fehler wieder an. Das aus den Normalengleichungen berechnete Polynom 25-ten Grades hat kaum mehr Ähnlichkeit mit der approximierten Funktion. Verwendet man orthogonale Polynome (Tschebyscheff-Polynome), treten diese numerischen Probleme nicht auf.

6.3 Ausgleichsgerade

Ein wichtiger Spezialfall der obigen Problemstellung: An $m + 1$ (mehr als zwei) gegebene Datenpunkte soll eine Gerade mit Gleichung $y = a + bx$ so angepasst werden, dass die Summe der Fehlerquadrate minimal wird.

$$a = \frac{s_2 t_0 - s_1 t_1}{s_0 s_2 - s_1^2}$$

$$b = \frac{s_0 t_1 - s_1 t_0}{s_0 s_2 - s_1^2}$$

Das Finden einer Ausgleichsgeraden wird oft auch als „lineare Regression“ bezeichnet. Das ist aber ein mißverständlicher Terminus, weil er leicht zur Verwechslung mit „linearen Datenmodellen“ führt.

6.4 Nichtlineare Datenmodelle

Dazu gibt es Material auf den Vorlesungsfolien und ausführlicher in den Übungsunterlagen. Kurzfassung: Jacobi-Matrix bilden und iterieren. Im Unterschied zum Newton-Verfahren, das Sie schon kennen, ist das lineare Gleichungssystem mit der Jacobimatrix nun überbestimmt und wird näherungsweise im Sinn der kleinsten Fehlerquadrate gelöst.

Dieses Verfahren wird als *Gauß-Newton-Verfahren* bezeichnet.

Bei nichtlinearen Ausgleichsproblemen gibt es neben dem Gauß-Newton-Verfahren noch weitere Verfahren (z. B. Levenberg-Marquardt-Algorithmus). Damit beschäftigt sich die Optimierung als Teilgebiet der Angewandten Mathematik. Dieses Skript kann nicht näher darauf eingehen.

6.5 Warum „kleinste Quadrate“

Die Methode der kleinsten Quadrate minimiert die euklidische Länge des Residuenvektors. Man kann aber die Größe des Residuenvektors durchaus auch anders messen und entsprechend andere Minimalbedingungen fordern. Wichtige Beispiele: Die Summe der *Absolutbeträge* der Fehler soll minimal werden, oder der *maximale Fehler* soll minimal werden („Minimax-Approximation“). Zwei Gründe sprechen für die kleinsten Quadrate:

- Einfache Herleitung und Durchführung: die Minimalwert-Aufgabe lässt sich mit elementarer Differentialrechnung lösen und führt auf ein einfaches algebraisches Problem

- Statistische Überlegungen: Wenn die Daten mit unabhängigen, zufälligen, normalverteilten Fehlern mit gleicher Standardabweichung behaftet sind, sind kleinste Quadrate in gewissem Sinn optimal (genauer: Die Methode liefert eine *maximum likelihood*-Schätzung der Parameter). Umgekehrt gilt aber: wenn die Fehler in den Daten *nicht* normalverteilt etc. sind, dann sind kleinste Quadrate unter Umständen ziemlich schlecht; siehe unten.

- Weitere statistische Eigenschaften: Wenn man eine Abschätzung für die Genauigkeit der Daten hat, kann man auf die Genauigkeit der berechneten Modellparameter schließen.

  > Angenommen, die Daten sind so skaliert, dass die Varianz der Messfehler gleich 1 ist. Sei $C = (A^T A)^{-1}$ die inverse Matrix des Systems der Normalengleichungen. Die Diagonalelemente von C sind die Varianzen der entsprechenden Modellparameter; die Elemente außerhalb der Hauptdiagonale sind die entsprechenden Kovarianzen.

Und was spricht dagegen?

- Die Methode reagiert empfindlich auf „Ausreißer“ in den Daten. Das Quadrieren der Fehler bestraft grosse Abweichungen streng. Deswegen ist die Methode der kleinsten Quadrate bereit, eine Kurve wild zu verzerren, um ein paar weit außen liegende Datenpunkte auch noch annähernd zu erreichen. Ein paar Ausreißer in ansonsten vernünftigen Daten können so eine völlig unsinnige Approximation bewirken.

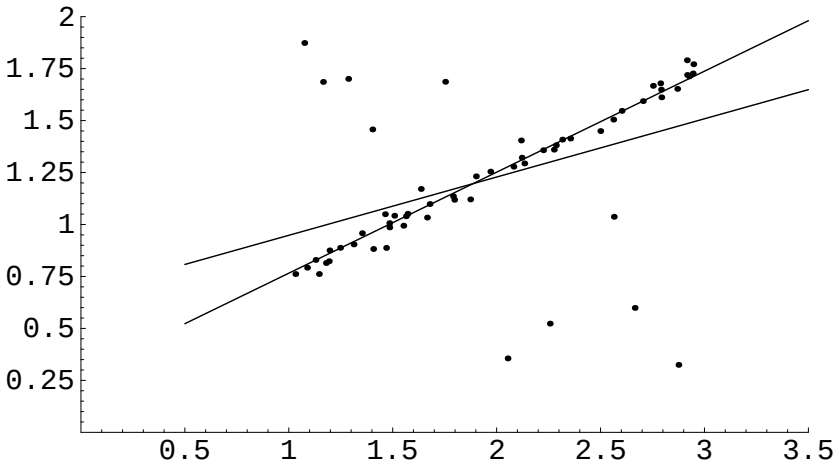


Abbildung 8: Anpassen einer Geraden an Datenpunkte. Die Ausgleichsgerade nach der Methode der kleinsten Quadrate lässt sich von den wenigen Ausreißern stark ablenken. Minimieren des absoluten Fehlers legt eine wesentlich plausiblere Gerade durch die Daten.

6.6 Ausgleichsgerade mit Minimieren der absoluten Fehler (L_1-Norm)

Gegeben: m Wertepaare $(x_i, y_i), i = 1, \ldots, m$
Gesucht: Eine Gerade in der Form $y = a + bx$, so dass die Summe der absoluten Fehler

$$\sum_{i=1}^{m} |p(x_i) - y_i|$$

minimal wird.
Die Lösung lautet: Für gegebenes b ergibt sich a als Median eines Datenfeldes,

$$a = \text{median}\{y_i - bx_i\}$$

Den Parameter b findet man als Lösung der Gleichung

$$0 = \sum_{i=1}^{m} x_i \,\text{sgn}(y_i - a - bx_i)$$

(wobei sgn(0) als Null interpretiert werden soll). Wenn man für a in dieser Gleichung die durch die vorigen Gleichung bestimmte Funktion $a(b)$ einsetzt, bleibt eine Gleichung in einer Unbekannten übrig. Intervallhalbierung (siehe Kapitel 1.7) ist die geeignete Lösungsmethode dafür.

6.7 Ausgleichsgerade mit Minimieren der Normalabstände

Dazu gibt es unter dem Titel *Total Least Squares* Material auf den Vorlesungsfolien und den Übungsunterlagen.

Ü 1 Erste Übungseinheit

Vier wichtige Regeln:
1. Es funktioniert nie beim ersten Mal.
2. Es wird auch beim zweiten Mal wahrscheinlich nicht funktionieren.
3. Es funktioniert besser, wenn alle Kabel angesteckt sind.
4. Wenn sonst nichts mehr hilft, lesen Sie die Anleitung.

Ü 1.1 MATLAB

Die Übungen beginnen mit einer Einführung in die Rechen- und Programmierumgebung MATLAB (steht abkürzend für „MATrix LABoratorium“). Dieses Programm ist im universitären Bereich und in der industriellen Praxis ein Standardwerkzeug für technisch-wissenschaftliche Berechnungen. Für viele numerischen Aufgaben bietet MATLAB Lösungsfunktionen und Methoden zur Visualisierung. Gleichzeitig ist es eine moderne Programmiersprache, in der Sie eigene Anwendungen entwickeln können.

Diese Übungseinheit soll Ihnen eine rasche Einführung geben.

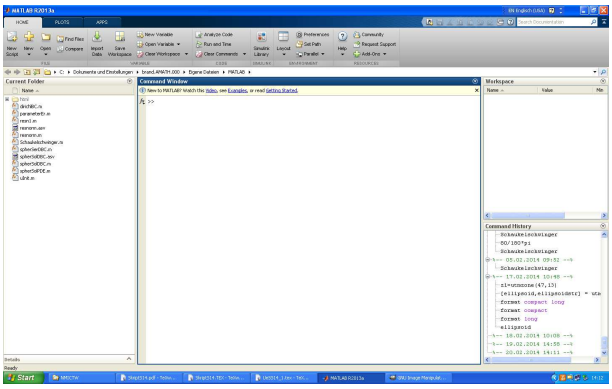
Inzwischen bietet auch die Programmiersprache Python mit ihren Zusatzpaketen (numpy, scipy) nahezu die gleiche Funktionalität wie MATLAB. Alle Übungsaufgaben lassen sich ebensogut in Python lösen. Wenn Sie mit Python vertraut sind, wird Ihnen numerisches Rechnen in MATLAB leicht fallen, und umgekehrt.

Einige weitere kostenlose Programmier- und Rechenumgebungen funktionieren ebenfalls mehr oder weniger ähnlich wie MATLAB: Scilab, Octave, FreeMat, Julia, R. Auch dafür gilt: Wenn Sie mit MATLAB gut vertraut sind, wird Ihnen der Einstieg zu diesen Softwarepaketen leicht fallen (und umgekehrt).

Ü 1.2 Der Anfang

Starten Sie MATLAB auf Ihren Rechnern wie übliche MS-Windows-Anwendungen durch Doppelklick auf das MATLAB-Zeichen. Sobald die MATLAB-Benutzeroberfläche (MATLAB *Desktop*) am Schirm erscheint, sind Sie bereit für die erste Lektion.

Sie sehen die MATLAB-Benutzeroberfläche mit der *Command Window* als großes Fenster in der Mitte.



Wenn die MATLAB-Benutzeroberfläche anders aussieht als hier nebenan abgebildet – es gibt sehr viele Konfigurationsmöglichkeiten – dann stellen Sie zuerst einmal die voreingestellte Standard-Oberfläche wieder her. Finden Sie das *Layout*-Symbol in der Mitte der oberen Leiste und klicken Sie auf *Layout–SELECT LAYOUT–Default*

Probieren Sie, wenn Sie wollen, einige der sonst noch angebotenen Möglichkeiten aus. Damit die Oberfläche wirklich so aussieht, wie hier gezeigt, müssen Sie *Layout–SHOW–Command History–Docked* einstellen.

Aber zum Schluss stellen Sie bitte mit *Default* das Standard-Layot wieder her. Sie sehen dann – von links nach rechts – vier Fenster: *Current Folder, Command Window, Workspace, Command History*. Wir arbeiten erst einmal im *Command Window*. Um die weiteren Fenster kümmern wir uns später.

Das *Command Window* liegt wie ein großes leeres Blatt vor Ihnen. Aber anders als in einem Textverarbeitungsprogramm können Sie nicht einfach irgendwohin schreiben, sondern immer nur in die aktuelle Befehlszeile. Die ist am Schirm durch die Eingabeaufforderung (den *prompt*) `>>` markiert.

Fangen Sie an und geben Sie die folgenden Befehle ein. Was Sie eintippen sollen, ist auch in den Unterlagen durch vorangestelltes `>>` markiert. Darunter folgt die Antwort von MATLAB (das, was Sie auf dem Schirm sehen sollten, nachdem Sie die Eingabetaste gedrückt haben).

```
>> 3-2

ans =

     1
```

Geben Sie 3-2 ein. Das Resultat wird unter dem Standard-Namen „ans“ (wie „eins“, oder „answer“, je nach Muttersprache) gespeichert.

```
>> ans

ans =

     1
```

Sie können das Resultat unter der Bezeichnung „ans“ jederzeit wieder abrufen...

```
>> zwa=ans+ans

zwa =

     2
```

...oder in weiteren Rechenausdrücken verwenden. Sie können den Wert eines Ausdrucks auch einer Variablen zuweisen.

```
>> y=2^2 + log(pi)*sin(zwa);
>> y

y =

    5.0409

>>
```

Sie berechnen $2^2 + \log(\pi) \cdot \sin(2)$. Ein **Strichpunkt** am Ende unterdrückt Ergebnis-Ausgabe und Zeilenvorschub. MATLAB erinnert sich trotzdem an y. Sie können den Wert von y jederzeit abrufen, indem sie einfach „y“ eintippen

```
>> format long g
>> y
y =
          5.04089993961332
>> y^10
ans =
          10594507.4098595
>>
>> format long e
>> y^10
ans =
     1.059450740985953e+07
>> format short g
>> y^10
ans =
   1.0595e+07

>>
```

MATLAB rechnet mit etwa sechzehnstelliger Genauigkeit. In der Standard-Einstellung zeigt es nur vier Nachkommastellen an. Sie können unterschiedliche Anzeigeformate einstellen. Das `e`-Format verwendet immer die Exponentialschreibweise, das `g`-Format schreibt, wenn möglich, das Ergebnis als Gleitkommazahl ohne Exponent.
`format compact` unterdrückt Leerzeilen bei der Ausgabe, damit passt mehr Ausgabe auf eine Bildschirmseite. `format loose` fügt Leerzeilen zur besseren Lesbarkeit ein.

```
>> theta=acos(-1)

theta =

    3.1416
```

MATLAB kennt trigonometrische Funktionen. Das ist der Arcus Cosinus von -1 (**Im Bogenmaß!**)

```
>> help elfun

  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    .
    .
    .
```

MATLAB kennt eine große Menge sogenannter elementarer Funktionen. Sie können so eine Liste verfügbarer Funktionen aufrufen. Zu jeder einzelnen Funktion lässt sich weitere Hilfe anfordern.

Versuchen Sie, über die verschiedenen Möglichkeiten der MATLAB-Hilfe herauszufinden: Wie heißt die Funktion `sinh` mit vollem Namen?

Arbeitsauftrag: Verwenden Sie den *help browser* und zeichnen Sie einen Funktionsgraph Die ausgiebige MATLAB-Hilfe (der *help browser*) gibt mehr Information zu `sinh` und zeigt auch gleich, wie sich die Funktion plotten lässt.

Lassen Sie MATLAB nach der Anleitung in der Hilfe einen Funktionsgraph von `sinh` zeichnen.

Ü 1.2.1 Eingabe wiederholen, frühere Befehle kopieren

Wenn Sie sich irgendwo vertippen und die Eingabe wiederholen müssen, brauchen Sie nicht alles erneut eintippen. Sie können mit der Taste ↑ früher eingegebene Befehle hereinholen und gegebenenfalls korrigieren.

Wenn Sie den oder die ersten Buchstaben eines früheren Befehls eingeben und dann ↑ tippen, erinnert sich MATLAB und vervollständigt jene Befehle, die mit diesen Buchstaben begannen.

Sie können auch aus dem Fenster rechts unten, der *Command History*, Befehle mit Rechtsklick-Copy kopieren und mit Rechtsklick-Paste in das *Command Window* einfügen. Noch schneller geht Rechtsklick-*Evaluate Selection* oder Markieren-F9

Arbeitsauftrag: Zeichnen Sie Ihnen vertrautere Funktionen: Sinus, Cosinus, Exponentialfunktion, nach dem Muster des vorigen Abschnitts. Tippen Sie nicht alles wieder neu ein. Verwenden Sie die Pfeiltaste und/oder die *Command History*, um frühere Befehle herzuholen und nur den Funktionsnamen auszubessern.

Ü 1.2.2 Ausgabe unterdrücken, Schirm reinigen, Notbremse

Geben Sie folgenden Befehl ein:

```
>>  0:10000

ans =

  Columns 1 through 13

    0     1     2     3
  .
  .
  .
       9990        9991

  Columns 9997 through 10001

       9996        9997

>>
```

Dieser Befehl erzeugt eine lange Liste. Er dient hier nur als Beispiel für eine Anweisung, die den Schirm mit Unmengen von Output vollramscht. Solche Befehle sollten Sie einen Strichpunkt abschließen. Dann werden intern die Berechnungen durchgeführt, aber Bildschirmausgabe unterdrückt.

```
>>  0:10000;
>>
```

So ists besser.

Wenn aber doch einmal versehentlich eine Anweisung nicht und nicht enden will: Die Strg C -Taste beendet die laufende MATLAB-Berechnung. Sie können damit auch begonnene Befehlszeilen löschen.

Wenn MATLAB auch auf Strg C nicht reagiert, lässt es sich nur mehr über den *Windows Task Manager* stoppen. Dann ist aber die gesamte Sitzung mit allen bisher berechneten Werten verloren.

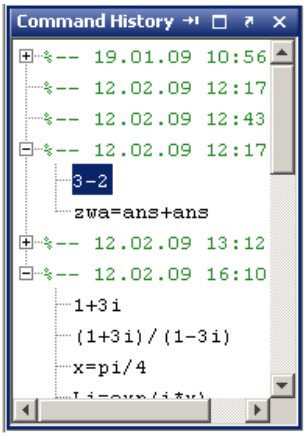
Wenn Sie alle Ein- und Ausgaben im *Command Window* löschen wollen, verwenden Sie den Befehl

```
>> clc
>>
```

Jetzt wird reiner Tisch gemacht

Ü 1.3 Speichern von Befehlen in einer Skript-Datei

Wenn Sie bis jetzt emsig gearbeitet haben, wollen Sie vielleicht auch Ihr Werk abspeichern. Das Fenster links unten, die *Command History*, hat alle Befehle soweit getreulich bewahrt. (Wenn Sie dieses Fenster nicht sehen, müssen Sie *Layout–SHOW–Command History–Docked* einstellen!)



Scrollen Sie durch die Befehle der *Command History*. Sie sehen alle von Ihnen eingegebenen Befehle. Wenn Sie weiter in die Vergangenheit zurückscrollen, finden Sie – durch Datum- und Zeitangabe gekennzeichnet – Befehle früherer Sitzungen. Sie lassen sich durch Anklicken öffnen oder schließen.

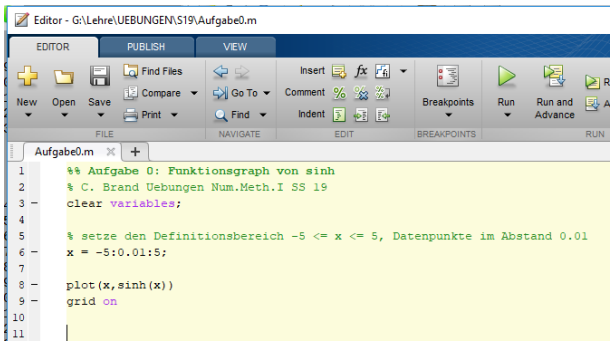
Angenommen, Sie wollen die Befehle der ersten Übungsaufgaben (Funktionsgraph von Sinus hyperbolicus zeichnen) speichern. Gehen Sie so vor:

Markieren Sie (Klick!) den ersten Befehl; halten Sie die Umschalttaste ⇑ gedrückt und scrollen sie in der Befehlsliste nach unten; markieren Sie (Klick!) den letzten Befehl. Die markierten

Befehle sind nun dunkelblau hinterlegt. Recktsklick→*Create Script*. Es öffnet sich ein Fenster des Editors.

Im Editor können Sie – anders als im *Command Window* – Befehle bearbeiten, wie in einem Texteditor. Sie können Zeichen an beliebiger Stelle einfügen und löschen

Empfehlenswert ist, *Kommentare* (beginnen mit %) einzufügen. Ihre Datei könnte dann so aussehen:



Speichern Sie Ihre Datei in der Windows-üblichen Weise (Menü *Save* – *Save as...*) – am besten gleich direkt auf einen mitgebrachten USB-Stick. MATLAB Skript-Dateien bekommen dabei automatisch den Typ `-.m`.

Mit dem Start-Symbol oben in der Menüleiste können Sie ihre Befehle als *Skript-M-Datei* starten. Das wirkt so als würden Sie die Befehle direkt ins *Command Window* eingeben. Die entsprechende Ausgabe erscheint im *Command Window*.

Die Skript-Datei soll aber tatsächlich alle Befehle enthalten, die zur Lösung einer Aufgabe notwendig sind. Typischer Anfängerfehler: das Skript verwendet Variablen, die im Skript nicht definiert sind. Es funktioniert, weil Sie die Variablen vorher definiert haben und daher die Variablen bereits im im Workspace vorhanden sind. Aber wenn Sie oder jemand anderer in einer neuen Matlab-Sitzung das Skript startet, kommt eine Fehlermeldung.

> Stellen Sie sicher, dass alle benötigten Variablen im Skript definiert werden!
> Tipp: Geben Sie ganz zu Beginn des Skripts den Befehl `clear variables`. Damit löschen Sie alle Variablen im Workspace und merken sofort, ob sie Definitionen im Skript vergessen haben

Aufgabe 1: Skript-Datei für Funktionsplot

Suchen Sie sich in MATLAB aus den Bereichen *Trigonometry, Exponents and Logarithms* oder, wenn Sie besonders neugierig sind, *Special Functions* eine Funktion aus, die Sie noch nicht kennen. Erstellen Sie ein Skript, das den Graphen dieser Funktion zeichnet. (Orientieren Sie sich am Screenshot auf Seite Ü-5)

> Tipp: Legen Sie sich für die Matlab-Dateien, die Sie in diesen Übungen erstellen, ein eigenes Unterverzeichnis an.
> Speichern Sie – falls Sie auf Uni-Rechnern arbeiten – am besten auf einem mitgebrachten USB-Datenträger!

Im Editor-Fenster sehen Sie, wenn die Skript-Datei fehlerfrei gespeichert ist, oben Mitte ein grünes „Run“-Pfeil-Symbol. Klick auf „Run“ führt alle Befehle im Skript erneut aus.

(Wenn eine Dialog-Box erscheint mit einem Text der Art *File D:/work/Aufgabe1.m is not found in the current folder blabla blabla*, dann klicken Sie einfach auf „Change Folder“)

Ü 1.4 Funktionen vom Typ $y = f(x)$ zeichnen

In diesem Abschnitt sollen Sie lernen, mit der Anweisung `plot` einfache Funktionsgraphen in der xy-Ebene (Die MATLAB-Hilfe spricht von „linear 2D-plots“) zu erstellen.

Arbeiten Sie im *Command Window* das folgende Beispiel durch. Es sollen die Graphen zweier Funktionen,

$$y = 3\cos x \quad \text{und } z = \log x$$

für den Definitionsbereich $0 < x \leq 25$ gezeichnet werden.

```
>> x=linspace(0.1,25,50)
```

Damit erzeugen Sie einen Zeilenvektor x, der 50 äquidistante Werte im Intervall $[0,1; 25]$ enthält. Entsprechend lang ist der Output am Schirm.

```
x =

  Columns 1 through 7

    0.1000    0.6082
1.1163    1.6245    2.1327
2.6408    3.1490
  .......

  Columns 43 through 49

   21.4429   21.9510   22.4592
22.9673   23.4755   23.9837
24.4918

  Column 50

   25.0000
```

Sie sehen, es werden tatsächlich 50 Werte erzeugt. In MATLABs Sichtweise ist x eine Matrix mit einer Zeile und 50 Spalten (deswegen „Columns“ im Output).

```
>> x=linspace(0.1,25,100);
```

Wiederholen Sie die Eingabe (mithilfe der Pfeiltaste), aber lassen Sie 100 Werte berechnen und fügen Sie einen Strichpunkt zum Abschluss an: Dadurch wird die Ausgabe unterdrückt und der Bildschirm bleibt übersichtlicher.
Übrigens werden, auch wenn Sie die Anzahl der Werte nicht angeben, also den Befehl in der Form `x=linspace(0.1,25);` verwenden, standardmäßig 100 Werte erzeugt.

```
>> y = 3*cos(x);
```

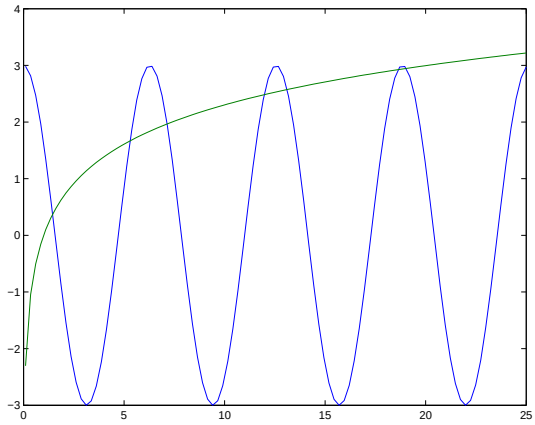
Dieser Befehl berechnet für jede Komponente im Vektor x den entsprechenden y-Wert.

```
>> plot(x,y)
```

Und damit erhalten Sie eine Zeichnung des Funktionsgraphen.

```
>> z=log(x);
>> plot(x,y,x,z)
```

Erzeugen Sie gleich noch einen zweiten Vektor, der den Funktionswerten von $\log x$ entspricht. Sie können beide Funktionsgraphen in ein Schaubild zeichnen. Achten Sie darauf, dass für die zweite Funktion nochmal die Angabe der x-Werte notwendig ist, obwohl beiden Funktionsgraphen die gleichen x-Werte zugrunde liegen.

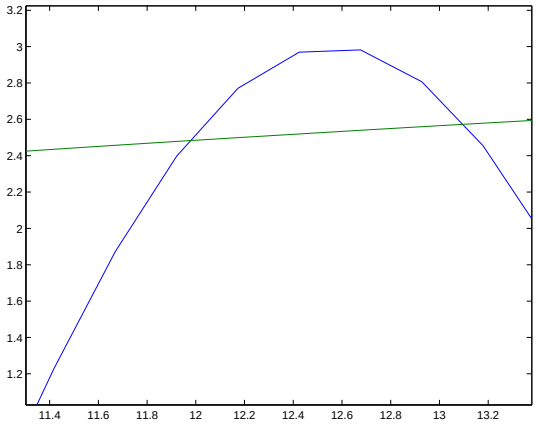


Der Graph sollte der obigen Abbildung entsprechen. Die Schnittpunkte der beiden Kurven entsprechen Lösungen der Gleichung

$$3\cos x = \log x$$

Vergleichen Sie dazu Abbildung 1 und Kapitel 1.4 im Vorlesungsskriptum, wo dieses Beispiel ausführlich diskutiert wird.

Wenn Sie die Zoom-in-Schaltfläche (mit dem Lupen-Symbol) anklicken, können Sie danach mit der Maus einen Bereich der Graphik vergrößert darstellen. Sie können also die Lösungen der Gleichung $3\cos x = \log x$ nähherungsweise aus der graphischen Darstellung ablesen. Das untenstehende Beispiel zeigt zwei Lösungen in der Nähe von $x = 11,9$ und $x = 13,1$.



Die Vergrößerung zeigt aber auch, dass nicht die „wirklichen“ Funktionen dargestellt werden, sondern die einzelnen Datenpunkten der Vektoren x, y und z stückweise durch Geraden verbunden sind. Deswegen kann man aus der Graphik die Lösungen der Gleichung $3\cos x = \log x$ natürlich nur im Rahmen der Zeichengenauigkeit darstellen.

Ü 1.5 Weitere Übungsbeispiele

Sie können, so weit Sie kommen, die folgenden Beispiele während dieser Übungseinheit ausarbeiten oder sonst auf Ihrem eigenen Rechner fertigstellen.

Speichern Sie die Aufgaben als Skript-Dateien auf einem USB-Datenträger, sodass Sie bei Befragungen Ihre Beispiele vorweisen können.

Aufgabe 2: Lösungen von $3\cos x = \log x$

Fertigen Sie als Skript-Datei eine genauere Version der obigen Funktionsgraphen mit 1000 Datenpunkten an. Geben Sie Ihrem Bild auch einen Titel und beschriften Sie die Achsen. (Informieren Sie sich in der MATLAB-Hilfe, beim Übungsleiter oder einer hilfreichen Nachbarin, wie das geht!)

Bestimmen Sie aus der Graphik alle Lösungen der Gleichung.

Hinweis: Beginnen Sie mit `x=linspace(0.1,25,1000);`

> Wenn Sie die Unterlagen in Papierform vor sich haben oder in Ihre digitale Kopie hineinschreiben können, tragen Sie hier Antworten ein. Schreiben Sie geforderte Antworten jedenfalls auch als Kommentarzeile in Ihr Skript.

Die Lösungen sind (auf drei Nachkommastellen genau):

_____

_____

Aufgabe 3: Quadratische Gleichung

Gesucht ist die betragskleinere Wurzel von

$$x^2 - 12345678x + 9 = 0$$

Verwenden Sie Rechenbefehle im *command window* und speichern Sie zum Schluss in übersichtlicher und kommentierter Form als Skript-Datei.

(Sinnvoller Weise setzen Sie dazu das Anzeigeformat `format long e`)

- Die gängige Lösungsformel $x_{1,2} = -p/2 \pm \sqrt{p^2/4 - q}$ liefert

  _____

- Die numerisch korrekte Berechnung der betragsmäßig *kleineren* Lösung einer quadratischen Gleichung berechnet zuerst die *betragsgrößere* Lösung x_{gr} mit der Standard-Formel. Die betragskleinere Lösung x_{kl} ergibt sich dann aus

  $$x_{\mathrm{kl}} = \frac{q}{x_{\mathrm{gr}}}$$

  Ergebnis: _____

- Auflösen nach dem linearen Term,

  $$x = \frac{x^2 + 9}{12345678}$$

  Fixpunkt-Iteration in der Form

```
>> x=0;
>> x = (x^2+9)/12345678
x =
    7.290000597780049e-007
```

Wiederhole, bis sich Wert nicht mehr ändert

Ergebnis: _____

Aufgabe 4: Fixpunkt-Iteration:

Finden Sie wie in der vorigen Aufgabe durch wiederholtes Auswerten einen Fixpunkt der Funktion $\phi(x) = 1/\exp x$. Startwert 5, Genauigkeit `format long e`.

Ergebnis: _____

Untersuchen Sie auch, wie sich von einem Schritt zum nächsten die Genauigkeit erhöht. Können Sie eine Regel der Art „pro Dezimalstelle Genauigkeit sind durchschnittlich x Iterationen notwendig“ finden?

Aufgabe 5: Wurzelbehandlung:

Schon die alten Babylonier berechneten Quadratwurzeln $\sqrt{a}$ mit der (oft als Heron-Verfahren bezeichneten) Iteration

$$x^{(0)} = a; \quad x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \text{ für } \quad k = 0, 1, 2, \ldots$$

Berechnen Sie $\sqrt{2}$ durch wiederholtes Auswerten der Iterationsvorschrift (Genauigkeit `format long e`). Testen Sie auch andere Wurzelberechnungen, z.B. $\sqrt{2005}$, $\sqrt{4711}$, $\sqrt{0{,}815}$ ... Untersuchen Sie bei allen Beispielen, wie sich von einem Schritt zum nächsten die Anzahl richtiger Stellen erhöht. Welche der folgenden Regeln beschreibt das Konvergenzverhalten am besten?

1. Der Fehler halbiert sich mit jeder Iteration
2. Pro Iteration gewinnt man zwei korrekte Dezimalstellen
3. Pro Iteration gewinnt man drei korrekte Dezimalstellen
4. Pro Iteration verdoppelt sich annähernd die Anzahl korrekter Stellen

Aufgabe 6: Newton-Verfahren:

Finden Sie mit dem Newton-Verfahren für $f(x) = x \tan x - 1$ die Nullstelle in der Nähe des Startwertes $x^{(0)} = 1$. Anleitung:

```
>> x=1;
>> f = x*tan(x)-1
f =
    5.574077246549023e-001
>> fstr = ...
fstr =
    4.982926545469661e+000
>> x = x - f/fstr
x =
    8.881364757099055e-001
```

Hinweis: Die Ableitung von $\tan x$ ist $1/\cos^2 x$.

Wiederholen Sie, bis das Ergebnis auf die volle Stellenanzahl genau ist. Prüfen Sie anhand des Endresultates die Anzahl korrekter Dezimalstellen bei den einzelnen Iterationsschritten.

Untersuchen Sie das Konvergenzverhalten (die zunehmende Anzahl korrekter Stellen), indem Sie eine Tabelle erstellen:

Schritt	korrekte Dezimalstellen
0	0
1	
2	
⋮	

Wie lässt sich das Konvergenzverhalten beschreiben?

Die wiederholte Ausführung eines Befehls der Art `x = x - f/fstr` in einem Programm schreit nach einer Schleife. MATLAB kennt natürlich solche Kontrollstrukturen, und wenn Sie damit vertraut sind, dann schreiben Sie in Ihrem Skript eine `for`-Schleife. Sonst reicht es auch, die Anweisung mit copy/paste mehrfach zu wiederholen.

So könnte in einer Skript-Datei der Code aussehen, wenn Sie eine Schleife verwenden:

```
format long e
x=1
for i=1:7
    f = x*tan(x)-1;
    fstr =  ...
    x = x-f/fstr
end
```

Aufgabe 7: Kepler-Gleichung

Die Vorlesungsfolien zur 1. Vorlesung zeigen als Beispiel die Kepler-Gleichung

$$x - \epsilon \sin x = m$$

(sie setzt verschiedene Parameter einer elliptischen Umlaufbahn in Beziehung – aber das müssen Sie nicht wissen!) Angenommen, $\epsilon = 1/10$ und $m = 2$ sind gegeben; x ist gesucht. Verschiedene Lösungswege sind möglich:

- Ablesen aus geeigneter graphischer Darstellung.
- Durch Fixpunkt-Iteration
- Newtonsches Verfahren
- Sekanten-Methode
- Intervallhalbierung

Suchen Sie sich zwei davon aus und schreiben Sie dazu ein Skript. Ihr Skript soll eine Folge von Näherungswerten ausgeben. Auf den Vorlesungsfolien sind auch Zahlenwerte angegeben, damit können Sie vergleichen.

Analysieren Sie das Konvergenzverhalten ihres Verfahrens und geben Sie an, wie sich der Fehler von einem Schritt zum nächsten reduziert.

Ü 1.6 Kurven vom Typ $\mathrm{x} = \mathrm{f}(\theta); \mathrm{y} = \mathrm{g}(\theta)$ zeichnen

Eine Funktion $f : x \mapsto f(x)$ weist jedem x-Wert genau einen y-Wert zu. Ein Kreis lässt sich in dieser Form nicht beschreiben, weil hier zum gleichen x-Wert zwei y-Werte gehören können.

In solchen Fällen kann man Kurven in Parameterform darstellen. Dabei sind sowohl die x- als auch die y-Werte Funktion eines Parameters (der hier θ heißt).

Wir wollen einen Kreis mit Radius 1 zeichnen. Damit der Kreis auch wirklich jenes Aussehen hat, das wir erwarten, erzeugen wir 100 Punkte, die auf dem Kreis liegen. Dazu verwenden wir die Beziehungen:

$$x = \cos\theta, \qquad y = \sin\theta, \qquad 0 \le \theta \le 2\pi$$

Die Folge der MATLAB-Anweisungen, die unsere Aufgabe lösen, könnte wie folgt aussehen:

```
>> theta = linspace(0, 2*pi, 100);
```
Erzeugt 100 äquidistante Werte zwischen 0 und 2π.

```
>> x=cos(theta);
>> y=sin(theta);
```
Erzeugt die x- und y- Koordinaten der 100 Punkte.

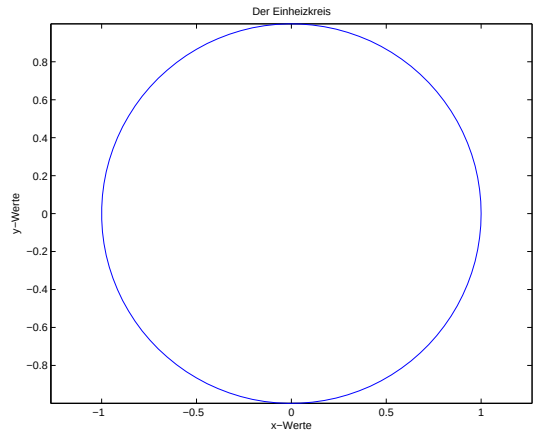
```
>> plot(x,y);
```
Zeichnet die Verbindunglinien von Punkt 1 zu Punkt 2 zu ... Punkt 100

Weitere Befehle, mit denen Sie Ihre Zeichnung formatieren können, sind

```
>> axis('equal');
```
Setzt die Skalierung der beiden Achsen gleich.

```
>> xlabel('x-Werte')
>> ylabel('y-Werte')
>> title('Der␣Einheizkreis')
```
Die x- und y-Achse werden beschriftet und Ihr Werk betitelt. (Strichpunkt oder nicht als Abschluss macht bei diesen Befehlen keinen Unterschied.)

Ihr Werk sollte nun so aussehen:



Sie können die Graphik auch interaktiv formatieren, wie Sie es von anderen Windows-Anwendungen gewohnt sind. Oben, neben dem Drucker-Symbol finden Sie die Schaltfläche „Edit plot“. Alternativ können Sie auch aus dem Menue „Tools–Edit Plot“ wählen. Danach können Sie durch Doppelklick Achsen oder andere Elemente der Zeichnung weiter editieren. Andere Schaltflächen erlauben Ihnen, Text, Pfeile oder Linien einzufügen.

Wichtig sind noch die Möglichkeiten, die Graphik zu speichern. Unter dem Menüpunkt „File–Export“ lässt sich die Graphik in verschiedenen Formaten speichern.

Ü 1.7 Elementweise Rechenoperationen

Es ist Ihnen vielleicht noch gar nicht bewusst gewesen: Sie haben beim Auswerten von Funktionstermen Rechenoperationen auf Datenvektoren angewandt. Der Befehl `x = cos(theta)` funktioniert sowohl für skalares θ, als auch für einen Datenvektor mit n Elementen. Auch Addition und Subtraktion lassen sich gleichermaßen auf skalare wie vektorielle Operanden anwenden.

Für Multiplikation, Division und die Potenzfunktion müssen Sie beim Verknüpfen von Datenvektoren aber unbedingt die elementweisen Operatoren `.*` `./` `.^` verwenden. Grund: Die „gewöhnlichen“ Operator-Symbole `*` `/` `^` verknüpfen in MATLAB Matrizen und Vektoren nach den Regeln der Matrix-Algebra. Diese Regeln wollen Sie *nicht* anwenden, wenn Sie Datenvektoren in Funktionsterme einsetzen.

Anfangs ist es oft schwer zu durchblicken, wann MATLAB Rechenoperationen von sich aus elementweise interpretiert, und wann man explizit den Punkt setzen muss.

> **Faustregel:** in Funktionstermen immer die elementweisen Operatoren verwenden, wenn Datenvektoren verknüpft werden.

Die Aufgaben 8 und 10 erfordern diese Unterscheidung. Dazu noch Hinweise:

```
>> t = linspace(0, 2*pi, 100);
>> x = sin(t)*2 - sin(t*2);
```

Erster Funktionsterm in Aufgabe 8: Hier reicht der „gewöhnliche“ `*` Operator, weil er nur in Operationen vom Typ „Skalar mal Vektor“ auftritt.

```
>> x = sin(t).*2 - sin(t.*2);
```

Es wäre aber auch nicht falsch, die Punkt-Form zu verwenden. Das Ergebnis bleibt gleich.

```
>> r = 2 - 2*sin(t) + sin(t) .* abs(cos(t)).^0.5 ./ (sin(t)+7/5)
```

Ein anderer Funktionsterm in Aufgabe 8: Aber hier muss überall, wo Vektoren verknüpft werden, ein Punkt-Operator stehen!

Ü 1.8 Weitere Übungsbeispiele

Dokumentieren Sie Ihre Übungsbeispiele als Skript-M-Dateien. Speichern Sie Zeichnungen als `JPEG`-Dateien ab, dann sind diese Dateien außerhalb MATLABs mit jedem Bildbetrachtungsprogramm anzusehen. Wenn Sie Zeichnungen als `.fig`-Dateien abspeichern – das ist MATLABs Voreinstellung – dann können Sie diese Zeichnungen nur innerhalb Matlabs öffnen.

Aufgabe 8: Zum Valentinstag (ist zwar schon vorbei, aber besser spät als nie)

Zeichnen Sie die Kardioide oder Herzkurve. Das ist eine Kurve, die durch folgende Parameterdarstellung für $0 \leq t < 2\pi$ gegeben ist.

$$x = 2\sin(t) - \sin(2t)$$
$$y = 2\cos(t) - \cos(2t)$$

Der Name Kardioide geht auf Giovanni di Castiglione, einen italienischen Mathematiker im 18. Jahrhundert, zurück. Für mich sieht sie mehr wie ein Apfel ohne Stängel aus. Eine schönere Herzkurven-Gleichung ist hier gegeben[10]:

$$r = 2 - 2\sin t + \sin t \frac{\sqrt{|\cos t|}}{\sin(t) + 7/5}$$
$$x = r\cos(t)$$
$$y = r\sin(t)$$

Hinweis: Verwenden Sie Sie für $0 \leq t < 2\pi$ mindestens 500 Teilpunkte und achten Sie darauf, an den richtigen Stellen `.*` und `./` zu verwenden!

Aufgabe 9: 3D-Plots

Zeichnen Sie mittels `plot3(x,y,z)` eine Spirale:

$$x(t) = \sin(t), \quad y(t) = \cos(t), \quad z(t) = t, \quad 0 \leq t \leq 20.$$

Verwenden Sie die Schaltfläche „Rotate 3D“, um Zeichnungen der Kurve aus zwei verschiedenen Blickwinkeln anzufertigen.

Aufgabe 10: Für die Fisch

Diskutieren Sie die Funktion

$$y = 3 - \frac{1}{2}\arctan(\frac{2\,x - 5}{x - 2}) + \frac{1}{10}\sin(7\,x) \quad 0 \leq x \leq 4$$

anhand eines Graphen. Wo liegen Supremum und Infimum? In welchen Bereichen ist die Funktion stetig? differenzierbar? An welcher Stelle ragt eine Haifischflosse aus den Wellen?

Wenn die letzte Frage für Sie sinnlos erscheint, haben Sie die Funktion vermutlich falsch gezeichnet. Beachten Sie bei der Berechnung der y-Werte, dass alle Rechenoperationen elementweise (für jeden einzelnen x-Wert) berechnet werden müssen. Wenn einer der Partner in einer Rechenoperation ein Skalar, der andere ein Vektor ist, funktioniert das automatisch, wie z.B. in der Anweisung `2*x-5`. Wenn beide Partner Vektoren sind, müssen Sie unbedingt die elementweisen Operatoren `.*` und `./` verwenden. Beispiel: der Ausdruck $(2x-5)/(x-2)$ muss geschrieben werden `(2*x-5)./(x-2)`.

[10] Quelle: `http://mathworld.wolfram.com/HeartCurve.html`

Ü 2 Zweite Übungseinheit

Inhalt der zweiten Übungseinheit:

- Zeilen- und Spaltenvektoren
- Matrizen, Abbildungen
- Funktionen
  - Funktionen als Funktions-M-Datei programmieren
  - Fixpunkte und Nullstellen von Funktionen
  - Befehle `fzero, roots`
  - Funktions-Einzeiler (*anonymous functions*)
- Kontrollstrukturen
  - Schleifen
  - Verzweigungen
- Fixpunkt-Iteration, ein- und mehrdimensional

Ü 2.1 Zeilen- und Spaltenvektoren

Sie haben in der vorigen Einheit bereits mit Vektoren gearbeitet: Beim Zeichnen von Funktionsgraphen haben Sie Bereiche für x- und y-Werte als Zeilenvektoren erzeugt. MATLAB unterscheidet zwischen Zeilen- und Spaltenvektoren.

Hier folgt eine kurze Zusammenfassung zum Erzeugen von und Rechnen mit Vektoren. Klicken Sie sich durch diese Anleitung, dann lernen Sie die wichtigsten Befehle dazu kennen. Überlegen Sie sich bei jedem Ergebnis, welche Rechenregel MATLAB dabei verwendet hat!

- Standard-Operationen der Vektorrechnung (Addition `+` , Subtraktion `-` , Multiplikation `*` mit Skalar, inneres Produkt `*` von Zeilen- mit Spaltenvektor, Vektor transponieren `'`)
- elementweise Multiplikation `.*` und Division `./` bei Vektoren gleicher Form. Ergebnis ist ein Vektor gleicher Form; die entsprechenden Elemente werden multipliziert bzw. dividiert.
- Wenn ein Operand Zeilen- und der andere Spaltenvektor ist, erzeugt Addition oder Subtraktion eine Matrix nach dem Muster „jedes Element mit jedem“.
- Bei der Multiplikation kommt es drauf an: linker Partner Zeile, rechter Partner Spalte berechnet inneres Produkt; linker Partner Spalte, rechter Partner Zeile berechnet eine Matrix nach dem Muster „jedes Element mit jedem“ (heißt auch äußeres, Kronecker- oder Tensor-Produkt - da ist sich die Fachwelt nicht einig).
- Bei der Division `/` zweier Zeilen- bzw. zweier Spaltenvektoren passieren völlig bizarre Dinge – was MATLAB da berechnet, versuchen wir hier erst gar nicht zu erklären.

Vektoren erstellen

```
>> x=[1 2 3]
x =
     1     2     3
```

x ist ein Zeilenvektor mit drei Elementen.
Vektoren sind von eckigen Klammern `[ ]` begrenzt.

```
>> x=[1, 2, 3]
x =
     1     2     3
```

So geht 's auch: Die einzelnen Komponenten in einer Zeile können Sie durch Leerzeichen (so wie gerade vorher) oder durch Beistriche (so wie hier) trennen.

```
>> y=[2;1;5]
y =
     2
     1
     5
```

y ist ein Spaltenvektor mit drei Elementen. Die Strichpunkte ; trennen die Zeilen im Vektor.

```
>> y = [2
1
5]
y =
     2
     1
     5
```

Statt des Strichpunkten können Sie im *command window* oder in Dateien auch eine neue Zeile beginnen.

Rechenoperationen

```
>> z=[2 1 0];
>> a=x+z

a =

     3     3     3
```

Sie können zwei Zeilen- oder zwei Spalten-Vektoren addieren oder subtrahieren – das entspricht den Standardregeln der Vektorrechnung

```
>> b=2*a

b =

     4     4     0
```

Skalar mal Vektor, das ist eine Standard-Regel

```
>> x/2
ans =
    0.5000    1.0000    1.5000
```

Division Vektor durch Skalar, ebenfalls Standard

```
>> 2/x
Error using  /
Matrix dimensions must agree.
```

in dieser Reihenfolge nicht möglich

```
>> 2./x
ans =
    2.0000    1.0000    0.6667
```

Das geht: Elementweise Division

```
>> b=x+y
b =
     3     4     5
     2     3     4
     6     7     8
```

Addition von Zeilen- plus Spaltenvektor ergibt eine Matrix – Das ist nicht gerade Standard, sondern MATLABs kreative Erweiterung des Plus-Operators.

Achtung bei älteren MATLAB-Versionen! Versionen vor R2016b erlauben nicht, Zeilen- zu Spaltenvektoren zu addieren. Obiger Befehl ergibt eine Fehlermeldung

```
» b=x+y
Error using +
Matrix dimensions must agree.
```

Die kreative Interpretation der Standard-Regeln in den neueren Versionen gilt für viele arithmetische und logische Operatoren. Regel: Wenn es irgendwie sinnvoll erscheint, erweitert MAT-

LAB die Operanden so, dass eine elementweise Operation möglich wird[11].

MATLAB führte dieses automatische Erweitern 2016 mit der Begründung: [12] ein: *"MATLAB has a long history of inventing notation that became widely accepted"* – die Kommentare im Blog sind eher kontroversiell. Nicht alle Anwender freut so einen lockerer und origineller Umgang mit den Rechenregeln.

> Vorsicht bei Rechenoperationen mit Vektoren! Wenn die Formate von Zeilen- und Spaltenvektoren nicht den Regeln der Standard-Vektor- und Matrizenrechnung entsprechen, findet MATLAB oft eine kreative, aber meist ungewollte Interpretation der Anweisungen.

```
>> a=x.*z
a =
     2     2     0
```

Sie können zwei Vektoren derselben Größe elementweise multiplizieren (oder dividieren); array operator `.*` (oder `./`)

```
>> x*y
ans =
    19
```

Skalares (oder inneres) Produkt. Standardregel für Zeilen- mal Spaltenvektor

```
>> y*x
ans =
     2     4     6
     1     2     3
     5    10    15
```

Spalten- mal Zeilenvektor ergibt Matrix. Heißt äusseres, Tensor- oder Kronecker-Produkt; auch das ist eine Standard-Rechenoperation.

```
>> x.*y
ans =
     2     4     6
     1     2     3
     5    10    15
```

Elementweise Multiplikation von Zeilen- mit Spaltenvektor ergibt dieselbe Matrix wie oben – MATLABs kreative Interpretation.

Vektoren transponieren

```
>> x
x =
     1     2     3
>> x'
ans =
     1
     2
     3
>> y
y =
     2
     1
     5
>> y'
ans =
     2     1     5
```

Höchste Zeit, dass Sie den ' Operator kennenlernen. Damit transponieren Sie Vektoren: Zeilen- wird Spaltenvektor, und umgekehrt.

[11] Für eine genauere Erklärung suchen Sie in der MATLAB Dokumentation nach *Compatible Array Sizes for Basic Operations*

[12] `https://blogs.mathworks.com/loren/2016/10/24/matlab-arithmetic-expands-in-r2016b/`

Im Matlab-Desktop sehen Sie übrigens die Registerkarte „Workspace“ links in der Mitte, und wenn Sie draufklicken, darüber ein Fenster. (Möglicherweise ist das Fenster schon offen, ohne dass Sie extra angeklickt haben.) Alle bisher verwendeten Variablen sind dort aufgelistet. Das Matrix-Symbol neben dem Variablennamen erinnert daran, dass MATLAB alle Variablen als Matrizen interpretiert - Skalare als 1×1-Matrizen, Zeilenvektoren mit n Elementen als $1 \times n$- und Spaltenvektoren mit m Einträgen als $m \times 1$-Matrizen. Doppelklick auf eine Zeile dieser Liste öffnet ein Fenster, den „Array Editor“, das die Werte der Variablen in Tabellenform anzeigt. Die Werte lassen sich (Doppelklick auf die entsprechende Zelle im Tabellenblatt) auch ändern.

Zeilenvektoren mit regelmäßigen Einträgen

Zum Erzeugen von Zeilenvektoren noch einige Beispiele:

```
>> x=linspace(0,10,5)

x =

         0    2.5000
5.0000    7.5000   10.0000
```

Dieser Befehl erzeugt einen Zeilenvektor der Länge 5, dessen Elemte äquidistant im Intervall [0,10] liegen. Ohne drittes Argument entsteht immer ein Vektor der Länge 100.

```
>> x=0:2.5:10
x =
         0    2.5000
5.0000    7.5000   10.0000
```

Auch mit dem Doppelpunkt-Operator lässt sich derselbe Zeilenvektor erzeugen nach dem Muster `Anfangswert:Schrittweite:Endwert`
Bei gewünschter Vektor-Länge ist `linspace` einfacher; wenn die Schrittweite vorgegeben ist, bietet sich die Doppelpunkt-Variante an.

```
>> x=1:6
x =
     1     2     3     4
5     6
```

Bei Schrittweite 1 ist die Doppelpunkt-Anweisung besonders einfach

Ü 2.2 Matrizen

Der Name MATLAB steht für *matrix laboratory*. Das signalisiert Ihnen: Das Arbeiten mit Matrizen ist in dieser Rechenumgebung ein ganz zentrales Thema. Hier eine kurze Einführung ins Erstellen von und in die Grundrechnungsarten für Matrizen.

Hoffentlich sind Ihnen die Rechenregeln der Matrizenrechnung noch in Erinnerung. Bevor Sie weiterarbeiten – ist Ihnen klar, wir die Multiplikation von Matrizen abläuft? Sie sollten mit Stift auf Papier nachrechnen können:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 0 \end{bmatrix} = \begin{bmatrix} 21 & 24 & \dots \\ 47 & \dots & ? \end{bmatrix}$$

Matrizen erstellen

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
>> B = [ 5, 6, 7
8, 9, 0]
B =
     5     6     7
     8     9     0
```

Eingabe: Leerzeichen oder Komma trennt Komponenten in einer Zeile; Strichpunkt oder neue Zeile trennt Zeilen.

Rechenoperationen, Zugriff auf Elemente

MATLAB bietet ähnlich wie für Vektoren folgende Operationen an:

`+ , -`	elementweise Addition, Subtraktion
`*`	multipliziert nach den Regeln der linearen Algebra
`.* , ./`	elementweise Multiplikation, Division
`A'`	Matrix `A` transponieren

Bei Rechenoperationen zwischen Matrizen ist wichtig, dass die Matrix-Dimensionen zueinander passen. Versuchen Sie für die gerade erstellten Matrizen `A` und `B`

```
>> C=A*B
C =
    21    24     7
    47    54    21
```

Matrix-Multiplikation nach den Regeln der linearen Algebra.

```
>> C=B*A
Error using  *
Incorrect dimensions ...
```

Innere Dimensionen müssen übereinstimmen.

Es is A eine 2×2-, B eine 2×3-Matrix. Bei $A \cdot B$ ist in $(2 \times 2) \cdot (2 \times 3)$ die innere Dimension 2. Bei $B \cdot A$ mit $(2 \times 3) \cdot (2 \times 2)$ passen die inneren Dimensionen nicht: $2 \neq 3$.

Elementweise Operationen lassen sich nur auf Matrizen gleicher Größe anwenden.

```
>> C=A+B
Arrays have incompatible sizes for this operation.
```

```
>> A(1,2)
ans =
     2
```

Zugriff auf Element a_{12}

```
>> B(1:2, 2:3)
ans =
     6     7
     9     0
```

Zugriff auf Elemente in Zeilen 1 bis 2, Spalten 2 bis 3

Ü 2.3 Der Doppelpunkt-Operator

Der Doppelpunkt (englisch: *colon*) ist einer der nützlichsten Operatoren im Umgang mit Matrizen und Vektoren. Er erzeugt Vektoren, Index-Bereiche und Iterationsindizes. Als Index-Ausdruck kann er aus Matrizen einzelne Zeilen oder Spalten herausgreifen.

In der MATLAB-Hilfe finden Sie Informationen dazu unter dem Stichwort *colon*.

Syntax-Beispiele

Erzeugen einfacher Zahlenfolgen

`x=3:8` erzeugt den Zeilenvektor `[3, 4, 5, 6, 7, 8]`

`x=0:2:6` erzeugt den Zeilenvektor `[0, 2, 4, 6]`

`x=10:-1:0` erzeugt den Countdown-Vektor `[10, 9, ..., 3, 2, 1, 0]`

Diese Syntax werden wir für Zählschleifen im Kapitel Ü 2.6 verwenden.

Index-Ausdrücke, Zugriff auf Vektor- und Matrixelemente

Als Matrix- oder Vektorindex bedeutet der Doppelpunkt soviel wie „alle Zeilen“ oder „alle Spalten“.

`A(:,4)` alle Zeilen, Spalte 4; die gesamte vierte Spalte von A

`A(3,:)` Zeile 3, alle Spalten; die gesamte dritte Zeile von A

`x(3:7)` verwendet den Vektor `3:7 = [3, 4, 5, 6, 7]` als Zugriffs-Index; greift den Teil-Vektor bestehend aus den x-Komponenten 3 bis 7 heraus

`A(2:4,3:5)` die Untermatrix bestehend aus den Zeilen 2 bis 4 und Spalten 3 bis 5 von A

Achtung, verwechseln Sie nicht `x(3:7)`, `x(3,7)` und `x([3,7])` ! Der erster Ausdruck ist ein Teilvektor von `x` (siehe oben); der zweite Ausdruck bezeichnet das Matrixelement x_{37}; der dritte verwendet den Vektor `[3, 7]` als Zugriffs-Index, greift also die 3- und 7- Komponente des Vektors x heraus. Das ergibt einen Vektor der Länge 2 mit den Komponenten `[x(3), x(7)]`.

Ü 2.4 Matrizen und Abbildungen

Die Matrix ist dazu gedacht,
dass sie aus einem Vektor einen and'ren macht.

Die linearen Abbildungen zwischen Vektorräumen $\mathbb{R}^m \to \mathbb{R}^n$ entsprechen genau den Matrix-Vektor-Multiplikationen $\mathbf{y} = A \cdot \mathbf{x}$ mit $n \times m$-Matrizen A.

Lineare Beziehungen zwischen Ein- und Ausgangsgrößen sind Grundbausteine für jede Art von Datenanalyse und -modellierung. Spannend und herausfordernd kann das für hochdimensionale Vektorräume werden. Im 2- und 3-dimensionalen Raum hingegen lassen sich Abbildungen noch gut visualisieren. Wenn Sie die Beispiele hier durchgearbeitet haben, kann die Verallgemeinerung auf mehrdimensionale Räume auch kein großer Schritt mehr sein. Laden Sie dazu die Datei `cat.dat` von der Übungs-Homepage in Ihr Arbeitsverzeichnis.

```
>> X=readmatrix('cat.dat');
>> size(X)
ans =
   119     2
>> X=X';
>> size(X)
ans =
     2   119
```

Sie lesen aus einer Textdatei einen Datensatz als Matrix ein. X hat 119 Zeilen und 2 Spalten. Wir vertauschen fürs weitere Arbeiten Zeilen und Spalten. X hat also nun 2 Zeilen und 119 Spalten.

> Für MATLAB-Versionen vor R2019a: Den Befehl `X=readmatrix('cat.dat')` gibt es noch nicht. Laden Sie die Skript-Datei datenX.m herunter und führen Sie diese aus. Danach ist die Matrix `X` im workspace geladen und Sie können mit den Befehlen `size(X)` und `X=X';` weiterarbeiten.

Die Spaltenvektoren in `X` entsprechen 119 Punkten im 2-dimensionalen Raum. Zeichnen Sie diese Punkte, dann können Sie sich die Daten besser vorstellen.

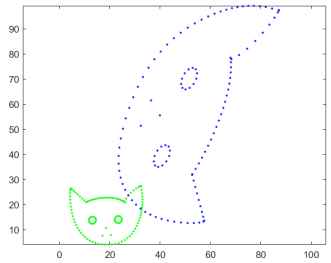
```
>> plot(X(1,:),X(2,:),'b.')
>> axis equal
```

Arnolds Katze[13] blickt Sie an. Gegeben sei nun die Matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} .$$

Wie wirkt diese Matrix auf Punkte des $\mathbb{R}^2$? Sie können die 2×2-Matrix A auf die 119 Spaltenvektoren, die in der Matrix X stehen, einfach durch Multiplikation anwenden: $Y = A \cdot X$ ist eine 2×119-Matrix, deren Spaltenvektoren jeweils die mit A multiplizierten Spaltenvektoren von X sind. Lassen Sie sich Ausgangs- und Bildpunkte zeichnen:

```
A=[ 1 2; 3 0];
Y=A*X;
plot(X(1,:),X(2,:),'g.',Y(1,:),Y(2,:),'b.')
axis equal
```



Sie sehen: A verdreht, vergrößert und verzerrt die Katze. Sie transformiert Ausgangsvektoren in Bildvektoren, die (außer in Sonderfällen) andere Länge und Richtung haben. (Spätestens jetzt sollten Sie das Merksprüchlein vom Anfang des Abschnittes verstehen!)

In diesem Beispiel werden alle Vektoren mehr oder weniger verlängert, aber nicht beliebig groß. Der linke Ohrzipfel, Punkt $\approx$ [32; 27], wird auf [86; 96] abgebildet; das entspricht einer Verlängerung um den Faktor $\approx 3{,}1$.

[13] Der Mathematiker Wladimir Igorewitsch Arnold (1937–2010) illustrierte damit in Lehrbüchern der klassischen Mechanik die Eigenschaften von Transformationen.

```
>> x=[32; 27];
>> y=A*x
y =
    86
    96
>> norm(y)/norm(x)
ans =
    3.0784
```

So können Sie Verlängerungs-Faktoren berechnen (in der 2-Norm).

Versuchen Sie andere Werte für x. Für welchen Vektor finden Sie den größten Verlängerungsfaktor? Umgekehrt: finden Sie auch einen Vektor mit besonders kleinem Verlängerungsfaktor?

Die Norm einer Matrix liefert den maximalen Verlängerungsfaktor

(Das gilt für 1-, 2- oder ∞-Norm; je nachdem, in welcher Norm gemessen wird, können die Werte leicht unterschiedlich sein. Für die Matrix A aus diesem Beispiel ist $\|A\|_2 = 3{,}5266$. Weil sich dieser nicht so einfach aus den Matrixelementen berechnen lässt, arbeitet man auch mit der 1-Norm (maximale Spaltenbetragssumme: 4, der Vektor [1; 0] verlängert sich um diesen Wert in der 1-Norm) oder der ∞-Norm (maximale Zeilenbetragssumme: 3, der Vektor [1;1] verlängert sich um diesen Wert in der ∞-Norm)).

```
>> [norm(A) norm(A,1) norm(A,'inf')]
ans =
    3.2566    4.0000    3.0000
```

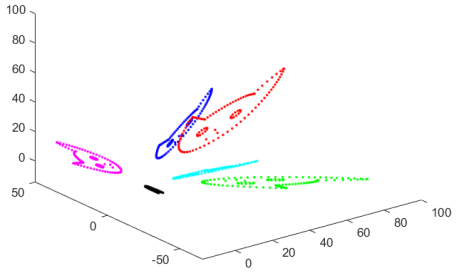
Das sind drei Matrix-Normen von A

Matrizen können aber auch lineare Abbildungen zwischen Vektorräumen unterschiedlicher Dimension vermitteln. Gegeben seien nun die Matrizen

$$B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad , \quad C = \frac{1}{15} \begin{bmatrix} -2 & 2 & 14 \\ -2 & -13 & -16 \\ 14 & 16 & 7 \end{bmatrix} .$$

Wenden Sie auf die Daten-Matrix X zuerst B und dann einige Male C an und sehen Sie Arnolds Katze durch den Raum wandern...

```
Z=B*X;
plot3(Z(1,:),Z(2,:),Z(3,:),'b.')
hold on
Z=C*Z;
plot3(Z(1,:),Z(2,:),Z(3,:),'r.')
Z=C*Z;
...
plot3(Z(1,:),Z(2,:),Z(3,:),'k.')
hold off
```



Aufgabe 11:

Gegeben sind die 2×119 Datenmatrix X wie oben und Matrizen

$$D = \begin{bmatrix} 1 & 1 \\ -\frac{1}{4} & \frac{3}{4} \end{bmatrix} \quad , \quad E = \begin{bmatrix} 1 & 2 \\ -\frac{1}{4} & \frac{3}{4} \end{bmatrix} \quad , \quad F = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ -1 & \frac{1}{2} \end{bmatrix} \quad , \quad G = \begin{bmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Wenden Sie die entsprechenden Abbildungen jeweils mehrere Male auf die Datenmatrix an und zeichnen Sie die Bildpunkte. Für welche Matrizen

- konvergiert die Fixpunkt-Iteration?

- liegt eine kontrahierende Abbildung vor?
  Eine kontrahierende Abbildung garantiert automatisch Konvergenz der Fixpunkt-Iteration. Aber auch für manche nicht-kontrahierende Abbildungen kann Fixpunktiteration konvergieren.
- liegt eine symplektische Abbildung vor?
  Das sind Abbildungen, die zwar Flächen verzerren, aber den Flächeninhalt unverändert lassen. Das können Sie aus den graphischen Darstellungen natürlich nur annähernd schätzen. Vielleicht finden Sie aber auch heraus: neben der Norm, die den Verlängerungsfaktor misst, gibt es eine weitere wichtige Matrix-Größe, die den Flächen- (oder allgemeiner: Volums-) Vergrößerungs-Faktor determiniert.

  Symplektische Abbildungen sind ein wichtiges Thema in der theoretischen Mechanik und für österreichische Radrennfahrerinnen[14].

Ü 2.5 Funktionen

Sie können in MATLAB eigene Funktionen definieren und in sogenannten Funktions-M-Dateien (*function M-files*) speichern. Bei einfache Funktionstermen ist es auch möglich, Funktions-Einzeiler, sogenannte *anonymous functions* zu verwenden.

Dieser Abschnitt erklärt folgende Punkte:

- Eine einfache Funktionsdatei
- Nullstellen mit `fzero` und `roots`
- Funktions-Einzeiler (*anonymous functions*)

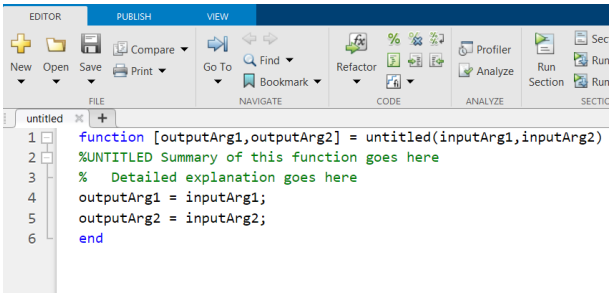
Ü 2.5.1 Eine einfache Funktionsdatei

Gegeben sei die Funktion

$$f : x \mapsto y = 3\cos x - \log x \ .$$

Sie ist ihnen in der Vorlesung und im Skriptum bereits begegnet.

Wählen Sie im MATLAB-Fenster links oben „New–Function“. Sie öffnen damit ein Fenster des MATLAB-Editors, in dem auch schon eine Muster-Funktion drinsteht.



Ergänzen Sie und erstellen Sie einen Quelltext. Orientieren Sie sich am folgenden Muster:

```
function [ y ] = meinf(x)
%MEINF Mein erster Versuch, eine Funktion zu programmieren
%   Funktion aus dem Skript, Abbildung 2
%
% Uebungen NM1, C.B. Feb.2014
y=3*cos(x)-log(x);
end
```

Die erste Zeile einer Funktionsdatei legt die Ein- und Ausgabeparameter sowie den Namen

[14] Anna Kiesenhofer, Integrable systems on b-symplectic manifolds. (Thesis, 2016)

einer Funktion fest. In diesem Fall heißt die Funktion `meinf`, hat eine Variable x als Argument (Input) und liefert einen Funktionswert y (Output). Die Argumente x und y können Skalare, aber auch Felder (Vektoren, Matrizen) sein. Sie können eigene Funktionsnamen verwenden, aber bitte nur zulässige Namen (keine Umlaute und Sonderzeichen, keine Ziffern zu Beginn)

Speichern Sie Ihre Funktion ab. Achten Sie darauf, dass die Datei unter dem Namen `meinf.m` gespeichert wird. Am besten legen Sie sich für die Matlab-Dateien, die Sie in diesen Übungen erstellen, ein eigenes Unterverzeichnis an.

Je nachdem, in welchem Verzeichnis sie gespeichert haben, kann für Sie eine kleine Zusatzarbeit anfallen:

> Skript- und Funktionsdateien lassen sich im *command window* nur aufrufen, wenn sie sich im aktuellen Ordner befinden.
> Wechseln Sie in der MATLAB-Oberfläche, linkes Fenster „Current Folder“ in das Verzeichnis, in dem Sie Ihre Funktion abgespeichert haben

Tipp: Schneller geht es, wenn Sie im Editor-Fenster, nachdem Sie die Funktion gespeichert haben, auf das grüne „Run“-Pfeil-Symbol klicken. Wenn eine Dialog-Box erscheint mit einem Text der Art *File D:/work/meinf.m is not found in the current folder blabla blabla*, dann klicken Sie einfach auf „Change Folder“ und ignorieren Sie weitere Meldungen (*mein.f requires more input...*) wurscht–das Verzeichnis ist gewechselt, mehr wollten wir nicht.

> Die Option *add to path* ist nicht zu empfehlen. Bei Namensgleichheit von Dateien verlieren Sie sonst die Übersicht, welche Datei nun tatsächlich ausgeführt wird.

Wechseln Sie wieder in die MATLAB *Command Window*. Stellen Sie zuerst gleich einmal die Anzeige von mehr Dezimalstellen ein und zusätzliche Leerzeilen ab:

```
>> format long g
>> format compact
>>
```

Andere Möglichkeiten:
short g gibt weniger Stellen.
short oder long e gibt Exponentialschreibweise.
format loose schiebt Leerzeilen ein.

Testen Sie nun, ob sich die Funktion aufrufen lässt:

```
>> y=meinf(1)
y =
   1.62090691760442
```

Sie haben die Funktion an der Stelle 1 ausgewertet.

Wenn Sie brav Kommentare eingegeben haben, gibt es nun eine Belohnung: Geben Sie ein:

```
>> help meinf
 meinf Mein erster Versuch, eine Funktion zu programmieren
    Funktion aus dem Skript, Abbildung 2

  Uebungen NM1, C.B Feb.2014
```

Und noch etwas ist toll an dieser Funktion: sie liefert nicht nur für einen skalaren Wert ein Ergebnis, sondern automatisch auch für eine ganze Werteliste. Erinnern Sie sich: Die Anweisung *Startwert*:*Schrittweite*:*Endwert* ist (alternativ zu `linspace`) eine zweite Möglichkeit, Zeilenvektoren zu erzeugen.

```
>> x=0:1:10

x =

     0     1     2     3
4     5     6     7     8
9    10
```

Werte von 0 bis zehn. Auch x=0:10 hätte funktioniert (Wenn Schrittweite fehlt, nimmt MATLAB automatisch 1)

Die Funktion `meinf` lässt sich elementweise auf den Zeilenvektor x anwenden.

```
>> meinf(x)
ans =   Inf    1.62090691760442
-1.94158769020137 ...
```

Kleiner Schönheitsfehler: $f(x) = 3\cos x - \log x$ ist für $x = 0$ nicht definiert. Der Funktionswert $f(0)$ wird als `Inf` gespeichert.

Ü 2.5.2 Nullstellen mit dem Newton-Verfahren

Dazu müssen Sie auch die Ableitung der Funktion f als Funktions-Datei programmieren. Ausnahmsweise helfen Ihnen die Unterlagen hier noch beim Differenzieren:

$$f'(x) = -3\sin x - \frac{1}{x}$$

```
function y = meindf(x)
%MEINDF Ableitung der Funktion meinf
% es ist so fad, Kommentare
% zu schreiben, aber es zahlt
% sich aus, wenn man eine Woche
% spaeter versucht, zu verstehen,
% was das hier sein soll
y=-3*sin(x)-1./x;
end
```

Öffnen Sie ein neues M-file, programmieren Sie die Ableitung, und speichern Sie unter dem Namen `meindf.m` ab.

```
>> meindf(1)
ans =
   -3.52441295442369
>> x=1:3
x =
     1     2     3
>> meindf(x)
ans =
   -3.52441295442369
-3.22789228047704
-0.75669335751293
```

Prüfen Sie, ob sich die Ableitung korrekt auswerten lässt

Ein kleines, aber wichtiges Detail: Wenn Sie die Division als `1/x` programmieren, funktioniert die Auswertung für *skalare* x problemlos, nicht aber, wenn x ein Vektor ist! Anfangs ist es oft schwer zu durchblicken, wann MATLAB Multiplikationen und Divisionen von sich aus elementweise interpretiert, und wann man explizit den Punkt setzen muss.

> **Regel:** (War schon in der vorigen Einheit da[15])
> In Funktionsdateien immer die elementweisen Operatoren `.* ./ .^` verwenden.

[15] *repetitio est mater studiorum*, auf Deutsch „Wiederholung ist die Mutter der Langeweile“, oder so ähnlich...

Dann hätten wir allerdings oben gemäß der Regel auch die erste Multiplikation mit `.*` schreiben sollen:

```
y=-3.*sin(x)-1./x;
```

In diesem Fall ist es wurscht, ob man `.*` oder `*` schreibt: wenn einer der beiden Operanden ein Skalar ist, wird automatisch elementweise gerechnet.

Zurück zum Newton-Verfahren. Informieren Sie sich notfalls im Skriptum, Kapitel 1.10, über die Rechenvorschrift.

```
>> x=1
x =
     1
>> x=x-meinf(x)/meindf(x)
x =
   1.45990834177644
```

Beginnen Sie mit Startwert 1 und rechnen Sie einen Schritt des Newton-Verfahrens.

Wiederholen Sie den Newton-Schritt, bis sich der Wert der Nullstelle nicht mehr ändert!

Vergleichen Sie ihre Werte mit folgender Tabelle, und beachten Sie das Konvergenzverhalten:

	korrekte Stellen:
1	1
1.45990834177644	1.4
1.44725583798192	1.44725
1.44725861727779	1.447258617277
1.44725861727790	alle

Die Anzahl der richtigen Stellen ist in jedem Schritt mindestens doppelt so groß wie im vorhergehenden. Diese außerordentlich rasche Konvergenz (quadratische Konvergenz) ist charakteristisch für das Newton-Verfahren.

Ü 2.5.3 Nullstellen mit der Sekantenmethode

```
>> xalt = 2
xalt =
     2
>> x = 1
x =
     1
>> xneu = x - meinf(x)*(x-xalt)/(meinf(x)-meinf(xalt))
xneu =
   1.45499210414322
```

Wenn Sie nicht wissen, was das hier soll, ziehen Sie das Skriptum, Kapitel 1.9 oder Vorlesungsunterlagen zu Rate.

```
>> xalt = x; x = xneu;
```

Mit diesen Befehlen setzen Sie die beiden Startwerte der Sekantenmethode neu auf die beiden zuletzt berechneten Werte. Jetzt können Sie die Formel erneut auswerten (aber nicht neu eintippen, Pfeiltaste verwenden!)

```
>> xneu = x - meinf(x)*(x-xalt)/(meinf(x)-meinf(xalt))
xneu =
   1.44716725175157
```

Wiederholen Sie die Auswertung der Sekantenmethode, bis sich der Wert der Nullstelle nicht mehr ändert!

Vergleichen Sie ihre Werte mit folgender Tabelle, und beachten Sie das Konvergenzverhalten:

	korrekte Stellen:
1	1
1.45499210414322	1.4
1.44716725175157	1.447
1.44725862822699	1.4472586
1.44725861727792	1.4472586172779
1.44725861727790	alle

Die Anzahl richtiger Stellen, sagt die Theorie, ist die Summe der richtigen Stellen der beiden vorherigen Näherungen. Das würde typischer Weise 0, 1, 1, 2, 3, 5, 8, 13, ... genaue Stellen bedeuten (kommt ihnen diese Folge bekannt vor?) Dem entsprechend sollte sich pro Schritt die Anzahl der richtigen Stellen um etwa 60% erhöhen.

Tatsächlich verdoppelt sich die Anzahl bei den ersten Schritten (1-2-4-8), und erhöht sich schliesslich immer noch um 75% (von 8 auf 14 Stellen). Die Sekantenmethode konvergiert hier schneller, als sie es den Regeln der Theorie entsprechend müsste.

Ü 2.5.4 Nullstellen mit fzero

MATLAB hat *numerische* Verfahren zur Nullstellensuche eingebaut, eine Kombination aus Intervallhalbierung, Sekantenmethode und inverser quadratischer Interpolation. Die Funktion `fzero` ruft diese Verfahren auf.

Für unser Beispiel $f(x) = 3\cos(x) - \log(x)$ lautet der Aufruf:

```
>> fzero(@meinf,1)
ans =
   1.44725861727790
>>
```

fzero steht für „find zero“

Wichtig: Dem Funktionsnamen (hier: `meinf`) müssen Sie den „Funktionshenkel“ `@` voranstellen (MATLAB nennt `@` *function handle*). Funktionsnamen mit Henkel `@` davor sind ein eigener Datentyp, damit lassen sich Funktionen an andere Funktionen übergeben. Stellen Sie sich vor, `@` sei der Henkel, mit dem Sie ein Kaffeehäferl weitergeben.

Aber `fzero` hat seine Tücken: es findet einen Punkt so nah wie möglich bei einem Vorzeichenwechsel der Funktion. Für stetige Funktionen (Zwischenwertsatz!) ist das zugleich ein Wert nahe einer Nullstelle. Für unstetige Funktionen kann `fzero` Werte liefern, die zu Singularitäten der Funktion gehören. Beispiel: der Tangens bei $\pi/2$

```
>> fzero(@tan,1)
ans =
   1.57079632679490
>>
```

Das ist keine Nullstelle der Tangensfunktion

Mehrfache Nullstellen (gerader Ordnung), bei denen die Funktion die x-Achse berührt, aber nicht schneidet, kann `fzero` auch nicht finden.

Funktions-Henkel können noch mehr. Damit lässt sich ohne Skript, als Einzeiler, eine Funktion definieren. Das erklärt der nächste Abschnitt.

Ü 2.5.5 Funktions-Einzeiler, *Anonymous Functions*

Bei einem einzeiligen Funktionsterm ist es nicht nötig, eigene Funktionsdateien zu schreiben. Unsere Beispiel-Funktion f mit Funktionsterm $f(x) = 3\cos(x) - \log(x)$ und ihre Ableitung $f'(x) = -3\sin x - \frac{1}{x}$ lassen sich auch als sogenannte *anonymous functions* definieren. Das sind Funktionen, die nicht in einer Programmdatei gespeichert sind, sondern mit einer Variablen vom Typ *function handle* (erkennbar am „Funktionshenkel“ `@` mit Funktionsargument in Klammer) verknüpft sind. Für unsere Beispiele:

```
>> f = @(x) 3*cos(x) -log(x)
f =
  function_handle with value:
    @(x)3*cos(x)-log(x)

>> df = @(x) -3*sin(x)-1./x
df =
  function_handle with value:
    @(x)-3*sin(x)-1./x
```

Sie können diese Einzeiler-Funktionen auswerten

```
>> f(1)
ans =
    1.6209
```

oder direkt `fzero` anwenden. Unbedingt ausprobieren! Achtung, weil `f` bereits vom Typ „Funktionshenkel“ ist, ist beim Argument von `fzero` *kein* Henkel `@` mehr erlaubt.

```
>> fzero(f,1)
ans =
    1.4473
>> fzero(f,10)
ans =
   11.9702
```

Die Schritte eines Newton-Verfahrens im *command window* könnten dann beispielsweise so aussehen:

```
>> x=1;
>> x=x-f(x)/df(x)
x =
   1.459908341776445
>> x=x-f(x)/df(x)
x =
   1.447255837981919
>> x=x-f(x)/df(x)
x =
   1.447258617277790
>> x=x-f(x)/df(x)
x =
   1.447258617277903
```

> Funktions-Einzeiler (*anonymous functions*) sind praktisch, wenn der Funktionsterm aus einem einzigen ausführbaren Befehl besteht.
> Die Bezeichnung eines Funktions-Einzeiler ist eine Variable vom Typ „Funktions-Henkel“, sie bezieht sich *nicht* auf eine Funktionsdatei

Ü 2.5.6 Nullstellen mit roots

Lösungen (man sagt auch Wurzeln) *polynomialer* Gleichungen sind für MATLAB leichter zu finden. Betrachten wir als Beispiel

$$p(x) = x^3 - 2x - 5$$

Ein Polynom ist durch die Angabe seiner Koeffizienten bestimmt. In diesem Beispiel lauten sie

$$1; 0; -2; -5$$

weil $p(x) = 1 \cdot x^3 + 0 \cdot x^2 - 2 \cdot x - 5 \cdot x^0$. Sie übergeben `roots` die Liste der Koeffizienten als Vektor `[1 0 -2 -5]`, und roots liefert *alle* (auch die komplexen) Nullstellen des Polynoms.

```
>> roots([1 0 -2 -5])
ans =
   2.09455148154233
  -1.04727574077116 + 1.13593988908893i
  -1.04727574077116 - 1.13593988908893i
>>
```

Übungsbeispiele

Aufgabe 12:

Wie groß ist das Molvolumen von Stickstoff bei 20 C und 1 bar nach der Van der Waals-Gleichung?

Die Zustandsgleichung

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT$$

beschreibt den Zusammenhang zwischen Druck p, Molvolumen V_{mol} und Temperatur T. Die Konstanten a und b haben für Stickstoff die Werte

$$a = 0{,}129\,\mathrm{Pa\,m^6/mol^2}, \quad b = 38{,}6 \times 10^{-6}\,\mathrm{m^3/mol}.$$

Die molare Gaskonstante ist $R = 8{,}3145\,\mathrm{J/molK}$. Nach Einsetzen der Zahlenwerte verbleibt als Gleichung für V_{mol}:

$$\left(100000 + \frac{0,129}{V_{mol}^2}\right)(V_{mol} - 0,0000386) = 2437,4$$

Lösen Sie diese Aufgabe mit der Sekanten- und der Newtonmethode und mittels `fzero`. Die Gleichung lässt sich auch auf polynomiale Form umformen. Verwenden Sie `roots` zur Lösung. Das Skriptum beschreibt in Kapitel 1.6 eine Umformung als Fixpunkt-Gleichung. Lösen Sie die Aufgabe auch mit Fixpunkt-Iteration!

Aufgabe 13:

Gegeben sei das Polynom

$$p(x) = -64 + 176\,x - 188\,x^2 + 101\,x^3 - 29\,x^4 + \frac{17\,x^5}{4} - \frac{x^6}{4}$$

Schreiben Sie dazu eine Funktions-M-Datei. Denken Sie dabei daran, die elementweisen Operatoren `.^` zu verwenden. Vergessen Sie nicht den Strichpunkt am Ende der Zeile (sonst liefert das M-file mächtig viel unnötige Ausgabezeilen im *Command Window*).

Zeichnen Sie das Polynom für $0 < x < 5$. Suchen Sie Nullstellen mit dem Newton-Verfahren, mit `fzero` (Sie müssen dazu geeignete Startwerte setzen) und mit `roots` .

Warum findet `fzero` nicht alle Nullstellen?

Überprüfen Sie, wie vorher in den Übungsunterlagen, die Konvergenzgeschwindigkeit des Newton-Verfahrens für alle drei Nullstellen. Verhält sich die Konvergenzgeschwindigkeit den Regeln entsprechend?

Ändern Sie im Polynom den Koeffizienten von x^3 um 1%; 0,1%; 0,01%. Bestimmen Sie Nullstellen mit `roots` und geben sie für jede ursprüngliche reelle Nullstelle an: Gibt es diese reelle Nullstelle noch? Wenn Ja, um wieviel hat sich ihr Wert relativ geändert? Oder hat sich eine mehrfache Nullstelle in mehrere verschiedene reelle Nullstellen aufgesplittet?

Diese Aufgabe illustriert: Die numerische Berechnung mehrfacher Nullstellen ist anfällig gegenüber kleinen Änderungen des Polynoms und Rundungsfehlern. Nullstellen verschwinden, verschieben oder vermehren sich. Man spricht von einem *schlecht konditionierten Problem*.

> **Schlecht konditioniertes Problem:** Kleine Änderungen in den Daten und/oder Rundungsfehler bewirken starke Änderungen im Ergebnis.

Ü 2.6 Kontrollstrukturen

Sie haben in den Aufgaben zur vorigen Übungseinheit Fixpunkt-Iterationen oder Iterationen des Newtonverfahrens oder der Sekantenmethode gleichsam „im Handbetrieb“ im *Command Window* eingegeben und, wenn die Ergebnisse sich nicht mehr geändert haben, das Verfahren beendet. Sie können diesen Ablauf auch als Programm formulieren. MATLAB bietet die üblichen Kontrollstrukturen (ähnlich wie in Java oder C++). Für den Anfang reichen `for`-Schleifen und `if`-Verzweigungen, wie sie dieser Abschnitt vorstellt. Mehr dazu, auch über `while`-Schleifen, erzählt Ihnen die MATLAB-Hilfe.

Schleifen

Eine for-Schleife hat zumeist die Form

```
for index = startwert:endwert
    anweisung
    anweisung
    ...
end
```

Der Index durchläuft dann die Werte `startwert, startwert+1, startwert+2...endwert`; er muss nicht (wie in Java oder C++) durch `index++` erhöht werden. Allgemeinere Form des Schleifenkopfes:

```
for index = startwert:schrittweite:endwert
```

Beispiel: ein Schleifenzähler s, der mit Schrittweite $-0,1$ die Werte $1; 0,9; 0,8; \ldots 0$ durchläuft:

```
for s = 1:-0.1:0
```

Verzweigungen

Eine bedingte Verzweigung mit `if` hat die Form

```
if ausdruck
    anweisung
    anweisung
    ...
end
```

MATLAB wertet „ausdruck“ aus, und wenn das Resultat logisch `true` oder (Unterschied zu Java!) ungleich 0 ist, führt es die folgenden Anweisungen bis zum `end` aus. Geschachtelte `if` sind möglich, jede Ebene muss mit dem entsprechenden `end` abgeschlossen werden.

Während Java sehr streng darauf achtet, dass in einer `if`-Anweisung nur ein logischer Ausdruck stehen darf, erlaubt MATLAB sogar Vektoren oder Matrizen. Wenn „ausdruck“ kein Skalar ist, muss jede einzelne Komponente `true` oder $\neq 0$ sein.

Die allgemeinere Form mit `elseif` und/oder `else` hat die Form

```
if ausdruck1
    anweisungen1
elseif ausdruck2
    anweisungen2
else
    anweisungen3
end
```

Musterprogramm

Das folgende Musterprogramm zur Fixpunkt-Iteration stellt Schleifen und Verzweigungen vor. Es führt eine Fixpunktiteration gemäß der Vorschrift

$$x^{(k+1)} = \phi(x^{(k)})$$

für den Startwert $x^{(0)}$ durch. Das Programm können Sie von der Übungs-Homepage herunterladen:

```
function x = fixpunkt(phi,x0)
%FIXPUNKT: Demo-Programm zur Fixpunkt-Iteration
% Das Verfahren iteriert gemaess der Vorschrift
%       x_i+1 = phi(x_i)
% bis Aenderungen unter eine Toleranzschwelle sinken
% oder maximale Iterationszahl ueberschritten wird.
%
% Eingabe   phi... Funktion, deren Fixpunkt gesucht ist
%                  (mit Funktions-Henkel @)
%           x0 ... Startwert oder -vektor
% Ausgabe   x .... Bei Konvergenz: Fixpunkt,
%                  sonst: NaN
% Beispiel  fixpunkt(@cos,1)
%
%-----------------------------------------NMI,SS09-18 C.Brand
itmax = 100;                % maximale Iterationszahl
errlim = 1.e-9;             % Fehlerschranke
for i=1:itmax
    x = phi(x0);            % Funktionsauswertung
    if norm(x-x0)<errlim    % Abbruchkriterium: 2-Norm des Fehlers
        return
    end
    x0 = x;
end
x = NaN;
end
```

Ü 2.7 Fixpunkt-Iteration ein- und mehrdimensional

Eindimensional

In Aufgabe 5 haben Sie die Qudratwurzel aus a als Fixpunkt der Funktion

$$y = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

berechnet. Wir wiederholen, verwenden Schleifen und das Fixpunkt-Musterprogramm und verallgemeinern auf den mehrdimensionalen Fall.

```
>> heron = @(x)(x+13/x)/2;
```

Definieren Sie die Iterations-Funktion als *anonymous function*. Hier soll $\sqrt{13}$ berechnet werden. Alternativ können Sie auch eine Funktions-Datei dafür schreiben.

```
>> heron(2)
ans =
    4.2500
```

Testen Sie unbedingt, ob sich die Funktion aufrufen lässt und ein passendes Ergebnis liefert.

```
>> x=1;
>> x=heron(x)
x =
     7
```

So sieht eine Fixpunkt-Iteration im „Handbetrieb“ aus: wiederholter Aufruf der Funktion in der Befehlszeile.

```
>> x=heron(x)
x =
    4.4286
>> x=heron(x)
x =
    3.6820
>> x=heron(x)
x =
    3.6063
```

```
>> fixpunkt(heron,1)
ans =
    3.6056
```

Und so berechnet das Fixpunkt-Musterprogramm dasselbe Ergebnis.

Mehrdimensionale Fixpunkt-Iteration, vektorwertige Funktionen

Dieses Programm kann aber ebenso mehrdimensionale Fixpunkt-Iteration durchführen. In der Vorlesung wurde der Fixpunkt einer vektorwertigen Funktion $\mathbf{\Phi} : \mathbb{R}^2 \to \mathbb{R}^2$ als Beispiel behandelt:

$$\begin{aligned} x_1 &= \frac{1}{4}(x_2 - x_1 x_2 + 1) \\ x_2 &= \frac{1}{6}(x_1 - \log(x_1 x_2) + 2) \end{aligned}$$

Die Funktion $\mathbf{\Phi}(\mathbf{x})$ kann als Funktions-Datei so implementiert werden:

```
function y = phi(x)
x1=x(1);
x2=x(2);
y = [ (x2 - x1.*x2 + 1)/4
      (x1-log(x1.*x2)+2)/6 ];
end
```

(Bemerkung: Eine einzeilige *anonymous function* wäre schon auch möglich, aber bei einem etwas umfangreicheren Funktionsterm ist die Funktions-Datei die übersichtlichere Implementierung!)

Aufruf der Fixpunkt-Funktion für die Funktionsdatei (Henkel `@`+Dateiname!)

```
>> fixpunkt(@phi,[1;1])

ans =

    0.3534
    0.6400
```

Ü 2.8 Weitere Aufgaben bis zur nächsten Übungseinheit

Aufgabe 14:

Formulieren Sie die Iterationsvorschrift des Newton-Verfahrens zur Lösung von

$$\sin x = x/2 \ .$$

Programmieren Sie die entsprechende Iterations-Funktion und rufen Sie das Fixpunkt-Musterprogramm auf. Finden Sie damit alle positiven Lösungen der obigen Gleichung.

Aufgabe 15:

Suchen Sie die Nullstellen der Funktion

$$f(x) = e^x - 3x^2$$

mit Hilfe des Newton-Verfahrens und einfacher Fixpunkt-Iteration (mehrere Umformungen möglich).

Vergleichen Sie (bei denselben Startwerten) die Anzahl der Iterationen. Schätzen Sie bei den Verfahren mit linearer Konvergenz den Reduktionsfaktor C. Findet Ihre Fixpunkt-Formulierung alle Nullstellen?

Aufgabe 16:

Das Vorlesungsskriptum diskutiert die Gleichung

$$r = K\frac{q-1}{1-q^{-n}} \quad \text{für } r = 900, K = 100\,000, n = 180.$$

Schreiben Sie ein MATLAB-Programm, das allgemein für Eingabewerte r, K und n die Lösung q findet. (Annahme: $n \cdot r > K$ und daher $q > 1$) Welches Verfahren Sie verwenden, bleibt Ihnen überlassen.

Hinweise dazu:

- Der Aufzinsungsfaktor q liegt realistischer Weise im Bereich $1 < q < 1.05$, entsprechend einer monatlichen Verzinsung über 0% und unter 5%. Für $q = 1$ ist die Gleichung nicht definiert (Nenner wird 0), ein Startwert $q = 1$ ist nicht sinnvoll.
- Die Fixpunkt-Form (Auflösen nach q im Zähler) konvergiert langsam, ist aber rasch implementiert und vergleichsweise unempfindlich bezüglich der Wahl des Startwertes (sofern $q > 1$).

- Newton-Verfahren und MATLABs `fzero` brauchen gute Startwerte.

Aufgabe 17:

Formulieren Sie analog zum Fixpunkt-Musterprogramm eine Funktions-M-Datei zur Intervallhalbierung. Der Aufruf

```
IntervHalb(@igendeineFunktion, x0, x1)
```

soll, ausgehend von den Startwerten $x^{(0)}$ und $x^{(1)}$ eine Nullstelle der Funktion finden. Testen Sie das Verfahren, indem Sie Nullstellen folgender Funktion suchen:

$$f(x) = x^2 - 3\tan x + 1$$

Anfangsintervalle mit Vorzeichenwechsel sind: $[0, 1]$, $[1, 2]$, $[4.5, 4.7]$. Findet Intervallhalbierung für alle drei Intervalle eine Nullstelle? Was findet `fzero`, wenn Sie obige Intervallgrenzen als Startwerte geben?

Aufgabe 18:

Die Vorlesungsfolien zur 1. Vorlesung beschreiben die Illinois-Variante der Regula Falsi. Dieses Verfahren zeigt im Regelfall die guten Konvergenzeigenschaften der Sekantenmethode und bietet gleichzeitig den sicheren Einschluss der Nullstelle im aktuellen Intervall. Wenn Sie dieses Verfahren implementieren, haben Sie ein sehr brauchbares Allzweck-Nullstellen-Programm.

Orientieren Sie sich am Fixpunkt-Musterprogramm und schreiben Sie eine Funktions-M-Datei zur Regula Falsi in der Illinois-Variante. Der Aufruf

```
Illinois(@igendeineFunktion, x0, x1)
```

soll, ausgehend von den Startwerten $x^{(0)}$ und $x^{(1)}$ eine Nullstelle der Funktion finden. Testen Sie das Verfahren, indem Sie alle positiven Lösungen der folgenden Gleichung suchen:

$$\sin x = x/2$$

Rechnen Sie zum Vergleich auch mit `fzero` die Lösung.

Aufgabe 19:

Gegeben sei das Gleichungssystem

$$\begin{array}{rcrcrcr} 8x_1 & + & x_2 & - & x_3 & = & 8 \;, \\ 2x_1 & + & x_2 & + & 9x_3 & = & 12 \;, \\ x_1 & - & 7x_2 & + & 2x_3 & = & -4 \;. \end{array}$$

Formulieren Sie ein Fixpunktverfahren. Überlegen Sie, aus welcher Gleichung sie x_1 ausdrücken sollen, aus welcher x_2 und x_3. Hinweis: Unbekannte möglichst aus jener Gleichung ausdrücken, in der sie den stärksten Einfluss (den größten Koeffizienten) haben. Testen Sie das Verfahren.

Aufgabe 20:

(Wenn Sie die Unterlagen durchgearbeitet haben, ist diese Aufgabe eigentlich schon gelöst)

Gegeben sei ein System von zwei Gleichungen in zwei Unbekannten:

$$\begin{aligned} f(x, y) = 4x - y + xy - 1 &= 0 \\ g(x, y) = -x + 6y + \log xy - 2 &= 0 \end{aligned}$$

Formulieren Sie dafür ein Fixpunktverfahren und testen Sie!

Ü 3 Dritte Übungseinheit

Inhalt der dritten Übungseinheit:

- Script- und Funktionsdateien, Live Scripts
- Rechenoperationen bei Eingabe von Matrizen und Vektoren
- Bergabstrich löst Gleichungssysteme
- Inverses Problem, Regularisierung
- Newton-Verfahren für Systeme
- Lokale Funktionen
- `fsolve`
- Isolinien- und Oberflächengrafiken

Ü 3.1 Skript- und Funktionsdateien: Zusammenfassung

Wir haben schon mit beiden gearbeitet: Siehe Abschnitte Ü 1.3 und Ü 2.5 . Wichtige Unterschiede bestehen in der Verwendung und dem Gültigkeitsbereich von Variablen. Hier eine kurze Wiederholung und Zusammenfassung.

- Skripts. Sie haben weder Eingabe-Argumente noch Rückgabewerte. Sie greifen auf Daten und Variable des aktuellen Workspace zu. Befehlszeilen in einer Skript-Datei wirken so, als ob sie direkt im *Command Window* eingegeben würden.

  Ein häufiger Fehler ist, dass das Skript Variable aus dem aktuellen Workspace verwendet, aber nicht selber definiert. Sie merken während der aktuellen MATLAB-Sitzung nichts. Erst beim nächsten Neustart meldet MATLAB: `Undefined function or variable`. Geben Sie deswegen ganz zu Beginn des Skripts den Befehl `clear variables`. Damit löschen Sie alle Variablen im Workspace und merken sofort, ob sie Definitionen im Skript vergessen haben. Siehe Abschnitt Ü 1.3.

- Funktions-Dateien. Sie übernehmen Eingabewerte (Argumente)und liefern Resultate. Innerhalb der Funktion deklarierte Variable gelten nur lokal, also innerhalb der Funktion.

- Lokale Funktionen: In einer Funktionsdatei können weitere Funktionen („Unter-Funktionen“) vorkommen. Die erste Funktion (die „Hauptfunktion“) lässt sich von der Befehlszeile im *Command Window* aus oder über Befehlszeilen in einem Skript starten. Alle weiteren Funktionen (Reihenfolge egal) sind nur lokal – also nur für andere Funktionen aus derselben Datei – verfügbar. Es kann ganz praktisch sein, ein Programm mit Unterfunktionen zu gliedern. Weitere Infos: MATLAB-Hilfe, Stichwort *“local functions”*

- Auch Skript-Dateien können (ab MATLAB R2016b) lokale Funktionen enthalten; die dürfen aber erst nach der letzten Zeile des Skript-Codes beginnen. Hilfe dazu suchen Sie unter *“Add Functions to Scripts”*

- Verschachtelte Funktionen (für Fortgeschrittene): das sind Funktionen, die innerhalb einer übergeordneten Funktion deklariert sind. Damit können Sie Zugriff auf Variable noch differenzierter steuern. Details finden Sie in der MATLAB-Hilfe, Stichwort *“nested functions”*.

- Funktions-Einzeiler (*anonymous functions*) sind praktisch, wenn der Funktionsterm aus einem einzigen ausführbaren Befehl besteht. Wurden in Abschnitt Ü 2.5.5 vorgestellt.

- Live-Scripts und -Funktionen (Für Fortgeschrittene): Diese Art von Programm-Dateien kann Quellcode und Output gemeinsam darstellen, Begleittext und Kommentare übersichtlich formatieren, Sie können Schalt- und Steuerflächen für interaktives Arbeiten einbauen und mehr. Solche Dateien haben Endung `.mlx` im Unterschied zu Endung `.m` bei „gewöhnlichen“ Skripts.

  Wenn Sie Ihre Dateien besonders ansprechend gestalten wollen, können Sie ja versuchsweise im Editor auf *Save/Save as/Save as type: Matlab Live Code files (*.mlx)* klicken und so Ihre Datei in ein Live-Script umwandeln. Sie finden mehr Informationen und Beispiele in der MATLAB-Hilfe, Stichwort *Live Scripts and Functions*. Eine nette Ausarbeitung[16] von Aufgabe 8 finden Sie hier (klick!)

Ü 3.2 Rechenausdrücke bei der Eingabe von Vektoren und Matrizen

Eine Erinnerung an die letzte Einheit: Bei der Eingabe von Vektoren und Matrizen wirken Komma, Strichpunkt, Leerzeichen und Zeilenumbruch als Trennzeichen.

Sie können nicht nur Zahlenwerte, sondern auch Rechenausdrücke angeben:

```
>> x=[1+2, 2+3 3+4]
x =
     3     5     7
```

Zeilenvektoren: Leerzeichen oder Komma trennt Komponenten. Hier (nicht zu empfehlen) einmal Komma, einmal Leerzeichen.

```
>> y=[1-2 3 + 4 5 +6 7+ 8]
y =
    -1     7     5     6    15
```

Vorsicht mit Leerzeichen bei + und −: kann als Vorzeichen, als Rechenoperation oder als Trennzeichen interpretiert werden.

> Typischer Fehler bei Rechenausdrücken in einem Vektor: Leerzeichen vor Operator wirkt ungewollt als Trennzeichen. Empfehlung: Bei Rechenoperationen entweder keine Leerzeichen oder beidseitig Leerzeichen um Operatoren.

Regeln bei Plus oder Minus in Termen:

- beidseitig kein Abstand: Rechenoperation wird ausgeführt (`+` und `-` wirken als *binäre Operatoren*)
- beidseitig Leerzeichen: ebenfalls als binäre Operation interpretiert.
- links Leerzeichen, rechts nicht: Leerzeichen links wirkt als Trennzeichen zwischen Matrix-Elementen, Operatoren `+` und `-` wirken als *unäre Operatoren* (als *Vorzeichen* des folgenden Terms).

  Passiert oft unabsichtlich, erzeugt Fehlermeldung oder falsches Ergebnis.
- rechts Leerzeichen, links nicht: binäre Operation (aber wenn Sie das absichtlich machen, ist das ziemlich verhaltensoriginell).

[16] von Chr. Tuschl, danke schön!

Ü 3.3 Gleichungssysteme: MATLABs schräge Schreibung

MATLAB verwendet – völlig normal – den Schrägstrich als Divisionsoperator: `x = 3/4` liefert wenig überraschend `x = 0.75000`.

Eher unüblich ist der andersrum gekippte „Bergabstrich“ `\` oder Backslash:

```
>> x=3\4
x =
    1.3333
```

Auch `\` dividiert, aber in der Form `x = Nenner\Zähler`. Es liegt ja auch wirklich, wenn Sie sich `\` als abwärts geneigten Bruchstrich vorstellen (oder den Bildschirm kurz mal nach links kippen), der linke Term *unter* und der rechte *über* dem Bruchstrich.

Grenzwertig originell ist jedoch MATLABs Interpretation des Bergabstrichs bei Gleichungssystemen. Das Wichtigste in Kürze:

`x = A\b` löst (oder „löst“) das Gleichungssystem $A\mathbf{x} = \mathbf{b}$

Nicht jedes von MATLAB so berechnete Ergebnis ist die Lösung eines Gleichungssystems im eigentlichen Sinn – deswegen die Anführungszeichen bei „löst“.

Niemand will sich im Detail merken, was genau MATLAB bei den verschiedenen Sonderfällen tut. Nur damit Sie sehen und gewarnt sind, was alles passieren kann, hier eine Aufzählung.

Der Befehl `x = A\b` liefert für ein Gleichungssystem $A\mathbf{x} = \mathbf{b}$

- bei nicht singulärer $n \times n$- Matrix die eindeutige Lösung;
- bei singulärer $n \times n$- Matrix eine Warnmeldung und, falls es Lösungen gibt, eine Lösung mit möglichst vielen Null-Komponenten. Falls es keine Lösung gibt, liefert MATLAB meistens unsinnige Zahlenwerte.
- bei einer $n \times m$- Matrix mit $n < m$ (*unterbestimmtes* Gleichungssystem), falls es Lösungen gibt, eine Lösung mit möglichst vielen Null-Komponenten.
- bei einer $n \times m$- Matrix mit $n > m$ (*überbestimmtes* Gleichungssystem), die „am wenigsten falsche Lösung“. Das ist jener Vektor $\mathbf{x}$, für den $A\mathbf{x} - \mathbf{b}$ die kleinste 2-Norm hat. Man spricht von der *Kleinste-Quadrate-Lösung*.
- Sonderfälle sind $n \times m$- Matrizen mit $n \neq m$ und Rang $< \min(m, n)$. Matlab warnt: „Warning: Rank deficient“ und liefert eine Kleinste-Quadrate-Lösung.

Gleichungssysteme mit mehreren rechten Seiten

Bei gleicher Matrix A und mehreren rechten Seiten $\mathbf{b}, \mathbf{c}, \mathbf{d}, \ldots$ lassen sich die Gleichungssysteme $A\mathbf{x} = \mathbf{b}, A\mathbf{y} = \mathbf{c}, A\mathbf{z} = \mathbf{d}, \ldots$ gemeinsam lösen. Stellen Sie alle rechten Seiten als Spaltenvektoren in einer Matrix B zusammen: `B = [b,c,d]`. MATLABs `\` liefert eine Matrix X, deren Spalten die Lösungsvektoren enthält: $X = [\mathbf{x}, \mathbf{y}, \mathbf{z}]$.

`X = A\B` löst (oder „löst“) die Gleichungssysteme $A \cdot X = B$

Schrägstriche zwischen Matrizen, allgemeiner Fall

Die Kurzfassung:

> Verwenden Sie den „Bergaufstrich“ `/` zwischen Skalaren als Divisionsoperator und den „Bergabstrich“ `\` zwischen Matrizen und Vektoren zum Lösen linearer Gleichungssysteme.

Wenn Sie mehr über MATLABs Umgang mit `/` und `\` wissen wollen, lesen Sie weiter...

Sie könnten $x = 3/4$ auch in der Form $x = 3 \cdot 4^{-1}$ schreiben, oder $x = 4^{-1} \cdot 3$. Niemand tut das bei skalaren Termen. Bei Matrizen ist die Schreibweise mit Inversen jedoch Standard, zum Beispiel $A \cdot B^{-1}$ oder $A^{-1} \cdot B$. MATLAB erlaubt dafür die Bruchstrich-Schreibung:

$$A \cdot B^{-1} = A/B \quad \text{und } A^{-1} \cdot B = A\backslash B$$

Stellen Sie sich `/B` als B^{-1} vor, weil B „unter“ dem Bruchstrich liegt, und entsprechend `A\` als A^{-1}. Die Schrägstriche stehen absichtlich *links* von B und *rechts* von A – Matrixmultiplikation ist nicht kommutativ! Je nachdem, von welcher Seite Sie mit einer Inversen multiplizieren wollen, verwenden Sie `/` oder `\`.

MATLAB berechnet nicht wirklich die inversen Matrizen und multipliziert damit. Das explizite Ausrechnen einer Inversen ist rechenaufwändig und mit Rundungsfehlern behaftet. In Wirklichkeit löst MATLAB mit der schrägen Bruchstrichschreibweise lineare Gleichungssysteme. Das ist nur bei nichtsingulären quadratischen Matrizen algebraisch äquivalent zur Multiplikation mit der Inversen.

Gleichungssysteme statt inverser Matrix

Wenn in mathematischen Ausdrücken eine Multiplikation mit der inversen Matrix auftritt, brauchen Sie für die rechnerische Auswertung die Inverse in expliziter Form nicht wirklich[17].

Für das numerische Rechnen besteht ein gewaltiger Unterschied im Hinblick auf Rechenaufwand und -genauigkeit, ob Sie eine Inverse berechnen und damit multiplizieren, oder so umformen, dass Sie Gleichungssysteme lösen.

> Vermeiden Sie explizite Multiplikation mit einer inversen Matrix! Es verursacht unnötigen Rechenaufwand und unerwünschte Rundungsfehler.
> `X = A\B` berechnet $A^{-1} \cdot B$ als Lösung des Gleichungssystems $A \cdot X = B$
> `X = A/B` berechnet $A \cdot B^{-1}$ als Lösung des Gleichungssystems $X \cdot B = A$

Zur Erklärung, warum sich Multiplikation mit Inverser auf Lösung eines Gleichungssystems zurückführen lässt, hier die entsprechenden Umformungen. Achtung, man muss „auf der richtigen Seite“ multiplizieren, damit die Inversen verschwinden: im ersten Fall multipliziert man A von links, das andere mal mit B von rechts. (Matrixmultiplikation ist nicht kommutativ!)

$$\begin{aligned} X &= A^{-1} \cdot B && \mid A\cdot \\ A \cdot X &= A \cdot A^{-1} \cdot B \\ A \cdot X &= B \end{aligned} \qquad\qquad \begin{aligned} X &= A \cdot B^{-1} && \mid \cdot B \\ X \cdot B &= A \cdot B^{-1} \cdot B \\ X \cdot B &= A \end{aligned}$$

[17] Damit Sie erst gar nicht in Versuchung kommen, eine Inverse zu berechnen, verschweigen diese Übungsunterlagen (vorerst einmal) absichtlich den MATLAB-Befehl zur Berechnung der Inversen.

Ü 3.4 Inverse Probleme, Regularisierung

Was eine Matrix tut,
macht die Inverse wieder gut.

Abschnitt Ü 2.4 hat erklärt: Die Matrix-Vektor-Multiplikation $\mathbf{y} = A \cdot \mathbf{x}$ definiert eine Transformation *Eingabedaten* $\rightarrow$ *Bilddaten*. Oft liegt das Problem in umgekehrter Form vor: Gegeben ist der Ergebnisvektor $\mathbf{y}$, welcher Vektor $\mathbf{x}$ führt zu diesem Ergebnis?

Wenn die inverse Matrix existiert, ist die theoretische Antwort einfach:

$$\mathbf{y} = A \cdot \mathbf{x} \qquad \Leftrightarrow \qquad \mathbf{x} = A^{-1} \cdot \mathbf{y} \quad ,$$

Es kann aber sein, dass mehrere $\mathbf{x}$-Vektoren denselben Ergebnisvektor $\mathbf{y}$ haben, oder es überhaupt keinen $\mathbf{x}$-Vektor gibt, der exakt zum Ergebnisvektor $\mathbf{y}$ führt. Dann wird es schwierig. Aber selbst wenn eine Inverse theoretisch existiert, kann es sein, dass die Rücktransformation damit praktisch unmöglich ist.

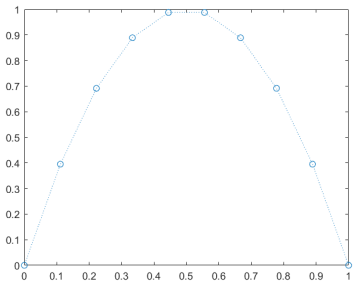
Inverses Problem: Ergebnis bekannt, Ursache gesucht. Oft schwierig.

Hier stellen wir Ihnen eine solche Situation vor, und zeigen, wie man damit umgehen kann.

Aufgabe 21: Schlecht konditioniertes inverses Problem

Erzeugen Sie eine Zeitreihe $\mathbf{x}(t)$ mit n Datenpunkten; hier mit parabelförmigen Verlauf; zeichnen Sie die Daten.

```
n=10;
t = linspace(0,1,n)';  % t Spaltenvektor!
x = 4*t.*(1-t);
plot(t,x,'o:')
```



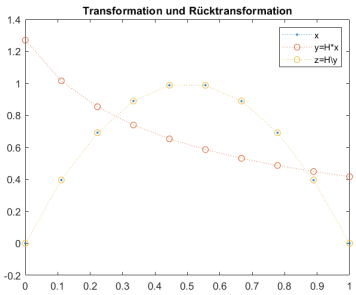
Wenden Sie auf $\mathbf{x}$ eine Transformation, die sogenannte Hilbert-Matrix, an: $\mathbf{y} = H \cdot \mathbf{x}$. Die Matrix ist einfach aufgebaut, ihre Inverse (sie enthält nur ganzzahlige Elemente) lässt sich explizit angeben. In MATLAB erzeuht sie der Befehl `hilb(n)` . Zum Beispiel für $n = 4$

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix} \qquad H^{-1} = \begin{bmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{bmatrix}$$

Auf den Ergebnisvektor $\mathbf{y}$ wenden Sie gleich wieder die Rücktransformation an. Das heißt, Sie lösen das Gleichungssystem $H \cdot \mathbf{z} = \mathbf{y}$. Theoretisch[18] muss dann matürlich $\mathbf{z}$ ident mit dem Ausgangsvektor $\mathbf{x}$ sein. Zeichnen Sie daher $\mathbf{x}, \mathbf{y}$ und $\mathbf{z}$. In MATLAB führen Sie das so aus:

[18] Alte Lebensweisheit: *Theoretisch ist kein Unterschied zwischen Theorie und Praxis. Praktisch schon.*

```
H = hilb(n);
y = H*x;
z = H\y;
plot(t,x,'.:',t,y,'o:',t,z,'o:')
legend('x','y=H*x','z=H\y')
title('Transformation und Rücktransformation')
```



In der Abbildung hier, für $n = 10$ Datenpunkte, liegen die **x**- und **z**-Datenpunkte perfekt übereinander.

Wiederholen Sie die Aufgabe für größere n. Ab welchem n liefert die Rücktransformation deutlich unterschiedliche Werte, ab wann katastrophal unbrauchbare?

Die Hilbert-Matrix ist nur ein Beispiel für Transformationen, die auf extrem schlecht konditionierte inverse Probleme führen.

Sie sollen nun trotzdem für $n = 50$ die Rücktransformation durchführen. In solchen Fällen wenden Sie Regularisierungsmethoden an. Hier das Kochrezept für Tichonov-Regularisierung.

Statt

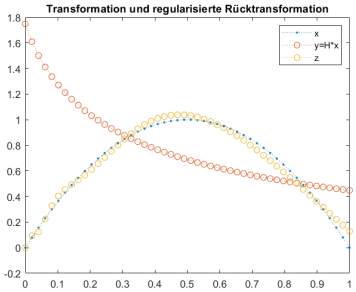
```
z = H\y;
```

verwenden Sie für die Rücktransformation

```
alpha= ...
z = (H'*H + alpha*eye(n))\H'*y;
```

für ein *sehr* kleines $\alpha > 0$.

Für welchen Wert α können Sie $n = 50$ Datenpunkte transformieren und rücktransformieren, so dass in der Zeichnung **x** und **z** möglichst gut übereinstimmen? Hier rechts ist die Übereinstimmung noch nicht besonders gut. Bei günstig gewähltem α sollte sie deutlich besser sein.



Regularisierung für inverse Probleme tritt in vielen Anwendungsgebieten auf. Das ist ein hochaktuelles Thema, obwohl es dazu noch nicht einmal deutsche Wikipedia-Einträge gibt[19].

Die Übungsunterlagen gehen auf die Theorie nicht ein[20], sie wollen hier nur zeigen: selbst theoretisch anspruchsvollen Methoden lassen sich mit einfachen MATLAB-Befehlen ausführen.

[19] siehe die Stichworte *Regularization (mathematics)* und *Tikhonov regularization* in der englischen Wikipedia

[20] Erklärung der Bedeutung von α (nur für Interessierte): Der Ergebnisvektor **y** ist in der Praxis nicht völlig exakt bestimmbar. Wenn man relative Fehler der Größenordnung α im Ergebnis **y** zulässt, dann gibt es unendlich viele Ausgangsvektoren **x**, die im Rahmen der Genauigkeit den Ergebnisvektor **y** hätten erzeugen können. Der obige Kochrezept-Befehl findet unter all diesen Vektoren jenen mit kleinster 2-Norm.

Das ist zwar nicht notwendig der „richtige“ Ausgangsvektor, aber unter bestimmten Zusatzannahmen der „plausibelste“.

Ü 3.5 Newton-Verfahren für Systeme: Anleitung

Lösen Sie das folgende nichtlineare Gleichungssystem mit dem Newton-Verfahren.

$$\begin{aligned} 2x_1 - x_2 &= e^{-x_1} \\ x_1 + 2\sin x_2 &= \cos x_2 \end{aligned}$$

Es folgt eine Schritt-für-Schritt-Anleitung. Am Ende dieses Abschnittes gibt es auch ein Link zu einer fertigen Lösung. Dringender Rat: arbeiten Sie zuerst diese Anleitung durch (insbesondere, wenn Sie der Meinung sind, dass fertige Musterlösungen Zeit und Mühe sparen. . .)

Gleichung auf Nullstellen-Problem umformen

Bringen Sie zuerst die Gleichungen in die Form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

$$\begin{aligned} 2x_1 - x_2 - e^{-x_1} &= 0 \\ x_1 + 2\sin x_2 - \cos x_2 &= 0 \end{aligned}$$

Jacobi-Matrix berechnen

Berechnen Sie die Jacobi-Matrix: $D_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} 2 + e^{-x_1} & -1 \\ 1 & 2\cos x_2 + \sin x_2 \end{bmatrix}$

Startvektor wählen

Beginnen Sie mit dem Startvektor $\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Werten Sie $\mathbf{f}$ und $D_{\mathbf{f}}$ für $\mathbf{x}^{(0)}$ aus – besser noch: lassen Sie MATLAB das tun (wird gleich erklärt).

Korrektur-Vektor berechnen

Sie berechnen einen Korrektur-Vektor $\Delta\mathbf{x}^{(0)}$ als Lösung eines linearen Gleichungssystems. Dessen Matrix ist die Jacobi-Matrix, auf der rechten Seite steht $-\mathbf{f}(\mathbf{x}^{(0)})$.

Zur Theorie lesen Sie bitte im Skriptum nach. Dort finden Sie:

> Iterationsvorschrift
>
> $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$
>
> mit $\Delta\mathbf{x}^{(k)}$ als Lösung von $D_f(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$

MATLAB löst lineare Gleichungssysteme mit dem „Bergab-Strich“-Operator `\`. Ein System $A\mathbf{x} = \mathbf{b}$ würden Sie in der Form `x = A\b` lösen. Für das Newton-Verfahren können Sie Gleichungslösen `Deltax = Df\(-f)` und Korrekturschritt `xneu = x + Deltax` als Einzeiler zusammenfassen: `xneu = x - Df\f`

Programmierung

Programmieren Sie $\mathbf{f}(\mathbf{x})$ und $D_{\mathbf{f}}(\mathbf{x})$ als MATLAB-Funktionen und speichern Sie die beiden als Dateien `f.m` und `Df.m`.

Abschnitte Ü 2.1 und Ü 2.2 haben die Eingabe von Vektoren und Matrizen schon kurz erklärt. Die Matrixelemente in einer Zeile können Sie durch Beistriche oder Leerzeichen trennen. Strichpunkte oder der Beginn einer neuen Zeile im Quelltext trennen die Zeilen in Matrizen und Spaltenvektoren.

Vervollständigen Sie die beiden folgenden Funktionsdateien!

```
function y = f(x)
y= [ ... %erste Komponente
     ... %zweite Komponente
   ];
end
```

Vektorwertige Funktion: Vektor als Eingabe, Vektor als Ergebnis!

```
function dy = Df(x)
dy= [ ... %erste Zeile
      ... %zweite Zeile
   ];
end
```

Matrixwertige Funktion: Vektor als Eingabe, Matrix als Ergebnis!

Prüfen Sie im *Command Window*, ob sich die beiden Funktionsdateien korrekt aufrufen lassen. Das sollte herauskommen:

```
>> x=[1; 1];
>> f(x)
ans =
    0.6321
    2.1426
>> Df(x)
ans =
    2.3679   -1.0000
    1.0000    1.9221
```

Achtung, setzen Sie Leerzeichen korrekt, wenn Sie die Jacobi-Matrix programmieren! Darauf weist schon Abschnitt Ü 3.2 hin; hier nochmal:

Leerzeichen um + und − bei Termen in Matrizen:
RICHTIG

- keine Lerzeichen. Beispiel `[ a+b, x+y ]`
- beiderseits Leerzeichen. Beispiel `[ a + b, x + y ]`

FALSCH:

- einseitig Lerzeichen. Beispiel `[ a +b, x +y ]`

Handbetriebene Iteration im Command Window

Wenn Funktion und Jacobi-Matrix korrekt programmiert sind, testen Sie im *Command Window* die Newton-Iteration. Beginnen Sie die Suche mit $\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

```
>> x=[1; 1];
>> x = x - Df(x)\f(x)
x =
          0.395158347619808
          0.199928444993336
>> x = x - Df(x)\f(x)
x =
          0.449399668569469
          0.261761474532574
>> x = x - Df(x)\f(x)
x =
          0.449562385013556
          0.261217530152151
>> x = x - Df(x)\f(x)
x =
          0.449562377948136
          0.261217503068493
```

Der Startwert ist mäßig genau. Die zweite Iteration liefert aber immerhin schon drei signifikante Stellen; ab dann lässt sich quadratisches Konvergenzverhalten beobachten.

Ü 3.6 Lokale Funktionen

Die beiden Funktionsdateien zusammen mit der Newton-Iteration sind in eine Datei zum Herunterladen gepackt. Dieses Programm zeigt Ihnen die Verwendung lokaler Funktionen.

> Ob Sie Aufgaben in einer Datei mit lokalen Funktionen oder mit anonymous functions programmieren, oder ob Sie Ihre Lösung lieber in Skripts und Funktionen in getrennten Dateien aufteilen, bleibt Ihnen überlassen! Alle Varianten haben Vor- und Nachteile.

Holen Sie sich die Datei und führen Sie sie aus. Für eine sauber formatierte Version können Sie die Datei auch in ein Live-Script umwandeln oder die *publish*-Möglichkeit von MATLAB nützen. Siehe MATLAB-Hilfe, Stichwort *Publish and Share MATLAB Code*. So sollte die Ausgabe aussehen:

```
>> NewtonMusterAufgabe
Konvergenz nach 4 Iterationen
Nullstelle:
          0.449562377948136
          0.261217503068493
```

Damit sollte es leicht fallen, die folgende Aufgabe zu lösen:

Aufgabe 22:

Aufgabe 20 fragt nach einer Lösung des Gleichungssystems

$$\begin{aligned} f(x,y) = 4x - y + xy - 1 &= 0 \\ g(x,y) = -x + 6y + \log xy - 2 &= 0 \end{aligned}$$

durch Fixpunkt-Iteration. Nun sollen Sie das Newton-Verfahren anwenden. Orientieren Sie sich am Beispiel des vorigen Abschnittes und am Musterprogramm `NewtonMusterAufgabe.m`. Im Skriptum (Seite 27) ist dieses Beispiel auch durchgerechnet.

Aufgabe 23:

Lösen Sie das folgende nichtlineare Gleichungssystem mit dem Newton-Verfahren. Startvektor $[1; -1; 0]$. Iterieren Sie, bis aufeinanderfolgende Lösungen in der ∞-Norm, das ist in MATLAB `norm(...,inf)`, auf 10^{-12} übereinstimmen.

$$\begin{aligned} x^2 + \sin^2 y + z &= 1 \\ e^x + e^{-x} - yz &= 2 \\ x + y + z^2 &= 0 \end{aligned}$$

Aufgabe 24:

Gegeben sind die Funktionsgleichungen

$$\begin{aligned} y &= 2x^2 - 1 \\ y &= \sin(3x) \quad \text{für } -1 \le x \le 1 \quad . \end{aligned}$$

Zeichnen Sie beide Kurven und lesen Sie aus der Graphik die Koordinaten der Schnittpunkte auf eine Nachkommastelle genau ab.

Formen Sie beide Gleichungen auf $\cdots = 0$ um und berechnen Sie die Lösung des nichtlinearen 2×2-Systems mit dem Newton-Verfahren.

Verwenden Sie die Näherungen aus der graphischen Darstellung als Startwerte. Konvergenzkriterium: sollen sich in keiner Komponente um mehr als 10^{-9} unterscheiden.

Aufgabe 25:

In einer wässrigen Natriumacetat-Lösung bestimmt ein nichtlineares System von vier Gleichungen die Konzentration der Ionen H^+, OH^-, Ac^- und der undissoziierten Säure HAc. Gegeben sind $K_w = 10^{-14}, K_s = 10^{-4,75}$ und eine variable Konzentration C.

$H^+ \cdot OH^-$	=	K_w	Ionenprodukt des Wassers
$H^+ \cdot Ac^-$	=	$K_s \cdot HAc$	Säure-Dissoziationsgleichgewicht
$HAc + Ac^-$	=	C	Acetat-Gesamtkonzentration
$Ac^- + OH^-$	=	$C + H^+$	Elektroneutralität

Wenn Sie die unbekannten Konzentrationen H^+, OH^-, Ac^-, HAc mit Variablen x_1, x_2, x_3, x_4 bezeichnen, lässt sich die Aufgabe umformulieren zu

$$\mathbf{f}(\mathbf{x}) = 0 \quad \text{mit} \quad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} -K_w + x_1 x_2 \\ x_1 x_3 - K_s x_4 \\ -C + x_3 + x_4 \\ C + x_1 - x_2 - x_3 \end{bmatrix}$$

Schreiben Sie eine MATLAB-Funktion `f(x,C)`, die für einen Eingabevektor $\mathbf{x} = (x_1, x_2, x_3, x_4)$ und gegebene Konzentration C den Vektor $\mathbf{f}(\mathbf{x})$ berechnet. ($K_w = 10^{-14}, K_s = 10^{-4,75}$ sind fix gegeben.) Schreiben Sie auch die entsprechende Funktion für die Jacobi-Matrix und ein Newton-Verfahren zur Nullstellenberechnung. Haben Sie eine Lösung gefunden, dann ist $-\log_{10} x_1$ der pH-Wert.

Bei diesem Beispiel sind gute Startwerte wichtig. Empfehlung:

$$\mathbf{x}^{(0)} = [10^{-7}; 10^{-7}; C; 0]$$

Zum Vergleich Testwerte für $C = 0.01$

```
>> x0=[1.e-7;1.e-7;0.01;0];
>> f(x0,0.01)
ans =
  1.0e-008 *

  -0.00000000000000
   0.10000000000000
                  0
                  0

>> df(x0)
ans =

   0.00000010000000   0.00000010000000                   0                   0
   0.01000000000000                   0   0.00000010000000  -0.00001778279410
                  0                   0   1.00000000000000   1.00000000000000
   1.00000000000000  -1.00000000000000  -1.00000000000000                   0

>> x0=x0-df(x0)\f(x0,0.01)
x0 =

   0.00000000035638
   0.00000019964362
   0.00999980071276
   0.00000019928724
```

Ü 3.7 Lösen von Systemen mit *fsolve*

Gleichungssysteme in der Form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ kann MATLABs *Optimization Toolbox* mit dem Befehl `fsolve` lösen. Der Aufruf funktioniert ähnlich wie `fzero`.

Angenommen, Sie haben die Funktion aus Abschnitt Ü 3.5 als Funktionsdatei `f.m` programmiert. Dann lauten Aufruf und Ergebnis

```
>> fsolve(@f,[1,1])

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>

ans =
 0.449562378440135 0.261217525788617
```

Sie ahnen vielleicht aufgrund der umfangreichen Ausgabe: so einfach kann das nicht gewesen sein. Ist es auch nicht – die zugrunde liegenden Methoden können wir im Rahmen dieser Übung nicht behandeln.

Ab der achten Stelle unterscheiden sich die Resultate von denen des Newton-Verfahrens. Wenn Sie auf hohe Genauigkeit Wert legen, müssten Sie die Standard-Einstellungen von `fsolve` anpassen[21].

Aufgabe 25 lässt sich mit Standardeinstellungen gar nicht lösen. (`fsolve` behauptet zwar auch „Equation solved“, aber das Resultat liegt total daneben.)

[21] Challenge: wer findet Einstellungen, so dass `fsolve` auf alle 16 double-Stellen genau dieselben Werte liefert wie das Newton-Verfahren? Ich hab probiert und es nicht geschafft. C.B.

Aufgabe 26:

Lösen Sie Aufgaben 22, 23 und 24 mit `fsolve`. Stellen Sie das Anzeigeformat im *command window* auf `format long g`, und vergleichen Sie die Genauigkeit mit den Resultaten des Newton-Verfahrens.

Ü 3.8 Isolinien- und Flächen-Diagramme

Aufgabe 27: Darstellung einer Funktion $z = f(x, y)$ durch Isolinien

Stellen Sie die Funktion

$$f : \mathbb{R}^2 \to \mathbb{R}, \quad z = xe^{-x^2-y^2}$$

als Isolinien-Diagramm dar. Am einfachsten geht das mit dem Befehl `ezcontour`: (amerikanisch-englisch ausgesprochen klingt das wie *ea-sy-contour*)

```
>> ezcontour('x*exp(-x^2 -y^2)')
```

Achtung, Hochkommas ' nicht vergessen!

Wenn Sie aber genauer festlegen wollen, wie und was sie darstellen wollen, verwenden Sie den „richtigen“ Befehl `contour`. Die MATLAB-Hilfe zum Stichwort `contour` erklärt, wie das geht. Sie können nach einem dort angegebenen Musterbeispiel (je nach Matlab-Version scrollen Sie drei, vier Abbildungen nach unten) arbeiten:

Für ein Isolinien-Diagramm (engl: *contour plot*) der Funktion $z = xe^{-x^2-y^2}$ im Bereich $-2 \leq x \leq 2, -2 \leq y \leq 3$ erzeugen Sie zuerst x- und y-Werte auf einem Gitter in der xy-Ebene.

```
>> x = -2:0.2:2;
>> y = -2:0.2:3;
>> [X,Y] = meshgrid(x,y);
```

Beachte: in englischen Texten steht oft `.2` (ohne 0 vor dem Dezimalpunkt), wo bei uns 0,2 stehen würde. Auch der MATLAB-Ausdruck `-2:.2:2` ist korrekt.

Für die (x, y)-Wertepaare berechnen Sie eine Matrix Z mit dem Befehl

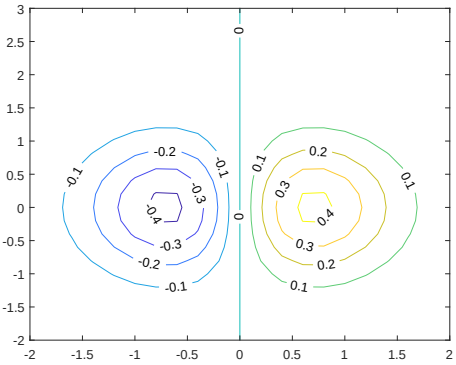
```
>> Z = X.*exp(-X.^2-Y.^2);
```

Anschließend erzeugen Sie den Isolinien-Plot:

```
>> contour(X,Y,Z,'ShowText','on')
>> colormap cool
```

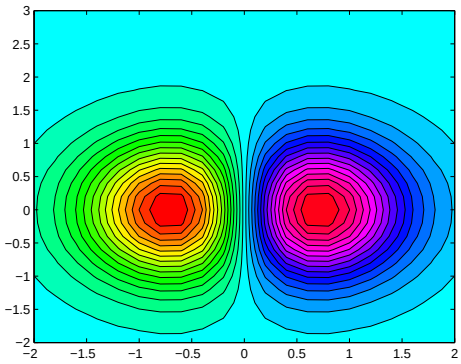
Es gibt verschiedene Varianten des contour-Befehls. In der MATLAB-Hilfe finden Sie weitere Beispiele (Befehle contourf, contour3)

So sollte das Isolinien-Diagramm aussehen, das Sie mit den obigen Befehlen erzeugt haben.



Weitere Varianten des `contour`-Befehls: `contour(X,Y,Z,30)` erzeugt 30 Isolinien; `contour(X,Y,Z,-1:0.1:1)` erzeugt Isolinien für Werte von -1 bis 1 in 0,1-Schritten.

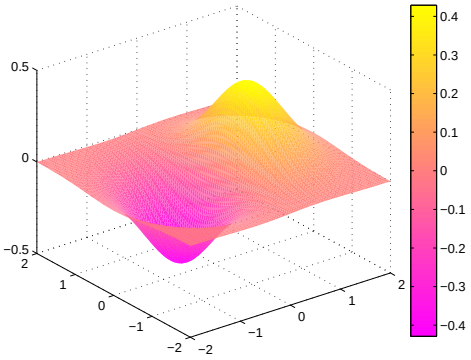
Erforschen Sie die Optionen des `contour`-Befehles. Ihre Aufgabe ist, herauszufinden, mit welchen Befehlen sie den Plot möglichst genau so aussehen lassen, wie hier gezeigt:



Aufgabe 28: Darstellung einer Funktion $z = f(x, y)$ als Fläche im Raum

Ganz ähnlich wie `contour` funktioniert der Befehl `surf`. Informieren Sie sich in der MATLAB-Hilfe und zeichnen Sie die Funktion aus Aufgabe 27 als Fläche im Raum.

Ihre Aufgabe ist, herauszufinden, mit welchen Befehlen sie den Plot möglichst genau so aussehen lassen, wie hier gezeigt (Datenbereich, Auflösung, Farbgebung, Farbbalken...).



Ü 3.9 Graphische Suche nach Nullstellen für nichtlineare Gleichungssysteme in zwei Variablen

Für eine Funktion $f : \mathbb{R} \to \mathbb{R}$ in der Form $y = f(x)$ lassen sich Nullstellen aus dem Funktionsgraph ablesen. Das kennen Sie schon aus der Mittelschule und der ersten Vorlesung.

Eine Funktion $f : \mathbb{R}^2 \to \mathbb{R}$ können Sie in der Form $z = f(x, y)$ als Fläche im Raum oder als Isoliniengrafik darstellen und so ebenfalls graphisch die Lage von Nullstellen finden. Eine Nullstelle ist in diesem Fall ein Wertepaar (x, y), das die Gleichung $f(x, y) = 0$ erfüllt.

Für ein System zweier nichtlinearer Gleichungen in der Form

$$f_1(x, y) = 0$$
$$f_2(x, y) = 0$$

kann man die Funktion

$$f(x, y) = [f_1(x, y)]^2 + [f_2(x, y)]^2$$

definieren. Die ist genau dann gleich 0, wenn f_1 und f_2 gleich Null sind. Kleiner als 0 kann f (aufgrund ihrer Definition als Summe zweier Quadrate) auch nicht werden. Die Nullstellen von f sind also zugleich auch globale Minima von f und Lösungen des nichtlinearen Gleichungssystems $f_1 = 0$, $f_2 = 0$.

Die nächsten beiden Aufgaben illustrieren diesen Zusammenhang.

Einleitung zu den Aufgaben 29 und 30

Die Funktion $f : \mathbb{R}^2 \to \mathbb{R}$, gegeben durch

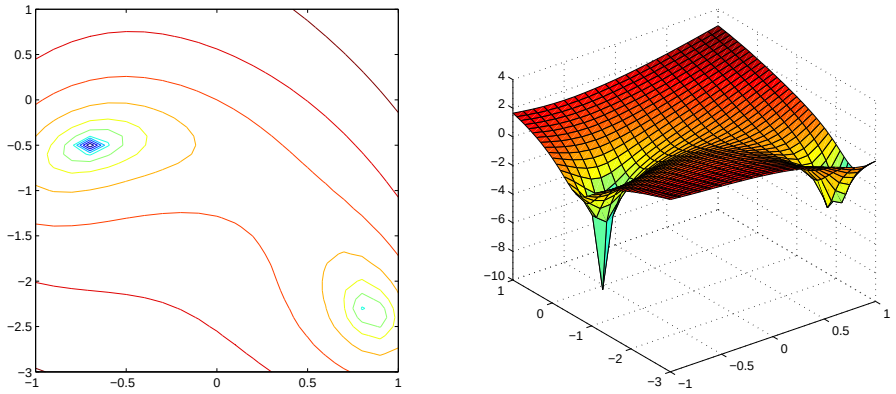
$$f(x, y) = \left[x^2 + \sin y\right]^2 + \left[e^x + y\right]^2$$

hat Minimalwert 0 für genau jene Wertepaare $(x; y)$, die Lösung des folgenden nichtlinearen Gleichungssystems sind:

$$\begin{aligned} x^2 + \sin y &= 0 \\ e^x + y &= 0 \end{aligned}$$

Aufgabe 29: Nullstellen einer Funktion in zwei Variablen (Teil 1)

Zeichnen Sie für $\log f$, den Logarithmus der oben gegebenen Funktion, im Bereich $-1 < x < 1$ und $-3 < y < 1$ eine `contour`- und eine `surf`-Grafik in der Art der Abbildung. (In der logarithmischen Darstellung erkennt man die Lage des Minimums deutlicher als in direkter Darstellung von f.) Verwenden Sie ≈ 20 Gitterpunkte in x- und ≈ 40 Gitterpunkte in y-Richtung.



Aufgabe 30: Nullstellen einer Funktion in zwei Variablen (Teil 2)

Lösen Sie das oben gegebene nichtlineare Gleichungssystem mit dem Newton-Verfahren. Finden Sie beide Lösungen im Bereich $-1 < x < 1$ und $-3 < y < 1$. Passende Startwerte können Sie aus der grafischen Darstellung (Aufgabe 29) entnehmen. Abbruchkriterium: Aufeinanderfolgende Iterationen unterscheiden sich, gemessen als Summe der Beträge aller Komponenten-Differenzen, weniger als 10^{-8}.

Ü 4 Vierte Übungseinheit

Inhalt der vierten Übungseinheit:

- Eindeutig, nicht eindeutig und gar nicht lösbare Systeme
- Fehlerempfindlichkeit, Matrixnormen, Konditionszahl
- Determinante, Komplexität
- Modelle an Daten anpassen
- Lineare Datenmodelle

Eine Anleitung für diese Übungseinheit

Im aktuellenVorlesungsskript behandelt Kapitel 3 lineare Gleichungssysteme mit quadratischer Matrix (mit genausoviel Gleichungen wie Unbekannten). In den Folien zur 4. Vorlesung können sie dazu die Abschnitte

1. Lösbarkeit, Fehlerempfindlichkeit und
4. Direkte Verfahren

zur Wiederholung durchklicken.

Insbesondere die Skriptum-Kapitel 3.2, 3.4, 3.6, 3.7.1 und 3.8 bringen Material, das direkt mit den Übungaufgaben 31 – 35 zusammenhängt.

Die weiteren Aufgaben in dieser Übungseinheit befassen sich damit, wie sich aus Datensätzen die Parameter eines Modells schätzen lassen; zum Beispiel: aus der Anzahl von infizierten Personen die Parameter eines exponentiellen Wachstumsmodells, Anleitung dazu in den Übungsunterlagen Ü 4.4.2.

Dazu sollten Sie in den Vorlesungsfolien das Thema

5. Überbestimmte Systeme

durchklicken und Kapitel 5 im Vorlesungsskript durchblättern. Wir werden die Theorie noch ausführlicher behandeln, aber zum praktischen Einsatz reicht vorläufig, zu wissen:

- Ein System mit mehr Gleichungen als Unbekannten ist im Regelfall nicht exakt lösbar
- Eine Näherungslösung (die „am wenigsten falsche Lösung“) lässt sich mit der Methode der kleinsten Fehlerquadrate bestimmen.
- MATLABS Operator \ findet bei überbestimmten Gleichungssystemen die kleinste-Quadrate-Näherungslösung.

Ü 4.1 Gleichungssysteme, Lösungsmannigfaltigkeiten

Können Sie ein lineares Gleichungssystem mit zwei Gleichungen und zwei Unbekannten immer lösen?

> Die Faustregel „Bei genausoviel Gleichungen wie Unbekannten gibt es immer eine Lösung“ ist falsch. FALSCH! **FALSCH!!**

Bei den drei Gleichungssystemen

$$\begin{aligned} x + y &= 2 \\ 2x + 2y &= 4 \end{aligned} \qquad \begin{aligned} x + y &= 2 \\ 2x + 2y &= 3 \end{aligned} \qquad \begin{aligned} x + y &= 2 \\ x + 2y &= 3 \end{aligned}$$

sehen Sie hoffentlich mit freiem Auge: Beim ersten und beim dritten sind $x = 1,\ y = 1$ Lösungen. Das mittlere System ist unlösbar. Beim ersten System gibt es allerdings noch unendlich viele weitere Lösungen.

Wiederholen Sie, was Sie dazu in Mathematik 1 gelernt haben. Im Skriptum, Kapitel 3.4 finden Sie weitere Informationen. Sie sollten mit folgenden Themen vertraut sein:

- Matrix-Rang, Rang der erweiterten Matrix, MATLAB-Befehle `rank(A), rank([A,b])`, Fallunterscheidungen zur Lösbarkeit.
- Stufenform eines Gleichungssystems, MATLAB-Befehl `rref([A,b])`
- Allgemeine Lösung des homogenen Systems, Nullraum, MATLAB-Befehl `null(A)`
- Spezielle Lösung des inhomogenen Systems, MATLAB-Befehle `A\b` bei vollem Rang, `pinv(A)*b` bei lösbaren Systemen mit Rang kleiner Zeilenzahl.
- Bei nicht lösbaren Systemen liefert `pinv(A)*b` die „am wenigsten falsche“ Antwort (mit kleinstem Fehler in der 2-Norm)

Aufgabe 31: Gleichungssysteme, Lösungsmannigfaltigkeiten

Welche der folgenden Gleichungssysteme $A\mathbf{x} = \mathbf{b}$ sind eindeutig, mehrdeutig oder gar nicht lösbar? Geben Sie, wenn möglich, die (oder eine) Lösung an. Bei mehrdeutigen Lösungen: wie viele freie Parameter hat die Lösungsschar?

Weitere Erläuterungen, Arbeitsschritte
Die möglichen Fälle zur Lösbarkeit entscheiden Sie über Rang der Matrix und Rang der erweiterten Matrix, Befehle `rank(A), rank([A,b])`. Als Alternative können Sie auch `rref([A,b])` verwenden.

Wenn eine eindeutige Lösung existiert, ist `A\b` der richtige Befehl.

Wenn eine Lösungsschar existiert, liefert `pinv(A)*b` eine mögliche Lösung, und zwar jene mit kleinstem Absolutbetrag. Der Standard-Befehl `A\b` kann auch funktionieren; er liefert einen Lösungsvektor mit möglichst vielen Komponenten gleich 0.

Die vollständige Darstellung einer Lösungsmannigfaltigkeit können Sie aus dem Ergebnis von `rref([A,b])` ablesen. Das erfordert aber noch etwas Umformen.

Alternative: Die allgemeine Lösung besteht aus einer speziellen Lösung (bestimmt mit `pinv(A)*b`) plus einer Linearkombination der Spaltenvektoren des Nullraumes (bestimmt mit `null(A)`).

1. $$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

2. $$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

3.
$$A = \begin{bmatrix} 10 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 \\ 2 & 1 & 0 & -1 \\ -2 & -3 & -4 & -5 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

4.
$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Aufgabe 32: LR-Zerlegung

Das Skriptum behandelt in Kapitel 3 Gauß-Elimination, LR-Zerlegung, Vorwärts-Elimination, Rücksubstitution. Dazu listet es auch JAVA-Codes. Hier eine Portierung in MATLAB. Wenn Sie diese Aufgabe durcharbeiten, haben Sie Ihren ersten eigenen Gleichungslöser programmiert.

> (Warnung: Die Programme dienen zur Illustration der Grundform der Verfahren. Für den praktischen Einsatz sind sieso noch nicht geeignet, es wäre zu ergänzen: Pivotierung, Absicherungen gegen singuläre Matrizen, Division durch Null...

Verwenden Sie die Angabe des Rechenbeispiels im Skript, Abschnitt3.6: Gegeben sei das Gleichungssystem $A \cdot \mathbf{x} = \mathbf{b}$ mit

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$

Sie können damit alle Zwischenergebnisse nachvollziehen.

Tippen Sie nicht ab, das ist zu fehleranfällig. Verwenden Sie Copy-Paste aus der PDF-Datei!

Schritt 1: LR-Zerlegung. Die beiden Matrizen L und R werden aus Effizienz-Gründen nicht extra gespeichert, sondern ersetzen die A-Matrix. Die L-Matrix ohne die Eins-Diagonale besetzt das Dreieck unterhalb der Hauptdiagonale; Die R-Matrix befüllt das obere Dreieck.

```
%% Zerlegung A = LR
% geht schief, wenn Null in der Diagonale auftritt!
for k=1:n
    for i=k+1:n
        A(i,k)=A(i,k)/A(k,k);
        for j=k+1:n
            A(i,j) = A(i,j)-A(i,k)*A(k,j);
        end
    end
end
```

Die Zerlegung ist damit abgeschlossen, vergleichen Sie (und beachten Sie: die beiden Ausgabe-Matrizen überschreiben die entsprechenden Bereiche in A):

$$A \text{ (Eingabe)} \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad L \cdot R \text{ (Ausgabe)} \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 0 & 8 & 9 \\ 0 & 0 & 10 \end{bmatrix}$$

Erst jetzt kommt die rechte Seite b ins Spiel: Die Vorwärts-Elimination löst $Ly = b$ und berechnet so das Zwischenresultat y. Im Code wird y gleich als x gespeichert.

```
%% Vorwärts-Elimination Ly=b
x = b;
for i=1:n
    for j=1:i-1
        x(i) = x(i) - A(i,j)*x(j);
    end
end
```

Das Zwischenresultat y bzw. x entspricht der letzten Spalte nach Abschluss des Eliminationsverfahrens, vor der Rücksubstitution, vergleichen Sie im Skript:

$$[A\,\mathbf{b}]^{(2)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

```
%% Rücksubstitution Rx=y
for i=n:-1:1
    for j=i+1:n
        x(i) = x(i)-A(i,j)*x(j);
    end
    x(i) = x(i)/A(i,i);
end
```

Nun ist x die Lösung des Systems.

Wie gut ist diese ganz naive Implementierung im Vergleich zu „richtigen“ Gleichungslöse-Programmen? Testen Sie dazu mit zufällig erzeugten Systemen

```
n=4;
A=floor(10*rand(n));
b=floor(10*rand(n,1));
```

und vergleichen Sie MATLABs Ergebnis `x=A\b` mit unserem Spielzeug-Programm. Meistens sollten die Ergebnisse gleich sein. Testen Sie viele Systeme und geben Sie an: in wieviel Prozent der Fälle versagt unser naives Programm? Warum?

Ü 4.2 Fehlerempfindlichkeit im Eliminationsverfahren

Aufgabe 33: Fehlerempfindlichkeit

Lösen Sie mit Matlabs Bergabstrich das Gleichungssystem $A\mathbf{x} = \mathbf{b}$ (es ist das Gleichungssystem Nr. 4 aus Aufgabe 31),

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Addieren Sie nun zur Matrix künstliche Fehler-Matrizen der Form $\delta A =$ `0.01*rand(4)` und vergleichen Sie die Lösung mit jener des ungestörten Systems.

Berechnen Sie die relativen Fehler der Matrixdaten[22] $||\delta A||/||A||$ und der Lösung $||\delta \mathbf{x}||/||\mathbf{x}||$. Testen Sie einige Fälle, auch mit kleineren oder größeren Störtermen, und schätzen Sie daraus das *maximale Verhältnis* von rel. Fehler der Lösungs zu rel. Fehler in den Daten ab.

Was bedeutet in diesem Zusammenhang der Wert `cond(A)`?

Ändern Sie nun die Matrix des Gleichungssystems zu

$$A = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

(Hinweis: `hilb(4)` liefert genau diese Matrix!) und wiederholen Sie Ihre Untersuchungen.

Ü 4.3 Entwicklung der Determinante, Komplexität

Aufgabe 34: Determinante

Gegeben sind

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, D = \begin{bmatrix} 10 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 \\ 2 & 1 & 0 & -1 \\ -2 & -3 & -4 & -5 \end{bmatrix}$$

Berechnen Sie von diesen Matrizen die Determinante auf verschiedene Arten:

1. Mit der Matlab-Funktion `det( )`
2. durch *LR*-Zerlegung, Matlab-Funktion `lu( )`, aus dem Produkt der Diagonalelemente von R[23]. Der Befehl `diag(R)` extrahiert übrigens die Hauptdiagonale einer Matrix als einen Vektor, der Befehl `prod` multipliziert alle Elemente.
3. Auch das primitive *LR*-Programm aus Aufgabe 32 liefert die Determinante als Produkt der Diagonalelemente (sogar vorzeichenrichtig, weil es keine Pivot-Suche durchführt). Testen Sie!
4. Die folgende Funktion implementiert das Standardverfahren zur Berechnung der Determinante durch Entwicklung nach Unterdeterminanten. (Copy-Paste aus PDF, lässt sich auch von der Übungs-Homepage herunterladen)

```
function d = mydet(A)
n = length(A);
if n==1
    d = A(1,1);
else
    d = 0;
```

[22] Sie brauchen Matrixnormen dazu! Informieren Sie sich in Skript und Vorlesungsfolien!

[23] Vorsicht, es kann sein, dass Matlabs `lu()` während der Rechnung Zeilen der Matrix vertauscht (Pivotierung). Mit jedem Zeilentausch wechselt das Vorzeichen der Determinante. Hat also `lu()` eine ungerade Anzahl von Zeilen vertauscht, hat das Produkt der Diagonalelemente das falsche Vorzeichen. Im Rahmen dieser Aufgabe sind Vorzeichen Glückssache.

```
        sig = 1;
        for i=1:n
            d = d + sig*A(1,i)*mydet(A(2:n,[1:i-1,i+1:n]));
            sig = -sig;
        end
    end
```

Wiederholen Sie auch die bekannten Standardverfahren (Regel von Sarrus, Entwicklung nach Unterdeterminanten) und rechnen Sie für A, B, C von Hand nach.

> Achtung: die Regel von Sarrus (Gitterzaunregel) gilt nur für 2×2- und 3×3-Matrizen. Wer 4×4-Determinanten auf diese Weise berechnen will, rechnet falsch. FALSCH! **FALSCH!!**

Aufgabe 35:

Ein Rechenverfahren, das zwar korrekt rechnet, aber ewig dafür braucht, hat nur theoretischen Wert. Die Berechnung der Determinante durch Entwicklung nach Unterdeterminanten ist ein Beispiel dafür.

Die Funktion `mydet` aus Aufgabe 34 implementiert das Standardverfahren zur rekursiven Berechnung der Determinante durch Entwicklung nach Unterdeterminanten.

Testen Sie die Rechenzeit`mydet` im Vergleich zum MATLAB-Standardbefeht `det`. Einfache Testmatrizen liefert z. B. die Funktion `magic(n)`.

MATLAB bietet mit den Befehlen `tic` und `toc` eine Art Stoppuhr an, mit der sie die Rechenzeit messen können. Sie könnten dazu Code der folgenden Art verwenden:

```
>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> tic; mydet(A); toc
Elapsed time is 0.000203 seconds.
```

Versuchen Sie nun auch etwas größere Matrizen als in Aufgabe 34. Bis zu welchem n lässt sich die Determinante

1. in weniger als zehn Sekunden,
2. in weniger als einer Minute
3. in weniger als zehn Minuten

berechnen? Das hängt natürlich von der Leistungsfähigkeit Ihres Rechners ab. Schätzen Sie aufgrund Ihrer Zeitmessungen und der Tabelle im Skriptum, Kapitel 3.7.1, wie lange die Berechnung für $n = 15$ und $n = 20$ dauern würde.

Ü 4.4 Modell-Funktionen an Daten anpassen

Gegeben sind Datenpunkte, gesucht sind Parameter für ein Modell, das diese Daten beschreibt. Im einfachsten Fall: eine „schöne Kurve“, die sich dem Verlauf der Datenpunkte gut anpasst.

Statistisch aussagekräftig ist das nur, wenn (deutlich) mehr Datenpunkte gegeben als Modellparameter gesucht sind. Das führt auf überbestimmte Gleichungssysteme.

Material und Links dazu siehe Einleitung. Noch einmal vorweg die wichtige Aussage:

> MATLABs Bergabstrich \ findet für überbestimmte Gleichungssystemen (die in der Regel nicht exakt lösbar sind) ein „am wenigsten falsches “ Ergebnis.

Etwas exakter gesagt: MATLABs Bergabstrich \ findet für überbestimmte Gleichungssystemen die bestmögliche Näherung im Sinn der kleinsten Fehlerquadrate.

MATLAB bietet in seiner *curve fitting toolbox* viele Hilfsmittel, um Kurven oder Flächen an gegebene Daten anzupassen. Bevor Sie sich darin vertiefen, sollten Sie die grundlegenden Ideen und Methoden verstehen. Darum geht es in dieser Übungseinheit.

Ü 4.4.1 Eine Variable: Beispiel aus der MATLAB-Hilfe

Die MATLAB-Hilfe (R2019–R2021) diskutiert Beispiele unter MATLAB › Data Import and Analysis › Descriptive Statistics › Programmatic Fitting › MATLAB Functions for Polynomial Models (geben Sie *“Programmatic Fitting”* im Suchfenster der Hilfe ein, dann finden Sie dieses Beispiel rasch.)

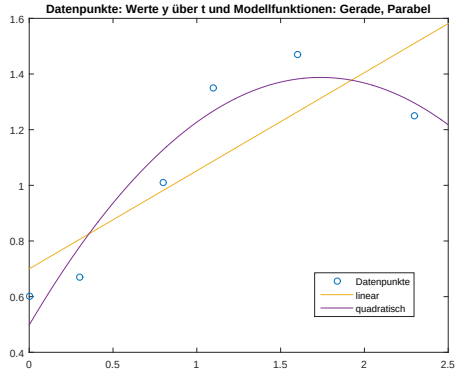
Hier ist das Beispiel im Abschnitt *MATLAB Functions for Polynomial Models* für Sie aufbereitet, so dass Sie aus den Angaben der MATLAB-Hilfe und den Zusatzinforrmationen folgende Aufgabe lösen können:[24]

Gegeben sind Messwerte y zu verschiedenen Zeiten t als Spaltenvektoren:

```
t = [0 0.3 0.8 1.1 1.6 2.3]';
y = [0.6 0.67 1.01 1.35 1.47 1.25]';
```

Sie sollen eine Funktion finden, die den Zusammenhang $y = y(t)$ gut modelliert. Einfache Ansätze sind: eine lineare Funktion, oder ein allgemeineres Polynom, zum Beispiel:

$$y = a_0 + a_1 t \qquad \text{oder}$$
$$y = a_0 + a_1 t + a_2 t^2$$



Dafür gibt es in MATLAB die Befehle `polyfit` und `polyval` und das *Basic Fitting User Interface*. Tipp: lassen Sie mit `plot` die Punkte zeichnen. Klicken Sie im *figure*-Fenster auf Tools › Basic Fitting und probieren Sie aus!

[24] Eigentlich müssen Sie nur die Beispiel-Befehle aus der Matlab-Hilfe in Ihre eigene Skript-Datei kopieren. Aber bitte schalten Sie ihr Hirn nicht aus, während Sie Strg + C und Strg + V verwenden!

Die Polynomfunktionen passen sich in diesem Beispiel nicht besonders gut an die Daten an. Versuchen Sie stattdessen ein lineares Modell[25] mit allgemeineren Funktionen, nämlich

$$y(t) = a_0 + a_1 e^{-t} + a_2 t e^{-t}$$

Vorgangsweise: Setzen Sie die gegebenen Datenpunkte in die Ansatzfunktion ein. Sie erhalten pro Wertepaar eine (lineare!) Gleichung in den drei Unbekannten a_0, a_1, a_2.

$$\begin{aligned} 0{,}60 &= a_0 + a_1 e^{0{,}0} + a_2 \cdot 0{,}0 \cdot e^{0{,}0} \\ 0{,}67 &= a_0 + a_1 e^{-0{,}3} + a_2 \cdot 0{,}3 \cdot e^{-0{,}3} \\ 1{,}01 &= a_0 + a_1 e^{-0{,}8} + a_2 \cdot 0{,}8 \cdot e^{-0{,}8} \\ \dots &\quad \dots \end{aligned}$$

Im MATLAB-Skript erstellen Sie die Koeffizientenmatrix X (die MATLAB-Hilfe sagt: *form the design matrix*). Achtung: `t` und `y` müssen Spaltenvektoren sein!

```
X = [ones(size(t))  exp(-t)  t.*exp(-t)];
```

Der Aufbau dieser Matrix ist der entscheidende Punkt, den Sie verstehen müssen. Die Koeffizienten von a_0 im obigen Gleichungssystem sind immer eins, daher enthält die erste Spalte von X lauter Einser, wie ein Volksschulzeugnis. Die Koeffizienten von a_1 sind e^{-t}-Werte, daher steht in der zweiten Spalte von X ein Vektor von e^{-t}-Werten, und so weiter.

Bitte mitdenken: Natürlich ist nicht immer automatisch die erste Spalte von X ein Volksschulzeugnis – das hängt von den gewählten Ansatzfunktionen ab, und von deren Reihenfolge. Aber als einfache Regel gilt: „Die Spalten von X enthalten die Werte der Ansatzfunktionen."

Die rechte Seite sind einfach nur die y-Werte. Die Modellkoeffizienten a_0, a_1, a_2 berechnet der Bergabstrich-Operator \ als bestmögliche Näherung aus dem überbestimmten Gleichungssystem:

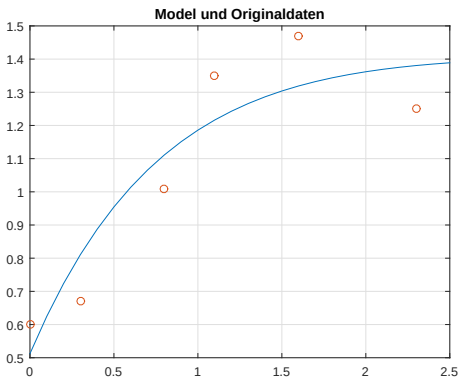
```
a = X\y
a =

    1.3983
   -0.8860
    0.3085
```

Also ist das bestmögliche Datenmodell gegeben durch

$$y = 1{,}3983 - 0{,}8860\, e^{-t} + 0{,}3085\, t e^{-t}.$$

Jetzt können Sie Ihre Modellfunktion auswerten (in genügend hoher Auflösung, also an vielen t-Punkten in engem Abstand) und zusammen mit den Originaldaten zeichnen.

```
T = (0:0.1:2.5)';
Y = [ones(size(T))  exp(-T)  T.*exp(-T)]*a;
plot(T,Y,'-',t,y,'o'), grid on
title('Modell und Originaldaten')
```



[25] *linear* sind hier nicht die verwendeten Ansatzfunktionen e^{-t} und te^{-t}. *Lineares Modell* bedeutet, dass diese Funktionen *linear kombiniert* werden: also ein Ansatz der Form „Koeffizient 1 mal Funktion 1 plus Koeffizient 2 mal Funktion 2…"

Auch hier bitte mitdenken: das Einsetzen von t-Daten in das Modell geschieht elegant wieder in Matrix-Vektor-Form: Die Matrix Y ist ähnlich aufgebaut wie die vorher verwendete X-Matrix. Multiplikation von Y mit Koeffizientenvektor $\mathbf{a}$ liefert Modellwerte.

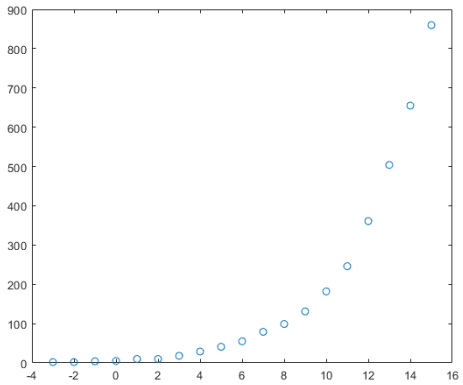
Alternative Auswertung: (in Wirklichkeit dieselbe Rechnung wie oben, nur für die a-Vektorkomponenten einzeln angeschrieben – vielleicht ist es so leichter verständlich)

```
Y = [ones(size(T))  exp(-T)  T.*exp(-T)]*a;  % Auswertung in Matrix-Vektor-Form
%
Y = a(1) + exp(-T)*a(2) + T.*exp(-T)*a(3); % Alternativ: komponentenweise Schreibung
```

Ü 4.4.2 Anpassen eines exponentiellen Wachstumsmodells

Gegeben sind (t, y)-Wertepaare. (MATLAB-Quelltext, copy-paste-fähig, nur die einfachen Anführungszeichen bei `'o'` müssen Sie ausbessern!)

```
t = -3:15;
y = [2 2 4 5 10 10 18 29 41 55 79 ...
99 131 182 246 361 504 655 860];
plot(t,y,'o')
```



Gesucht sind die Parameter a und b eines exponentiellen Wachstumsmodells

$$y = a \exp(bt) \ .$$

Es handelt sich bei y um die COVID-19-Fälle in Österreich (aufsummiert), mit Zeitachse t in Tagen vom 26. Februar bis 15. März 2020.

Wenn Sie hier, so wie vorher im Abschnitt Ü 4.4.1, die y- und t-Werte in die Modellgleichung einsetzen, erhalten Sie pro Wertepaar eine Gleichung für die unbekannten Parameter a und b.

$$\begin{aligned} 2 &= a \exp(-3b) \\ 2 &= a \exp(-2b) \\ 4 &= a \exp(-b) \\ \vdots &\qquad \vdots \\ 860 &= a \exp(15b) \end{aligned}$$

Im Unterschied zu vorher sind die Gleichungen jedoch nichtlinear in a und b. Wir behandeln echt nichtlineare überbestimmte Systeme erst später. Dieses System lässt sich aber zu einem linearen Problem vereinfachen.

Dieser einfache Trick zeigt dir, wie du ein nichtlineares System loswirst. . .

Logarithmieren Sie die Gleichungen:

$$\begin{aligned} \log 2 &= \log a - 3b \\ \log 2 &= \log a - 2b \\ \log 4 &= \log a - b \\ &\vdots \qquad\quad \vdots \\ \log 860 &= \log a - 15b \end{aligned}$$

(Natürlich steht log hier für den natürlichen Logarithmus!) Nennen Sie noch $\log a = \bar{a}$, dann steht hier ein überbestimmtes lineares Gleichungssystem in den Unbekannten $\bar{a}$ und b.

In MATLAB transformieren Sie die Datenvektoren von Zeilen- zu Spaltenvektoren. Die Koeffizientenmatrix X und rechte Seite dieses Systems erzeugen Sie dann so

```
t = t';
y = y';
X = [ones(size(t)) t];
rhs = log(y);
```

Der Bergabschrägstrich liefert die kleinste-Quadrate-Schätzung für die unbekannten Parameter $\bar{a} = \log a$ und b.

```
ab=X\rhs
```

Für die hier gegebenen Daten ist `ab(1)` $= \bar{a} = \log a = 1.7602$ und damit der Parameter $a = \exp(\bar{a}) = 5.8135$. Das ist der (vom Modell geschätzte) Anfangswert für $y(0)$, wobei hier $t = 0$ der 29. Februar ist.

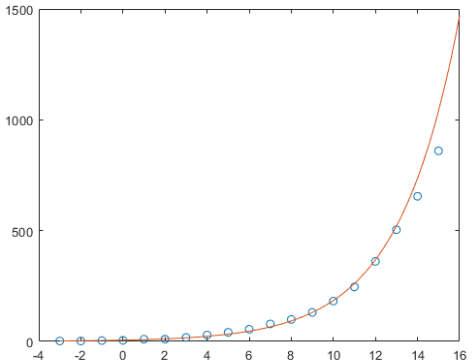
Der Wert `ab(2)` $= b = 0.3458$ ist die Wachstumskonstante im exponentiellen Modell. Um den Faktor $\exp(b) = 1.4132$ multiplizieren sich pro Tag die y-Werte, das entspricht einem täglichen Zuwachs um 41%. Die Verdopplungszeit T_2 in Tagen erhalten Sie aus

$$T_2 = \frac{\log 2}{b} = 2.0043 \quad .$$

Erinnern Sie sich noch, das war die beunruhigende Datenlage vor drei Jahren. Schauen wir uns die Daten und das geschätzte Wachstumsmodell genauer an.

Wir brauchen für die Zeitachse Zeitpunkte in feinerer Auflösung und gleich mal einen Tag in die Zukunft extrapoliert. `linspace` liefert hundert Zeitpunkte, das reicht. Für diesen `T`-Vektor werten wir das Modell aus.

```
a = exp(ab(1));
b = ab(2);
T = linspace(-3,16)';
Y = a*exp(b*T);
plot(t,y,'o',T,Y)
```



Zum Glück liegen die letzten beiden Datenpunkte schon unter der Modellkurve.

Vorsicht! Daraus allein hätte sich (mit Datenstand von März 2020) keine fundierte Vorhersage ableiten lassen. Dazu sind das Modell und unsere Datenanpassung zu stark vereinfacht. Was sich damals aussagen ließ: Ein exponentielles Wachstumsmodell mit den hier berechneten Parametern sieht in der grafischen Darstellung – bis auf die letzten beiden Datenpunkte – plausibel aus; **wenn sonst nichts das Wachstum bremst**, sagt dieses Modell in zwei Wochen katastrophale Fallzahlen voraus.

Im Nachhinein lassen sich die Modell-Voraussagen überprüfen. Am Ende dieses Abschnittes ist dazu eine grafische Darstellung bis Anfang April ergänzt.

Vorher versuchen wir aber noch eine etwas verfeinerte Modellrechnung. Denn unser logarithmierender Linearisierungs-Trick verursacht einen statistischen Fehler: die Parameter-Schätzung minimiert die Fehler in den logarithmierten Daten. Dadurch werden die (weniger aussagekräftigen) Datenpunkte zu Beginn stärker gewichtet als die (aktuelleren) Daten gegen Ende des Zeitintervalls.

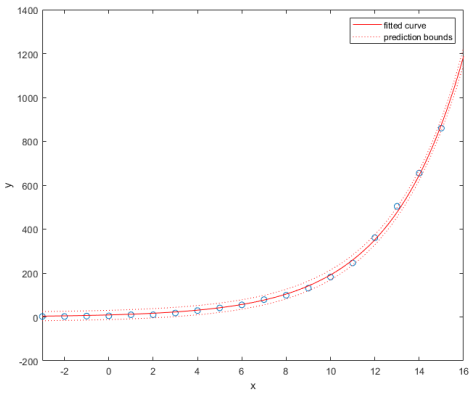
Wir werden uns noch eingehender mit der direkten Anpassung des nichtlinearen Datenmodells $y = a\exp(bt)$ befassen, aber jetzt lassen wir MATLAB arbeiten. Wenn bei Ihnen die *Curve Fitting Toolbox* installiert ist, funktioniert der Befehl
`modell = fit(t,y,'exp1')` Achtung, da steht 1, eins, nicht klein-ell bei `'exp1'` !)
und liefert als Ergebnis

```
modell =
     General model Exp1:
     modell(x) = a*exp(b*x)
     Coefficients (with 95% confidence bounds):
       a =       9.162  (7.889, 10.44)
       b =       0.304  (0.294, 0.314)
```

Zu den geschätzten Parametern wird auch ein 95%-Konfidenzintervall angegeben. **Vorsicht!** Diese Angaben gehen davon aus, dass die Daten tatsächlich einem exponentiellen Gesetz gehorchen und die einzelnen Datenpunkte nur zufallsbedingt mit einer gewissen Wahrscheinlichkeitsverteilung von der exponentiellen Kurve abweichen. Die Aussage ist also: *Wenn* ein exponentielles Wachstum vorläge, *dann wären das* die geschätzten Parameter.

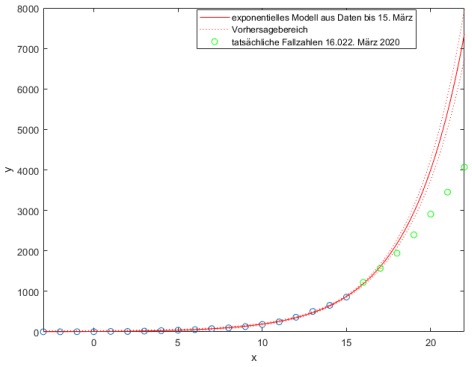
Wenn das Modell über MATLABs `fit`-Befehl berechnet ist, gibt es zusätzliche Optionen für die graphische Darstellung: Das Schlüsselwort `'predobs'` zeichnet auch Unsicherheits-Grenzen ein. Der `xlim([-3,16])`-Befehl stellt den x- (hier t-) Achsenbereich ein; ein größerer Endwert würde das Modell weiter in die Zukunft hin extrapolieren.

```
plot(t,y,'o')
hold on
xlim([-3,16])
plot(modell,'predobs')
hold off
```



Nochmal **Vorsicht!** Die engen Unsicherheitsgrenzen sollen nicht den Eindruck hoher Genauigkeit oder Verlässlichkeit entstehen lassen. Die Kurven fußen auf der Annahme, dass der Verlauf wirklich einem exponentiellen Modell folgt.

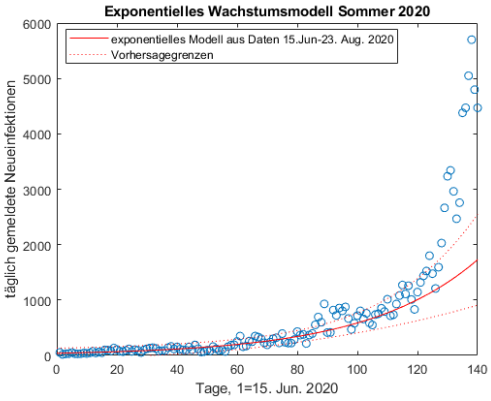
Wir haben das Modell in der obigen Graphik nicht weiter in die Zukunft extrapoliert, weil die Modellannahmen stark vereinfacht und unsicher waren. Tatsächlich konnte der erste Lockdown das exponentielle Wachstum bremsen. Die mit mehr Datenpunkten ergänzte Grafik rechts zeigt, dass einige Tage nach Beginn des Lockdowns die gemeldeten Fallzahlen von der exponentiellen Modellkurve abweichen. Es stellt sich ein anderes Wachstumsmodell ein.



Aufgabe 36: COVID-19 im Sommer und Herbst 2020

Sie können hier Daten zu den täglichen Neuinfektionen in Österreich, Zeitraum 140 Tage (15. Juni bis 1. Nov 2020) herunterladen.

Berechnen Sie aus den ersten 70 Datenpunkten die Parameter eines exponentiellen Wachstumsmodells, sowohl in der einfachen Form (linearer Fit an die logarithmierten Daten), als auch mit MATLABS *curve fitting toolbox*, Befehl `fit`. Vergleichen Sie das angepasste exponentielle Modell mit den tatsächlichen Daten. Stellen Sie Ihre Modellrechnung etwa so dar, wie in der Grafik rechts.



Interpretation: Bis etwa Tag 120 liegen die tatsächlichen Fallzahlen einigermaßen innerhalb der Vorhersagegrenzen, aber dann brechen die Daten nach oben aus. Offensichtlich hatte sich zu diesem Zeitpunkt (Mitte Oktober) der Ausbreitungsmodus signifikant verändert. Konsequenter Weise mussten ab Anfang November wieder Ausgangsbeschränkungen verordnet werden.

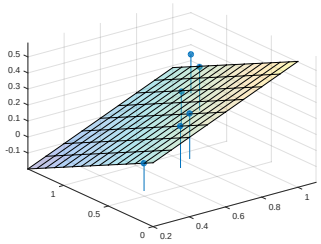
Ü 4.4.3 Mehrere Variable

Die MATLAB-Hilfe zeigt *Multiple Regression* als nächstes Beispiel unter
MATLAB » Data Import and Analysis » Descriptive Statistics » Programmatic Fitting » Multiple Regression .

Beschreibung und Illustration siehe auch 4. Vorlesung, vorletzte Folie. Auch das Skriptum erklärt in Kapitel 5.4: Anpassen eines linearen Modells (einer Ausgleichsebene).

Hier eine Musterlösung zur Aufgabe der vorletzten Vorlesungsfolie. Im Vergleich zur Lösung in der MATLAB-Hilfe zeigt sie auch das Zeichnen der Ebene und der Originaldaten.

```
%% Datenpunkte: Variable x1, x2 und Messwert y
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
y  = [.17 .26 .28 .23 .27 .24]';

%% Matrix des ueberbestimmten Systems
% fuer  Ansatz y = a0 + a1*x1 + a2*x2
X = [ones(size(x1))  x1  x2];

%% Loesung: Koeffizientenvektor
a = X\y
%% Auswertung und Zeichnung auf feinem Gitter
[XX,YY]=meshgrid(0.2:0.1:1,0:0.1:1.4);
ZZ= a(1) + a(2)*XX + a(3)*YY;

surf(XX,YY,ZZ,'FaceAlpha',0.3),
axis tight, hold on
% Originaldaten
stem3(x1,x2,y,'.','MarkerSize',25)
hold off
```



Ü 4.5 Weitere Aufgaben zu Linearen Datenmodellen

Aufgabe 37: Hochwasser

Für die Blies (einen Nebenfluss der Saar) sollen die Hochwasserstände am Pegel Neunkirchen aus den Wasserständen des Pegels Ottweiler und des Pegels Hangard vorhergesagt werden. Es liegen die Daten der Scheitelwasserstände von 12 Winterhochwässern aus den Jahren 1963–1971 vor: (Daten können Sie von der Übungs-Homepage runterladen)

Wasserstand in cm												
Neunkirchen y	172	309	302	283	443	298	319	419	361	267	337	230
Ottweiler x_1	93	193	187	174	291	184	205	260	212	169	216	144
Hangard x_2	120	258	255	238	317	246	265	304	292	242	272	191

Quelle: U. Maniak, Hydrologie und Wasserwirtschaft, Springer, 1988

Wir unterstellen den Daten das lineare Modell $a_0 + a_1 x_1 + a_2 x_2 = y$. In diesem Ansatz sind a_0, a_1 und a_2 unbekannte Koeffizienten, die aus den zwölf gegebenen Werte-Tripeln möglichst gut bestimmt werden sollen.

Berechnen Sie die Koeffizienten des linearen Modells, und geben Sie auch den Differenzenvektor zwischen Modellvorhersage und Messwerten an. (Dieses Beispiel ist im Skriptum Kapitel 5.4 ausführlich durchgerechnet)

Aufgabe 38: Jack Sparrows Kompass

Egal, ob Sie eine Geophysik-Vorlesung besucht oder Johnny Depp in *Pirates of the Caribbean* gesehen haben, Sie haben gelernt, dass eine Kompassnadel nicht immer nach Norden zeigt.

Die Abweichung heißt magnetische Deklination, ihre Kenntnis ist nach wie vor von großer Wichtigkeit für die Seefahrt. Deswegen gibt die Zentralanstalt für Meteorologie und Geodynamik regelmäßig für alle Landeshauptstädte aktuelle Werte der magnetischen Deklination an:[a] (Daten von Übungs-Hompage herunterladbar!)

[a] Update 2017: Diese Tabelle wird von der ZAMG inzwischen nicht mehr bereitgestellt, vermutlich, weil die Bedeutung der Seefahrt in den Landeshauptstädten abgenommen hat)

Deklinationswerte der einzelnen Landeshauptstädte
(östliche Deklination bezogen auf Jahresmitte 2008)

Stadt	Länge x	Breite y	Deklination D
Wien (WIK)	16.37	48.20	3.0000
Eisenstadt	16.52	47.85	3.0333
St.Pölten	15.63	48.20	2.9000
Graz	15.45	47.07	2.7833
Linz	14.30	48.30	2.5500
Klagenfurt	14.31	46.62	2.5000
Salzburg	13.03	47.80	2.2500
Innsbruck	11.40	47.27	1.8833
Bregenz	9.77	47.50	1.4000

Finden Sie für diese Daten eine Anpassung der Form

$$z(x, y) = a_1 + a_2 x + a_3 x^2 + a_4 y$$

Geben Sie die Differenz $D - z(x, y)$ zwischen den tatsächlichen Deklinationswerten und den angepassten Werten an. Welcher Wert wird am schlechtesten approximiert?

Moderne Modelle für das Erdmagnetfeld (International Geomagnetic Reference Field IGRF, World Magnetic Model WMM) verwenden im Prinzip denselben Ansatz: Sie passen ein Modell an Messdaten an. Aber im Unterschied zu unserem Polynom-Ansatz für 9 Datenpunkte verarbeiten Sie riesige Datenmengen und verwenden als Ansatz Kugelflächenfunktionen.

Aufgabe 39: Für Weicheier

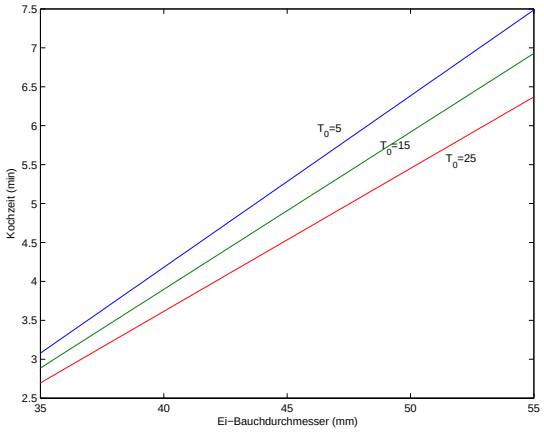
Der Wiener Physiker Werner Gruber beschäftigt sich mit der Thermodynamik des Eierkochens und gibt nebenstehende Werte für die Kochzeit eines perfekt weichen Frühstücks-Eis an, abhängig von Bauchdurchmesser d (nicht seiner, der des Eis) und Ausgangstemperatur T_0 (Daten können Sie von der Übungs-Homepage runterladen)

Durchmesser d (mm)	Ausgangstemperatur T_0 (C)	Kochzeit (min:sek)
35.0	4	3:10
40.0	4	4:10
45.0	4	5:20
50.0	4	6:30
35.0	20	3:00
40.0	20	3:40
45.0	20	4:40
50.0	20	5:50

Finden Sie für diese Daten eine Anpassung der Form

$$t(d, T_0) = a_1 + a_2 d + a_3 T_0 + a_4 d T_0$$

Zeichnen Sie ein Diagramm ähnlich dem hier gezeigten (x-Achse: Eidurchmesser, y-Achse: Kochzeit), in dem Sie für Anfangs/-temperaturen 5,15,25 Grad die Kochzeiten eintragen.

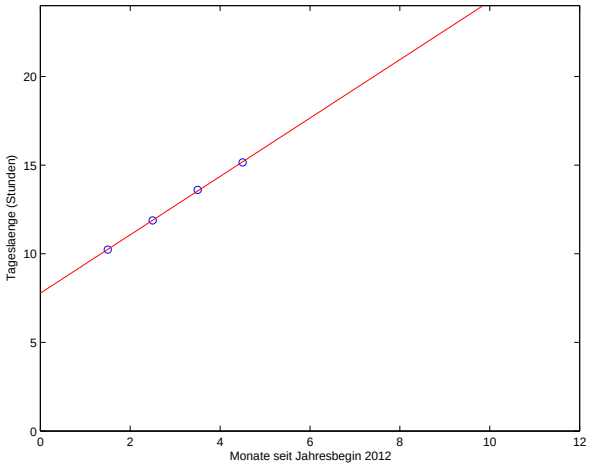


Aufgabe 40: Ein Reich, in dem die Sonne nicht untergeht

Eine Aufgabe zu Datenmodellen mit polynomialen und allgemeineren Ansatzfunktionen.

Die folgende Tabelle gibt für Leoben die Tageslänge (von Sonnenauf- bis -untergang) in Stunden für vier Tage von Februar bis Mai an. Klar ersichtlich ist ein eindeutiger Trend zu immer längeren Tagen.

Datum kalendarisch	Monate seit Jahresbeg.	Tageslänge Stunden
15. Feb. 2019	1,5	10,23
15. Mär. 2019	2,5	11,88
15. Apr. 2019	3,5	13,60
15. Mai. 2019	4,5	15,15



Anlageberater und andere begabte Verkäufer könnten Ihnen anhand der Grafik zu diesen Daten überzeugend erklären, dass spätestens Ende Oktober die Sonne nicht mehr untergeht.

Bevor Sie nun in Solar-Aktien investieren, erstellen Sie selber Modelle für diesen Datensatz

1. Lineare Regression: $y = a_0 + a_1 x$
2. Polynom-Ansatz $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$
3. Datenmodell $y = a_1 + a_2 \cos\left(x\frac{\pi}{6}\right) + a_3 \sin\left(x\frac{\pi}{6}\right)$

Stellen Sie die Datenpunkte und die drei Modelle in einer Grafik dar (Achsen-Bereich wie oben). Beantworten Sie (durch Ablesen aus der grafischen Darstellung) folgende Fragen: Ab wann geht laut Modell 1 die Sonne in Leoben nicht mehr unter (Tageslänge 24 h)? Für wann sagt Modell 2 ewige Nacht voraus (Tageslänge 0 h)? Wann ist laut Modell 3 Sommersonnenwende (Tageslänge maximal)?

Nur Modell 3 kann verlässliche Voraussagen treffen, weil es einen problemgerechten Ansatz verwendet: die periodischen Schwankungen des Sonnenlaufes werden durch Sinus- und Cosinusterme genähert.