

Einführung in die Programmierung mit MATLAB

Dominik Edelmann

Numerische Analysis, Eberhard Karls Universität Tübingen

October 7, 2021

Ablauf

- ▶ Vormittags
 - ▶ Theoretische Erklärung über Zoom.
 - ▶ Live-Programmierung kleiner Programme.
- ▶ Nachmittags
 - ▶ Programme auf Übungsblatt werden programmiert.
 - ▶ Wahlweise zuhause oder unter Betreuung im N16 (oder gar nicht!)

Einleitung

Grundlagen der Syntax

Kontrollstrukturen

Skripte und Funktionen

Vektoren und Matrizen

Visualisierung

Aufbau eines Matlab-Programms

Debugging, Fehlersuche, Performance

Performance

Fazit

Warum Numerik?

- ▶ Viele analytischen Probleme nicht explizit lösbar oder
- ▶ explizite Darstellung der Lösung nicht zur schnellen Berechnung geeignet.

Beispiele:

$$\int_{-1}^1 e^{-x^2} dx =? \quad \frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \pm \dots$$

Warum Numerik?

Numerische Mathematik (kurz: Numerik):

- ▶ Konstruktion von Algorithmen zur näherungsweise Berechnung (**Approximation**) von Lösungen kontinuierlicher Probleme.
- ▶ Analyse der *Genauigkeit* der berechneten Lösung (Fehleranalyse).

Numerik und Programmieren?

- ▶ Mathematisch formulierte Algorithmen werden in Programmiersprache übersetzt.
- ▶ Algorithmen können zuverlässig und schnell berechnet werden.
- ▶ Algorithmen können mit unterschiedlichen Parametern durchgeführt werden.

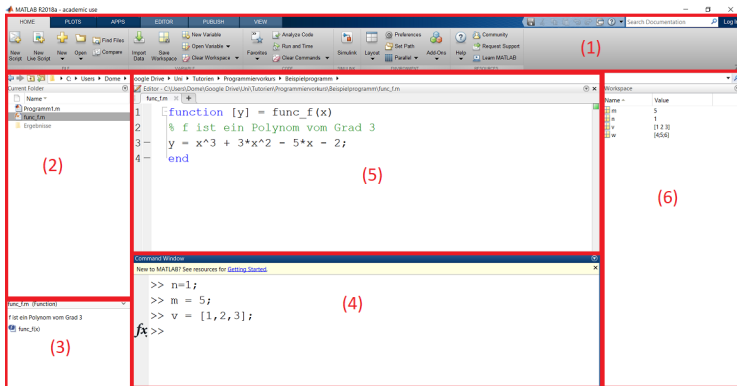
Was bedeutet Programmieren?

- ▶ Übersetzen von *umgangssprachlichen* Anweisungen in Computersprache.
- ▶ Erstellen von Computerprogrammen.

Matlab ...

- ▶ ... ist ein Softwarepaket zur numerischen Lösung mathematischer Probleme.
- ▶ ... ist eine Programmiersprache, in der Algorithmen implementiert werden.
- ▶ ... bietet massenweise vorgefertigter Algorithmen.

Aufbau von Matlab (Beispiel)



Der Aufbau kann im Menüpunkt **Layout** angepasst werden.

Aufbau von Matlab

- ▶ Im **Menü (1)** kann das Layout angepasst werden, Funktionen und Skripte erstellt werden, allgemeine Einstellungen vorgenommen werden (Preferences) und vieles mehr.
- ▶ Bei **Current folder (2)** wird der Ordner angezeigt, in dem Matlab gerade arbeitet.
 - ▶ Hier können Funktionen und Skripte geöffnet werden ...
 - ▶ ... oder auf Unter- und Überordner zugegriffen werden.
- ▶ Der Bereich **Details (3)** zeigt Details der in (2) ausgewählten Datei

Aufbau von Matlab

- ▶ Im **Command Window (4)** können Befehle eingegeben werden.
- ▶ Im **Editor (5)** werden Funktionen und Skripte erstellt.
- ▶ Im Bereich **Workspace (6)** werden die Variablen angezeigt, die aktuell im Arbeitsspeicher gespeichert sind.

Aufgaben

Aufgabe 1

- (a) Öffnen Sie Matlab.
- (b) Deklarieren Sie im Workspace einige Variablen, z. B. $n=3$ und $m=5$.
- (c) Geben Sie beispielsweise die folgenden Befehle ein: $n+m$, $n+m;$, und $k=n+m$. Machen Sie sich die Unterschiede klar, indem Sie darauf achten, welche Werte im Command Window zurückgegeben werden und welche Werte die Variablen danach haben (Workspace).

Syntax

- ▶ Computer kann keine Umgangssprache verstehen.
- ▶ Im Gegensatz zu (den meisten) Menschen kann ein Computer keine Sätze verstehen, die im Aufbau falsch sind.
- ▶ Syntax legt Regeln über zulässige *Sprachelemente* der Programmiersprache fest.

Syntax

- ▶ Damit ein Programmierbefehl durchgeführt werden kann, muss die richtige *Syntax* verwendet werden:

```
a = 3;
```

```
b = 2a; ← Nicht lauffähig!
```

- ▶ Jeder Befehl wird typischerweise mit einem Semikolon beendet, um die Ausgabe des Ergebnisses zu unterdrücken¹.

- ▶ Mit dem Prozentzeichen können Sie Kommentare in Ihr Programm einfügen:

```
p = 3.14; % a ist ein Näherungswert für pi
```

⇒ Alles ab dem Prozentzeichen bis zum Ende der Zeile wird von Matlab bei der Durchführung des Programms ignoriert.

¹Manchmal ist es aber hilfreich, wenn bestimmte Werte ausgegeben werden.

Syntax

- ▶ Vordefinierte Konstanten, z. B. `pi`, `eps` (Maschinengenauigkeit), `i` (Imaginäre Einheit). Nicht: `e` (man verwendet `exp(1)`).
- ▶ Vordefinierte Funktionen, z. B. `sin`, `exp` usw.
- ▶ Strings (Zeichenketten) werden mit einfachen Anführungszeichen umschlossen:
`string = 'Hallo.'`
- ▶ Code kann mit dem 3-Punkt-Operator auf mehrere Zeilen verteilt werden (fortgeschritten)
`summe = 1 + 2 + 3 + 4 + 5 ...`
`+ 6 + 7 + 8 + 9 + 10;`

Variablen

- ▶ Variablen und Operatoren (+, -, *, /, ...) sind die wichtigsten Bausteine beim Programmieren.
- ▶ Variablen werden durch Ausdrücke wie $a=3+5$, $v = [1 \ 2 \ 3]$ usw. deklariert.
- ▶ Das Gleichheitszeichen ist wie ein $:=$ in mathematischen Texten zu verstehen: Der Ausdruck rechts vom Gleichheitszeichen wird berechnet (falls möglich), und dann als Variable gespeichert, deren Namen links vom Gleichheitszeichen steht.
⇒ $n = n+1$ erhöht den Wert der Variable n um 1.

Regeln für Variablennamen

- ▶ Variablennamen dürfen aus Klein- und Großbuchstaben, Zahlen und Unterstrichen gebildet werden.
- ▶ Variablennamen dürfen nur mit Buchstaben beginnen.
- ▶ Matlab unterscheidet zwischen Groß- und Kleinschreibung!
- ▶ Es gehört zu einem guten Programmierstil, weitgehend selbsterklärende, intuitive und möglichst kurze Variablennamen zu verwenden.
- ▶ Halten Sie sich möglichst auch an Konventionen, die Sie sonst aus der Mathematik kennen: z. B. n, m, \dots für natürliche Zahlen, v, w für Vektoren und A für Matrizen etc.

Skripte

- ▶ Einfache Matlab-Programme bestehen aus Skripten und Funktionen
- ▶ Technisch gesehen sind beide eine `.m`-Datei
- ▶ Großer Unterschied: Ein Skript *kennt* alle Variablen, die im Workspace vorhanden sind und verändert diese.
(Im Gegensatz dazu hat bei einer Funktion jeder Funktionsaufruf seinen eigenen Workspace. Dazu später mehr)

⇒ Der Aufruf eines Skripts ist äquivalent dazu, die dort enthaltenen Befehle nacheinander in das Command Window einzugeben.

Aufgaben

Aufgabe 2

- (a) Navigieren Sie in der *Current Folder* Spalte zu Ihrem Ordner für den Programmierkurs.
- (b) Erstellen Sie per Rechtsklick ein neues Skript (Rechtsklick → New File → New Script) mit dem Namen `meinErstesSkript.m`
- (c) Öffnen Sie dieses Skript per Doppelklick. Die leere Datei erscheint im Editor.
- (d) Programmieren Sie nun (im Editor) einige der Befehle, die Sie bisher kennen. Deklarieren Sie nach Belieben einige Variablen und führen einige Rechenoperationen durch.
- (e) Rufen Sie ihr Skript auf, indem Sie im Command Window den Befehl `meinErstesSkript` eingeben.

Aufgaben

Aufgabe 3

1. Geben Sie, nachdem Sie das Skript aus der vorigen Aufgabe aufgerufen haben, den Befehl `clear all` ein. Was passiert mit den im Workspace gespeicherten Variablen?
2. Geben Sie den Befehl `clc` ein. Was passiert?

⇒ In den meisten Fällen ist es sinnvoll, diese beiden Befehle an den Anfang eines Skripts zu stellen.

Relationale Operatoren

- ▶ Neben dem Zuweisungsoperator = und den arithmetischen Operatoren + - * / ^ gibt es weitere Operatoren.
- ▶ Die relationalen Operatoren sind == ~= > >= < <=.
`n = (3 ~= 4);`
`disp(n); % Ausgabe: true`
- ▶ Diese werden in der Regel verwendet, um bestimmte Bedingungen abzufragen.
- ▶ Auch hier gilt: Die Operation rechts von =, also hier die Abfrage, ob $3 \neq 4$ gilt, wird zuerst durchgeführt. Das Ergebnis true wird in *n* gespeichert.

Logische Operatoren

- ▶ Neben den relationalen Operatoren gibt es auch logische Operatoren: `&&` `||` `~` stehen für and, or bzw. not.
- ▶ Mit logischen Operatoren werden logische Variablen (also `true` und `false`) verknüpft. Bei `||` handelt es sich um ein sog. *einschließendes Oder*.

Beispiel:

```
x = 5;  
y = (x < 6) || (x > 4);  
disp(y); % Ausgabe true  
z = (x > 0) && ~(x > 3);  
disp(z); % Ausgabe false
```

if-Abfragen

- ▶ In sehr vielen Programmen gibt es einzelne Schritte, die nur unter bestimmten Bedingungen durchgeführt werden.
- ▶ Dafür werden if-Abfragen gepaart mit logischen Ausdrücken verwendet. Die Struktur ist dabei immer Folgende:

if *logischer Ausdruck*

 Anweisung(en)

elseif *logischer Ausdruck*

 Anweisung(en)

 ⋮

else

 Anweisung(en)

end

if-Abfragen

- ▶ Das letzte else darf keinen logischen Ausdruck haben.
- ▶ Die Zeilen mit `if`, `elseif`, `else` und `end` werden **nicht** mit einem Semikolon beendet!
- ▶ `elseif` und `else` sind nicht erforderlich:

```
if x ~= 0  
    y = 1/x;  
end
```

⇒ Da keine Alternative mit `else` angegeben wurde, werden diese Zeilen einfach übersprungen, falls $x = 0$ ist.
- ▶ if-else-Strukturen können beliebig ineinander geschachtelt werden. Matlab rückt die Anweisungen zwischen `if` und `else` um vier Leerzeichen ein. Behalten Sie diese Einrückung bei! Dadurch wird der Code übersichtlicher.

if-Abfragen

Aufgabe 4

Schreiben Sie ein Skript, in dem Sie zwei Zahlen x und y deklarieren. Dann soll das Programm $z = \frac{x}{y}$ ausrechnen, falls $y \neq 0$ ist und eine Fehlermeldung, falls $y = 0$.

Aufgabe 5

Schreiben Sie ein Skript, in dem Sie zwei Zahlen x und y deklarieren. Das Programm soll ausgeben, ob $x > y$, $x = y$ oder $x < y$ ist.

Schleifen

- ▶ Zur wiederholten, *iterativen* Durchführung bestimmter Operationen beispielsweise bei
 - ▶ Partialsummen
 - ▶ Integration mit Riemann-Summen
 - ▶ Gauß-Elimination
- ▶ Schleifen sind gemeinsam mit `if`-Abfragen die wichtigsten Kontrollstrukturen beim Programmieren.

for-Schleifen

- ▶ for-Schleifen haben die Struktur
for *Variable = Startwert:Endwert*
Anweisung(en)
end
- ▶ Beispiel: Berechnung von $S = \sum_{k=0}^n \frac{(-1)^k}{2k+1}$ für gegebenes n ,
z. B. $n = 5$:
 - 1: `n = 5;`
 - 2: `S = 0; % Variable für die Summe`
 - 3: `for k=0:n`
 - 4: `S = S + (-1)^k / (2*k+1);`
 - 5: `end`

Erläuterungen

- ▶ In Zeile 1 wird zunächst der Index n , bis zu der die Partialsumme berechnet werden soll, deklariert.
- ▶ In Zeile 2 wird eine Variable S deklariert, die am Ende den Wert der Summe enthalten soll. Dies ist notwendig, da die einzelnen Summanden in Zeile 4 zu einer bereits bestehenden Variable addiert werden müssen.
(Bei Produkten würde man mit dem Wert 1 initialisieren.)

for-Schleifen

Erläuterung:

- ▶ In Zeile 3 wird zunächst eine Variable $k=0$ definiert.
- ▶ Alle Anweisungen die zwischen dieser Zeile und `end` auftreten werden nacheinander ausgeführt, wobei die Variable k den Wert 0 hat.
- ▶ Dann wird k um 1 erhöht, hat also nun den Wert 1.
- ▶ Alle Anweisungen werden nacheinander ausgeführt, diesmal mit $k=1$
- ▶ Dann wird k um 1 erhöht und die Anweisungen erneut durchgeführt
- ▶ ... so lange bis $k=5$ ist, wobei alle Anweisungen dann ein letztes Mal durchgeführt werden.

for-Schleifen

Aufgabe 6

- (a) Schreiben Sie ein Skript, in dem Sie zwei ganze Zahlen n und N definieren. Das Programm soll nun alle ganzen Zahlen von n bis einschließlich N ausgeben.
- (b) Was passiert, wenn Sie n größer als N wählen?
- (c) Verbessern Sie ihr Programm, so dass es eine Fehlermeldung ausgibt und die Schleife nicht aufgerufen wird, wenn $n > N$ ist.

Aufgabe 7

Schreiben Sie ein Skript, das für eine gegebene Zahl n die Fakultät $n!$ berechnet.

for-Schleifen

- ▶ Manchmal möchte man nicht, dass die Schleifenvariable in jedem Schritt um 1 erhöht wird.
- ▶ Beispiel: Berechnung von $7! = 7 \cdot 6 \cdot \dots \cdot 2 \cdot 1$

```
n = 7;
```

```
x = n;
```

```
for i=(n-1):-1:1
```

```
    x = x*i;
```

```
end
```

- ▶ Erklärung: i hat anfangs den Wert $n-1 = 6$ und wird in jedem Schleifendurchlauf um -1 erhöht, solange, bis die Schleife mit $i = 1$ das letzte Mal durchgeführt wird.

while-Schleifen

- ▶ Manchmal ist nicht von vornherein klar, wie oft die Anweisungen in einer Schleife wiederholt werden müssen.
- ▶ Beispiel: Man kann zeigen, dass die rekursiv definierte Folge

$$x_{k+1} = \frac{1}{2} \left(x_k + \frac{2}{x_k} \right), \quad x_1 = 2$$

gegen $\sqrt{2}$ konvergiert.

- ▶ Möchte man dieses Verfahren realisieren, berechnet man nacheinander diese Folge, bis sich zwei aufeinander folgende Folgenglieder nicht mehr stark unterscheiden, also z. B. solange, bis $|x_{k+1} - x_k| < \text{tol}$ mit einer vorgegebenen *Toleranz* tol .
⇒ Es ist nicht klar, wie viele Iterationen erforderlich sind.

while-Schleifen

- ▶ Eine while-Schleife hat die Struktur
while *logischer Ausdruck*
 Anweisung(en)
end
- ▶ Vor jedem Schleifendurchlauf wird gecheckt, ob *logischer Ausdruck* den Wert `true` liefert. Dann werden die Anweisungen zwischen `while` und `end` durchgeführt und *logischer Ausdruck* wird erneut gecheckt.
- ▶ **Achtung:** Es ist möglich, dass *logischer Ausdruck* immer `true` liefert. In diesem Fall bricht die Schleife nicht ab. Man spricht auch von einer **Endlosschleife**².

²Das ist bei einer for-Schleife auch möglich, dann hat man aber i. d. R. einen Programmierfehler gemacht.

Aufgabe 8

Schreiben Sie ein Programm, welches die rekursive Folge $x_{k+1} = \frac{1}{2}(x_k + \frac{2}{x_k})$ mit Startwert $x_1 = 2$ solange berechnet, bis die Differenz zweier aufeinander folgender Werte kleiner als eine vorgegebene Toleranz ist.

Dazu soll in einer While-Schleife jeweils ein `xNeu` aus `xAlt` gemäß der obigen Rechenvorschrift berechnet werden.

break und continue in Schleifen

- ▶ **continue**: Der aktuelle Schleifendurchlauf wird unterbrochen und es wird an den Schleifenanfang gesprungen.
- ▶ **break**: Die komplette Schleife wird unterbrochen und die Abarbeitung wird nach der Schleife fortgesetzt.

```
for i=1:6
    if i == 2
        continue
    end
    if i == 4
        break
    end
    disp(i)
end
```

Welche Zahlen gibt das Programm aus?

Skripte

- ▶ Ein Skript auszuführen ist gleichbedeutend damit, die Befehle nacheinander im Command Window einzugeben. Das bedeutet insbesondere:
 - ▶ Ein Skript kennt alle Variablen, die bereits vorher deklariert wurden (und sich im Workspace befinden).
 - ▶ Werden bereits vorhandene Variablen geändert oder neue deklariert, so sind diese Änderungen nach der Ausführung des Skripts vorhanden.

Funktionen

- ▶ Im Gegensatz zu Skripten haben Funktionen ihren *eigenen Workspace*. Das bedeutet:
 - ▶ Wird eine Funktion aufgerufen, so kennt diese die bereits im Workspace vorhandenen Variablen nicht und kann diese somit auch nicht ändern.
 - ▶ Werden in einer Funktion neue Variablen definiert, so sind diese nach Auswertung der Funktion nicht im Workspace vorhanden.
- ▶ Funktionen erhalten typischerweise Input-Parameter (*Variablen*) und liefern als Ergebnis Output-Parameter (*Funktionswerte*)

Beispielfunktion

Die folgende Funktion berechnet die Fläche und den Umfang eines Rechtecks:

```
function [ar, per] = area(a,b)
%Berechne Fläche und Umfang eines Rechtecks mit
Seitenlängen a,b.
ar = a*b;
per = 2*a+2*b;
end
```

Dabei sind a und b die Input-Parameter und ar und per die Output-Parameter.

Wichtiges über Funktionen

- ▶ **Wichtig:** Bei einer Funktion müssen Datei- und Funktionsnamen übereinstimmen. Die obige Funktion muss also in einer Datei `area.m` gespeichert werden.
- ▶ Obige Funktion erwartet beim Aufruf zwei Input-Parameter.
- ▶ Die Rückgabewerte müssen nicht speziell gesetzt werden (s. folgende Aufgabe).
- ▶ Benötigt eine Funktion keine Eingabewerte, so ist die runde Klammer hinter dem Funktionsnamen leer: `()`.
- ▶ Liefert eine Funktion keine Rückgabewerte, so bleibt die eckige Klammer hinter `function` leer: `[]`.

Aufgaben

Aufgabe 9

Schreiben Sie eine Funktion `function [ar,per] = area(r)`, welche die Fläche (= area) und den Umfang (= perimeter) eines Kreises mit Radius r zurückgibt.

Rufen Sie dann die Funktion (im Command Window) auf unterschiedliche Weisen auf (ohne Semikolon ; am Ende).

Betrachten Sie nach jedem Aufruf den Workspace.

- ▶ `area(r)`
- ▶ `[ar,per] = area(r)`
- ▶ `x = area(r)`

Welchen Wert bekommt x ? Machen Sie sich die unterschiedlichen Bedeutungen der Aufrufe klar.

Wichtige Funktionen in Matlab

- ▶ Trigonometrische Funktionen `sin`, `cos`, `tan`
- ▶ Exponentialfunktion und natürlicher Logarithmus `exp`, `log`
- ▶ Wurzelfunktion `sqrt`
- ▶ Absolutbetrag `abs`
- ▶ Verwaltungsfunktionen `length`, `size`
- ▶ Laden, Speichern `load`, `save` (wird nicht besprochen)
- ▶ Grafische Funktionen `plot` (später)
- ▶ *Mächtigere* Funktionen `lu`, `qr`, `norm` ... (wird nicht besprochen)

Vektoren und Matrizen

- ▶ Matlab basiert auf Matrizen. (Matlab steht für **Matrix Laboratory**)
- ▶ Skalare sind (1×1) -Matrizen.
- ▶ Spalten- bzw. Zeilenvektoren sind $(n \times 1)$ - bzw. $(1 \times n)$ -Matrizen.

Vektoren: Erzeugung

- ▶ Vektoren werden in eckigen Klammern `[]` deklariert.
- ▶ Spalten werden durch Kommas getrennt, Zeilen durch Semikolons.
- ▶ `a = [1, 2, 3, 4]`; erzeugt einen Zeilenvektor der Dimension 4.
- ▶ `b = [1 2 3 4]`; ist identisch zu `a`.
- ▶ `c = [1;2;3]`; erzeugt einen Spaltenvektor der Dimension 3.
- ▶ `d = 1:0.5:3`; erzeugt den Vektor `[1 1.5 2 2.5 3]`.
- ▶ `e = []`; erzeugt einen *leeren Vektor*.

Vektoren: Zugriff

- ▶ $a = [1 \ 2 \ 5 \ 9 \ 3 \ 7]$; erzeugt Zeilenvektor der Dimension 6.
- ▶ Elementweiser Zugriff:
 - $g = a(1)$; erzeugt Skalar mit dem Wert $a_1 = 1$.
 - $i = 3$; $g = a(i)$; erzeugt Skalar mit dem Wert $a_i = a_3 = 5$.
 - $g = a(\text{end})$; erzeugt Skalar mit dem letzten Wert von a , d. h. $g = 7$.
- ▶ Bereichswahl:
 - $g = a(1:3)$; enthält die ersten drei Werte von a .
 - $v = [1 \ 2 \ 4]$; $g = a(v)$; \Rightarrow Was ist g ?
- ▶ **Im Unterschied zu anderen Programmiersprachen** beginnt in Matlab die Nummerierung mit 1, nicht mit 0.

Vektoren: Operationen

Wichtig: Indizes können nur positive natürliche Zahlen sein!

- ▶ Transponieren: Ist c ein Spaltenvektor, so ist $d = c'$ ein Zeilenvektor und vice versa.
- ▶ Länge eines Vektors ermitteln, z. B. für Schleifen:

```
n = length(a);  
for i = 1:n  
    disp(a(i));  
end
```

Vektoren

Aufgabe 10

Schreiben Sie eine Funktion `function [c] = add(a,b)`, welche zwei Vektoren `a` und `b` addiert. Die Addition soll mittels einer `for`-Schleife elementweise realisiert werden. Dazu soll vorher die Länge der Vektoren `a` und `b` ermittelt werden.

Vektoren: Operationen

Operatoren wirken auf Vektoren möglicherweise anders als auf Skalare.

- ▶ Zuweisungsoperator: =
 $a = b: \quad a(i) = b(i)$ für alle i .
- ▶ Arithmetische Operatoren: + -
 $a + b: \quad a(i) + b(i)$ für alle i .

Vorsicht!

Matlab addiert auch Vektoren und Skalare sowie Vektoren unterschiedlicher Dimensionen, insbesondere auch Zeilen- und Spaltenvektoren selber Länge. Dies kann zu unerwarteten Ergebnissen führen.

- ▶ Arithmetische Operatoren: *
 $a * b$ funktioniert nur dann, wenn a ein Zeilen- und b ein Spaltenvektor ist (Skalarprodukt) oder umgekehrt (dyadisches Produkt).

Vektoren: Operationen

- ▶ Relationale Operatoren werden elementweise ausgeführt: `a == b` liefert nicht `true` oder `false`, sondern einen Vektor, deren Einträge `true` oder `false` sind:

```
a = [1 2 3 4];
```

```
b = [1 9 3 4];
```

```
c = (a == b); % c = [1 0 1 1]
```

- ▶ Entsprechendes gilt für `>`, `>=`, usw.
- ▶ `if`-Abfragen werden nur dann durchgeführt, wenn alle Werte `true` sind:

```
if a == b disp(a); end
```

liefert keine Ausgabe.

Matrizen: Erzeugung

- ▶ `A = [1 2 3 4; 5 6 7 8]; % 2 x 4 - Matrix`

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

- ▶ `B = [1 5; 2 6; 3 7; 4 8]'; % identisch zu A`
- ▶ `B(2,3) = 15; % ändert Element von B`

$$B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 15 & 8 \end{pmatrix}$$

Matrizen: Erzeugung

- ▶ `I = eye(n); % n x n - Einheitsmatrix`
- ▶ `A = zeros(n,m); % n x m - Matrix mit Nullen`
- ▶ `B = ones(n,m); % n x m - Matrix mit Einsen`
- ▶ `v = [1 2 3]; D = diag(v);` erzeugt die Matrix

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

- ▶ `M = diag(v,1);` erzeugt die Matrix

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Matrizen: Zugriff

- ▶ Als Ganzes: $B = A;$
- ▶ Elementweise:
 $b = A(3,1);$
 $i = 2; j = 3; g = A(i,j);$
 $g = A(\text{end},1);$ g ist Skalar und enthält das erste Element aus der letzten Zeile von A .
- ▶ Bereichswahl:
 $B = A(2:4,2:4);$ ist (3×3) -Untermatrix

$$B = \begin{pmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Für Vektoren v, w mit positiven ganzen Zahlen liefert $A(v, w)$ die Matrix mit den Einträgen a_{v_i, w_j} (fortgeschritten).

Matrizen: Operationen

- ▶ Transponieren:

`B = A'; % B ist Transponierte von A`

- ▶ Größe ermitteln:

`[n m] = size(A); % n Zeilen, m Spalten`
(gemäß der Notation $A \in \mathbb{R}^{n \times m}$)

Aufgabe 11

Schreiben Sie eine Funktion `function [C] = addMat(A,B)`, welche zwei Matrizen A und B addiert. Dazu soll zunächst die Größe von A und B ermittelt werden und die Addition soll in zwei ineinander geschachtelten for-Schleifen realisiert werden.

Matrizen: Rechenoperationen

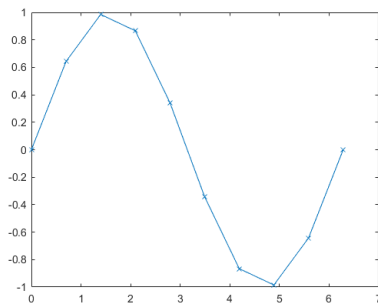
- ▶ $A \pm B$ berechnet die Matrix mit den Einträgen $A(i, j) \pm B(i, j)$.
- ▶ $A * B$ hat drei Bedeutungen:
 1. Sind A und B Matrizen, so entspricht $A*B$ der üblichen Matrixmultiplikation (sofern die Dimensionen passen).
 2. Ist A eine $(m \times n)$ -Matrix und B ein $(n \times 1)$ -Spaltenvektor, dann ist $A*B$ eine Matrix-Vektor-Multiplikation (fällt eigentlich unter (1)).
 3. Ist A eine Matrix und B ein Skalar, dann ist $A * B$ die Matrix mit den Einträgen $A(i, j)*B$.

2d Plots

- ▶ Matlab plottet nicht direkt Funktionen, sondern interpoliert Punktmengen $(x(i), y(i))_{i=1}^N$.

```
x = linspace(0,2*pi,10); y = sin(x);
```

```
plot(x,y); % Plotte Vektoren der Dimension 10
```



Vorgehen beim Plotten

Gegeben sei eine zu plottende Funktion $f : [a, b] \rightarrow \mathbb{R}$.

Vorgehensweise:

1. Erstelle Vektor x mit *Stützstellen* (Auswertungspunkte). Bsp:

```
x = -1:0.1:1;
```

```
% Intervall [-1,1], Abstand 0.1 zwischen zwei  
Punkten
```

```
x2 = linspace(-1,1,21); % entspricht x
```

2. Werte die Funktion in allen Stützstellen aus, beispielsweise mit einer Schleife.

```
for i=1:length(x) y(i)=f(x(i)); end
```

3. Die Punktmenge wird an Matlab zum Plotten übergeben

```
plot(x,y, 'Marker', 'x')
```

4. Nach Wunsch können an dem Plot noch verschiedene Einstellungen vorgenommen werden.

Plotten: Beispiel

```
1 % Plotte sin(x) fuer x in [0,2pi]
2
3 % Wertetabelle erstellen
4
5 % Vektor mit 50 aequidistanten Stuetzstellen erzeugen
6 x = linspace(0,2*pi,50);
7 y = zeros(50,1); % Initialisierung
8
9 % Berechne sin(x(i)) fuer alle x(i)
10 for i=1:length(x)
11 y(i) = sin(x(i));
12 end
13
14 % Plotten mit roter Farbe
15 % Die Punkte (x(i),y(i)) werden mit einem Kreis
markiert
16 plot(x,y,'r','Marker','o');
```


Plotten: Beispiel

```
18 % Ueberschrift und Achsenbeschriftungen!  Matlab  
kann auch LaTeX  
19 title('Mein erster plot');  
20 xlabel('0 \leq x \leq 2 \pi');  
21 ylabel('sin(x)');  
22  
23 % Lege Achsen fest:  [xmin,xmax,ymin,ymax]  
24 axis([0,2*pi,-1.1,1.1]);  
25  
26 % Speichere Plot als png-Datei  
27 % Erstes Argument:  Dateiformat  
28 % Zweites Argument:  Aufloesung  
29 % Letztes Argument:  Dateiname  
30 print('-dpng', '-r100', 'sin_plot.png');
```

Mehrere Plots in einem Schaubild

- ▶ Eine Möglichkeit:

```
% x1, y1 Vektoren der Dimension n
```

```
% x2, y2 Vektoren der Dimension m
```

```
plot(x1,y1);
```

```
hold on; % Garantiert, dass nochmal in dieselbe  
Figur geplottet wird.
```

```
plot(x2,y2);
```

```
legend('Erste Linie', 'Zweite Linie');
```

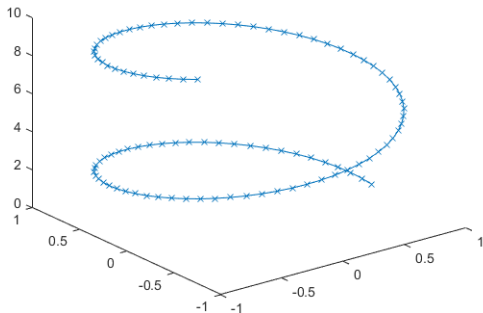
- ▶ Andere Möglichkeit: `plot(x1,x2,y1,y2)`

3D-Plot einer Kurve

- ▶ 3D-Plot einer Kurve $\gamma : \mathbb{R} \supset I \rightarrow \mathbb{R}^3$

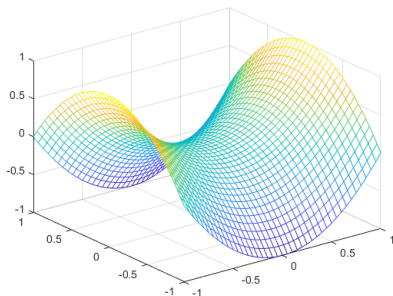
```
plot3(x,y,z);
```

```
% x, y, z Vektoren der Dimension n
```



3D-Plot eines Graphen

- ▶ 3D-Plot einer Funktion $f : [a_1, b_1] \times [a_2, b_2] \rightarrow \mathbb{R}$
- ▶ Geplottet wird die Punktmenge $(x_i, y_j, f(x_i, y_j))$
 - 1: `a = -1:0.05:1; % Diskretisierung von [a1,b1]`
 - 2: `b = -1:0.05:1; % Diskretisierung von [a2,b2]`
 - 3: `[x,y] = meshgrid(a,b) % Generiere Gitterpunkte`
 - 4: `z = x.^2 - y.^2;`
 - 5: `mesh(x,y,z);`



Aufbau eines (kleinen) Programms

- ▶ Wir haben gelernt, wie man mit Skripten und Funktionen umgeht.
- ▶ Ein Programm besteht typischerweise aus einem Skript und mehreren Funktionen und lässt sich in drei Blöcke gliedern:
 1. **Preprocessing:** Hier werden Parameter eingegeben und Einstellungen vorgenommen. Dieser Block gehört in das Skript.
 2. **Processing:** Code, in dem der Algorithmus durchgeführt wird. Dazu werden im Skript typischerweise eine oder mehrere Funktionen aufgerufen.
 3. **Postprocessing:** Ausgabe, Speicherung, Visualisierung von Ergebnissen.

Aufbau eines (kleinen) Programms

- ▶ Es ist eine Frage des Stils, wie viele und wie große Funktionen verwendet werden.
Faustregel: Eine Funktion sollte auf einen Bildschirm passen.
- ▶ In einem fertigen Programm sollte der Benutzer nur noch die Input-Parameter im Skript eingeben und dieses ausführen.
Das Processing und Postprocessing sollten dann vollständig automatisch ablaufen.

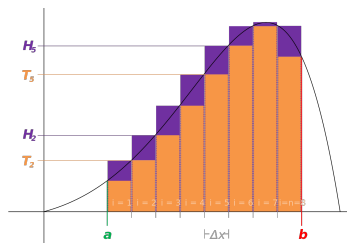
Einschub: Function Handles

- ▶ Man kann mithilfe des Codes
$$f = @(x) (x^2+1)$$
eine Funktion (genauer: ein *function handle* definieren, wobei in der letzten Klammer der Funktionsausdruck steht).
- ▶ Diese Funktion kann, nachdem sie definiert wurde, genauso aufgerufen werden, wie die Funktionen, die als Dateien gespeichert wurden.
- ▶ Nachdem der Workspace gelöscht oder Matlab beendet wurde ist ein function handle nicht mehr verfügbar.
- ▶ Vorteil: Function handles können wie Variablen an andere Funktionen übergeben und dort verwendet werden, sofern die Syntax dort für eine Funktion ausgelegt ist.

Numerische Integration

Das Integral $\int_a^b f(x)dx$ einer Funktion ist per Definition der Grenzwert von Riemann-Summen.

Man kann also ein Integral näherungsweise berechnen, indem man Riemann-Summen berechnet.



Quelle: https://de.wikipedia.org/wiki/Riemannsches_Integral

Numerische Integration

Vorgehensweise:

- ▶ Zerlege das Intervall $[a, b]$ in N Teilintervalle mit den Grenzen

$$a < a + h < a + 2h < \dots < a + Nh = b,$$

d. h. $h = \frac{b-a}{N}$.

- ▶ Es gilt

$$\int_a^b f(x) dx = \sum_{k=0}^{N-1} \int_{a+kh}^{a+(k+1)h} f(x) dx.$$

Numerische Integration

Die Idee ist nun, dass auf einem *kleinen* Intervall $[c, c + h]$

$$\int_c^{c+h} f(x) dx \approx hf(c).$$

Daraus ergibt sich die (linke) **Rechtecksregel**

$$I := \int_a^b f(x) dx \approx \sum_{k=0}^{N-1} hf(a + kh) =: I(h).$$

Der von h abhängige Wert $I(h)$ ist eine *Approximation* an den exakten Wert I des Integrals.

Numerische Integration

Numerische Aspekte:

- ▶ Konvergiert der approximierte Wert für $h \rightarrow 0$ auch wirklich gegen den exakten Wert, also gilt $\lim_{h \rightarrow 0} I(h) = I$?
- ▶ Konvergiert der approximierte Wert für jede beliebige (integrierbare) Funktion?
- ▶ Falls ja, wie *schnell* konvergiert $I(h)$ gegen I , d. h. lässt sich etwas über $|I(h) - I|$ aussagen³?

³Spoiler: Es gibt eine Konstante c , so dass $|I(h) - I| \leq ch$ für jede stetig differenzierbare Funktion. ▶

Numerische Integration: Planung des Programms

- ▶ Was sind die Inputparameter? Die Funktion f , die Intervallgrenzen a, b und die Anzahl N der Teilintervalle. \Rightarrow Preprocessing.
- ▶ Was muss der Algorithmus berechnen? Funktionsauswertungen $f(a + kh)$ für $k = 0, \dots, N - 1$ und die Summe $\sum hf(a + kh)$. \Rightarrow Processing.
- ▶ Welche Ergebnisse soll der Algorithmus liefern und wie sollen sie visualisiert werden? Hier nur das approximierte Ergebnis $I(h)$. \Rightarrow Postprocessing.

Numerische Integration: Pseudocode

1. Input: f , a , b , N .
2. Berechne die Funktionswerte $f_k = f(a + kh)$ für $k = 0, \dots, N - 1$.
3. Berechne die Produkte hf_k .
4. Bilde die Summe $I(h) = \sum_{k=0}^{N-1} hf_k$.
5. Gebe das Ergebnis $I(h)$ aus.

Fehlersuche

- ▶ Hat man einen Code fertiggestellt, wird dieser in den meisten Fällen nicht auf Anhieb funktionieren.
- ▶ Es ist normal, einen großen Teil der Zeit mit der Fehlersuche (Debugging) zu verbringen.
- ▶ Es gibt verschiedene Arten von Fehlern:
 - ▶ Syntax-Fehler, z. B. $A = * A$: Leicht zu beheben, da Matlab euch sagt, was falsch ist.
 - ▶ Logische Fehler: Schwer zu finden, da das Programm aus Sicht des Computers fehlerfrei ist, aber nicht das richtige Ergebnis liefert.

Typische Fehlerquellen

- ▶ Falsche Syntax, z. B. $y = 2x+1$;
- ▶ Unvollständiges Programm, z. B. wenn man sich nicht genug Gedanken über negative Zahlen, falsche Dimensionen, Zeilen- und Spaltenvektoren etc. macht.
- ▶ Unerwartete Argumente für Funktionen, z. B. wenn man einer Funktion, die Skalare erwartet, Vektoren übergibt.
- ▶ Unerwarteter Zustand von Daten, z. B. eine Variable, die später noch gebraucht wird, wird aus Versehen überschrieben.
- ▶ Logische Fehler, z. B. falsche Bedingungen in if-Abfragen.

Fehlersuche

Ein paar Tipps:

- ▶ Bleiben Sie ruhig: Fehler beim Programmieren sind absolut normal.
- ▶ Systematik: Gehen Sie systematisch vor!
- ▶ Fehlermeldungen: Lesen Sie die Fehlermeldungen! Zumindest die Zeilenangaben sind verständlich. Viele Fehler lassen sich dadurch leicht beheben.
- ▶ Fehlereinschätzung: Ist es eher ein einfacher Syntax-Fehler?
- ▶ Fehlereingrenzung: Isolieren Sie den Fehler:
 - ▶ Kommentieren Sie Teile des Codes aus, die nicht unbedingt nötig sind. Taucht der Fehler immernoch auf?
 - ▶ Hat der Code früher bereits funktioniert? Was genau haben Sie verändert?
- ▶ **Fragen Sie andere!** Das ist meistens die schnellste Methode.

Performance

- ▶ Performance ist das *Zeitverhalten* eines Programms.
- ▶ Je schneller ein Programm läuft und je weniger Speicher es benötigt, desto besser.
- ▶ Hier: Performance ist wichtig, wenn es in der Programmieraufgabe ausdrücklich verlangt ist.
- ▶ Performance kann leicht mit `tic` und `toc` gemessen werden (zurückgegeben wird die Zeit, die zwischen `tic` und `toc` vergangen ist), oder mit dem Matlab-Profiler (fortgeschritten, wird nicht besprochen).

Performance und Effizienz

- ▶ Es gibt keine einheitliche Definition, wann ein Programm *effizient* genannt wird.
- ▶ Häufig hängt die Laufzeit bzw. die Anzahl der nötigen Rechenoperationen von einer Größe N ab:
 - ▶ (Quadratische) Gleichungssysteme mit $(N \times N)$ -Matrizen,
 - ▶ Numerische Integration mit N Teilintervallen,
 - ▶ Berechnung der ersten N Folgenglieder einer (rekursiven) Folge.
- ▶ Das Ziel ist, dass die von N abhängige Laufzeit des Programms nicht zu schnell mit N wächst.

Performance und Effizienz

- ▶ Manchmal nennt man einen Algorithmus *effizient*, falls er eine polynomiale Laufzeit hat, also falls die Laufzeit in etwa proportional zu cN^k mit einer Konstante c und einem Exponenten k .
- ▶ Beispiel (später in Numerik): Aufwand zur Berechnung LR-Zerlegung (= Gauß-Elimination) einer $(N \times N)$ -Matrix $\approx \frac{1}{3}N^3$.
- ▶ Obige Definition unbrauchbar, falls k groß, sie schließt aber Algorithmen, deren Laufzeit exponentiell wächst, aus.

Beispiel: Fibonacci-Folge

Die Fibonacci-Zahlenfolge ist definiert durch

$$f_{n+2} = f_{n+1} + f_n, \quad f_1 = f_2 = 1.$$

Man kann leicht eine *rekursive Funktion* schreiben:

```
1 function [fn] = fib_recursive(n)
2 % Berechnet das n-te Element der Fibonacci-Folge
  rekursiv
3 if n <= 2
4   fn = 1;
5 else
6   fn = fib_recursive(n-1)+fib_recursive(n-2);
7 end
```

Beispiel: Fibonacci-Folge

- ▶ Ruft man obige Funktion mit $n = 3$ auf, so wird der if-Block übersprungen und es wird Zeile 6 ausgeführt.
- ▶ Die Funktion ruft sich selbst zweimal auf mit $n = 2$ und $n = 1$.
- ▶ In diesen Aufrufen wird $f_n = 1$ zurückgegeben.
- ▶ Im ursprünglichen Funktionsaufruf wird dann mit den zurückgegebenen Werten $f_n = 1+1$ berechnet.

Beispiel: Fibonacci-Folge

Obiges Programm benötigt bereits ca. eine Minute, um f_{45} auszurechnen. Denken Sie, dass die Fibonacci-Folge kompliziert genug ist, um das zu rechtfertigen?

Woran könnte es liegen, dass dieses Programm so langsam ist?

Hinweis: Überlegen Sie sich, wie oft obige Funktion mit einem $n \in 1, 2$ aufgerufen wird, wenn sie f_4 , f_5 , f_6 etc. berechnen wollen!

⇒ Mehr dazu im Präsenzteil!

Beispiel: Fibonacci-Folge

Erklärung: Nur für $n = 1$ und $n = 2$ kann die Funktion direkt ein Ergebnis liefern, anderenfalls muss sie sich selbst aufrufen. So wird beispielsweise f_6 folgendermaßen berechnet:

$$\begin{aligned}f_6 &= f_5 + f_4 \\&= (f_4 + f_3) + (f_3 + f_2) \\&= ((f_3 + f_2) + (f_2 + f_1)) + ((f_2 + f_1) + 1) \\&= (((f_2 + f_1) + 1) + (1 + 1)) + ((1 + 1) + 1) \\&= (((1 + 1) + 1) + 2) + (2 + 1) \\&= ((2 + 1) + 2) + 3 \\&= (3 + 2) + 3 = 5 + 3 = 8\end{aligned}$$

Dies führt zu einer unnötig hohen Anzahl an Funktionsaufrufen und Folgenglieder werden mehrfach berechnet (z. B. wird f_3 dreimal berechnet).

⇒ Ein iteratives Programm ist effizienter (Übung).

Performance und Effizienz

- ▶ Die Performance eines Programms sollte i. d. R. erst optimiert werden, wenn das Programm bereits läuft.
- ▶ Matlab bietet eine Funktion `profile`, welche genau misst, welche Funktion und welche Code-Zeilen wie viel Zeit in Anspruch nehmen.
- ▶ Für Numerik 1 eher irrelevant, wichtig bei größeren Programmen.

Was man mitnehmen sollte

- ▶ Matlab arbeitet mit Matrizen (Zahlen sind (1×1) -Matrizen).
- ▶ Unterschied zwischen Skript und Funktion.
- ▶ Bei einem = wird der Ausdruck rechts ausgewertet und in die Variable links gespeichert.
- ▶ Schleifen (for und while)
- ▶ **Nicht hilfreich:** Alle Befehle auswendig kennen.
Stattdessen: Wissen, welche Möglichkeiten bestehen und bei Bedarf Google benutzen!

Fragen?