# Programmierkurs

Patrick Dondl

# Allgemeine Infos

- Dozent: Patrick Dondl (patrick.dondl@mathematik.uni-freiburg.de)
- Assistent: Keith Anguige (keith.anguige@mathematik.uni-freiburg.de)
- Website zur Vorlesung: https://aam.uni-freiburg.de/agdo/lehre/ss17/prog/

# Allgemeine Infos (cont.)

- Etwa die Hälfte des Kurses beschäftigt sich mit MATLAB, die andere Hälfte mit C++ (bzw. C).
- Es gibt zum Abschluss einen Überblick über andere populäre Programmiersprachen, z.B. Python
- Insgesamt werden wir hier sehr anwendungsbezogen arbeiten
- **WICHTIG:** Besorgen Sie sich einen MATLAB-Account und eine Lizenz (kostenlos für eingeschriebene Studierende, Infos dazu auf der Kurswebsite, Fragen bitte ans Rechenzentrum)

# Infos vom ZfS

- Anwesenheit: Anwesenheitspflicht im Sinne einer regelmäßigen Teilnahme (bei den **Tutoraten**)

- Fehlzeit: maximal 20% der Präsenzzeit, also maximal **zwei mal**

- Überschreiten der maximal möglichen Fehlzeit:
  - nachweisliche Kollision mit einer Pflichtveranstaltung oder Krankheit: keine Teilnahme mehr möglich, keine Sperrung, im Krankheitsfall bitte dem ZfS ärztliches Attest vorlegen
  - sonst: Sperrung für den jeweiligen Kompetenzbereich (wirksam ab nächster Belegphase)

# Infos vom ZfS (cont.)

Leistungsanforderungen:

- Arbeitsaufwand: 1 ECTS = 30 Arbeitsstunden (4 ECTS = 120 Arbeitsstunden)
- ECTS-Punkte werden ganz oder gar nicht vergeben (d.h. nicht für Teile der Leistung oder bloße Anwesenheit)
- Leistungsanforderungen: 50% der Übungspunkte sowie Bestehen der Prüfung
- keine Unterscheidung zwischen Bachelor- und Nicht-Bachelor-Studierenden in den Leistungsanforderungen
- falls die geforderten Leistungen nicht erfolgreich oder fristgemäß erbracht werden, kann das zur Sperrung der/des Studierenden im jeweiligen Kompetenzbereich (wirksam ab Folgesemester) führen. Nähere Informationen unter: www.zfs.uni-freiburg.de

# Infos vom ZfS (cont.)

Teilnahmebescheinigung:

- Nachweis der Studienleistung über Ihre Online-Leistungsübersicht; seit dem SoSe2015 sehen Jurastudierende ihre Notenpunkte nach erfolgreicher Teilnahme direkt in ihrer Leistungsübersicht. Es muss kein Leistungsnachweis mehr beim ZfS abgeholt werden.

- Nachweise als ‘Schein’ in Papierform für EUCOR- und ERASMUS-Studierende

- Ausgabe von Bescheinigungen:
  - Mo bis Do 09:00 – 12:00 in Raum 01 004, Universitätsstraße 9, 1. OG Allgemeine Sprechstunde (keine Scheinausgabe):
  - Do 15:00 – 16:00 Uhr in Raum 01 006, Universitätsstraße 9

# Infos vom ZfS (cont.)

- Übungsgruppen: Bitte in HISinOne anmelden, Kursnummer 5105T
- Anmeldung zum Kurs: Ab sofort bitte per Email an `zfs-info@zfs.uni-freiburg.de`
- Abmeldung: Nur mit triftigem Grund, Email an abmeldung@zfs.uni-freiburg.de
- Evaluierung online

# MATLAB

- MATLAB ist ein großes Programmpaket mit einer Vielzahl von Anwendungsmöglichkeiten
- Standardwerkzeug in Forschung und Industrie
- Viele sog. Toolboxen (beispielsweise Statistik, Optimierung, Partielle Differentialgleichungen, …) mit speziell auf bestimmte Anwendungen angepassten Funktionalitäten
- Mächtige Programmiersprache, kann aber auch als eine Art ‘Taschenrechner’ verwendet werden.

# Freie Alternative

- MATLAB wird von Mathworks hergestellt und ist ein kommerzielles Programmpaket
- Octave ist eine (fast) vollständig kompatible, freie (Open Source) und kostenlose Alternative
- Installationspakete von Octave gibt es für Windows, Installation auf Linux und Mac (vermutlich) am besten über Package-Manager
- Support von unserer Seite etwas eingeschränkt

# Weiterführende Literatur

- Desmond J. Higham: Matlab Guide
- Cleve B. Moler: Numerical Computing with Matlab
- Wolfgang Schweizer: Matlab kompakt

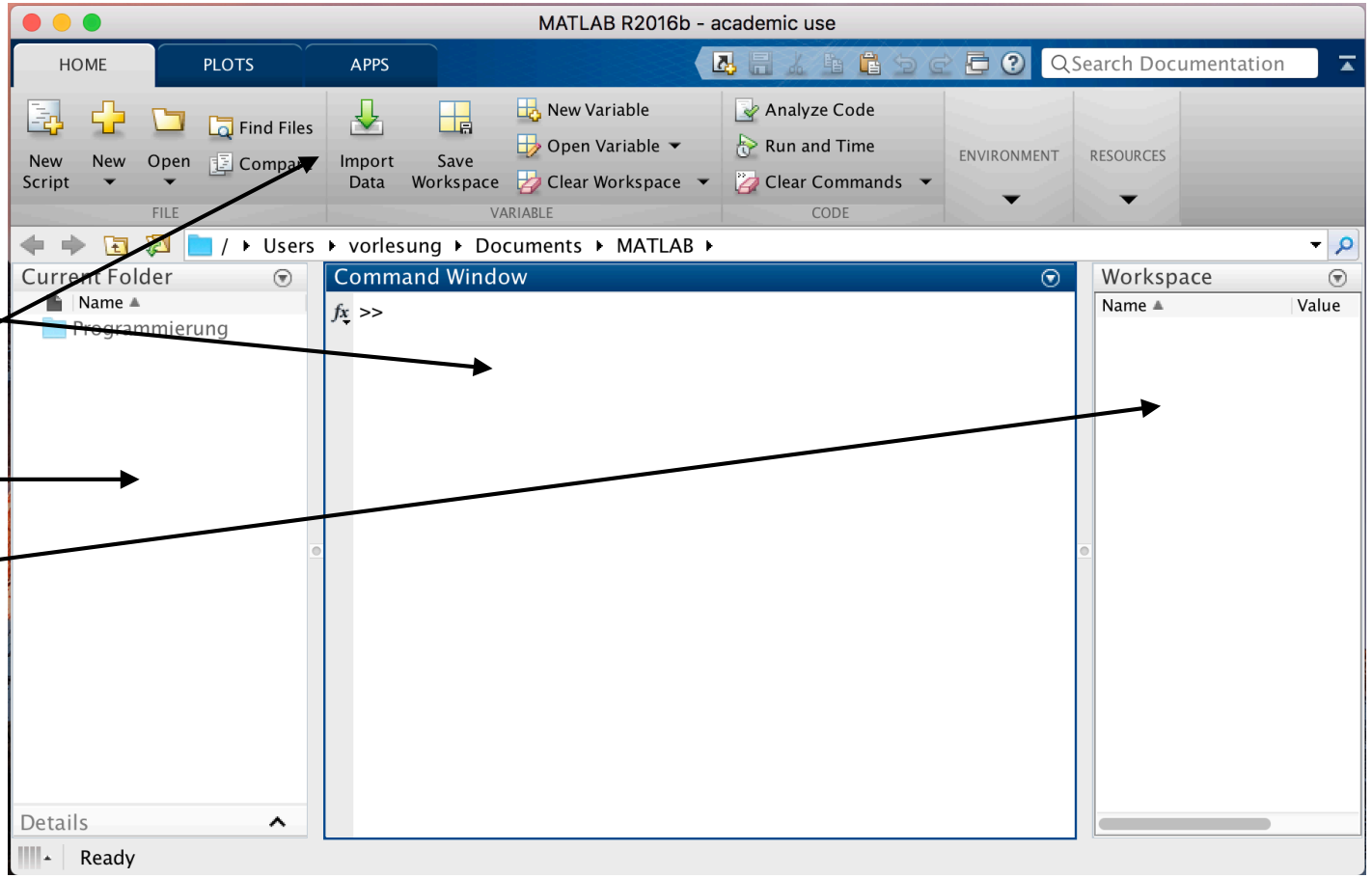
# Nutzung von MATLAB

Es gibt mehrere Möglichkeiten für Sie, MATLAB zu nutzen

- Poolraum im 2. Stock (Sie benötigen ein Login, MATLAB ist installiert, keine weitere Lizenz nötig)
- MATLAB-Installation auf Ihrem Rechner (Lizenz für Studierende kostenlos, siehe Kurswebpage)
- MATLAB im Webbrowser (Login mit Mathworks-Account, verlinkt mit Studierendenlizenz) https://de.mathworks.com/products/matlab-online.html

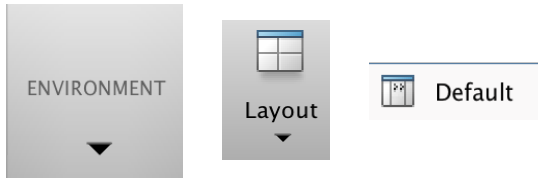
# MATLAB-Desktop

Kommandozeile
Menüoptionen
Dateiverzeichnis
Definierte Variablen



# MATLAB-Desktop (cont.)

- In der Kommandozeile können direkt Befehle eingegeben werden, diese werden nach Eingabe von ‘Enter’ sofort ausgeführt.
- Im Dateiverzeichnis werden die Dateien im aktuellen Verzeichnis angezeigt (nützlich später)
- Im Workspace werden aktuell definierte Variablen angezeigt (dazu gleich mehr)
- In der Toolbar können z.B. neue Dateien erzeugt werden (dazu auch später mehr) oder (besonders nützlich): mit einem Druck auf ‘Layout’ (manchmal versteckt unter ‘Environment’ und dann ‘Default’ kann der Standarddesktop wiederhergestellt werden



# Die Kommandozeile

## MATLAB als Taschenrechner

```
>>
>> 4*3
ans =
     12
>> a = 3
a =
      3
>> a+2.1
ans =
      5.1
```

Variablenzuweisung
ACHTUNG! Die Bedeutung ist nicht zu verwechseln mit dem mathematischen ‘=‘

# Variablen

- Mit dem Kommando ‘`a = 3`’ wird ein Stück im Computerspeicher reserviert, welches später mit dem Namen ‘a’ wieder referenziert werden kann. Der Wert wird auf ‘`3`’ gesetzt.
- Die Namen unterscheiden Groß- und Kleinschreibung, es gibt auch bestimmte Einschränkungen (darf beispielsweise nicht mit einer Zahl beginnen, keine Lehrzeichen)
- Das ist nicht zu verwechseln mit dem mathematischen ‘=’
- Beispiel:

```
>> a = 3
a =
    3
>> a = a+7.2
a =
    10.2000
```

# Variablen (cont.)

Variablen in MATLAB können eine ganze Reihe verschiedener Datentypen referenzieren, beispielsweise:

- ganze Zahlen
- Fließkommazahlen (Computerapproximation von reelen Zahlen)
- Vektoren
- Matrizen
- Zeichenketten (‘Strings’, brauchen wir eigentlich hier selten — werden durch Hochkommas begrenzt: `s = 's ist jetzt ein String'`)
- …..

Generell ist in MATLAB alles eine Matrix (MATLAB steht für MATrix LABoratory), auch einfache Zahlen werden als 1×1-Matrix gespeichert. Das ist erstaunlich nützlich, aber ab und an etwas verwirrend.

Im Gegensatz zu vielen anderen Programmiersprachen ist MATLAB nicht sehr streng, was Variablentypen angeht, zum Beispiel wird nur wenn nötig zwischen ganzen und Fließkommazahlen unterschieden.

# Zahlen in MATLAB

- Intern gespeichert werden (fast immer) 16 Nachkommastellen, ausgegeben üblicherweise nur 5. Mit einer Eingabe von '`format long`' kann das geändert werden. Ein '`format short`' macht das rückgängig.

```
>> format long
>> a
a =

   10.199999999999999
```

# Zahlen (cont.)

- Es gibt auch komplexe Zahlen in MATLAB

```
>> a = a + 2*i
a =

    10.2000 + 2.0000i
```

- Achtung: Man kann auch einer Variablen den Namen `i` geben. Das führt u.U. zu seltsamen Verhalten…

# Zahlen (cont.)

- MATLAB benutzt den IEEE-Standard für Fließkommazahlen, d.h. es werden ein Faktor und eine Zehnerpotenz gespeichert. Üblicherweise ist das ein ‘double’ mit insgesamt 64bit Speicher.

- Die Zahlen können in Exponentialschreibweise aus- und eingegeben werden

```
>> b = 1.23e-14
b =
    1.2300e-14
```

$1.23 \cdot 10^{-14}$

# Vektoren und Matrizen

```
>> v = [1; 2; 3]
v =
     1
     2
     3
>> A = [1.1 2.3 4.5; 8 7 2; 4.1 1.1 0];
>> A*v

ans =

   19.2000
   28.0000
    6.3000
```

Ein Semikolon am Ende unterdrückt die Ausgabe

# Vektoren und Matrizen (cont.)

- Man kann auf Einträge in einem Vektor oder einer Matrix separat zugreifen

```
>> A(3,2)
ans =

    1.1000
>> v(2,3) = 7.0;
>> size(A)
ans =

    3   3
```


- Es gibt noch eine große Menge sehr nützlicher Matrixindizierungsmethoden
- Wie gesagt, eigentlich ist in MATLAB alles eine Matrix (und wird auch so behandelt). Ein Skalar ist eine 1×1-Matrix, ein Spaltenvektor eine k×1-Matrix, ein Zeilenvektor eine 1×k-Matrix.

# Rechenoperationen

Der Additionsoperator ‘+’ kann folgendes bedeuten:

- Addition von zwei Skalaren
- Addition von Matrizen oder Vektoren (Achtung: Größen müssen natürlich übereinstimmen, sonst gibt es eine Fehlermeldung)
- Addition von Matrix und Skalar (???) — zu jedem Eintrag der Matrix wird der gegebene Skalar hinzuaddiert. Das ist manchmal nützlich, kann aber natürlich verwirrend sein. Man kann (leider?) beispielsweise auch eine 3x3-Matrix und einen 3-Vektor addieren, dann wird zu jeder Spalte der Matrix der Vektor addiert.

# Rechenoperationen (cont.)

Der Subtraktionsoperator ‘–’: Verhält sich wie ‘+’

# Rechenoperationen (cont.)

## Multiplikation ‘*’

- Skalar mit Skalar: wie üblich
- Skalar mit Matrix: Standard-Skalarmultiplikation
- Matrix mit Vektor: Matrix-Vektorprodukt
- Matrix mit Matrix: Matrixprodukt
- Will man *eintragweise* multiplizieren kann man ‘`.*`’ benutzen

# Rechenoperationen (cont.)

## Division ‘/’

- Division von Skalar, Vektor oder Matrix durch einen Skalar: wie üblich
- Eintragweise Division von Vektoren und Matrizen: wieder mit ‘**./**’
- Man kann auch durch eine Matrix *A* dividieren, das ist gleichbedeutend mit der Multiplikation mit $A^{-1}$ von rechts, Größen müssen natürlich zusammenpassen.
- Es gibt auch den (berühmten) MATLAB Backslash Operator ‘\’, zu diesem später mehr.

# Rechenoperationen (cont.)

## Potenzierung ‘^’

- Für Skalare klar
- Für Matrizen: `A^k = A*A*…*A` (k-mal) bei ganzzahligem `k`. Nichtganzzahliges `k`: nunja…
- Eintragweise wieder mit ‘`.^`’

# Rechenoperationen (cont.)

## Weitere Matrixoperationen…

- Transposition ‘`.'`’
- Komplex-konjugiert und Transponiert ‘`'`’
- Es gilt ‘Punkt vor Strich’, man kann natürlich (runde) Klammen setzen. Eckige Klammern sind zur Matrixkonstruktion gedacht
- Es gibt noch eine ganze Reihe weiterer Operationen, die MATLAB-Hilfe ist hier sehr nützlich.

# Hilfe!

```
>> help exp
 exp    Exponential.
    exp(X) is the exponential of the elements of X, e to the X.
    For complex Z=X+i*Y, exp(Z) = exp(X)*(COS(Y)+i*SIN(Y)).

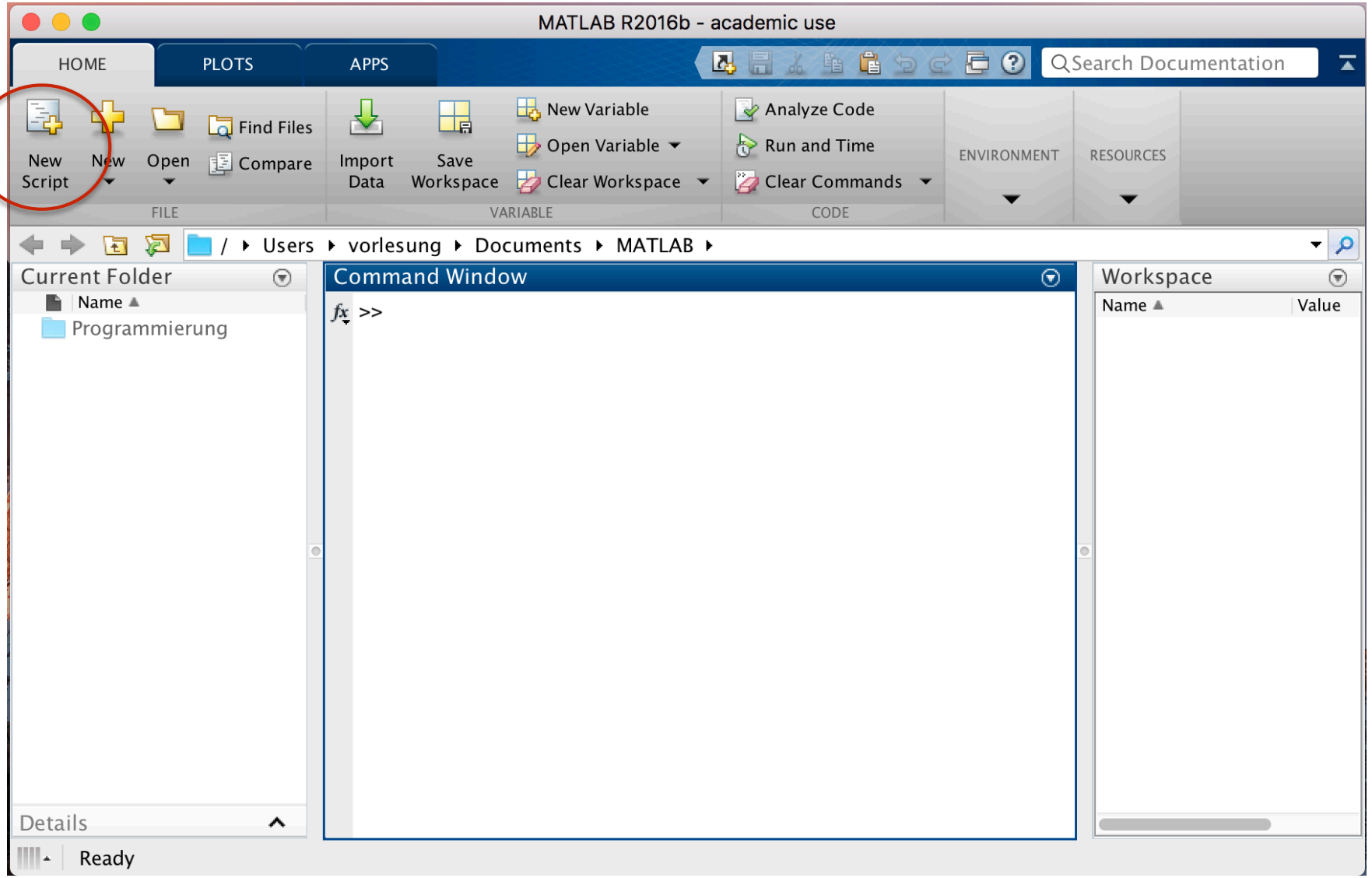
    See also expm1, log, log10, expm, expint.

    Reference page for exp
    Other functions named exp

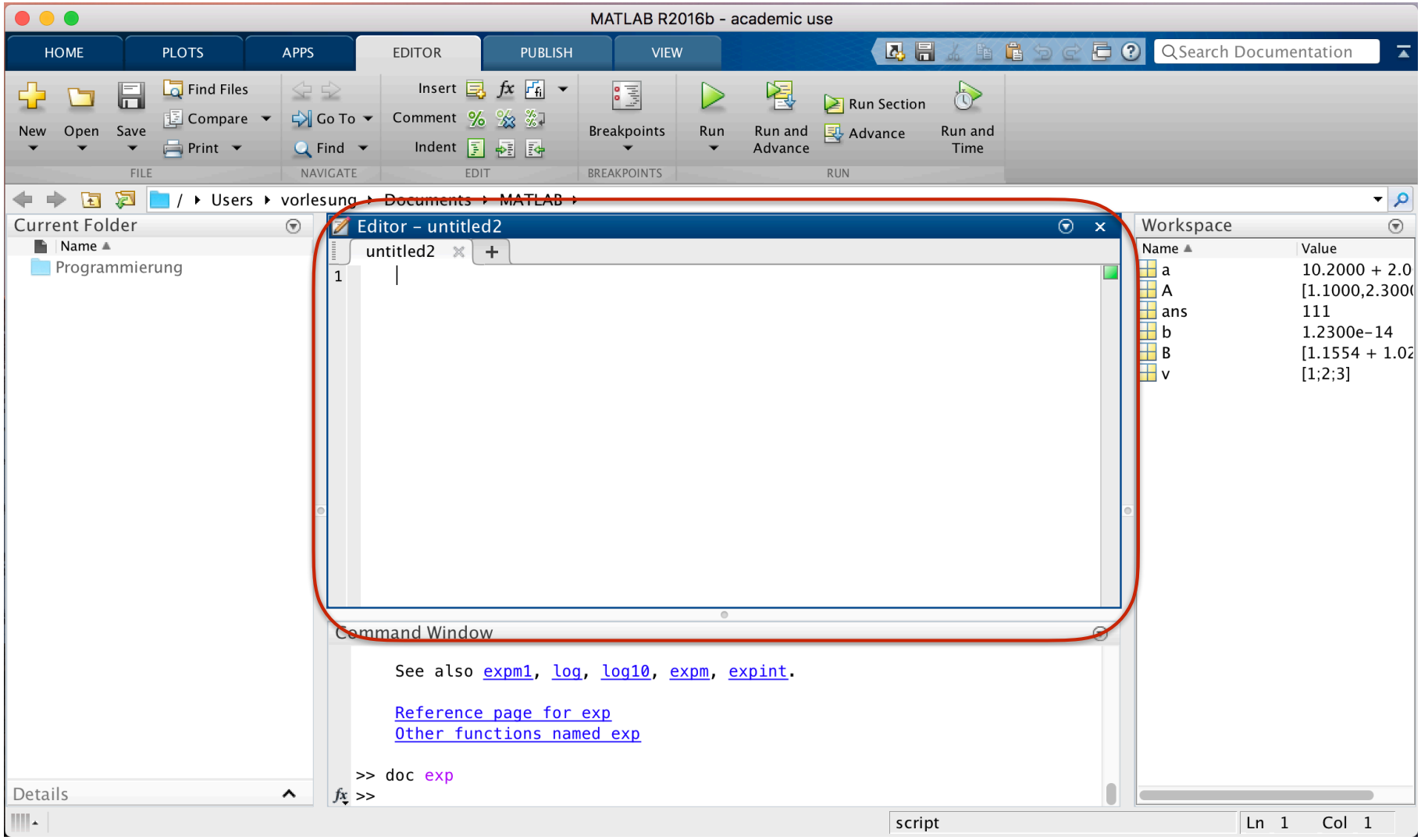
>> doc exp
```

Ich bitte darum, von der MATLAB-Hilfe
*ausgiebig* Gebrauch zu machen.

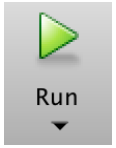
# MATLAB Skripte

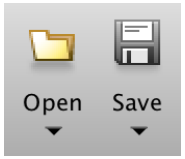


# Matlab Skripte (cont.)



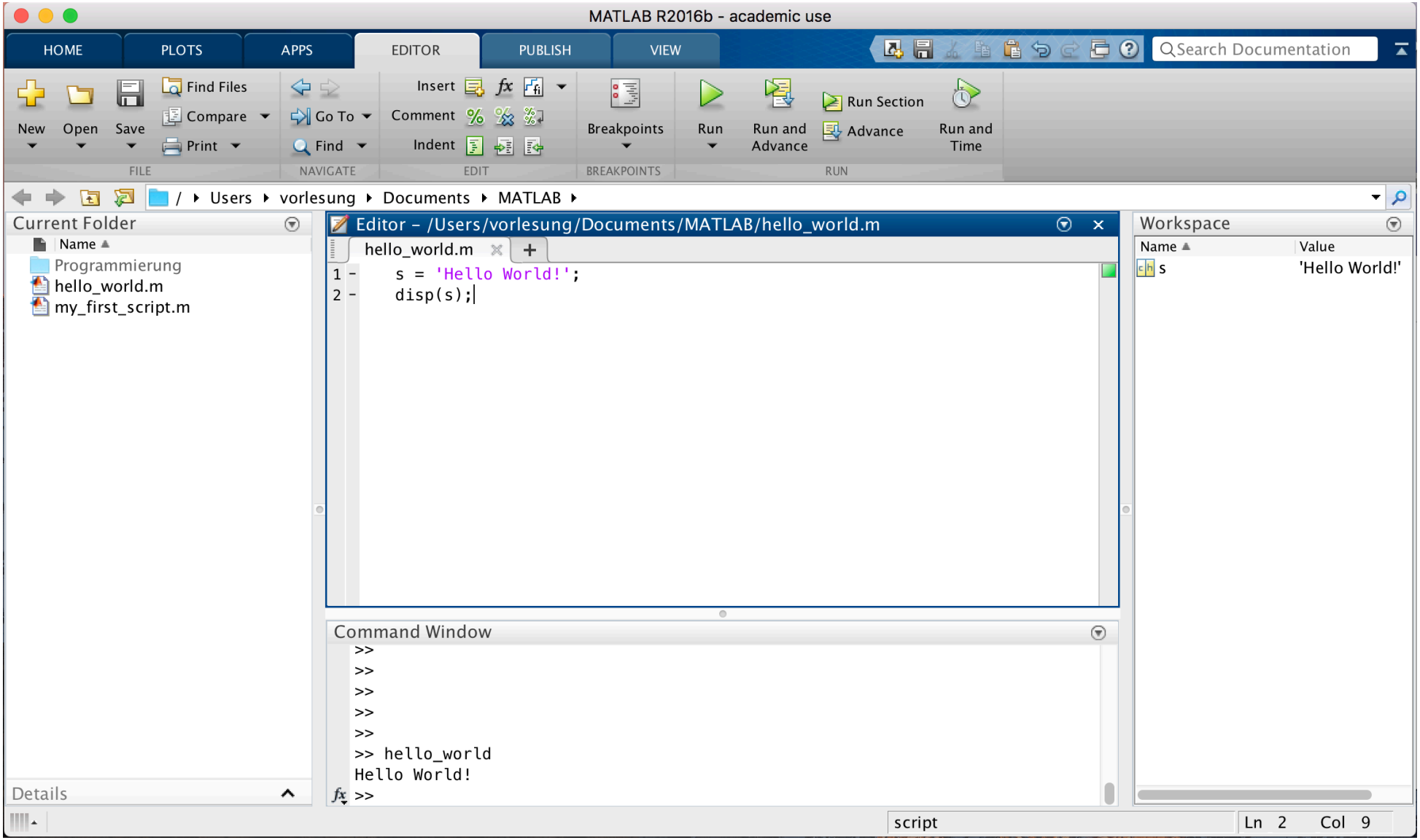
# Matlab Skripte (cont.)

- Kommandos können nun Zeile für Zeile als Skript gespeichert werden.
- Ausführung durch Klick auf 

- Laden und Speichern mit 

- Standard-Dateiendung ‘.m’ — ein paar Einschränkungen im Dateinamen: erstes Zeichen keine Zahl, keine Leerstellen,…

# Obligatorisch



# Ein Skript mit Rechenoperationen

```
% ich bin ein Kommentar
clear          % loescht alle Variablen aus dem Workspace
v = [1;2;3];
A = [1.1 2.3 4.5; 8 7 2; 4.1 1.1 0];
B = A*v;
B              % Ohne Semikolon erfolgt die uebliche Ausgabe
               % im Command Window
```

# Numerische Datentypen in MATLAB

- Generell werden am Rechner Zahlen immer im Binärsystem gespeichert, d.h. eine ganze Zahl wird geschrieben als $a_j \cdot 2^j$, $j$=0..($k$-1) mit $k$ der Anzahl der verwendeten Bit, eventuell mit einem Bit für ein Vorzeichen

- Standardmäßig wird jede Zahl in MATLAB als sog. `double` gespeichert, **auch** bei einem Kommando der Form `a=3`

- Ein double ist eine Fließkommazahl (eine Computerapproximation einer reellen Zahl), d.h. es wird eine gewisse Menge Bit auf einen Vorfaktor $m$ verwendet, der Rest auf einen Exponenten $a$ — die Zahl ist dann von der Form $m \cdot 2^a$.

- Genauigkeit von double: etwa 16 Dezimalstellen (relativ), größte darstellbare Zahl etwa $10^{308}$, kleinste positive darstellbare Zahl etwa $10^{-308}$

- Es gibt auch `single`, (Verwendung `a=single(10.234)`) diese haben die Hälfte der Präzision (und benötigen die Hälfte des Speicherplatzes, 32 anstelle von 64 Bit).

# Numerische Datentypen in MATLAB (cont.)

- MATLAB versucht allerdings, ganze Zahlen etwas anders zu behandeln, obwohl sie nicht anders gespeichert werden als (approximationen von) reellen Zahlen, beispielsweise können diese als Indizes von Einträgen in Vektoren benutzt werden
- Will man tatsächlich eine ‘echte’ ganze Zahl verwenden, so kann man einen Ausdruck der Form
  `a = uint16(1337)`
  verwenden

- Es gibt hier (`u`)`int8`, (`u`)`int16`, (`u`)`int32`, (`u`)`int64`.

- Das u steht für ‘unsigned’, also nicht vorzeichenbehaftet, die Zahl gibt die Anzahl Bit vor. Wertebereich?

- Bei Operationen mit Integertypen wird immer gerundet

- Fürs Erste empfehle ich aber, einfach beim Standardtyp zu bleiben, Probleme gibt es nur in Ausnahmefällen

# Wichtige Numerische Funktionen

- Die Standardoperatoren (+,-,…) haben wir bereits kennen gelernt
- Eine Auswahl von anderen Funktionen:
  `exp`, `log`, `log10`, `log2`, `sqrt`, `mod`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`, `atan2`, `abs`, `imag`, `real`, `conj`, `sign`, `round`, `floor`, `ceil`, `fix`
- Die MATLAB-Hilfe ist hier sehr nützlich

# Wichtige Funktionen auf Vektoren und Matrizen

- Die Standardoperatoren haben wir auch hier bereits kennen gelernt (Achtung: Unterscheidung zwischen punktweisen Operatoren und Matrixoperationen)

- Ansonsten gibt es z.B.
  `max`, `min`, `mean`, `median`, `sum`, `prod`, `sort`, `transpose`, `det`, `inv`, `diag`

- Alle, die nicht natürlicherweise auf Matrizen operieren, behandeln jeden Spaltenvektor einer Matrix separat und geben einen Reihenvektor aus

# Strings

- Mit `s = 'hallo'` wird in MATLAB der Variable `s` ein String zugewiesen
- Stringoperationen in MATLAB z.B. `strcat`, `strfind`, `strcmp`

# fprintf, sprintf

- Mit der Funktion `fprintf` (bzw. `sprintf`) kann ein schön formatierter String ausgegeben werden.
- Die Syntax ist (bestenfalls) gewöhnungsbedürftig
- Beispiel: `fprintf('hallo %.5f\n', 0.2);`
- `\n` : Neue Zeile
- Jedes Vorkommen von Einem Ausdruck der Form `%f` (o.Ä) wird durch die jeweilige Zahl im Argument ersetzt
- Es gibt z.B. `%d`, `%u`, `%f`, `%e`
- Allgemein: `%8.4x`, die erste Zahl gibt an auf wieviele Zeichen die Ausgabe formatiert werden soll (es werden einfach Leerzeichen eingefügt), die zweite Zahl gibt bei `f` und `e` die Anzahl von Nachkommastellen an, bei `d` und `u` die wird mit vorausgehenden Nullen aufgefüllt
- `fprintf` schreibt in eine Datei (dazu später mehr) bzw. auf die Konsole, `sprintf` in einen anderen String (`s = sprintf('hello %f', 17.1);`)

# Relationsoperatoren

- Es sind definiert
  >, <, <=, >=, ==, ~=, also (in der Reihenfolge) kleiner als, größer als, kleiner oder gleich, größer oder gleich, ist gleich, ist ungleich

- Die Ausgebe ist ein *logical*, `0` wenn falsch (‘false’), `1` wenn korrekt (‘true’).

- ```
  >> 5<2
  ans =
   logical
  0
  ```

- Angewendet auf Vektoren oder Matrizen erfolgt die Auswertung eintragweise! Vorsicht bei komplexen Zahlen

- Man kann ein `logical` auch als Variable erzeugen, und zwar mit
  `a = logical(1);` oder
  `a = ( 5<2 );`
  ```

# Logische Operatoren

| A | B | A&B | A\|B | ~A | xor(A,B) |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 |

# Logische Operatoren (cont.)

- Es wird von links nach rechts ausgewertet, also ist zum Beispiel
  `~a|b|c = ((~a)|b)|c`

- & hat aber Priorität, also gilt
  `a|b&c = a|(b&c)`

- Für skalare Größen kann man auch `||` und `&&` verwenden, diese sind etwas schneller (duch Verwendung eines ‘Kurzschlusses’)

# Bedingte Ausführung: `if`

```
if (logischer Ausdruck1)
  Anweisungen 1;
elseif (logischer Ausdruck2)
  Anweisungen 2;
else
  Anweisungen 3;
end
```

- Falls (logischer Ausdruck 1) als wahr evaluiert wird, so werden die Anweisungen 1 ausgeführt
- Falls (logischer Ausdruck 1) falsch, aber (logischer Ausdruck 2) wahr ist, so werden die Anweisungen 2 ausgeführt
- Falls weder (logischer Ausdruck 1) noch (logischer Ausdruck 2) wahr sind, so werden die Anweisungen 3 ausgeführt
- Im Fall von nicht-skalaren logischen Ausdrücken werden alle Einträge mit `&` verknüpft

# Fallunterscheidung: `switch`

```
switch Ausdruck
  case a
    Anweisungen 1;
  case b
    Anweisungen 2;
  otherwise
    Anweisungen 3;
end
```

Es wird erst der Ausdruck ausgewertet, falls dieser gleich a ist, so werden die Anweisungen 1 ausgeführt etc.

# `for`-Schleifen

```
for variable = Matrix oder Vektor
  Anweisungen;
end
```

Der Variable werden nacheinander die Spalten der Matrix bzw. (im einfachen, üblicheren Fall) die Einträge des Vektors zugeordnet. Dann werden jeweils die gegebenen Anweisungen ausgeführt. Die Variable kann in den Anweisungen verwendet werden.

Für eine einfache Schleife verwendet man den Ausdruck

```
for index = 1:100
 …
end
```

# `while`-Schleifen

```
while logischer Ausdruck
  Anweisungen;
end
```

Die Anweisungen werden solange ausgeführt, wie der logische Ausdruck als ‘wahr’ evaluiert wird.

# `break` und `continue`

- In einer (`for` oder `while`) Schleife können die Befehle `break` und `continue` (üblicherweise innerhalb einer weiteren `if`-Abfrage) verwendet werden.
- Mit `break` wird die Schleife verlassen
- Mit `continue` wird der Rest der Anweisungen innerhalb der Schleife übergangen

# Vektoren, Matrizen und lineare Gleichungssysteme (siehe File matvec.m)

- Matrizen und Vektoren erstellen
- Standard Matrix-Vektoroperationen
- Matrix- und Vektorindizes

  https://de.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html

- Max, Min, Finden, etc.

# find

- Der MATLAB-Befehl `find(A)` findet im Argument von Null verschiedene Einträge und gibt einen Vektor mit linearen Indizes für diese Einträge zurück
- Falls `A` ein logisches Array ist, so werden alle Indizes gefunden, die '`true`' (also 1) sind
- `find(A,n)` findet die ersten n Einträge

# Max, Min

- Diese Befehle haben wir bereits gesehen, sie können aber zum Beispiel auch die Indizes der maximalen Elemente zurückgeben:
  ```
  [m,I] = max(A);
  ```

- Wenn man nur die Indizes haben möchte, kann man den ersten zurückgegebenen Wert auch wegwerfen:
  [~,I] = max(A);

- Generell gibt es einige Funktionen, die mehr als eine Variable zurückgeben — dies kann immer mit
  ```
  [a,b,c]  = function(A,B,C);
  ```
  ausgewertet werden

# Reshape

- Die Funktion
  `A = reshape(B, sz);`
  macht aus der Matrix `B` eine Matrix `A` (mit ebensovielen Einträgen), die die Dimensionen wie im Vektor `sz` angegeben besitzt

# Backslash

- Der MATLAB-Backslash löst lineare Gleichungssysteme:
  `x = A\b;`

# Funktionendefinitionen

- Bekannt sind bereits eine ganze Reihe von eingebauten MATLAB-Funktionen, wie zum Beispiel sin, max, find, etc.

- Man kann aber auch eigene Funktionen definieren, hier ein Beispiel:

```
function y = myfun(x)
y = sin(x)+x.^2
```

  Dieser Text wird in einer Datei namens `myfun.m` gespeichert (Achtung: Dateiname und Funktionsname müssen übereinstimmen).

- Danach kann diese Funktion aufgerufen werden (in der Kommandozeile oder in einem Skript:

```
a = myfun(3);
```

# Funktionendefinitionen (cont.)

- Genau wie die eingebauten Funktionen können selbstdefinierte Funktionen mehrere Argumente sowie mehrere Rückgabewerte besitzen:
  `function [out1, out2, …, outN] = myfun2(in1,in2,…inN)`
  … in der Funktion werden nun `out1` bis `outN` berechnet.

- Aufruf mit
  `[y1, y2, …, yN] = myfun2(a1, a2, …, aN).`

# Funktionendefinitionen (cont.)

- Werden weniger Rückgabewerte angegeben, so werden die restlichen (von vorne gezählt) verworfen.

- Werden weniger Eingabeargumente angegeben, so gibt es eine Fehlermeldung, wenn diese verwendet werden.

- Ob ein Argument angegeben wurde, kann man in der Funktion mittels
```
if ~exist('in2','var')
  …
end
```
überprüft werden. Die Funktionen `nargin` bzw. `nargout` gibt die Anzahl der übergebenen Argumente bzw. Rückgabewerte zurück.

- Auch hier gilt: will man nur den zweiten Rückgabewert bekommen, so kann man einen Aufruf der Form
```
[~,a] = myfun2(1,2,3);
```
benutzen (Achtung: in diesem Beispiel ist `nargout` gleich 2).

# Mehrere Funktionen in einem File (alte Variante)

- Leider ist das etwas umständlich. Man kann ein File testfun.m schreiben mit dem Inhalt

```
function [] = testfun()
% keine Argumente, keine Rückgabewerte.
  fprintf('%f %f\n', hello(12), hello2(2))
end

function y = hello(x)
  y = sin(x)+12;
end

function z = hello2(x)
  z = cos(x).^2;
end
```

# Mehrere Funktionen in einem File (seit 2016b)

- Skript mit beliebigem Namen

```
fprintf('%f %f\n', hello(12), hello2(2))

function y = hello(x)
  y = sin(x)+12;
end

function z = hello2(x)
  z = cos(x).^2;
end
```

# Anonyme Funktionen

- Mit `test = @(x) sin(x)/x` kann eine anonyme Funktion erzeugt und anschließend `test` genannt werden

- Beispiel:

```
test = @(x,y) sin(x)./y;
fprintf('%f\n', test(0.7,0.1));
```

# Vektoren und Matrizen als Arrays und Ausgabewerte

- Funktioniert genauso, wie man es sich vorstellt.
- Es ist in MATLAB sehr üblich, den Programmcode zu vektorisieren, selbstgeschriebene Funktionen sollten also (sofern das in irgend einer Weise sinnvoll ist) mit vektoriellem Input umgehen können.

# Funktionen als Argument

- Beispiel:

```
func = @(x) sin(x)/x;
a = test(func, 2);
```

mit `test.m` mit Inhalt

```
function z = test(f, x)
  z = f(x).^2;
```

oder der neueren Variante mit der Funktion `test` lokal definiert (`end` nicht vergessen).

- Genau genommen erzeugt der Code oben ein Function-Handle `func`. Will man eine extern definierte Funktion (eingebaut oder selbstdefiniert) übergeben, so muss man sie mittels vorangestelltem @ in ein Handle verwandeln:
`a = test(@sin, 2);`

# Graphische Ausgabe von Funktionen

- Eine sehr einfache Variante:
  `fplot(@sin,[-pi,pi]);`

- Diese Funktion erzeugt einen Plot der (als Handle übergebenen) Funktion. Besonders nützlich in der Kombination mit einer anonymen Funktion:
  `fplot(@(x) sin(x)./x, [-2*pi, 2*pi]);`

- Eine alte Variante heißt `ezplot` und funktioniert ähnlich.

# 2d-Plots

- Beispiel

```
x = [0.1 0.2 0.3 0.6 1 1.5 2.5];
y = [0.1 0.3 0.2 1.8 2.5 2.8 3.0];
figure;
plot(x,y); %erzeugt einen Linienplot
figure;
stairs(x,y); %erzeugt einen Stufenplot
```

# Mehrere Linien

- Beispiel:

```
x = [0.1 0.2 0.3 0.6 1 1.5 2.5];
y1 = [0.1 0.3 0.2 1.8 2.5 2.8 3.0];
y2 = [0.6 0.2 3.0 1.8 4.0 1.5 1.2];
figure;
plot(x,y1,x,y2);
figure;
% mit unterschiedlichen Stilen
% und Farben
plot(x,y1,'--g',x,y2,':*r');
```

- Allgemein: Linienstil (weglassen für keine Linien), dann Marker (weglassen für keinen Marker), dann Farbe. Verschiedene Möglichkeiten siehe MATLAB-Hilfe zu `plot` und LineSpec

- Kompliziertere Variante beispielsweise für Marker in anderer Farbe als die Linie

# Titel und Achsenbeschriftungen

- Beispiel:

```
x = linspace(0, 4*pi, 100);
y = sin(x);
plot(x,y);
title('test');
xlabel('x');
ylabel('y=f(x)');
grid on;
axis tight;
text(pi+0.01,0, 'Nullstelle');
```

# Wichtige Befehle

- axis([xmin xmax ymin ymax]);
  Legt Achsenintervalle fest.
- axis tight
  Setzt Achsen eng anliegend.
- axis equal
  Gleiche Skalierung in x und y
- axis square
  Quadratisches Bild
- grid on/off
  Gitter anzeigen bzw. löschen
- xlabel/ylabel/title (jeweils mit einem String als Argument)
  Achsebeschrifung und Titel
- legend (ein String pro Datensatz als Argument)
  Legende anzeigen
- text(x,y,string)
  Text ins Bild einbauen

# Zusammenfügen von Plots

- Der Befehl hold on führt dazu, das weitere Plots dem aktuellen Plot hinzugefügt werden:

```
plot(X, sin(X));
hold on
plot(X,sin(X.^2));
hold off
legend('sin(x)', 'sin(x^2)');
```

# Andere Plots

- `loglog`
- `semilogx`
- `semilogy`

# Modifikationen und Export

- Nacheinander mehrere Plots mittels `hold on` und `hold off`
- Mehrere Unterplots in einem Bild mittels `subplot`
- Weitere Modifikationen können interaktiv im Plot vorgenommen werden
- Export via Menü im Plot. Es gibt (im Wesentlichen) zwei Möglichkeiten, als `.pdf` oder als `.png`
- Export auch möglich mittels

```
print -dpdf dateiname.pdf
print -dpng dateiname.png
```

# 3d-Plots von Funktionen

- Einfache Ausgabe von Funktionen:

```
fsurf(@(x,y) sin(x)+cos(y), [-pi pi -pi pi]);
```

# Surface Plots

- Gegeben eine Matrix `S`, lässt diese sich einfach durch

  ```
  surf(S)
  ```

  als Fläche darstellen.

- Will man eine Funktion gegenüber `x`- und `y`-Werten darstellen, so ist `meshgrid` sehr nützlich.

```
x = linspace(0,2,30);
y = linspace(0,3,45);
[X,Y] = meshgrid(x,y);
surf(X,Y, sin(X)+cos(Y));
```

# Farbpalette

- Die Funktion `colormap` stellt einige Standardpaletten zur Verfügung:

```
colormap summer
```

- Man kann in das Argument von `surf` auch separat eine Farbe für jeden Punkt vorgeben:

```
Z = sin(X)+cos(Y);
C = sin(X);
surf(X,Y,Z,C);
```

- `colorbar` erzeugt eine Farblegende, `caxis` ändert die Skalierung der Farbpalette

# Shading, Lighting

- Der Befehl `shading` gibt an, wie die Fläche aussehen soll. Es gibt die Möglichkeiten `faceted` (default), `flat` und `interp`.
- Der Befehl `lighting` gibt an, wie Oberflächen dargestellt werden sollen. Es gibt die Möglichkeiten `flat`, `gouraud` und `none`.
- `light, camlight` erzeugen eine Lichtquelle.
- `material` ändert die Reflektionseigenschaften der Fläche, es gibt `shiny`, `metal` und `dull`.

# Andere Plots

- `mesh`
- `contour`
- `surfc`
- `pcolor`

# Wichtige Befehle zu Surface Plots

- alle von 2d-Plots
- `axis ij`, `axis xy`
- Achsenskalierungen haben jetzt natürlich sechs Parameter
- `view(az, el)` — Betrachtungswinkel.

# Filme

- Siehe Beispielprogramme.

# Dünnbesetzte Matrizen

- Sehr häufig hat man es in der angewandten Mathematik mit sehr großen Matrizen zu tun, bei denen allerdings fast alle Einträge Null sind.

- Eine Matrix, bei der so viele Einträge Null sind, das man das am Rechner effizient ausnutzen kann, heißt dünnbesetzt. Es können dann eventuell, in geeigneter Weise, nur die von Null verschiedenen Einträge der Matrix, sowie ihre Indizes gespeichert werden.

- In MATLAB gibt es hierfür einen Datentyp: `sparse`

# Verwendung von sparse

- Initialisieren einer Nullmatrix:

```
A = sparse(n, n);
```

- Initialisieren einer Nullmatrix mit vorreserviertem Speicher für Einträge:

```
A = spalloc(n, n, nnz);
```

- Sparse-Matrix aus vollbesetzter Matrix erstellen:

```
A = sparse(B);
```

- Sparse-Matrix mit Diagonaleinträgen:

```
d = ones(100000,1);
A = spdiags([d, -2*d, d], -1:1, 100000, 100000);
% erstellt Sparse-Matrix mit den Spaltenvektoren d auf den
Nebendiagonalen und -2*d auf der Diagonalen
```

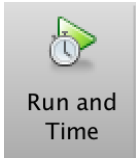
# Auffüllen von Sparse-Matrizen

- Ein einfaches `A(i,j) = A(i,j) + 3` funktioniert zwar, ist aber leider eher langsam.

- Besser: Zuerst die Matrixeinträge als Listen von Zahlen und Indizes erstellen.

```
i = [6 6 6 5 10 10 9 9]';
j = [1 1 1 2 3 3 10 10]';
v = [100 202 173 305 410 550 323 121]';
A = sparse(i, j, v); % mehrfach
vorkommende                    %Indizes
werden aufsummiert!
```

# Der MATLAB-Profiler




- Der Profiler misst die Zeit, die das Programm in jeder einzelnen Zeile verbringt.
- Damit lassen sich sehr gut herausfinden, wo sich eine Optimierung lohnt!

# Vektorisierung

- Insbesondere for-Schleifen sind in MATLAB oftmals eher langsam.
- Es ist deshalb häufig von Nutzen, ein Programm zu vektorisieren.
- Die Vorgehensweise hängt sehr vom individuellen Problem ab. Siehe Beispielprogramm.

# C++

- Ein ausgezeichnetes (aber leider etwas veraltetes) Buch ist

  A. Koenig, B.E. Moo; “Accelerated C++” Addison-Wesley 2000

# Grundlegende Unterschiede

- Im Gegensatz zu MATLAB ist C++ eine kompilierte Programmiersprache, d.h. ein Programm muss vor dem Ausführen in Maschinensprache übersetzt werden
- Variablentypen sind statisch (d.h. müssen vor der Übersetzung feststehen
- C++ ist sehr flexibel und höchst effizient

# Ein erstes Beispielprogramm

- hello.cpp

```
// ein kleines Programm
#include <iostream>

int main() {
  std::cout << “Hello, world!” << std::endl;
  return 0;
}
```

# Kompilieren

- Mit gcc
  ```
  > g++ -o hello hello.cpp
  ```
- Mit clang//llvm
  ```
  > clang++ -o hello hello.cpp
  ```
- danach auf Linux und Mac
  ```
  > ./hello
  ```
  auf Windows
  ```
  > hello.exe
  ```

# Installation von Compilern

- Linux:
  gcc-Paket installieren (evtl. auch clang). Pakete sollten für alle Linuxdistributionen leicht verfügbar sein

- Mac:
  XCode installieren, damit ist automatisch auch clang verfügbar (kein gcc)

- Windows:
  Mingw-w64 ist vermutlich die einfachste Variante. Auch die Microsoft-Programmierumgebung Visual Studio ist geeignet.

# Zurück zu unserem Programm

- hello.cpp

```
// ein kleines Programm
#include <iostream>

int main() {
  std::cout << “Hello, world!” << std::endl;
  return 0;
}
```

# Zurück zu unserem Programm

- `// ein kleines Programm`
- Die erste Zeile ist ein Kommentar.
- Es gibt für Kommentare zwei Möglichkeiten: ein // kommentiert den Rest der Zeile aus, /* …. */ kommentiert alles zwischen den Markierungen (auch mehrzeilig).

# Zurück zu unserem Programm

- `#include <iostream>`
- Die meisten fundamentalen Möglichkeiten von C++ sind nicht Teil der Sprache selbst, sondern werden von der **Standardbibliothek** bereitgestellt.
- Solche Art Funktionalität muss mit einem `#include`-Statement eingebunden werden.
- `iostream` stellt Ein- und Ausgabemöglichkeiten für Zeichenketten bereit.

# Zurück zu unserem Programm

- `int main()`

- Eine Funktion in C++ ist ähnlich wie in MATLAB ein Teil des Programms, dem ein Name gegeben wurde, dem (evtl.) Parameter übergeben werden, und das (evtl.) einen Rückgabewert besitzt.

- Jedes C++-Programm muss eine main-Funktion beinhalten, diese wird bei Aufruf des Programms ausgeführt. In unserem Fall besitzt die Funktion keine Parameter `()` und gibt einen `int` (also eine ganze Zahl) zurück.

# Zurück zu unserem Programm

- `{ ..... }`
- Mit geschweiften Klammern werden in C++ Blöcke von Code gruppiert.
- In unserem Fall heißt das, dass die Anweisungen in diesen Klammern die Funktion `main()` bilden.

# Zurück zu unserem Programm

- `std::cout << ”Hello” << std::endl;`

- Dies ist die eigentliche Anweisung. Der Sogenannte Ausgabeoperator `<<` wird benutzt, den String “Hello” in die Standardausgabe `cout` (die Konsole) zu schreiben. `endl` fügt einen Zeilenumbruch hinzu. Jede Anweisung muss mit einem Semikolon beendet werden.

- Das vorgesetzte `std::` muss angegeben werden, nachdem `cout` und `endl` Teil der Standardbibliothek sind.

# Zurück zu unserem Programm

- `return 0;`
- Unsere Funktion gibt einen `int` zurück, in diesem Fall einfach Null.
- Generell bedeutet ein return in einer Funktion, dass diese zu dem aufrufenden Programm zurückkehren soll. Im Fall von `main()` ist das aber nichts anderes als die Kommandozeile.

# Eingabe

- Wir betrachten ein weiteres einfaches Programm:

```
// Hallo!
#include <iostream>
#include <string>
int main() {
  // wir fragen nach einem Namen
  std::cout << “Bitte Name eingeben: ";
  // read the name
  std::string name;     // definiere eine Variable
  std::cin >> name;     // Einlesen der Variable
  // Ausgabe der Begruessung
  std::cout << "Hallo, " << name << "!" << std::endl;
  return 0;
}
```

# Eingabe

- Wir betrachten ein weiteres einfaches Programm:

```
// Hallo!
#include <iostream>
#include <string>
int main() {
  // wir fragen nach einem Namen
  std::cout << “Bitte Name eingeben: ";
  // read the name
  std::string name;      // definiere eine Variable
  std::cin >> name;      // Einlesen der Variable
  // Ausgabe der Begruessung
  std::cout << "Hallo, " << name << "!" << std::endl;
  return 0;
}
```

# Variablen

- `std::string name;`
  definiert eine Variable vom Typ string (Teil der Standardbibliothek) und gibt dieser den Namen ‘name’.

- `std::cin >> name;`
  weist der Variable mittels des Eingabeoperators einen Wert zu, der von der Konsole gelesen wird.

# Blöcke und Scope

- Mit weiteren geschweiften Klammern kann man geschachtelte Blöcke definieren.

- Generell leben Variablen in C++ nur in dem ‘Scope’, d.h. in dem Programmblock, in dem Sie definiert wurden. Wird also mittels geschweifter Klammern ein neuer Block erzeugt und darin eine Variable definiert, so ist diese außerhalb nicht benutzbar!

```
int main() {
  std::string s1;
  {
    std::string s2;
    s1 = ”hello1”; // OKAY
  }
  s2 = ”hello”; // NICHT OKAY!
}
```

funktioniert nicht!

# Strings

- `std::string s;`
  definiert eine Variable s und weist eine leere Zeichenkette zu
- `std::string t = s;`
  definiert die Variable t und weist dieser den Inhalt von s zu.
- `os << s;`
  schreibt die Zeichen aus s in den ‘Output Stream’ os (z.B. std::cout)
- `is >> s;`
  liest Zeichen aus dem ‘Input Stream’ is (z.B. std::cin) und schreibt diese in den String s
- `s+t`
  resultiert in einem String mit den Zeichen in s und (danach) den Zeichen in t
- `s.size()`
  gibt die Anzahl der Zeichen in s zurück.
- `”Hallo”`
  ist ein String (genauer gesagt ein “String Literal” mit dem Inhalt “Hallo” (ohne die Anführungszeichen).

# Integers

- `int j;`
  definiert eine Variable j in der ganze Zahlen gespeichert werden können (ohne etwas zuzuweisen, der Inhalt der Variable ist vor der ersten Zuweisung nicht zu gebrauchen!).
- `int j = 0;`
  definiert und weist Null zu.
- Die üblichen arithmetischen Operationen funktionieren.
  https://www.tutorialspoint.com/cplusplus/cpp_operators.htm
  Besonders interessant sind die Inkrement- und Dekrementoperatoren ++ bzw. –
- `os << j;`
  `is >> j;`
  funktionieren auch.
- Vergleiche
  `i < j`
  etc (>, >=, <=, != ) geben einen Wert vom Typ `bool` zurück, dieser kann verwendet werden in

# Bedingte Ausführung

- …

```
int i;
std::cout >> i;
if ( i<10 ) {
  std::cout << ”Die eingegebene Zahl
ist kleiner als Zehn.” << std::endl;
} elseif ( i==10 ) {
  …
} else {
  …
}
…
```

# While

- ```
  …
  int j = 3;
  while ( j<1000 ) {
    j = j*j;
    std::cout << j << std::endl;
  }
  …
  ```
  ```

# for

- ```
  for ( int j = 0; j<10; ++j ) {
    std::cout << j << std::endl;
  }
  ```
  ist äquivalent zu
  ```
  {
   int j = 0;
    while ( j<10 ) {
      std::cout << j << std::endl;
      ++j;
    }
  }
  ```
  ```

# Ein paar Details

- Als “Ausdruck” wird ein Konstrukt bezeichnet, das von der Implementierung ausgewertet wird.

- Jeder Ausdruck besitzt ein Ergebnis und kann Nebeneffekte haben.
  Der Ausdruck “4+5” besitzt das Ergebnis 9 und hat keine Nebeneffekte.
  Unser Ausdruck
  std::cout << ”Hello, World!” << std::endl;
  hat den Nebeneffekt, dass “Hello World” auf der Konsole ausgegeben wird.

# Details (cont.)

- Jeder Ausdruck besteht aus Operatoren und Operanden. Im vorigen Beispiel sind “<<" die Operatoren, der Rest Operanden.

- Jeder Operand besitzt einen Typ — “int” ist beispielsweise ein Typ, der von der Sprache selbst (der “Core Language”) definiert ist. “std::ostream” ist ein Typ, der von der Standardbibliothek definiert wird, und datenstrombasierte Ausgabe handhabt. std::cout ist ein Objekt vom Typ std::ostream.

- Der Operator “<<“ benötigt zwei Operanden — diese werden links-assoziativ behandelt, d.h. der Ausdruck ist äquivalent zu
(std::cout << “Hello World”) << std::endl;

- Es gilt: << schreibt die Zeichenkette auf der rechten Seite in das Objekt auf der linken Seite, und gibt als Resultat das Objekt auf der linken Seite nochmals zurück. Damit wird std::endl einfach danach in std::cout geschrieben.

# Details (cont.)

- Scope im Sinne von Bereichen, in denen definierte Variablen gültig sind, haben wir bereits betrachtet.
- std::cout ist ein sogenannter “qualifizierter Name”, “::” heißt auch “Scope Operator”. Links steht der Name eines Gültigkeitsbereiches, rechts der Name eines Objekts (z.B. einer Typenbezeichnung), das im Gültigkeitsbereich definiert ist.

# Wichtige Datentypen und Operatoren

- std::string — ein gar nicht so einfaches Objekt, insbesondere wegen Unicode. Wir benötigen aber eigentlich nur die Operatoren + (Concatenation), <<, >> (für Streams), sowie .size()

- int — Operatoren: +,-,*,/,%,++,--,-=, +=, *=, /=, %=

- bool — Werte true bzw false. a>b etc. ergeben als Resultat ein bool. Operatoren: &&, !, ||

- double — kennen wir auch schon. Operatoren wie bei int (außer denjenigen, die keinen echten Sinn ergeben).

# Using

- using std::cout;
  Nach dieser Zeile ist cout in den aktuellen Scope verfrachtet (d.h., man kann cout anstellen von std::cout schreiben).

- using namespace std;
  sehr beliebt (aber evtl. eher nicht zu empfehlen), importiert alles aus std in den aktuellen Scope. Kann man auch in den Header schreiben, dann gilt es in jeder Funktion.

# Container

- Die Standardbibliothek stellt eine Reihe von sehr nützlichen Datenstrukturen zur Verfügung. Der häufigstverwendete ist wohl std::vector

- ```
  #include <vector>
  ...
  std::vector<double> vec;
  vec.push_back( 12.3 );
  vec.push_back( -1.7 );
  ```
  ```

# std::vector

- std::vector<double> v;
- v.begin() — erster Eintrag
- v.end() — Element nach dem letzten Eintrag
- v.push_back(e) — vergrößert den Vektor um einen Eintrag, dieser wird auf e gesetzt.
- v[i] — i-tes Element
- v.size() — Anzahl Elemente

# for-Schleife über einen Vektor

- Klassisch:

```
std::vector<double>::iterator vi =
v.begin(), ve = v.end();
for ( ; vi != ve; ++vi ) {
  std::cout << *vi << std::endl;
}
```

- Modern:

```
for (auto i : v) {
  std::cout << i << std::endl;
}
```

# Pointer und Arrays

- Werden verwendet zur dynamischen Speicherverwaltung. Siehe Beispielprogramm.

# Iteratoren

- Die Container der Standardbibliothek (also z.B. `std::vector` stellen Iteratoren zur Verfügung.
- Diese sind mit Pointern verwandt in dem Sinne, dass sie wie Zeiger auf Objekte zu verstehen sind -- in diesem Fall Zeiger auf Objekte im Container.
- Standardmäßig stellt ein Container `c` die Iteratoren `c.begin()` und `c.end()` bereit. Diese sind vom Typ *container_type*`::iterator`
- Es gibt auch `const_iterator`.
- Achtung: `c.begin()` zeigt auf das erste Element, `c.end()` auf das Element **nach** dem letzten. Warum?
- Übliche Verwendung: in Schleifen und als Anfangs- und Endpunkt für Algorithmen wie `std::sort`
- Auf Iteratoren können die Inkrement- und Dekrementoperatoren (`++`/`--`) angewendet werden, Dereferenzierung mit `*` -- wie bei Pointern.

# Funktionen

- Einfachste Variante

```
double f(double x, double y) {
  double z = x*x+y*y;
  return sqrt(z);
}
```

- Wir können auch Pointer und Arrays übergeben, eine Rückgabe eines Pointers ist üblicherweise nicht so günstig.

- Übergabe einer Referenz:

```
void f(double &z, double x, double y) {
  z = x*x+y*y;
}
```

- `const` (???)

# Sequentielle Container

- Das sind beispielsweise vector, list, deque, forward_list. http://en.cppreference.com/w/cpp/concept/SequenceContainer
- Je nach Container sind unterschiedliche Anwendungen unterschiedlich schnell!

# Sequentielle Container (cont.)

- Es werden unterstützt:
  `c.begin()`, `c.end()`, `c.rbegin()`, `c.rend()`.
  `container<T> c;` // leerer neuer Container/
  `container<T> c(c2);` // neuer Container mit Inhalt von `c2`
  `container<T> c(n);` // Container mit Platz für `n` Einträge (bei eingebauten Typen mit Null initialisert)
  `container<T> c(b,e);` // Container mit den Elementen von Iterator `b` bis (ausschließlich) `e`
  `container<T> c = {x, y, z};` // Initlist (neu in C++11)
  `c = c2;` // Ersetzt die Elemente in `c` durch die in `c2`
  `c.size();` // Anzahl Elemente
  `c.empty();` // bool, gibt an ob der Container leer ist
  `c.insert(d, b, e)` // Copiert `[b,e)` in `c`, direkt vor `d`
  `c.erase(it)` // Löscht Element `it`
  `c.erase(b,e)` // Löscht [b,e).
  `c.push_back(t)` // Fügt Kopie von Element `t` hinten ein.
  `c[n]` // Zugriff auf Element n (nicht für alle Container).

# Assoziative Container

- http://en.cppreference.com/w/cpp/concept/AssociativeContainer

- `std::pair<K,V> p;` // Paar aus Typen K und V
  `p.first(), p.second()` // klar.
  `std::map<K,V> m;` // Assoziatives Array
  `m[k] = v;` // Zuweisung
  `m[k]` // Zugriff auf durch k indizierten Eintrag von m
  `it = m.begin();`
  `it = m.end();`
  `it = m.find(k);`
  `it->first();` // == (*it).first();
  `it->second();`
  `std::set<V> s;` // Mengenartige Struktur
  `pair<std::set<V>::iterator, bool> ret = s.insert(v);` // Einfügen. In `ret.first()` steht dann der Ort wo das Element zu finden ist, `ret.second()` ob eingefügt wurde (false wenn das Element bereits in `s` war).

# Algorithmen in der Standardbibliothek

- Wir haben schon gesehen:
  `std::sort(b, e);` // sortiert die Elemente in [b,e);
- `it = std::find(b,e, v);` // findet v in [b,e)
- `t = std::accumulate(b, e, init);` // Summiert (o. ä).
- `std::partition` // ...

# Beispielcode

- In großen Mengen auf cppreference.com zu finden.

# Klassen

- Man kann in C++ Objekte, sogenannte "Klassen" definieren
- Eine Klasse besteht aus verschiedenen Mitgliedern (Members)
- *Data Members* oder *Member Variables* sind Variablen, die innerhalb einer Klasse auftauchen. Sie speichern den Zustand eines Objekts.
- Member Functions sind Operationen, die mit diesem Objekt ausgeführt werden können. Diese Funktionen verändern oder arbeiten üblicherweise mit den Member Variables.

# Arten von Member Funtions

- *Konstruktoren* initialisieren neue Instantiierungen einer Klasse. Wir werden ein paar Beispiele sehen.
- *Destruktoren* räumen auf, wenn die Instanz einer Klasse aufhört zu existieren.
- *Accessors* erlauben Zugriff auf den internen Zustand der Klasse -- man kann also genau kontrollieren welche Daten 'von außen' sichtbar sind.
- *Mutators* ändern den internen Zustand.

# Abstraktion und Kapselung

- Ein Objekt (eine Klasse) sollte als eine funktionale Einheit betrachtet werden. Beispiel: Ein Vektor im zweidimensionalen Raum besitzt als Objekt einen Zustand (x- und y-Koordinate), und hat bestimmte natürliche Eigenschaften (z.B. Länge, Abstand zu einem anderen Punkt), die abgefragt werden können.

- Generell sollte man versuchen, mit einer Klasse ein sauberes, einfaches Interface zu schaffen. Das Objekt sollte seinen internen Zustand selbst verwalten. Beispielsweise sollte es beim Zugriff auf den Vektor irrelevant sein, ob intern die Koordinaten als (x,y) oder (r,phi) gespeichert werden.

# Zugriffskontrolle

- Innerhalb der Deklaration einer Klasse kann man mit den Schlüsselworten 'public' bzw. 'private' regeln, ob die definierten Variablen und Funktionen von außen sichtbar sind.

# Beispiel: Punkt in 2d.

- Siehe Beispielcode.

# Operator Overloading

- `Point& operator+=(const Point& rhs);`

- ```
  Point& Point::operator+=(const Point& rhs) {
    x_coord += rhs.x_coord;
    y_coord += rhs.y_coord;
    return *this;
  }
  ```

- Achtung: Man sollte sich unbedingt an die Konventionen der eingebauten Typen halten. Insgesamt ist das Operator Overloading eher eine fortgeschrittene Technik.

- Schauen Sie am besten nach bei http://en.cppreference.com/w/cpp/language/operator_comparison
  ```

# Templates

oder: Was bedeuten eigentlich die spitzen Klammern?

- ```
  template <typename T> class Point {
  public:
    void setX( T x ) {
      coordX = x;
    }
  private:
    T coordX;
  };
  ```
  ```

# Auch für 'Generic Programming'

- ```
  template <typename T> T sqr( T s ) {
    return s*s;
  }
  ```
  ```