Introduction to Scientific Programming

Part I: Matlab

Prof. Dr. Dirk Praetorius

Julian Streitberger, MSc

Institute of Analysis
and Scientific Computing

General information

Homepages + Accounts

- TISS homepage (login required)
  - https://tiss.tuwien.ac.at + search for lecture
  - Registration (mandatory)
- TUWEL homepage (login required)
  - https://tuwel.tuwien.ac.at/course/view.php?id=54001
  - Schedule & Course material
  - Weekly assignments (download & handing-in)
  - Forum to ask questions on lecture + exercises
- Server `lva.student.tuwien.ac.at`
  - Remote login via
    - `ssh -X name@lva.student.tuwien.ac.at`
    - `name` = e + student ID, e.g., `e12173378`
  - Requires valid VPN connection outside TU Wien
    - see TU.it (vpn)
  - Working on a remote server will be important later when you work on the VSC supercomputer
- If you have problems with your TU passwords, you must contact TU.it (TU accounts)

Course contents

- Quick introduction to Unix
  - needed to work on the VSC
- Quick introduction to MATLAB
  - needed for exercises on Numerics of ODEs and Numerics of PDEs
  - basics must be available until March 10
- Introduction to C
  - needed for Parallel Computing on the supercomputer VSC (Vienna Scientific Cluster)
  - full proficiency must be reached until May 10

Course organization

- The course will deal with hands-on programming of mathematical problems in MATLAB and C
- It accompanies the lectures *Numerics of ODEs*, *Numerics of PDEs*, and *Parallel Computing*
- The regular course takes place 6h per week
  - Friday 08:30–10:00: Presentation of homework
  - Friday 10:30–12:00: Joint work on theory
  - Friday 13:00–14:30: Hands-on programming
- Course start: March 03, 2023 (but only 2h)
  - only 08:30–10:00: Introduction to Unix
  - homework: make yourself familiar with MATLAB
- Course dates:
  - MATLAB: 10.03 + 17.03 + 24.03 + 31.03
  - C: 21.04 + 28.04 + 05.05 + 12.05
- Course end: May 12, 2023
- No exam, but grades according to homework
  - A positive grade requires the solution of $\geq 50\%$ of all exercises
  - Active contribution to class will have positive impact on the final grade
    - and non-contribution has negative impact!

General information

- start and quit MATLAB
- MATLAB online help
- m-files

- `help`

What is MATLAB?

- MATLAB (MATrix LABoratory) is a numeric computing environment that provides a full programming language together with an IDE (integrated development environment)
- 1970: developed for academic teaching
  - on Linear Algebra
  - on Numerical Mathematics
- Powerful tool for mathematicians and engineers
  - Numerical solution of mathematical problems

Why MATLAB?

- Easy development of mathematical algorithms
  - Most mathematical core functionality is already provided by MATLAB functions
    - e.g., `x = A\b` to solve $Ax = b$ via Gaussian elimination
- Matrices & vectors are built-in ingredients
- MATLAB allows the programmer to concentrate on mathematical key problems
- Therefore, MATLAB is the first choice for developing mathematical algorithms

Selling points of MATLAB

- Easy to learn
- Quick implementation of “strong” algorithms
- Built-in & powerful MATLAB editor
  - code folding
  - break points
  - real-time debugger
  - profiler
- MATLAB can be combined with C, C++, Fortran
  - first: development of algorithms in MATLAB
  - then: successive re-implementation for speed-up
    - e.g., in C
- Many (free) online tutorials
  - e.g., MATLAB Onramp
- Large and active community
  - MATLAB file exchange

Availability

- MATLAB is a commercial product
- Available on server `lva.student.tuwien.ac.at`
- Free student version for all students of TU Wien
  - http://www.sss.tuwien.ac.at/sss/mla/
  - https://de.mathworks.com/academia/tah-portal/technische-universitat-wien-30338656.html
- Free MATLAB clone: Octave
  - http://www.octave.org

Toolboxes

- Toolbox = library for MATLAB
- To solve special math problems, e.g.,
  - Symbolic Math Toolbox
  - Partial Differential Equations Toolbox
  - Statistics Toolbox
  - Parallel Computing Toolbox . . .
- Usually, one must buy MATLAB and toolboxes separately
- TU Wien has a quite strong bundle of toolboxes included in its campus license

Program

- A computer program (or, briefly, a program) is a collection of statements, written in a programming language, that performs a specific task when executed by a computer
  - Statement = declaration or instruction
    - Declaration = e.g., definition of variables
    - Instruction = 'do something'
  - Example: Search for a phonebook entry
  - Example: Compute the value of an integral

Algorithm

- An algorithm is a finite sequence of unambiguous operations which specifies how to solve a problem (or a class of problems)
  - Example: Compute the solution of a linear system of equations via Gaussian elimination
  - Example: Compute the zero of a quadratic polynomial using the quadratic formula
  - Note: A program is only an algorithm if it stops eventually
- There exist many algorithms to solve a problem
  - Not all algorithms are "good"
    - What does "good" mean? (see later)

Source code

- Text of a computer program written in a programming language
- It is processed step-by-step while executing or compiling
- In the easiest situation: sequentially
  - Line-by-line
  - From the top to the bottom

Programming language

- Programming languages can be classified into interpreted and compiled languages
- The interpreter executes source code line-by-line during the "translation"
  - i.e., translate and execute at the same time
  - e.g., Matlab, Java, PHP, Python
- The compiler "translates" the source code and produces a stand-alone program written in assembly language (executable)
  - i.e., first translate, then execute
  - e.g., C, C++, Fortran
- Alternative classification:
  - Imperative languages, e.g., Matlab, C, Fortran
  - Object-oriented languages, e.g., C++, Java
  - Functional languages, e.g., Lisp, Haskell

Start MATLAB

- Windows/Mac OS
  - graphical interface
- UNIX/Linux
  - Enter `matlab` in UNIX-Shell to start
    - note: UNIX is case sensitive
  - If possible: graphical interface
  - Or: text-only `matlab -nodisplay`
  - Or: text-based with figures `matlab -nodesktop`

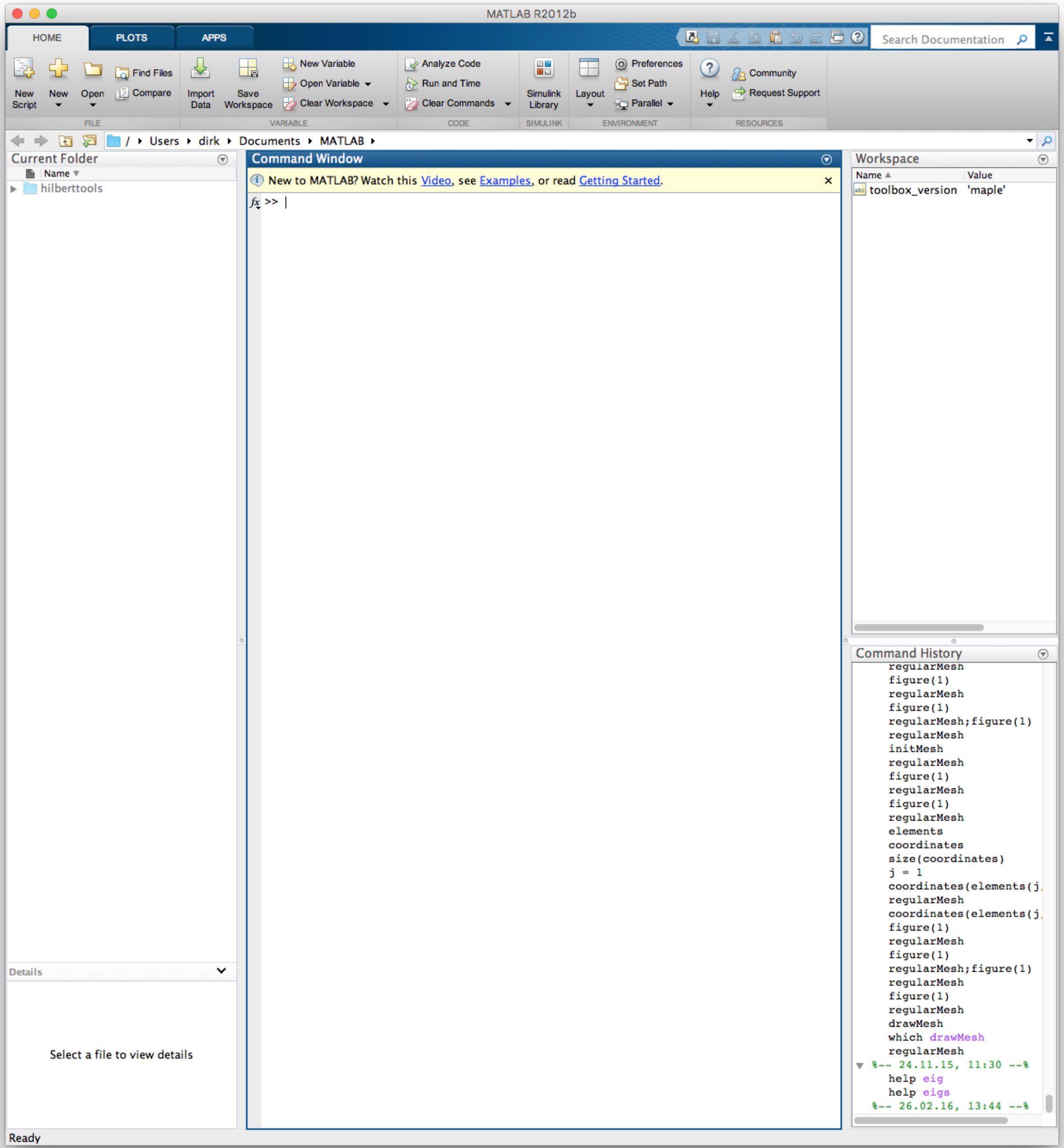
MATLAB Command Window

- Main window of MATLAB is Command Window
- MATLAB shell is a command line
  - The MATLAB shells knows the most important UNIX commands, e.g., `ls`, `mkdir`, ...
  - Further UNIX commands are available in MATLAB shell via `!command`
- MATLAB can be used like a pocket calculator

Quit MATLAB

- Enter `exit` into MATLAB shell

Screenshot MATLAB



- middle = MATLAB shell
- left = current directory
- upper/right = variables in workspace
- lower/right = last commands entered

Variable

- Variable = symbolic name (identifier) of a storage location (memory address) containing some quantity of information (value)

Variable names (identifiers)

- Made of letters, digits and underscore `_`
  - in MATLAB: maximum length = 63
  - in MATLAB: The first character must be a letter
- in MATLAB (and usually): Variable names are case-sensitive
  - i.e. `Var`, `var`, `VAR` are three different variables
- **Usual convention:** `lowercase_with_underscores`

Data types

- Usually, the data type of a variable must be declared before using it
- Elementary data types:
  - Floating-point numbers for values in $\mathbb{Q}$, $\mathbb{R}$, e.g., `double`
  - Integer for values in $\mathbb{N}$, $\mathbb{Z}$
  - Characters (letters), e.g., `char`

Working in Workspace

- Dynamic declaration of variables
  - i.e., variables are generated by first assignment
  - No formal declaration (and data type) is needed
- By default, all variables are `double`
- All arithmetic operations can be used
- End of statement by line feed
- Some statements provide an echo / output
  - that can be suppressed by use of a semicolon

Example

- Start MATLAB by entering `matlab` in Unix shell
- Create variables $a = 3$ and $b = 2.5$

```
>> a=3
```

  - leads to echo: `a = 3`

```
>> b=2.5;
```

  - No echo because of semicolon
- Compute $\sqrt{ab}$

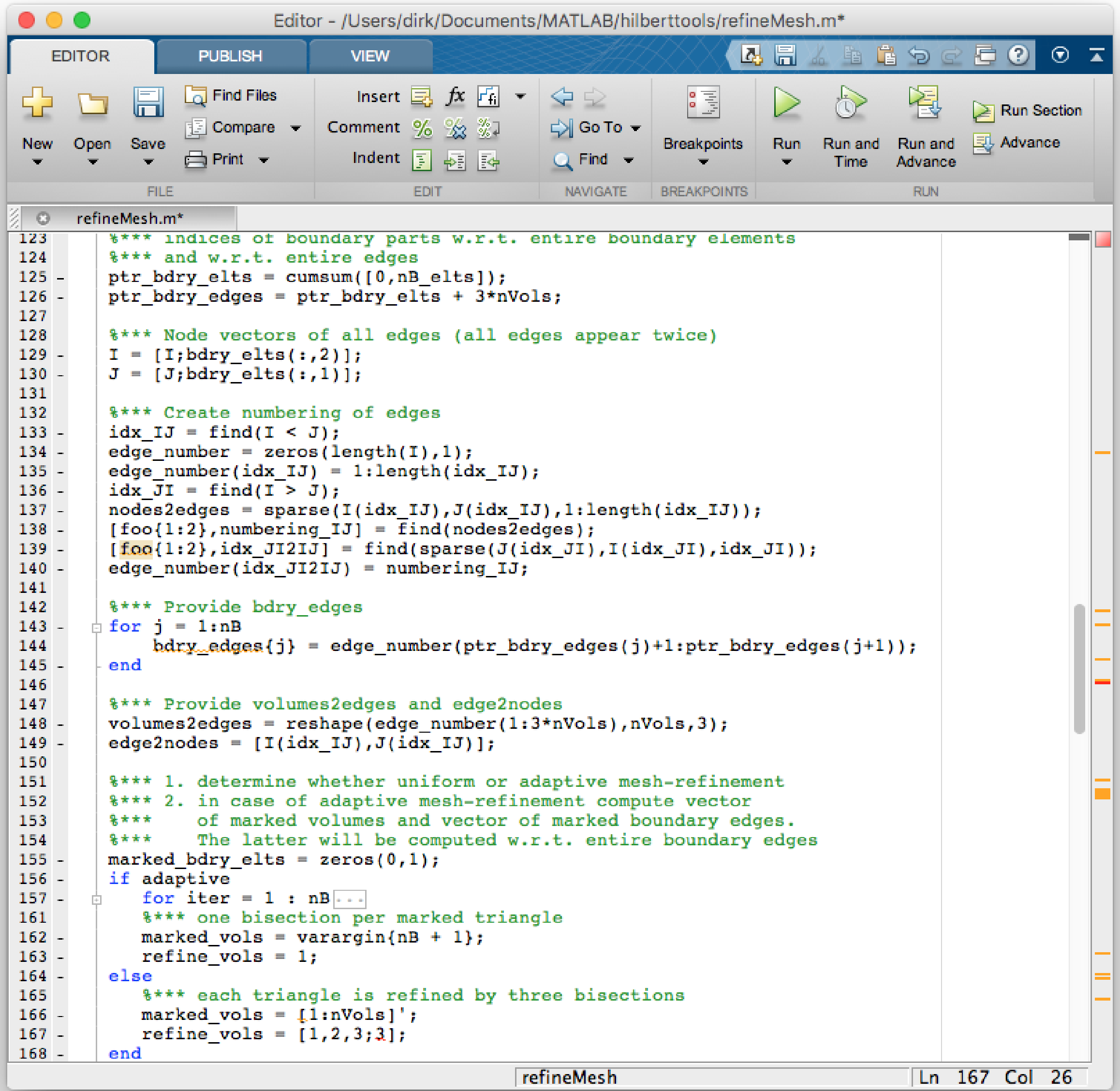
```
>> sqrt(a*b)
```

  - Leads to echo: `ans = 2.7386`
- Result of last computation is always stored in system variable `ans` ("answer")
- `sqrt` is square root function in MATLAB

MATLAB files

- MATLAB is an interpreted language
- MATLAB files are called `name.m`, and there are two types of MATLAB files `name.m`
  - Script files
    - They are executed by entering `name` at the MATLAB command line
    - They contain a sequence of statements that are sequentially executed
    - They modify the workspace memory (i.e., variables are changed)
    - They must not start with functions, but may contain local functions (which can only be called from inside the script)
  - Function files
    - First line of file declares the main function `function output = name(input)`
    - Function name `name` ⇔ file name `name.m`
    - End of function is given by `end`
    - Then, the function can be called from outside by `out = name(in)`
    - A file can contain further functions, but only the main function can be called from outside
    - All variables in functions are local variables, i.e., they can only be accessed during runtime of the function
- Executing is interrupted by `Crtl+C` during runtime

Screenshot MATLAB-Editor



- MATLAB command edit opens editor window
- left = line numbers (and possible break points)
- left = code folding for loops
  - line 143 (non-folded), line 157 (folded loop)
- right = real-time code check
  - green = code OK
  - orange = recommendations / improvements
  - red = errors

Help!

- MATLAB has built-in help / documentation of all functions
  - `help command`
    - text based in MATLAB shell
  - `doc command`
    - opens full documentation in help window
- full online documentation
  - http://www.mathworks.com/products/matlab/

Good to know

- MATLAB is *case sensitive* for names of variables and functions
- many MATLAB commands are actually m-files
  - Exception: all linear algebra functions are taken from the LAPACK library
    - so-called MATLAB built-in functions
  - `which command` returns directory + filename
    - One can copy and adapt `command` if needed
  - `type command` shows MATLAB code if m-file
  - `edit command` opens MATLAB code in editor
    - if you are working with MATLAB in a graphical environment
- Example: `lu`, `fft` (built-in), `pcg` (m-file)

Variables

- dynamic declaration
- dynamic memory allocation
- all variables are matrices
- complex numbers
- assignment operator
- semicolon

- `double`, `char`, `logical`
- `real`, `imag`
- `'...'`
- imaginary unit `i`

Dynamic declaration

- = is the assignment operator

- Variables are declared through first assigment
  - `var = 7;` assigns `var` the value 7
    - Data type is chosen according to assignment
    - Standard data type for all numbers is `double`
  - `var = 'hello';` assigns `var` the string `hello`
    - String = row vector of type `char`

- Each assignment updates the data type
  - e.g., `var = 7; var = 'hallo';` is admissible

Semicolon

- Lines ending with semicolon ; suppress the result output (so-called echo)
  - `var = 7` assignment with echo

    ≫ `var = 7`
  - `var = 7;` assignment without echo

Data types

- All numeric variables are a-priori `double`
  - according to IEEE 754 standard
  - i.e., floating point numbers with approximately 16 significant digits
- MATLAB provides also other numeric data types
  - e.g., `single`, `int8`, etc.
- Data type `char` for characters (letters)
- Data type `logical` for logical results
  - Takes only two values: 0 false, 1 true
  - Numeric values $\neq 0$ are interpreted as `true`

Complex numbers

- All MATLAB arithmetics is provided for complex numbers
  - imaginary unit is `i` or `1i` (and also `j` or `1j`)
  - `var = 7 + 5i;` assigns `var` the value $7 + 5i$
    - Note: Only here, `*` can be omitted
    - e.g., `5.5i` and `5.5*i` is both OK
  - real and imaginary part are stored as `double`
    - also other data types are possible

Further numeric data types

- MATLAB knows further numeric data types
  - `single`
    - according to IEEE 754 standard
    - i.e., floating point numbers with approximately 8 significant digits
    - corresponds to `float` in C/C++
  - `int8`, `int16`, `int32`, `int64`
    - for integers (with fixed bit length)
    - `int32` (4 Byte) corresponds to `int` in C/C++
  - `uint8` `uint16`, `uint32`, `uint64`
    - unsigned integer

- Will not be used in this lecture!

- MATLAB behaves differently than other programming languages. Therefore, I do not use other numeric data types than `double`.

  - `>> a = int8(3.7)`
    Echo: `a = 4`
    - C would return the value 3 (truncation instead of rounding)

  - `>> b = single(4)*double(3)`
    Echo: `b = 12`

  - But `b` has data type `single`!
    - C would compute the `double` value 12 and would cast it according to the prescribed data type of `b`!

Names of variables

- Variables have a unique name
  - MATLAB is case-sensitive
  - The maximal length is 63, further characters are ignored

- Admissible characters for names of variables and functions are
  - letters (no special letters like German ö)
  - digits
  - underscore

- A name must begin with a letter!

- Admissible names are, e.g.,
  - `A`, `a`, `A_p_e`, `a2Dsju__s`

- Non-admissible names are, e.g.,

  `3a`, `äöüß`, `some-Variable`

- Some names are pre-defined like `pi` or `sin` or `i`
  - They can be overwritten, but this is probably not a good idea in practice
    - e.g., `pi = 3;` would be admissible

  - One can delete a variable `var` via `clear var`
    - If the name was pre-defined, then you return to the original meaning (e.g., `pi`).

All variables are matrices!

- In MATLAB, all variables are matrices:
  - `var = 7;` declares a 1×1 matrix
  - row vector $=$ $1 \times N$ matrix
  - column vector $=$ $N \times 1$ matrix

Strings

- There are two ways to handle strings in MATLAB
  - either `"hello"` or `'hello'`
- `a = 'hello'` creates a 1×4 matrix of type `char`
- `a = "hello"` creates a 1×1 matrix which contains a string
  - This variant is better if you need a vector of strings that are not of the same length

Dynamic memory allocation

- First assigment to a non-existing entry extends a matrix accordingly
  - new `double` entries are initialized with `0`
  - new `char` entries are initialized with blanks
    - Example: `A = 1; A(3) = 7;` extends `A` to a 1×3 matrix $A = (1, 0, 7)$

- Numerical matrices have always a real and an imaginary part
  - The imaginary part is dynamically allocated if the first entry becomes complex (instead of real)
    - A complex matrix $A \in \mathbb{C}^{n\times n}$ is internally stored by two real $n \times n$ matrices
  - `real(var)` returns the matrix of the real parts
  - `imag(var)` returns the matrix of imaginary parts

- Try to avoid dynamic re-allocation of matrices, since this leads to unnecessarily high runtime
  - Each allocation calls the memory management of the operating system!
  - MATLAB stores matrices column-wise
    - since the MATLAB kernel is based on LAPACK, which is a Fortran library
    - Whenever a matrix gets new rows, essentially all old entries have to be copied and moved. This leads to a hidden runtime inefficiency!
  - Therefore, allocate matrices at the needed size, before you work with them!

Vectors

- Vectors
- Indexing of vectors and sub-vectors

- `double`, `char`
- `length`
- `sort`, `unique`, `find`
- `min`, `max`
- `abs`
- `sum`, `prod`
- `zeros`, `ones`, `rand`

- Operator `'` and `.'`
- `help strfun`, `doc strfun`

Vectors

- Create a row vector
  - `x = [1 2 3 4 5 6 7 8];`
  - `x = [1,2,3,4,5,6,7,8];`
    - Entries are separated by blanks or commas

- Create a column vector
  - `x = [1;2;3;4;5;6;7;8];`
    - Entries are separated by semicolons
  - `x = [1 2 3 4 5 6 7 8]';`
    - Operator `'` means $A \mapsto A^H := \overline{A}^T$
    - Operator `.'` means $A \mapsto A^T$

- If `x` is a vector, then `x(j)` is the j-th entry x_j
  - Indices run from $j = 1, \dots, N$ for $x \in \mathbb{C}^N$
  - The length of a vector is returned by `length(x)`
  - Access to a column vector can be done by `x(j,1)`
  - Access to a row vector can be done by `x(1,j)`

- Dynamic allocation
  - `x = 0;` creates 1×1 matrix $=$ scalar
  - `x(10,1) = 1;` extends `x` to column vector
    - `x` is 10×1 matrix $=$ column vector
    - all entries but `x(10)` are `0`
  - analogously for row vector

Allocating a vector

- `x = zeros(N,1);` creates a zero column vector
  - `x = zeros(1,N);` for row vector
- `x = ones(N,1);` creates col. vector with entries 1
  - `x = ones(1,N);` for row vector
- `x = rand(N,1);` for col. vector with random entries
  - `x = rand(1,N);` for row vector
- Function `rand` creates random numbers $\in [0,1]$
- Function `irand` creates random integer numbers
  - see `help irand`

Creating a row vector

- `x = start:stepsize:stop;` creates row vector
  - from `start` to $\leq$`stop` for `stepsize`> 0
  - from `start` to $\geq$`stop` for `stepsize`< 0
  - `stepsize` is optional, default stepsize is 1
    - e.g., `x = 1:8;` yields $x = (1,2,3,4,5,6,7,8)$
    - e.g., `x = 1:3:8;` yields $x = (1,4,7)$;
    - e.g., `x = 8:-3:1;` yields $x = (8,5,2)$;
    - nonsense creates `empty matrix`, e.g., `x = 6:2;`
- Further useful functions are `linspace` and `logspace`

Concatenating vectors

- `x` and `y` row vectors
  - `[x y]` concatenated row vector
  - Example: `x = [1 2 3]; y = [4 5];`
    - `[x y]` yields `[1 2 3 4 5]`

- `x` and `y` column vectors
  - `[x;y]` concatenated column vector
  - Example: `x = [1;2;3]; y = [4;5];`
    - `[x;y]` yields `[1;2;3;4;5]`

Indexing

- $x \in \mathbb{C}^N$ row or column vector
- `j` $\in 1, \ldots, N \quad \Rightarrow \quad$ `x(j)` returns x_j

- J index vector with entries $\in \{1, \ldots, N\}$
- `x(J)` is admissible and returns a vector
  - Length depends on length of J
  - Row or column shape depends on x
    - `x` column vector $\Rightarrow$ `x(J)` column vector
    - `x` row vector $\Rightarrow$ `x(J)` row vector

- Example `x = [1 8 2 7 3 6 4 5 1];`
- `J = [1 2 1 3]`
  - `x(J)` yields `[1 8 1 2]`
- `J = 1:2:9`
  - `x(J)` yields `[1 2 3 4 1]`
- `x(10)` returns an error, since `x` has length 9
  - `Index exceeds the number of array elements.`

Assignment

- $x \in \mathbb{C}^N$ row or column vector
- $J \in \mathbb{R}^n$ index vector with entries $\in \{1, \ldots, N\}$
- `x(J) = y` is admissible,
  - if `y` is a scalar
    - Then, assignment `x(j) = y` for all j ∈ J
  - if `y` is a vector of length n with the same shape as `x`, i.e., both are row vectors or both are column vectors
    - Then, `x(J(j)) = y(j)` for all $j = 1, \ldots, n$
- Example: `x = [1 2 3 4 5]`
  - `x([1 1 1 2]) = [4 3 2 1]`
  - yields `x = [2 1 3 4 5]`

Examples on MATLAB elegance

- Create row vector $x = (0, 1, 0, 1, ...) \in \mathbb{R}^N$

```
x = zeros(1,N);
x(2:2:N) = 1;
```

or

```
x = zeros(1,N);
x(2:2:end) = 1;
```

- keyword `end` is short-hand notation for `length(x)`

- Create row vector $x = (0, 1, 0, 2, 0, 3, 0, 4, ...) \in \mathbb{R}^N$

```
x = zeros(1,N);
x(2:2:end) = 1:N/2;
```

- Create $x = (N, 0, N-1, 0, N-2, 0, ..., 1) \in \mathbb{R}^{2N-1}$

```
x = zeros(1,2*N-1);
x(1:2:end) = N:-1:1;
```

- Take $x = (x_1, \ldots, x_N)$ and return $y = (x_N, \ldots, x_1)$

```
y = x(end:-1:1);
```

or

```
y = flip(x);
```

Useful functions on vectors

- `sort` : sorts a vector in ascending order
  - e.g., `x = [1 8 2 7 3 6 4 5 1];`
  - `sort(x)` yields $(1, 1, 2, 3, 4, 5, 6, 7, 8)$
- `unique` : sorts a vector in ascending order and eliminates multiple values
  - `unique(x)` yields $(1, 2, 3, 4, 5, 6, 7, 8)$
- `find` : returns those indices j, where the coefficients x_j satisfy a given condition
  - `find(x>3)` yields $(2, 4, 6, 7, 8)$
  - `x(find(x>3))` yields $(8, 7, 6, 4, 5)$
    - see also later $\rightarrow$ *logical indexing*
- `max`, `min` : returns maximum / minimum of a vector
  - plus indices, where these are attained
- `abs` : returns the vector of the absolute values
- `sum` : computes the sum of entries $\sum_{j=1}^{N} x_j$
- `prod` : computes the product of entries $\prod_{j=1}^{N} x_j$

Examples

- Compute the factorial $n!$
  - `factorial = prod(1:n);`
- Sort a vector in descending order
  - `x = sort(x); x = x(end:-1:1);`
  - or: `x(end:-1:1) = sort(x);`
  - or: `x = sort(x,'descend');`
- Eliminate the minimal entries of a vector
  - e.g., $x = (1, 2, 1, 2, 3, 1, 4, 5) \mapsto x = (2, 2, 3, 4, 5)$
  - `x = x( find(x > min(x)) );`
- Count the number of the minimal entries
  - e.g., $x = (1, 2, 1, 2, 3, 1, 4, 5) \rightarrow 3\times$ minimum
  - `count = length( find(x == min(x)) );`

An example function

```
function [mean,n] = meanDeviation(x,C)
  mean = sum(x)/length(x);
  idx = find( (x > mean + C) | (x < mean - C) );
  n = length(idx);
end
```

- What is the mean of a vector and how many entries are “far” from the mean?

Strings

- If we use row vectors of `char` to create strings,
  - then manipulation as for `double` vectors
    - `hello = 'Hello';`
    - `world = 'World!';`
    - `helloworld = [hello,' ',world];`
    - `helloworld(2:5)` yields `ello`

- If we use “real” MATLAB strings (e.g., `"hello"`),
  - then we need string functions
    - see `help strfun` or `doc strfun`
  - concatenation via `"Hello" + " " + "World"`

- Use `disp(text)` to print a string to the shell
  - for both types of strings

Matrices

- Matrices
- Indexing of matrices and sub-matrices

- `length`, `size`
- `zeros`, `ones`, `rand`, `eye`

- Operator `:`

- `help matfun`, `doc matfun`

Matrices

- We can define matrices row-wise (as for vectors)
  - `A = [1 2 3;4 5 6];` creates $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
- And we can replace the semicolon by a line break
  - ```
    A = [1 2 3
         4 5 6];
    ```
- Or we can define the same matrix column-wise
  - `A = [[1;4] [2;5] [3;6]];`
- Or we can define the same matrix block-wise
  - `A = [[1;4] [2 3;5 6]];`
  - The dimensions of the blocks must be consistent
- `C = [A B]` or `C = [A,B]` concatenates row-wise
  - creates $C \in \mathbb{R}^{M\times(N+n)}$ from $A \in \mathbb{R}^{M\times N}$, $B \in \mathbb{R}^{M\times n}$
  - with error if the dimensions mismatch:
    ```
    Error using horzcat
    Dimensions of matrices being concatenated
    are not consistent.
    ```
- `C = [A;B]` concatenates column-wise
  - creates $C \in \mathbb{R}^{(M+n)\times N}$ from $A \in \mathbb{R}^{M\times N}$, $B \in \mathbb{R}^{m\times N}$
  - with error if the dimensions mismatch:
    ```
    Error using vertcat
    Dimensions of matrices being concatenated
    are not consistent.
    ```
    ```

Allocating a matrix

- `A = zeros(M,N);` creates zero matrix $A \in \mathbb{R}^{M \times N}$
- `A = ones(M,N);` creates $A \in \mathbb{R}^{M \times N}$ with $A_{jk} = 1$
- `A = rand(M,N);` creates A with random $A_{jk} \in [0, 1]$
- `A = eye(N);` creates the identity matrix $A \in \mathbb{R}^{N \times N}$
- Dynamic memory allocation
  - `x = 1:3:12` yields row vector $x = (1, 4, 7, 10)$
  - `x(100,3) = 5` extends it to $x \in \mathbb{R}^{100 \times 4}$
    - only 5 non-zero entries
- Recall that changing the size of a matrix is a costly operator due to the internal storage and the memory management
  - Hence, it is recommended to allocate matrices in advance!

Indexing 1/3

- `A(j,k)` yields access to entry A_{jk}
  - with $j = 1, \dots, M$, $k = 1, \dots, N$ for $A \in \mathbb{C}^{M \times N}$

- Since matrices are stored columnwise as a vector, MATLAB allows access via `A(ℓ)` for $1 \le \ell \le MN$
  - e.g., $\texttt{A}(4) = 5$ for $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

- The dimensions of $A \in \mathbb{C}^{M \times N}$ are returned by
  - `[M,N] = size(A);`
  - `M = size(A,1);` and `N = size(A,2);`
  - `length(A)` yields $\max\{M, N\}$
  - `numel(A)` yields MN

Indexing 2/3

- MATLAB allows block-wise indexing of matrices
  - $A \in \mathbb{C}^{M \times N}$
  - J vector with entries $\in \{1, \ldots, M\}$
  - K vector with entries $\in \{1, \ldots, N\}$
  - Then, `A(J,K)` returns a matrix, whose dimension depends on the lengths of J and K
- `A = [1 2 3;4 5 6];` declares $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
- `A([1 2 1],[1 3])` yields $\begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 1 & 3 \end{pmatrix}$.
- Operator `:` stands for the full index set
  - `A(1,:)` yields the first row of A
  - `A(:,[1 2])` yields $\begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$.
- `A(:)` returns A as its storage vector
  - and yields $(1, 4, 2, 5, 3, 6)$
  - Note the columnwise storage of A
- The keyword `end` stands for the maximum index per dimension
  - `A(:,1:2:end)` yields `A(:,[1 3])`
  - since `end` is `size(A,2)` for this use

Indexing 3/3

- Indexing allows to cancel rows from a matrix
  - e.g., $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - `A(1,:) = [ ];` yields $A = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$
    - where `[ ]` is the empty matrix

- Indexing allows to cancel columns from a matrix
  - `A(:,2) = [ ];` yields $A = \begin{pmatrix} 1 & 3 \\ 4 & 6 \end{pmatrix}$
  - `A(:,[2 3]) = [ ];` yields $A = \begin{pmatrix} 1 \\ 4 \end{pmatrix}$

- Alternatively, use `A = A(I,J)` with index vectors `I`,`J`
  - e.g., $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
  - `A = A([1 2],:)` yields $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - `A = A([1 2],[2 3])` yields $A = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$

Useful functions on matrices

- Essentially all MATLAB functions are natively provided for matrices
  - e.g., `help sort` or `doc sort`
    - `[...] For vectors, sort(X) sorts the elements of X in ascending order. For matrices, sort(X) sorts each column of X in ascending order. [...]`

- The same applies for math functions, which usually return the matrix with entries $f(A_{ij})$
  - e.g., `exp`, `log`, `sin`, `cos`, `tan`

- Available functions from numerical linear algebra:
  - `help matfun`, `doc matfun`

Operators

- matrix arithmetics
- scalar-matrix arithmetics
- entry-wise arithmetics
- logical operators

- `+    -    *    /    \`
- `.*    ./    .\    .^`
- `^`

Matrix arithmetics 1/3

- All variables are matrices
- Therefore, the standard arithmetics is a matrix arithmetics
- +, - depends on the dimensions:
  - either matrix $\pm$ matrix (entry-wise)
  - or scalar $\pm$ matrix in each entry
  - or matrix $\pm$ scalar in each entry
  - Recall: Same dimension or one is a scalar!
    - Otherwise, you get an error:

```
Error using +
Matrix dimensions must agree.
```

- e.g., $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A + 10` yields $C = \begin{pmatrix} 11 & 12 \\ 13 & 14 \end{pmatrix}$
  - `C = 10 + A` yields $C = \begin{pmatrix} 11 & 12 \\ 13 & 14 \end{pmatrix}$
  - `C = 1 - A` yields $C = \begin{pmatrix} 0 & -1 \\ -2 & -3 \end{pmatrix}$
  - `C = A + B` yields $C = \begin{pmatrix} 11 & 22 \\ 33 & 44 \end{pmatrix}$

Matrix arithmetics 2/3

- * depends on the dimensions:
  - either matrix $*$ matrix (usual matrix product)
  - or scalar $*$ matrix in each entry
  - or matrix $*$ scalar in each entry
  - Recall: Fitting dimension or one is a scalar!

- e.g., $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A * 10` yields $C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = 10 * A` yields $C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A * B` yields $C = \begin{pmatrix} 70 & 100 \\ 150 & 220 \end{pmatrix}$

Matrix arithmetics 3/3

- Division `\` and `/` depends on the dimensions:
  - either matrix-scalar Division (entry-wise)
  - or solution of a linear system
    - for `x` scalar and `A` matrix, `x\A` $=$ `A/x`
    - for `X` and `A` matrices the order matters:
    - `X\A` $\mapsto X^{-1}A$
    - `A\X` $\mapsto A^{-1}X$
    - `X/A` $\mapsto XA^{-1}$
    - `A/X` $\mapsto AX^{-1}$
  - NOTE: `\` and `/` are also defined for non-invertible matrices via regression

- e.g., $A = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$
  - `A / 2` yields $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
  - `2 \ A` yields $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
  - $X = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$, $B = AX = \begin{pmatrix} 6 & 12 & 18 \\ 14 & 28 & 42 \end{pmatrix}$
  - `A \ B` yields `X`
  - `B / A` yields error

```
Error using  /
Matrix dimensions must agree.
```

Entry-wise arithmetics 1/2

- `+`, `-` are entry-wise addition/subtraction
  - matrix $\pm$ matrix for matrices of the same dim.
  - or scalar $\pm$ matrix
  - or matrix $\pm$ scalar

- `.*` is entry-wise multiplication
  - for matrices of the same dimension
    - i.e., `X.*A` yields matrix with entries $X_{jk}A_{jk}$
  - or scalar-matrix multiplication
  - or matrix-scalar multiplication
    - identical to `*` in the latter cases

- e.g., $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A*10` and `C = A.*10` yield $C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A*B` yields matrix product $C = \begin{pmatrix} 70 & 100 \\ 150 & 220 \end{pmatrix}$
  - `C = A.*B` yields $C = \begin{pmatrix} 10 & 40 \\ 90 & 160 \end{pmatrix}$

Entry-wise arithmetics 2/2

- `./` and `.\` entry-wise division
  - for matrices of the same dimension
  - or scalar-matrix division
  - or matrix-scalar division

- `.^` entry-wise power
  - for matrices of the same dimension
    - i.e., `X.^A` yields matrix with entries $X_{jk}^{A_{jk}}$
  - or scalar-matrix `x.^A` yields matrix with $x^{A_{jk}}$
  - or matrix-scalar `X.^a` yields matrix with X_{jk}^{a}

- `^` normal matrix power
  - matrix `^` scalar is only defined for quadratic matrices!
    - `A^3` means `A*A*A`

- e..g., $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
  - `C = A^2` yields $C = \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}$
  - `C = A.^2` yields $C = \begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}$
  - `C = 2.^A` yields $C = \begin{pmatrix} 2 & 4 \\ 8 & 16 \end{pmatrix}$

Arithmetics instead of loops

- Often, loops from other programming languages can be avoided in MATLAB by means of vector arithmetics (and appropriate vector functions)
- Usually, this is more efficient, since built-in functions are optimized
  - i.e., same computational cost as for loops
  - but much faster due to precompiled kernel-code

Example: Supremum norm

- $\|x\| = \max_{j=1,\ldots,N} |x_j|$ on $\mathbb{R}^N$, e.g., in C

```
int j = 0;
double tmp = 0;
double norm = fabs(x[0]);
for (j=1; j<N; ++j) {
  tmp = fabs(x[j]);
  if (tmp > norm) {
    norm = tmp;
  }
}
```

- MATLAB is closer to mathematical thinking:
  - Create vector of absolute values `abs(x)`
  - Take the maximum of this vector
  - `result = max(abs(x));`
- Computational cost is still $\mathcal{O}(N)$!

Example: Scalar product

- $x \cdot y = \sum_{j=1}^{N} x_j y_j$ is the scalar product on $\mathbb{R}^N$
  - We interpret this as a matrix-matrix product
    - with $(1 \times N)$ matrix `x` and $(N \times 1)$ matrix `y`
  - If `x`, `y` are row vectors, we get `result = x*y';`
  - or: `result = sum(x.*y);`
    - which also works if `x`, `y` are column vectors

Example: Frobenius norm

- The Frobenius norm reads $\|A\| = \left(\sum_{j,k=1}^{N} A_{jk}^2\right)^{1/2}$
- in C:

```
int j,k;
double norm = 0;
for (j=0; j<N; ++j) {
  for (k=0; k<N; ++k) {
    norm = norm + A[j][k]*A[j][k];
  }
}
result = sqrt(norm);
```

- in MATLAB: Square all entries and sum it up
  - `result = sqrt( sum( sum(A.^2, 2) ) );`
  - `result = sqrt( sum(A(:).^2) );`
  - `result = norm(A,'fro');`

Example: Evaluate polynomial

- Consider the polynomial $p(x) = \sum_{j=0}^{N} a_j x^j$
  - suppose that `a` is a row vector, `x` is a scalar
  - Recall that MATLAB indices are $j = 1, 2, \ldots$
    - i.e., $N = $ `length(a)` $- 1$
  - `result = sum( a.*(x.^[0:length(a)-1]) );`
  - or: `result = a*(x.^[0:length(a)-1])';`

Example: Vandermonde matrix

- Given $x \in \mathbb{R}^n$, create $X = \begin{pmatrix} x_1 & x_1^2 & x_1^3 & \cdots & x_1^n \\ x_2 & x_2^2 & x_2^3 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n & x_n^2 & x_n^3 & \cdots & x_n^n \end{pmatrix}$
- Idea: `X` $= \begin{pmatrix} x_1 & x_1 & \cdots & x_1 \\ x_2 & x_2 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_n & \cdots & x_n \end{pmatrix}$.^ $\begin{pmatrix} 1 & 2 & \cdots & n \\ 1 & 2 & \cdots & n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & \cdots & n \end{pmatrix}$
- With column vector `x`, this is done as follows

```
n = length(x);
X = x * ones(1,n);
X = X .^ ( ones(n,1) * (1:n) );
```

Logical operators

- logical NOT $\sim$
- logical OR | (general) , || (short circuit, only scalars)
- logical AND & (general), && (short circuit, only scalars)

- less <
- less or equial <=
- greater >
- greater or equal >=
- equal ==
- unequal $\sim$=

- These operators apply entry-wise
  - for matrices of the same dimension
  - or for matrix-scalar or scalar-matrix
- They return the matrix with the corresponding logical results

- `any(a > b)` is an iterated OR for vectors
- `all(a > b)` is an iterated AND for vectors
  - For matrices, see `help any` and `help all`

Logical indexing

- `x` vector of length N
- `J` `logical` vector of length N with `J(k)` $\in \{0, 1\}$
- Then, `x(J)` is sub-vector of all `x(k)` with `J(k)==1`
- e.g., `x = [1 8 2 7 3 6 4 5 1];`
  - `x > 3` yields `[0 1 0 1 0 1 1 1 0]`
    - resulting data type is `logical`, *not* `double`
  - `x(x > 3)` yields `[8 7 6 4 5]`
- NOTE: Indexing with `logical` vs. `double`
  - `J = [1 1 1 1 1 1 1 1 1];` (hence `double`)
    - `x(J)` yields `[4 4 4 4 4 4 4 4 4]`
    - `x(logical(J))` yields `[1 8 2 7 3 6 4 5 1]`
  - Note the error for `x([0 1 0 1 0 1 1 1 0])`
    - `Subscript indices must either be real positive integers or logicals.`
- `find` returns indices of non-zero entries of a vector
  - `x > 3` yields `[0 1 0 1 0 1 1 1 0]`
  - `find(x > 3)` yields `[2 4 6 7 8]`
- Example: How many entries of `x` are > 3?
  - `count = length( find(x > 3) );`
  - or: `count = sum(x > 3);`
  - or: `count = nnz(x > 3);`

Examples

- Has $x \in \mathbb{R}^N$ at least one positive entry?
  - `answer = any( x > 0 );`
- Has $x \in \mathbb{R}^N$ only positive entries?
  - `answer = all( x > 0 );`
- Replace all entries of $x \in \mathbb{R}^N$ with $|x_j| > C$ by sign$(x_j)\,C$
  - `x( x > C ) = C;`
  - `x( x < -C ) = -C;`
- Delete minimal entries from $x \in \mathbb{R}^N$
  - `x = x( x > min(x) );`
  - or: `x( x == min(x) ) = [];`

Functions

- Structure of a MATLAB function
- Comment lines
- Call by Value
- local and global variables

- `function`
- `%`
- `global`
- `return`

Structure of a function

```
function output = name(input)

% This text will be shown if "help name" is input
% at the MATLAB prompt. Therefore, this text
% should comment on
% - How can the function be called?
% - What will be done?
% - What is the necessary (and optional) input?
% - What is the output?
% This is the final line of the help text.

% After the empty line, one should comment on
% author / source / copyright / last modified etc.

% Here comes the function body (ended by "end").

end

function y = subfunction(x)

% This is a subfunction that can only be called
% from functions inside this file. There should
% be comments on what is done / what is input.

end
```

- `%` indicates a comment, i.e., the text after `%` until the end of line is only for the programmer and will not be executed by the MATLAB interpreter
- The first contiguous block of comment lines right after `function` will be shown when `help name` is input at the MATLAB prompt
- Line numbers are not part of the source code

Possible function declarations

- **now:** fixed number of input and output parameters
- `function name` or `function name()`
  - no input parameters
  - no output parameters
  - called by: `name;` or `name();`
- `function name(in1,in2,...)`
  - finitely many input parameters indicated by `...`
  - no output
  - called by: `name(in1,in2,...);`
- `function out = name`
  - no input [optional `()` for declaration and call]
  - one single output parameter
  - called by: `out = name;`
- `function out = name(in1,in2,...)`
  - finitely many input parameters indicated by `...`
  - one single output parameter
  - called by: `out = name(in1,in2,...);`
- `function [out1,out2,...] = name`
  - no input [optional `()` for declaration and call]
  - finitely many output parameters indicated by `...`
  - called by: `[out1,out2,...] = name;`
- `function [out1,out2,...] = name(in1,in2,...)`
  - finitely many input parameters indicated by `...`
  - finitely many output parameters indicated by `...`
  - called by: `[out1,out2,...] = name(in1,in2,...);`

Call by value

- MATLAB employs **call by value**, i.e., functions get all input as values and store these in local variables (with dynamic declaration)

- All variables that are declared in the signature as well as the body of a function are local variables
  - If a function changes a variable, this has no effect for the calling code (or the workspace)
    - i.e., `fct(var);` does not change the value of `var` for the calling code
  - All variables that are declared in a function lose their lifetime when the function terminates

- There is no **call by reference** for standard MATLAB functions
  - If a function should change the value of a variable, then it must return this value
    - i.e., one must employ `var = fct(var);`

Output and return value

- The names of the output variables are fixed by the function declaration
  - The data type of the output is dynamic
- The return value of an output variable is the value that is assigned, when the function terminates
- A function terminates if the interpreter meets the function's `end` or when it meets the keyword `return`
  - Unlike other programming languages, `return` does not have any argument

Keyword `global`

- MATLAB knows global variables, but these should only be used for debugging
  - Global variables must be declared by `global var` in calling code and called function `fct`
  - And `var` must not be an input parameter of `fct`
  - Then, changes of `var` in `fct` also change the value of `var` in the calling code.

Example: supremum norm

```
function result = supremumNorm(x)

% This function computes the supremum norm
%
%   || x || = max_{j=1...N} |x_j|
%
% of a vector x in C^N.
%
% RESULT = supremumNorm(X) returns the supremum
% norm of X, where X is a numeric row or
% column vector.

% author: Dirk Praetorius
% last modified: 06.03.2022

result = max(abs(x));
```

- This is also provided by MATLAB as `norm(x,Inf)`

Example: evaluate polynomial

```
function px = evaluatePolynomial(a,x)

% This function evaluates a polynomial p(x) that
% is given in terms of its coefficient vector.
%
% PX = evaluatePolynomial(A,X), where A is a row
% vector and X is a scalar. The return value is
%
%   PX = sum(j=1...length(A)) A(j)*X^(j-1)
%
% i.e., A(1) is the coefficient in front of
% the smallest power X^0 and p(x) is of
% degree n = length(A)-1.

% author: Dirk Praetorius
% last modified: 06.03.2022

px = a * (x.^[0:length(a)-1])' ;
```

- MATLAB employs indexing $j = 1, ..., N+1$
- $p(x) = \sum_{j=1}^{N+1} a_j x^{j-1}$ is a polynomial of degree N
  - Given: $x \in \mathbb{R}$ and $a \in \mathbb{R}^{N+1}$
  - Goal: compute $p(x)$

Ex: polynomial interpolation 1/4

- Given: Values of a continuous function $f : [a, b] \to \mathbb{R}$
- Sought: Polynomial p of degree N with $p \approx f$
- Fix p by $p(x_j) = f(x_j)$ for $j = 0, \dots, N$
  - with $x_0, \dots, x_N \in [a, b]$ being pairwise different

- Mathematical questions:
  - Existence and uniqueness of p?
  - How to compute p?

- Consider the space $\mathbb{P}_N = \{p \text{ poly. of degree } \leq N\}$
  - i.e., $p \in \mathbb{P}_N$ can be written as $p(x) = \sum_{j=0}^{N} a_j x^j$
  - clearly: $\mathbb{P}_N$ is a vector space with $\dim \mathbb{P}_N \leq N+1$
  - next step: $\dim \mathbb{P}_N \geq N+1$ by construction

- Define $L_j(x) := \displaystyle\prod_{\substack{k=0\\k\neq j}}^{N} \frac{x - x_k}{x_j - x_k}$ for all $j = 0, \dots, N$
  - clearly: $L_j \in \mathbb{P}_N$, $L_j(x_j) = 1$, $L_j(x_k) = 0$ for $k \neq j$
  - next: $\{L_0, \dots, L_N\} \subseteq \mathbb{P}_N$ are linearly independent
    - Let $a \in \mathbb{R}^{N+1}$ with $0 = \displaystyle\sum_{j=0}^{N} a_j L_j$
    - Then, $0 = \displaystyle\sum_{j=0}^{N} a_j L_j(x_k) = a_k$ for all k
    - thus: $a = 0$, which proves linear independence

Ex: polynomial interpolation 2/4

- $\mathbb{P}_N = \{p \text{ polynomial of degree } \leq N\}$
  - $\dim \mathbb{P}_N \leq N + 1$
  - $\{L_0, \ldots, L_N\} \subseteq \mathbb{P}_N$ linearly independent
  - hence: $\dim \mathbb{P}_N = N + 1$

- Consider the evaluation $Tp := \big(p(x_0), \ldots, p(x_N)\big)$
  - $T : \mathbb{P}_N \to \mathbb{R}^{N+1}$
  - clearly: T linear
  - goal: T is surjective
    - show that: $\forall a \in \mathbb{R}^{N+1} \exists p \in \mathbb{P}_N : \quad Tp = a$
    - Given $a \in \mathbb{R}^{N+1}$, define $p := \sum_{j=0}^{N} a_j L_j$
    - Then, $p(x_k) = \sum_{j=0}^{N} a_j L_j(x_k) = a_k$

- One main theorem of Linear Algebra:
    - dim(domain) = dim(range) + dim(nullspace)
- here: $\dim \mathbb{P}_N = \dim T(\mathbb{P}_N) + \dim \ker(T)$
- hence: $\dim \ker(T) = 0$
- hence: T is injective and hence even bijective
    - overall: $\forall a \in \mathbb{R}^{N+1} \exists! p \in \mathbb{P}_N : \quad Tp = a$

Ex: polynomial interpolation 3/4

- $T : \mathbb{P}_N \to \mathbb{R}^{N+1}$, $Tp := \big(p(x_0), \ldots, p(x_N)\big)$
  - linear and bijective

- Given: Values of a continuous function $f : [a, b] \to \mathbb{R}$
  - Then, there exists a unique $p \in \mathbb{P}_N$
    - with $p(x_j) = f(x_j)$ for all $j = 0, \ldots, N$

- Question: How to compute p?

- Consider the monome basis $p(x) = \sum_{j=0}^{N} a_j x^j$
  - then: $a \in \mathbb{R}^{N+1} \mapsto \big(p(x_0), \ldots, p(x_N)\big) = \mathbf{T}a$
  - clearly: The matrix takes the following form

$$\mathbf{T} = \begin{pmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ x_N^0 & x_N^1 & x_N^2 & \cdots & x_N^N \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{pmatrix}$$

- from Linear Algebra: $\mathbf{T}$ ist invertible

- Define $b := \big(f(x_0), \ldots, f(x_N)\big) \in \mathbb{R}^{N+1}$
  - Define $a := \mathbf{T}^{-1} b$
  - Then, $p(x) := \sum_{j=0}^{N} a_j x^j$ is the unique $p \in \mathbb{P}_N$ with $p(x_j) = f(x_j)$ for all $j = 0, \ldots, N$

Ex: polynomial interpolation 4/4

```
function a = fitpol(b,x)

% For given vectors X and B in R^n with pairwise
% different entries X(j), this function computes
% the coefficient vector A of the unique Lagrange
% interpolation polynomial of degree n-1.
%
% A = fitpol(B,X), where A, B, and X are column
% vectors of the same length n. Then, the
% polynomial
%
%   p(x) = sum(j=1...length(B)) A(j)*x^(j-1)
%
% satisfies
%
%   p(X(j)) = B(j) for all j = 1,...,n

% author: Dirk Praetorius
% last modified: 07.03.2022

n = length(x);
T = (x * ones(1,n)) .^ (ones(n,1) * (0:n-1));
a = T\b;
```

- $$\mathbf{T} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{N-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ x_N^0 & x_N^1 & x_N^2 & \cdots & x_N^{N-1} \end{pmatrix} \in \mathbb{R}^{N\times N}$$
- $a = \mathbf{T}^{-1}b$

Function handles

- Recall that in MATLAB the use of a function name `fct` calls the function.
- If a function should be the input argument of another function, then one must use so-called function handles
- If `fct` is a function, then `@fct` provides the function handle.
  - Let `fct` be of the type `output = fct(input)`
  - Then, `ptr = @fct` assigns the function handle of `fct` to `ptr`
  - One can call `fct` also by `output = ptr(input)`
    - cf. function pointers in C
- In particular, `@fct` can be the input argument of another function (e.g., an implementation of Newton's method)

Passing functions to functions

- Functions can take other functions as input by taking either their name or their function handle
- In either case, one can evaluate this argument via `output = feval(fct,input)`
  - e.g., `y = feval('sin',x);`
  - e.g., `y = feval(@sin,x);`

Anonymous functions

- Sometimes it is useful to create simple functions just in one line of code
- Using the function handle operator `@`, this is done as follows:

  `f = @(input) output`

  - Then, `f` takes a list of input parameters `input`
  - and returns the result `output`
    - e.g., `f = @(x) x.^2+exp(x)-2;`
    - defines $f(x) = x^2 + \exp(x) - 2$
    - e.g.,`f = @(x,y) x.*exp(-x.^2-y.^2);`
    - defines $f(x,y) = x \cdot e^{-(x^2+y^2)}$
- These so-called anonymous functions are used as normal functions
  - i.e., `output = f(input)`
  - Formally, they define a function handle

Conditionals

- Conditional statement

- `if - elseif - else - end`
- `switch - case - otherwise - end`

Simple conditionals

```
if a > b

  % The following code is executed if and only
  % if the condition (a > b) is evaluated true
  disp('a > b');

elseif a == b

  % MATLAB allows arbitrarily many "else if"
  disp('a == b');

else

  % if none of the preceding conditions was
  % evaluated true, then this case is executed
  disp('a < b');

end
```

- Unlike other programming languages, MATLAB does not enforce brackets around conditions
  - But `if (a > b)` is more readable than `if a > b`
- The conditional code is indicated by keywords
  - `if` – `elseif` – `else` – `end`
  - All cases are exclusive
- The branches with `elseif` and `else` are optional

Example: adding polynomials

```
function c = addPolynomials(a,b)

% Compute the coefficient vector of the polynomial
%
%   (p+q)(x) = sum(ell=1...) C(ell) * x^(ell-1)
%
% where the polynomials
%
%   p(x) = sum(j=1...M) A(j) * x^(j-1)
%   q(x) = sum(k=1...N) B(k) * x^(k-1)
%
% are given in terms of their coefficient vectors.
%
% C = addPolynomials(A,B) returns C, where A and
% B are either both column vectors or row vectors.

% author: Dirk Praetorius
% last modified: 07.03.2022

m = length(a);
n = length(b);
if (m < n)
  c = b;
  c(1:m) = c(1:m) + a;
else
  c = a;
  c(1:n) = c(1:n) + b;
end
```

- given: $p(x) = \sum_{j=1}^{M} a_j x^{j-1}, \quad q(x) = \sum_{k=1}^{N} b_k x^{k-1}$
- sought: $(p+q)(x) = \sum_{\ell=1}^{\max\{M,N\}} c_\ell x^{\ell-1}$

Multi-case conditionals

```
switch x
  case 1
    disp("x==1")
  case {2,3}
    disp("x==2 or x==3")
  otherwise
    disp("x~=1,2,3")
end
```

- Variable `x` must be a scalar or a string
  - Optionally, one may also write `switch(x)`
- `case` provides conditions on the value of `x`
  - The code after `case` is executed if `x` has the stated value
  - Multiple equivalent values are possible via `{...}`
  - All cases are exclusive
- The code after `otherwise` is excuted if none of the preceding cases was met.
  - `otherwise` is optional
- The above code can also be stated with `if ... end`

```
if (x==1)
  disp("x==1");
elseif (x==2 || x==3)
  disp("x==2 or x==3");
else
  disp("x~=1,2,3");
end
```

Example: days per month

```
function days = daysPerMonth(month,year)

switch(month)
  case {1,3,5,7,8,10,12}
    days = 31;
  case {4,6,9,11}
    days = 30;
  case 2
    if (mod(year,400) == 0)
      days = 29;
    elseif (mod(year,100) == 0)
      days = 28;
    elseif (mod(year,4) == 0)
      days = 29;
    else
      days = 28;
    end
  otherwise
    days = -1;
end
```

- Determine the number of days per month
- A year is a leap year (and the February has 29 days) if the year is divisible by 4
  - Exeption: The year is not a leap year if it is divisible by 100 (& 4)
  - Another Exeption: The year is nevertheless a leap year if it is divisible by 400 (& 100 and 4)
- `mod(x,y)` returns the remainder after division of two integers `x` and `y`
  - see `help mod` for arbitrary `double` input

Loops

- Counting loop `for`
- Conditional loop `while`

- `for - end`
- `while - end`
- `break`

for loop 1/3

```
out = 0;
for j = rowvector
  out = out + j;
end
```

- The `for` loop iterates some code for `length(rowvector)` times
  - For the 1st iteration, it holds `j` = `rowvector(1)`
  - For the 2nd iteration, it holds `j` = `rowvector(2)`
  - etc.
- By definition, there is no iteration if `rowvector` is empty
- The iteration terminates after `length(rowvector)` iterations
- `break` can be used to prematurely terminate the loop at any time
- `break` applies only to the current (innermost) loop
  - and cannot be used to terminate nested loops

```
out = 0;
for j = rowvector
  out = out + j;
  j = 42;
  rowvector = 42;
end
```

- The code leads to the same result as above
- `j` takes the entries of `rowvector` as values, where `rowvector` is fixed before the iteration starts

for loop 2/3

```
result = 0;
for j = 1:2:100
  result = result + j^2;
end
disp(result)
```

- Often: `rowvector` takes the form `start:step:end`
- Example: Compute $\sum_{\substack{j=1 \\ j \text{ odd}}}^{100} j^2 = 166650$
- But: Such a computation can often be replaced by vector arithmetics, which is more efficient in MATLAB
  - e.g., `result = sum( (1:2:100).^2 );`

for loop 3/3

```
A = [1 2 ; 3 4 ; 5 6 ; 7 8];
for j = A
  j
end
```

- If `for` is applied to a matrix A (instead of a row vector), then `for` iterates over the columns of A
- It is an often made mistake to apply a `for` loop to a column vector (instead of a row vector)
- Output:

```
j =

     1
     3
     5
     7


j =

     2
     4
     6
     8
```

Example: product of polynomials

```
function c = multiplyPolynomials(a,b)

% Return the coefficient vector of the polynomial
%
%   r = \sum_{\ell=1}^{m+n-1} c_\ell x^{\ell-1}
%
% obtained by multiplication r = p*q of
%
%   p = \sum_{j=1}^m a_j x^{j-1}
%   q = \sum_{k=1}^n b_k x^{k-1}
%
% C = multiplyPolynomials(A,B) takes the
% coefficient vectors A and B and returns the row
% vector C of the coefficients of the product
% polynomial r = p*q

m = length(a);
n = length(b);
c = zeros(1,m+n-1);

for j = 1:m
  for k = 1:n
    c(j+k-1) = c(j+k-1) + a(j)*b(k);
  end
end
```

- Compute the product $r = pq$ of two polynomials
  - $a \in \mathbb{C}^m$, $p(x) = \sum_{j=1}^{m} a_j x^{j-1}$, degree($p$) $= m - 1$
  - $b \in \mathbb{C}^n$, $q(x) = \sum_{k=1}^{n} b_k x^{k-1}$, degree($q$) $= n - 1$
  - note: $c \in \mathbb{C}^{m+n-1}$, $r(x) = \sum_{\ell=1}^{m+n-1} c_\ell x^{\ell-1}$
    - with $c_\ell = \sum_{j+k=\ell+1} a_j b_k$
- This is already provided by MATLAB as `conv`

`while` loop

- Syntax:

```
while condition
   body
end
```

- The `while` loop iterates some code as long as `condition` (of type `logical`) remains true,
  - There is no iteration if `condition` is false
- Unlike other programming languages, MATLAB does not enforce brackets around conditions
  - But `while (condition)` is more readable

Example: Euclidean algorithm

```
function a = euclid(a,b)

% Compute the greatest common divisor (gcd) of
% two positive integers by means of Euclidean
% algorithm which is based on
%   gcd(A,B) = gcd(B,A)
% and, for A>B,
%   gcd(A,B) = gcd(A-B,B)
%
% RESULT = EUCLID(A,B) returns the gcd of two
% positive integer scalars A and B

 while (a~=b)
  if (a<b)  % guarantee a>=b
   tmp = a;
   a = b;
   b = tmp;
  end
  a = a-b;
 end
end
```

- The Euclidean algorithm computes the greatest common divisor (gcd) of two positive integers `a` and `b`
  - It exploits the observations that
    - $\text{gcd}(a, b) = \text{gcd}(b, a)$
    - $\text{gcd}(a, b) = \text{gcd}(a - b, b)$ if $a > b$
    - $\text{gcd}(a, a) = a$
- This is already provided by MATLAB as `gcd`

Example: binary search

```
function index = binsearch(vector,query)

% Seek an index J such that the J-th entry X(J)
% of a vector X coincides with a sought query Q.
% Return -1 if no such index exists. The vector X
% is required to be sorted in ascending order
%
% J = binsearch(X,Q) with X being a numeric
% vector and Q being a scalar.

lower = 1;
upper = length(vector);
while (lower <= upper)
  index = floor(0.5*(lower + upper));
  if (vector(index) == query)
    return
  elseif (vector(index) > query)
    upper = index - 1;
  else
    lower = index + 1;
  end
end
index = -1;
```

- Suppose that `vector` is sorted in ascending order and that searching for equality makes sense (e.g., integers)
- Find an index `j` with `vector(j) == query`
  - Return -1 if no such index exists
- Use bisection as for searching a dictionary
  - Consider the middle entry of `vector` and reduce the search to a vector of half length

“while” vs. “repeat ... until”

- Recall the syntax

```
while (condition)
   % body
end
```

- where `while` iterates as long as `condition` is true
- However, most mathematical algorithms have a termination criterion `done`
  - i.e., the algorithm is terminated as soon as `done` is true
  - Logically, `done` is the negation of `condition`
- This is easily implemented by use of a formally infinite loop to avoid errors if `done` is complicated
- Suggested syntax:

```
while true
   if (done)
     break
   end
   % body
end
```

Example: Heron's method

- Realization via negation of termination condition

```
function xn = heron(x,tol)

% XN = heron(X,TOL) realizes the Heron algorithm
% for the computation of sqrt(X). For a given
% tolerance TOL > 0, it returns the first iterate
% XN such that | XN^2 - X | <= TOL.

xn = x;
while ( abs(xn^2 - x) > tol )
  xn = 0.5*(xn + x/xn);
end
```

- Realization via infinite loop and break

```
function xn = heron(x,tol)

% XN = heron(X,TOL) realizes the Heron algorithm
% for the computation of sqrt(X). For a given
% tolerance TOL > 0, it returns the first iterate
% XN such that | XN^2 - X | <= TOL.

xn = x;
while true
  xn = 0.5*(xn + x/xn);
  if ( abs(xn^2 - x) <= tol )
    break
  end
end
```

- Define $x_0 := x$ and $x_{n+1} := (x_n + x/x_n)/2$
- Then: There holds convergence $x_n \to \sqrt{x}$
- Given $\varepsilon > 0$, return the first x_n with $|x_n^2 - x| \leq \varepsilon$

Example: bisection method

```
function [c,fc] = bisection(f,a,b,tol)

% Given a continuous real-valued function F on a
% compact interval [A,B] with F(A)*F(B) <= 0, the
% intermediate value theorem guarantees a root
% F(X) = 0 in [A,B]. Given a tolerance tol > 0,
% the bisection algorithm returns X0 such that
%|X - X0| <= tol and |F(X0)| <= tol
%
% [X0,FX0] = BISECTION(F,A,B tol) takes the
% function handle of F, the scalar endpoint A, B
% of the interval [A,B], and the scalar tolerance.
% It returns the approximate root X0 as well as
% the corresponding function value F(X0).

fa = feval(f,a);
fb = feval(f,b);

while true
  c = (a+b)/2;
  fc = feval(f,c);
  if ( abs(b-a)<=2*tol && abs(fc)<=tol )
    return
  elseif ( fa*fc <= 0 )
    b = c;
    fb = fc;
  else
    a = c;
    fa = fc;
  end
end
```

- Adapts the idea of binary search for continuous `f`
- The input `f` is either a function handle or the name of a function (as a string)

Basic graphics

- Export of figures as EPS-files

- `plot`
- `figure, clf, close`
- `hold on, hold off`
- `axis, axis on, axis off`
- `axis equal, axis tight, axis square`
- `grid on, grid off`
- `box on, box off`
- `title, xlabel, ylabel, legend`
- `text`
- `print`

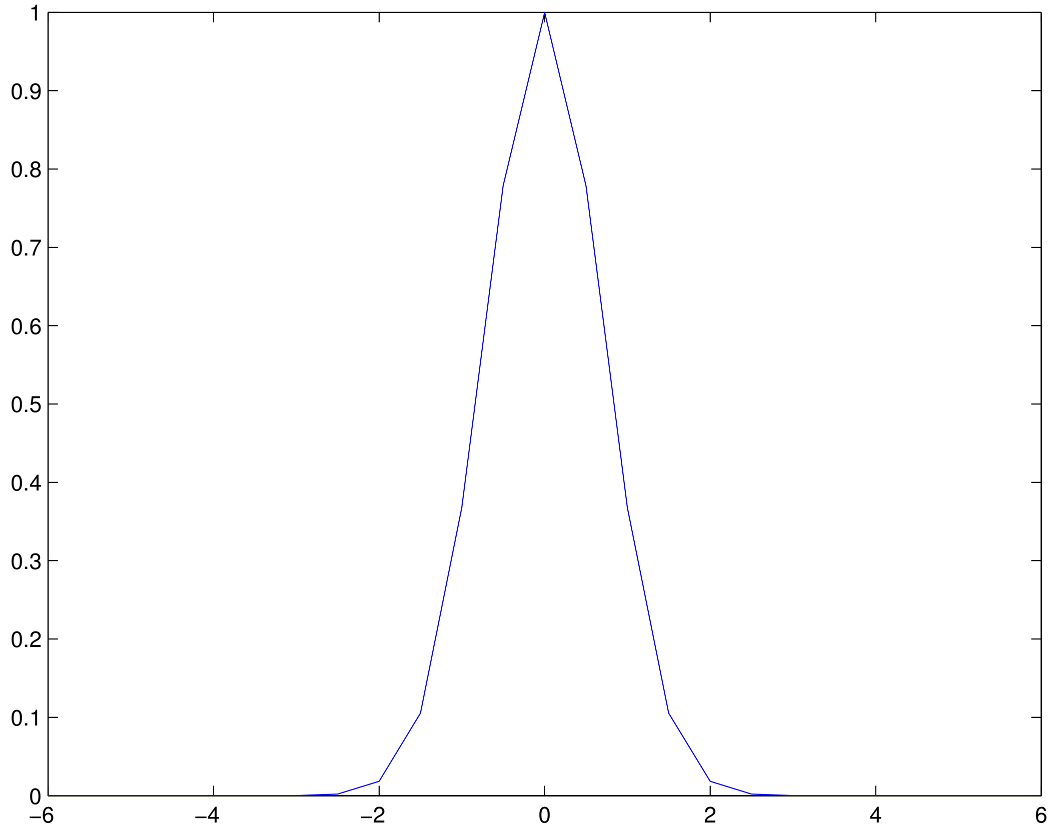
The `plot` command

```
figure(1)
x = -6:.5:6;
y = exp(-x.^2);
plot(x,y)

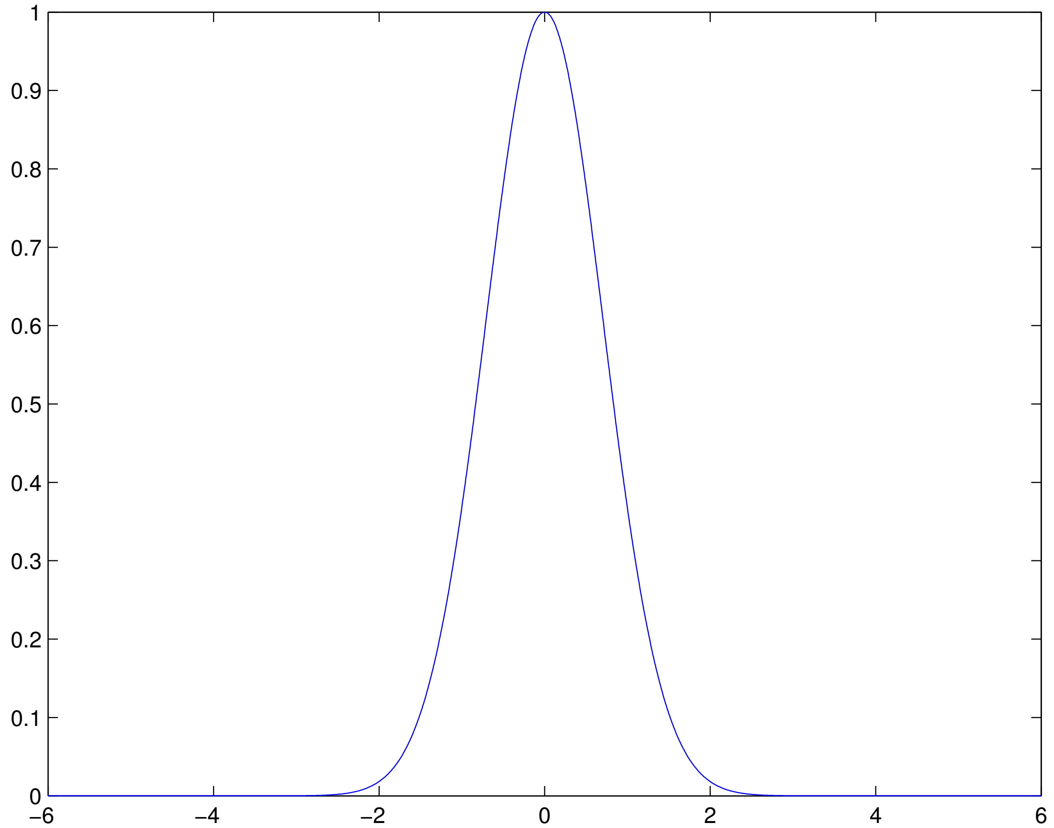
figure(2)
x = -6:.01:6;
y = exp(-x.^2);
plot(x,y)
```

- `plot(x,y)` plots y_j over x_j
  - $x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$ are vectors of same length
  - Points (x_j, y_j) are connected with lines
- `figure(nr)` selects active figure
  - All graphics commands are applied to active figures
  - If figure `nr` does not exist, a new window is spawned
- `close(nr)` closes figure `nr`
  - `close` closes active figure
  - `close all` closes all figures
- `clf(nr)` deletes figure `nr`
  - `clf` deletes active figure
    - Windows are preserved

▶ `figure(1)`



▶ `figure(2)`



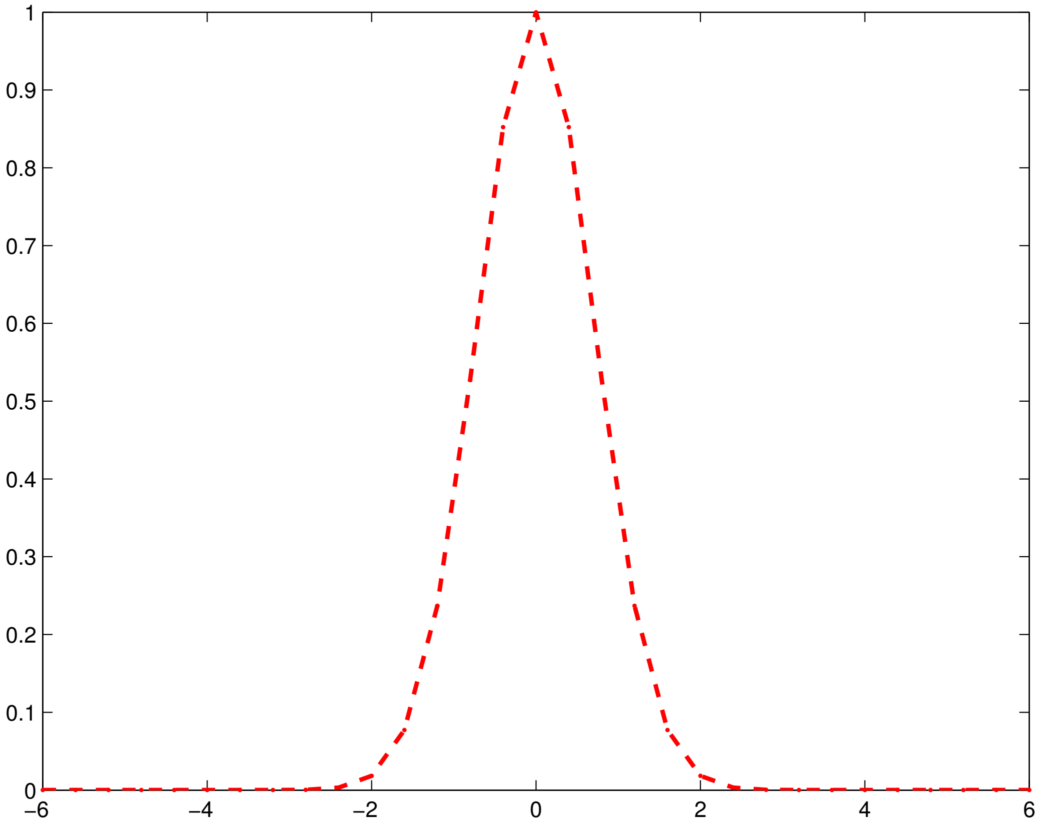
Optional parameters of `plot`

```
figure(1)
x = -6:.4:6;
y = exp(-x.^2);
plot(x,y,'r.--','LineWidth',2)

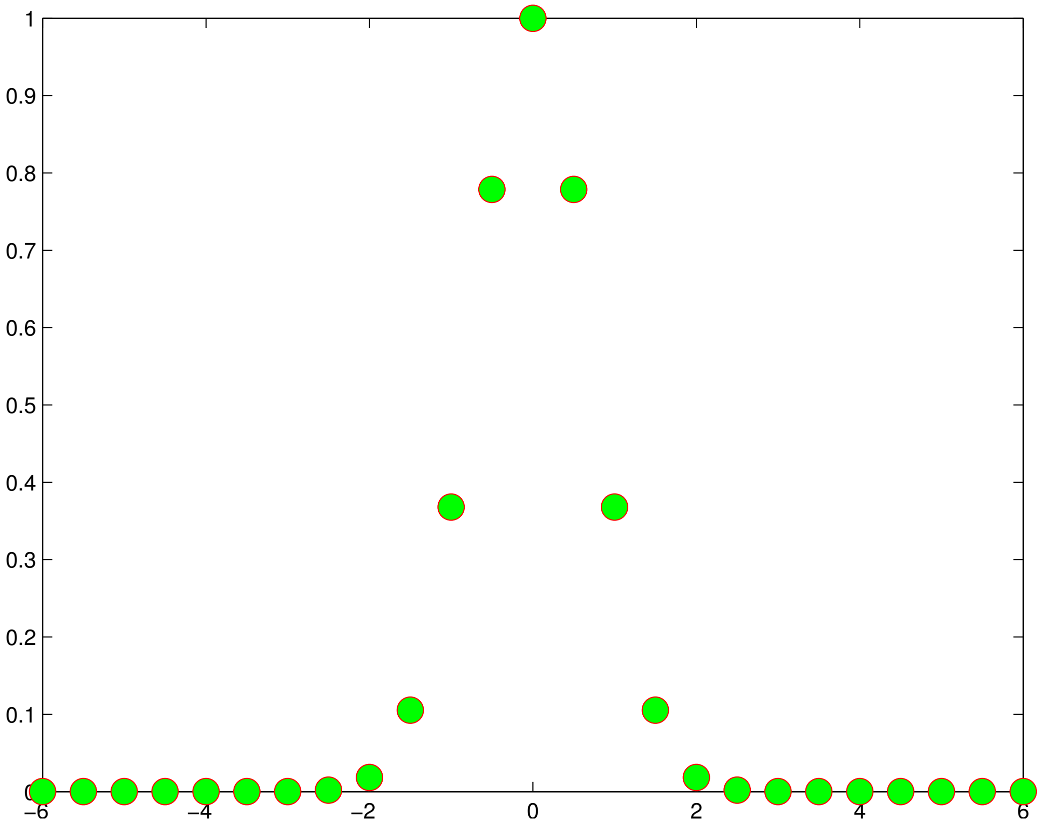
figure(2)
x = -6:.5:6;
y = exp(-x.^2);
plot(x,y,'ro','MarkerSize',12, ...
              'MarkerFaceColor','g')
```

- `plot(x,y,string)`
  - Optional `string` defines plot style
    - blue `b`, red `r`, green `g`, black `k`
    - dot `.`, circle `o`, cross `x`, plus `+`, star `*`
    - solid `-`, dotted `:`, dash-dotted `-.`, dashed `--`
  - 1 option for color/marker/line-style each
    - All options: `help plot` or `doc linespec`
    - Default `'b-'` = blue/no marker/solid line
- `plot(x,y,string, opt1,val1,... )`
  - Further options for all plot commands
    - opt1 = predefined string
    - val1 = new value
  - e.g., `'LineWidth'` (default = 0.5)
  - e.g., `'MarkerSize'` (default = 6)
  - e.g., `'MarkerEdgeColor'` (default = 'auto')
  - e.g., `'MarkerFaceColor'` (default = 'none')

▶ figure(1)



▶ figure(2)



Multiple plots in one figure

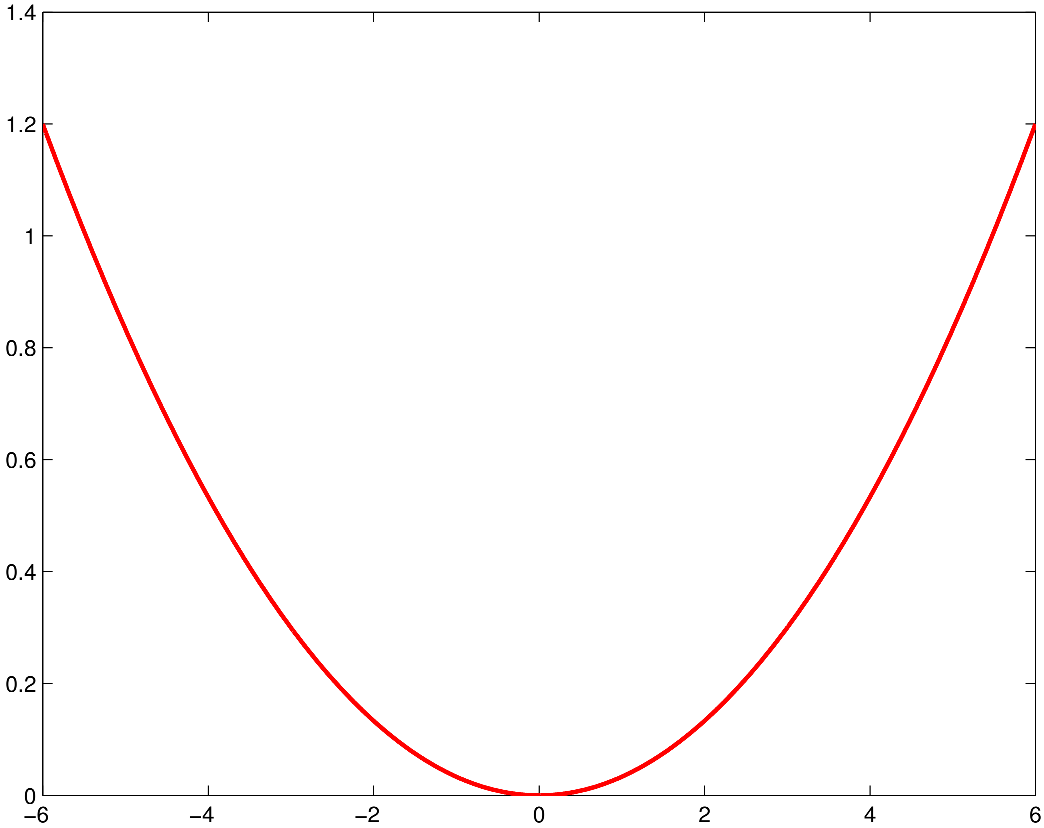
```
x = -6:.01:6;
y = exp(-x.^2);
z = x.^2/30;

figure(1)
plot(x,y,'b','LineWidth',2)
plot(x,z,'r','LineWidth',2)

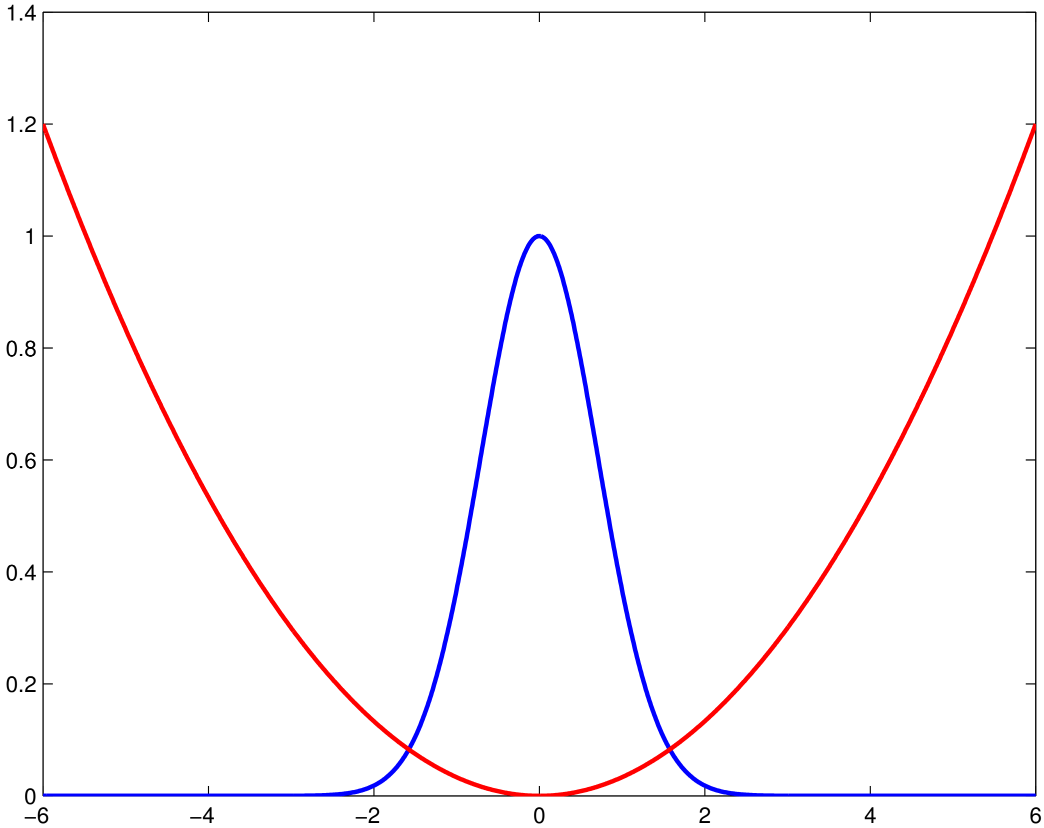
figure(2)
plot(x,y,'b','LineWidth',2)
hold on
plot(x,z,'r','LineWidth',2)
hold off
```

- Often, one wants multiple plots in one figure
  - Each new plot executes `clf` per default
- `hold off` = automatic `clf` in active figure
  - This is the default
- `hold on` = no automatic `clf` in active figure

▶ figure(1)



▶ figure(2)



Axes in plots 1/2

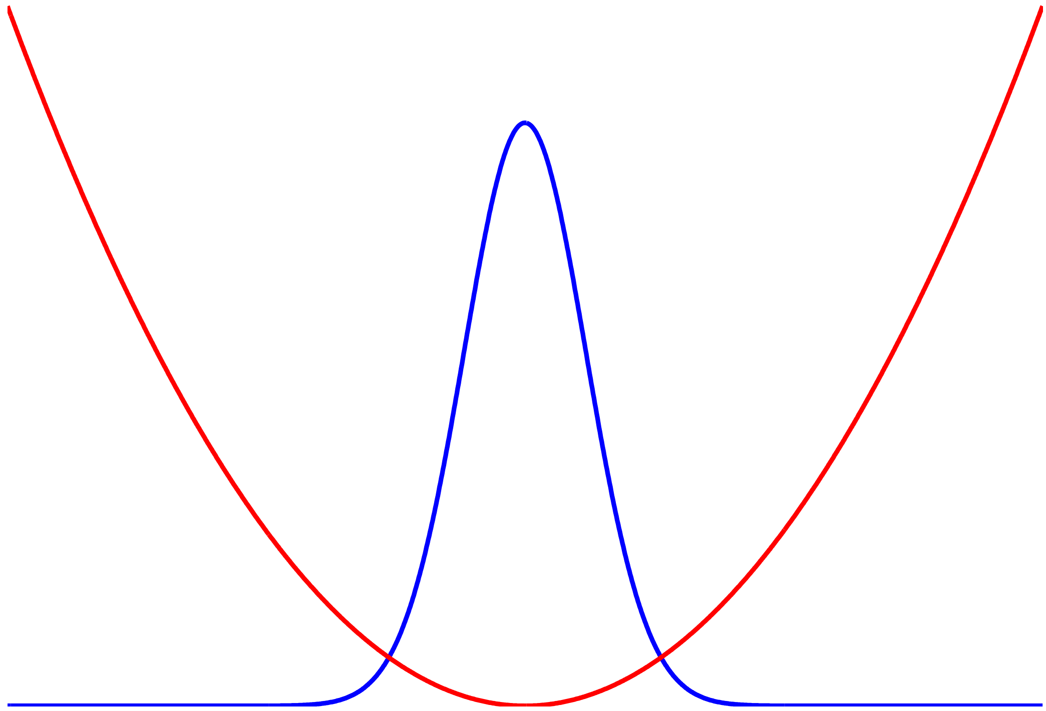
```
x = -6:.01:6;
y = exp(-x.^2);
z = x.^2/30;

figure(1)
plot(x,y,'b','LineWidth',2)
hold on
plot(x,z,'r','LineWidth',2)
hold off
axis off

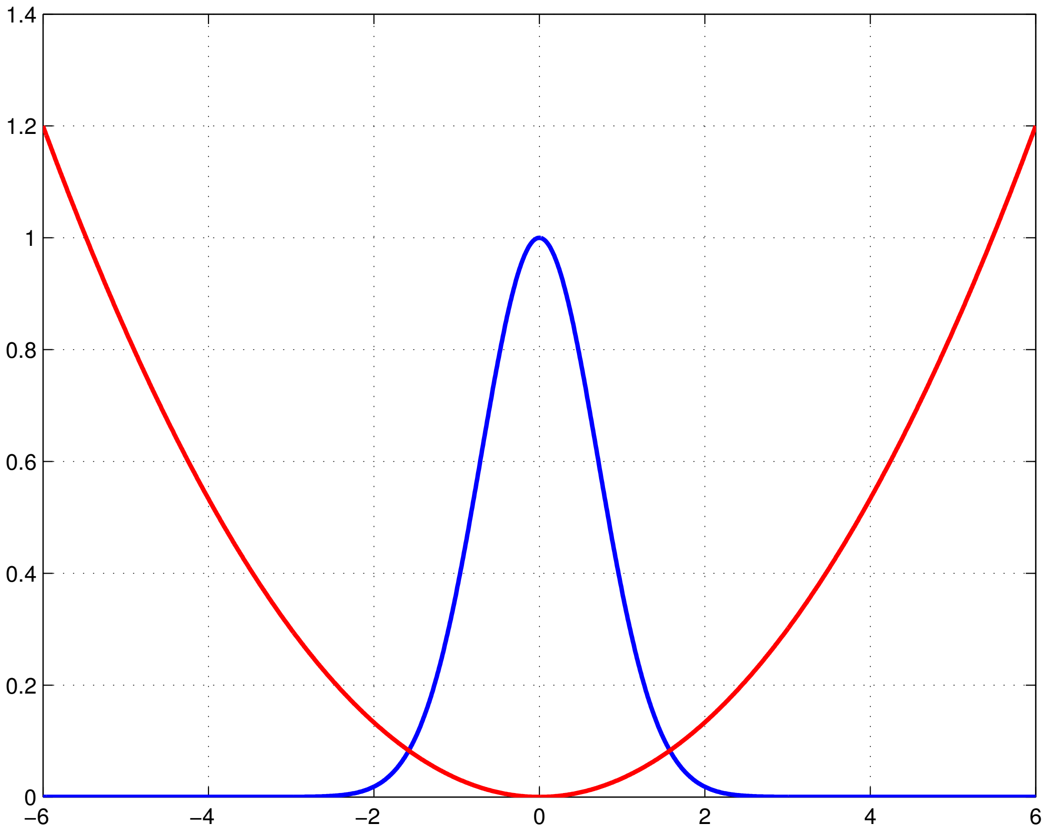
figure(2)
plot(x,y,'b','LineWidth',2)
hold on
plot(x,z,'r','LineWidth',2)
hold off
grid on
```

- `axis on` (`axis off`) = coordinate axes
- `grid off` (`grid on`) = grid lines
- `box on` (`box off`) = coordinate axes as box
- `axis([xmin,xmax,ymin,ymax])` sets axis limits
  - `axis` returns current vector of axis limits

▶ figure(1)



▶ figure(2)



Axes in plots 2/2

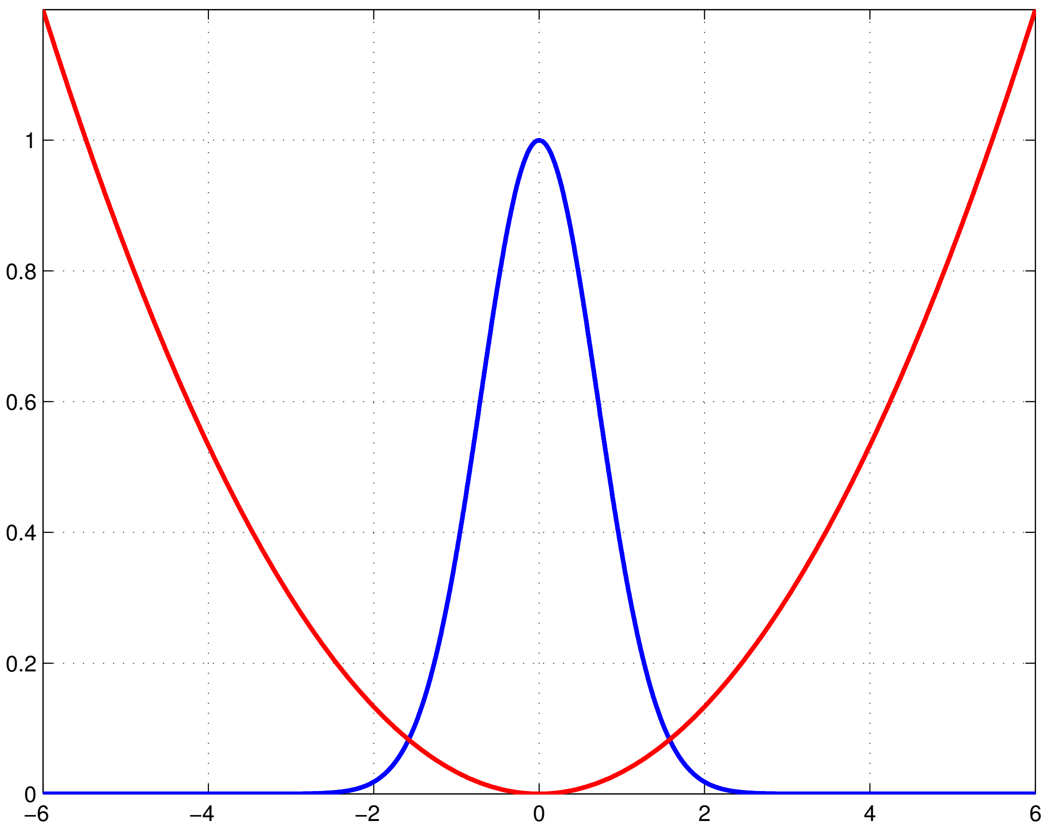
```
x = -6:.01:6;
y = exp(-x.^2);
z = x.^2/30;

figure(1)
plot(x,y,'b','LineWidth',2)
hold on
plot(x,z,'r','LineWidth',2)
axis tight
grid on

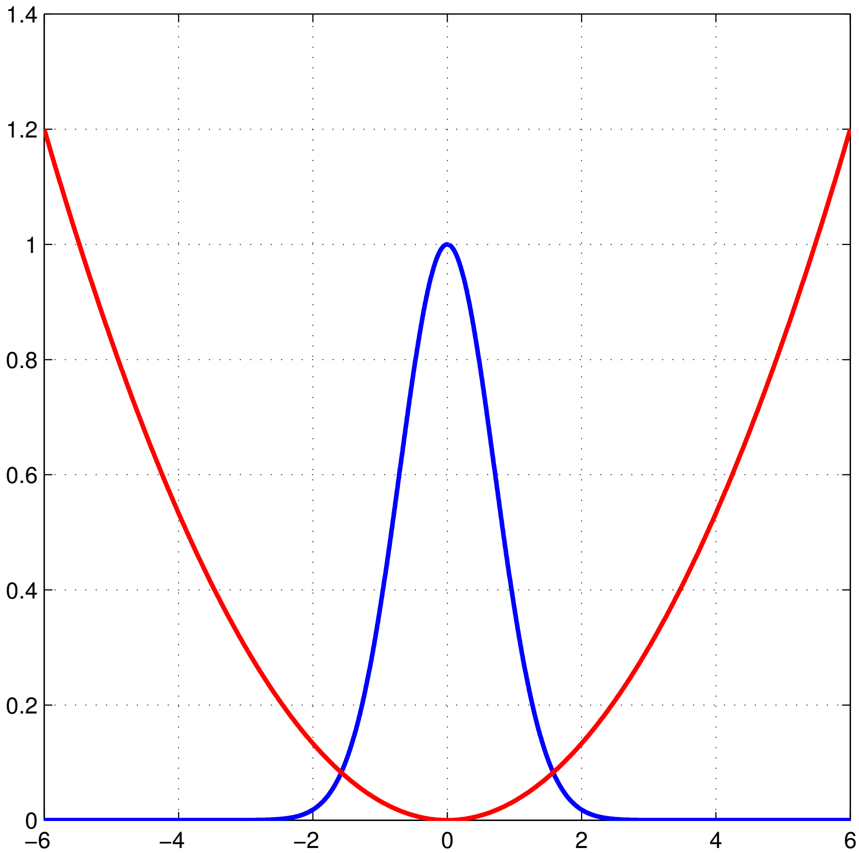
figure(2)
plot(x,y,'b','LineWidth',2)
hold on
plot(x,z,'r','LineWidth',2)
axis square
grid on
```

- `axis equal` = equal unit lenghts on both axes
- `axis tight` = image section as small as possible
- `axis square` = square image section

▶ `figure(1)`



▶ `figure(2)`



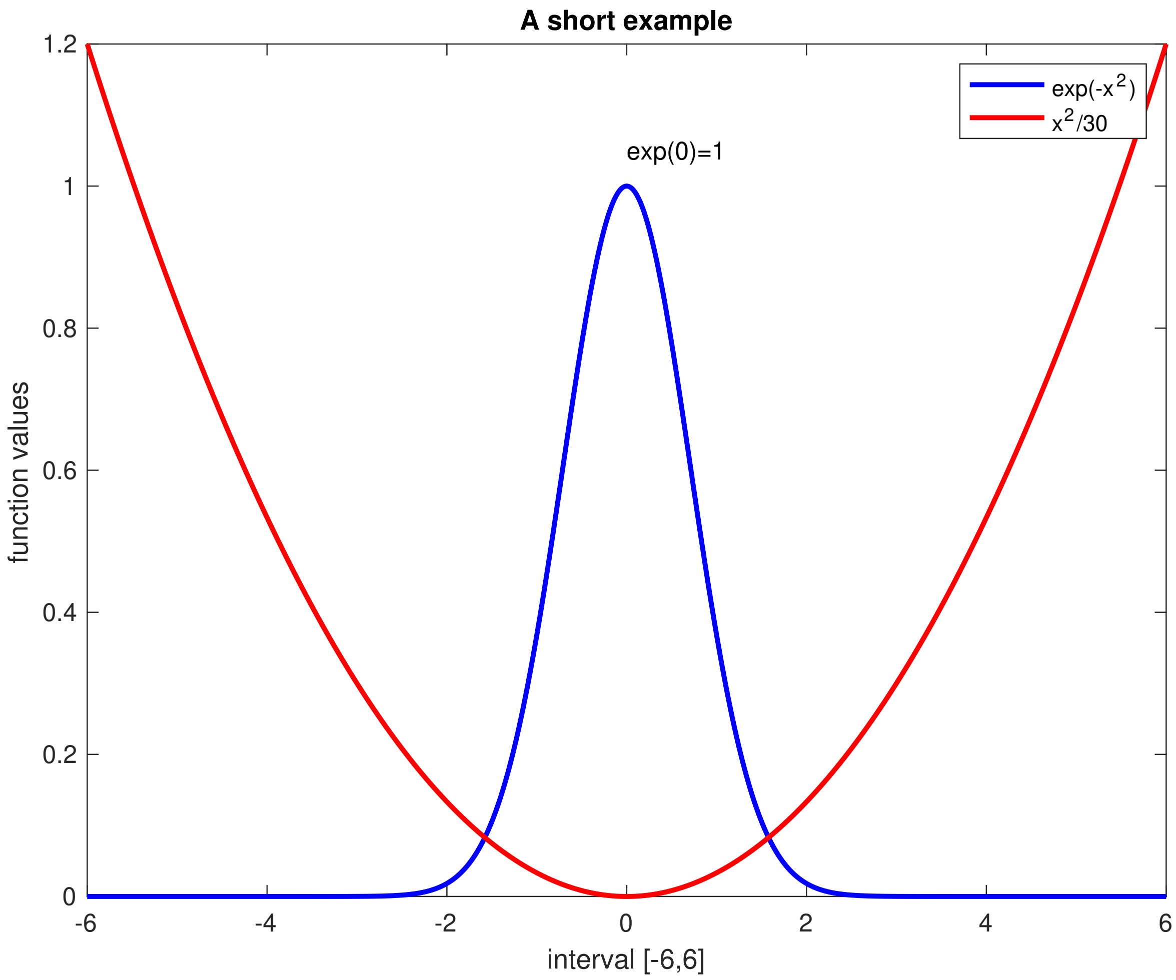
Labeling of plots

```
x = -6:.01:6;
y = exp(-x.^2);
z = x.^2/30;

plot(x,y,'b','LineWidth',2)
hold on
plot(x,z,'r','LineWidth',2)
text(0,1.05,'exp(0)=1')
hold off

legend('exp(-x^2)','x^2/30')
xlabel('interval [-6,6]')
ylabel('function values')
title('A short example')
```

- `legend(text1,text2,...)` creates legend
  - in order of the used plot commands
- `legend(...,'Location',lcn)` positions legend
  - e.g., `'northeast'` or `'southoutside'`
- `legend boxoff` = no box outline around legend
  - better for LaTeX-replacements (below!)
- `xlabel(text)` labels x-axis
- `ylabel(text)` labels y-axis
- `title(text)` creates title
- `text(x,y,text)` writes text `text` at coordinate (x, y)
- MATLAB can deal with basic LaTeX,
  - e.g., `x^2/30` in above code


A short example
exp(-x²)
x²/30
exp(0)=1
function values
interval [-6,6]
1.2
1
0.8
0.6
0.4
0.2
0
-6
-4
-2
0
2
4
6

Export of images

```
% demoprint.m
x = -6:.01:6;
y = exp(-x.^2);
z = x.^2/30;

plot(x,y,'b--')
hold on
plot(x,z,'r')
text(0,1.05,'exp(0)=1')
hold off

legend('exp(-x^2)','x^2/30')
xlabel('interval [-6,6]')
ylabel('function values')
title('A short example')

print('-r600','-depsc2','demoprint.eps')
print('-r600','-djpeg','demoprint.jpg')

close
```

- `print(opt1,opt2,...,name)` creates file `name`
  - Optional strings `opt` specify

    e.g., resolution: `'-r200'` = 200dpi (def. 150dpi)

    e.g., data type:
    - `'-deps'`, `'-deps2'` = EPS grayscale
    - `'-depsc'`, `'-depsc2'` = EPS colored
    - `'-djpeg90'` = JPG, quality 90% (def. 75%)
- Colored plots should be recognizable in grayscale
- If you use MATLAB figures for LaTeX documents, then the EPS format allows to replace any text in the graphics in LaTeX by use of the `psfrag` package

loglog

- Experimental convergence rate

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

Convergence rate of a method

- In numerical mathematics, $h > 0$ is often the discretization parameter
  - e.g., $\Phi(h) = \dfrac{f(x+h) - f(x)}{h}$ as approximation of the derivative $\Phi(0) = f'(x)$
- Clearly: $\Phi(h) \to f'(x)$ as $h \to 0$
- Question: Can something be said about the size of the approximation error?
  - Taylor theorem
    - For $f \in C^2(\mathbb{R})$, it holds that

$$f(x+h) = f(x) + f'(x)h + R_1(f, x, h)$$

    - with remainder term

$$R_n = \int_x^{x+h} \frac{(x+h-t)^n}{n!} f^{(n+1)}(t)\,\mathrm{d}t = \mathcal{O}(h^{n+1})$$

  - Hence,

$$\Phi(h) = \frac{f'(x)h + R_1(f, x, h)}{h} = f'(x) + \mathcal{O}(h)$$

Experimental convergence rate

- Approximation errors $e_h = |\Phi(h) - \Phi(0)|$ usually satisfy that
  - $e_h = \mathcal{O}(h^\alpha)$ for $h \to 0$ and $\alpha > 0$
    - i.e., $e_h \le C\, h^\alpha$ with a constant $C > 0$
  - α is called **convergence rate**
    - In general, C and α are unknown and only known for special cases, e.g., $f \in C^2(\mathbb{R})$

- One can experimentally determine C and α
  - Ansatz: Let $e_h = Ch^\alpha$
  - For $h_1 > h_2 > 0$ compute $e_1 = e_{h_1}$, $e_2 = e_{h_2}$
  - Division yields $e_1/e_2 = (h_1/h_2)^\alpha$
  - Taking the logarithm yields $\alpha = \dfrac{\log(e_1/e_2)}{\log(h_1/h_2)}$
    - so-called **experimental convergence rate**

Visualization

- Let $h_1 > h_2 > 0$ and corresponding e_1, e_2 be given
- Plot points in a graph:
  - x-axis is $x = \log(1/h)$
  - y-axis is $y = \log(e)$
- Straight line through $\big(\log(1/h_j)\,,\,\log(e_j)\big)$ has slope
  - $$m = \frac{\log(e_1) - \log(e_2)}{\log(1/h_1) - \log(1/h_2)} = \frac{\log(e_1/e_2)}{\log(h_2/h_1)}$$
  - hence, $-m = \dfrac{\log(e_1/e_2)}{\log(h_1/h_2)} = \alpha$ is exp. conv. rate

The `loglog` command

- `loglog(x,y)` corresponds to `plot(log(x),log(y))`
  - Optional parameters as for `plot`
- `loglog(x,y)` is used to visualize algebraic dependence $y = \mathcal{O}(x^\alpha)$
  - α can be observed as slope of a line!
  - e.g., for experimental conv. rate $e_h = \mathcal{O}(h^\alpha)$
  - e.g., for complexity time$(N) = \mathcal{O}(N^\alpha)$
- Further variants of `plot`:
  - `semilogx`, `semilogy`

A smooth example

```
%*** problem
h = 2.^-[0:16];
x = 1;
f = @(x) exp(x);          % def. f(x) = exp(x)
fprime = @(x) exp(x);     % def. fprime(x) = exp(x)

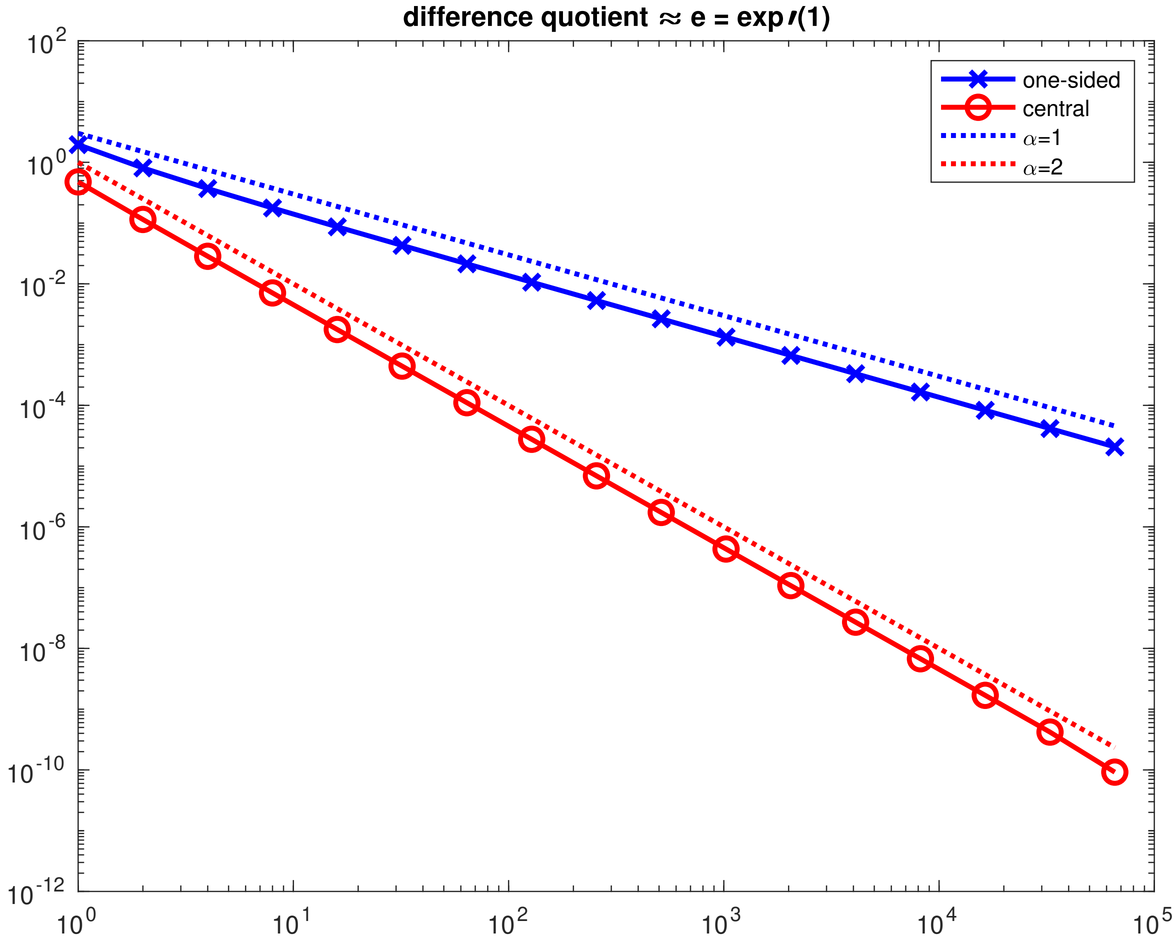
%*** one-sided difference quotient
phi = (f(x+h)-f(x))./h;
e = abs(fprime(x)-phi);
loglog(1./h,e,'bx-','LineWidth',2,'MarkerSize',9)
hold on

%*** central difference quotient
phi = 0.5*(f(x+h)-f(x-h))./h;
e = abs(fprime(x)-phi);
loglog(1./h,e,'ro-','LineWidth',2,'MarkerSize',9)

%*** reference lines
loglog(1./h,3*h,'b:','LineWidth',2)  % alpha = 1
loglog(1./h,h.^2,'r:','LineWidth',2) % alpha = 2
hold off

title(['difference quotient ',...
       '\approx e = exp\prime(1)'])
legend('one-sided','central', ...
       '\alpha=1','\alpha=2')
```

- One-sided $\Phi(h) = \frac{f(x+h) - f(x)}{h}$
  - maximal convergence rate $\alpha = 1$ for $f \in C^2$
- Central $\Phi(h) = \frac{f(x+h) - f(x-h)}{2h}$
  - maximal convergence rate $\alpha = 2$ for $f \in C^3$



- $f \in C^2 \;\Rightarrow\;$ one-sided diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h)$
- $f \in C^3 \;\Rightarrow\;$ central diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h^2)$
- Example: $f(x) = \exp(x)$ satisfies $f \in C^\infty$
  - Experiment confirms theory

A less smooth example

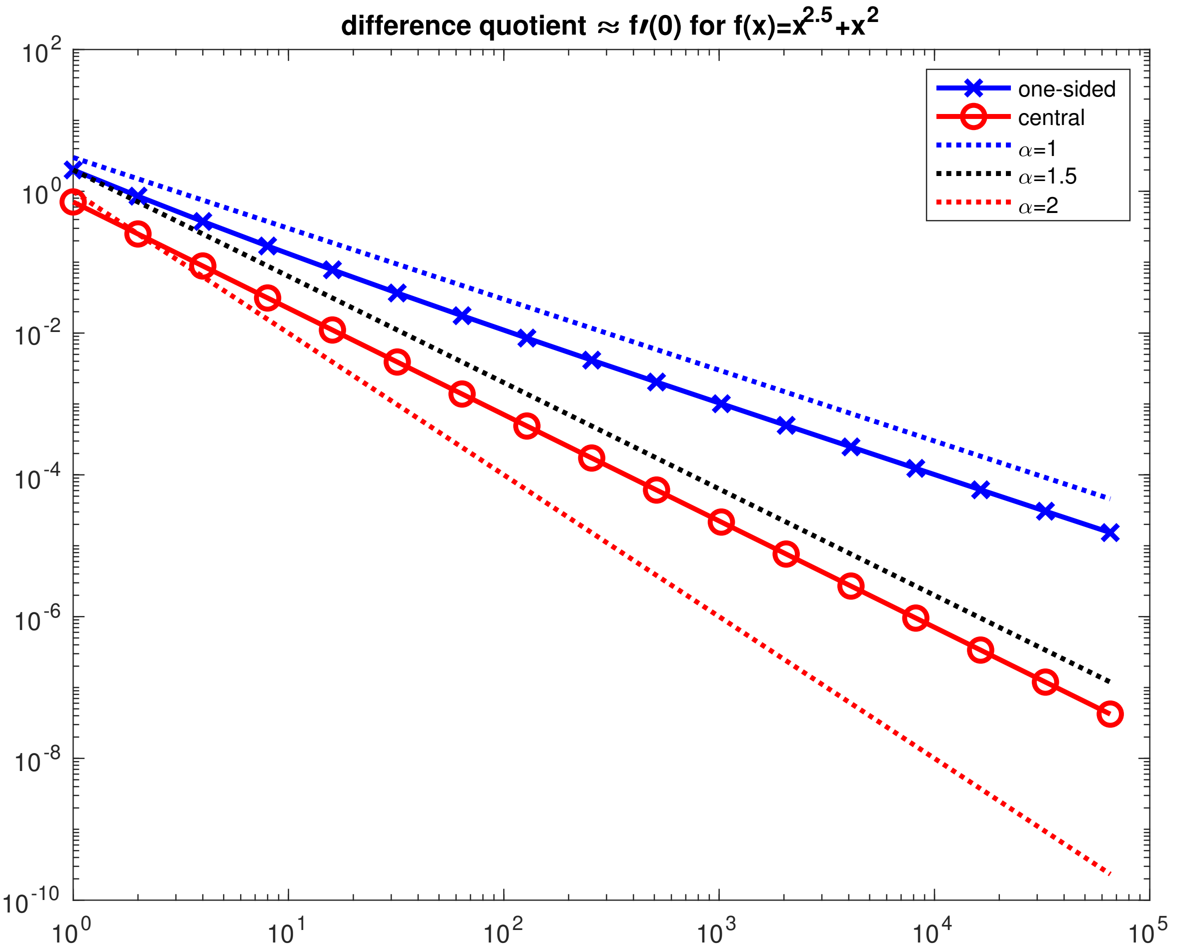
```
%*** problem
h = 2.^-[0:16];
x = 0;
f = @(x) x.^2.5 + x.^2;
fprime = @(x) 2.5*x.^1.5 + 2*x;

%*** one-sided difference quotient
phi = (f(x+h)-f(x))./h;
e = abs(fprime(x)-phi);
loglog(1./h,e,'bx-','LineWidth',2,'MarkerSize',9)
hold on

%*** central difference quotient
phi = (f(x+h)-f(x-h))./h/2;
e = abs(fprime(x)-phi);
loglog(1./h,e,'ro-','LineWidth',2,'MarkerSize',9)

%*** reference lines
loglog(1./h,3*h,'b:','LineWidth',2)
loglog(1./h,2*h.^1.5,'k:','LineWidth',2)
loglog(1./h,h.^2,'r:','LineWidth',2)
hold off

title(['difference quotient ',...
       '\approx f\prime(0) ',...
       'for f(x)=x^{2.5}+x^2'])
legend('one-sided','central', ...
       '\alpha=1','\alpha=1.5','\alpha=2')
```



- $f \in C^2 \;\Rightarrow\;$ one-sided diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h)$
- $f \in C^3 \;\Rightarrow\;$ central diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h^2)$
- Example: $f(x) = x^{2.5} + x^2$ satisfies only $f \in C^2 \backslash C^3$
  - Experiment does not contradict theory!

Input / Output

- Input from keyboard
- Output in MATLAB shell
- Load and save variables
- Load matrices from text files
- Save matrices to text files

- `input`
- `disp`
- `fprintf`
- `load`
- `save`
- `fopen`, `fclose`

Input from keyboard

- `var = input(string);`
  - displays the text `string` in the MATLAB shell
  - waits for input from the keyboard
  - interprets the input and assigns the value to `var`
    - e.g., from the input `2 + [1 2 3]`, the variable `var` takes the value `[3 4 5]`
  - If input cannot be interpreted, MATLAB returns an error
    - e.g., the input `Hello World` leads to `Error: Unexpected MATLAB expression.`
- `var = input(string,'s');`
  - displays the text `string` in the MATLAB shell
  - waits for input from the keyboard
  - assigns the input to `var` (as array of characters)

Output to MATLAB shell

- `disp(var)` displays the value of the variable `var` in the MATLAB shell
- `fprintf(string,var1,var2,...)`
  - displays the text `string` in the MATLAB shell
  - works like `printf` in C
  - `string` can contain conversion specifiers indicated by `%`, e.g.,
    - `%d` for an integer
    - `%f` or `%e` for a floating point number
    - `%s` for string
    - See `help fprintf` for details on the specifiers
  - The conversion specifiers are replaced by the given values `var1` etc. (from left to right)
  - The number of conversion specifiers and additional values must coincide
  - Line breaks are indicated by the so-called escape sequence `\n` in `string`
  - Note that `fprintf` works only for real numbers, not for complex numbers!
    - Use `real()` and `imag()` to output real and imaginary part separately
- e.g., `fprintf('%1.4f\n',pi)` gives `3.1416`
  - where `%1.4f` also specifies the number of digits
- e.g., `fprintf('%1.8e\n',2/3)` gives `6.66666667e-01`
- e.g., `fprintf('%f\n',2+3i)` gives `2.000000`

Functions `load` / `save`

- Goal: Save (partial) results from computations
  - This avoids the need to compute everything from scratch, if the computation is aborted (e.g., when the PC crashes)
  - Moreover, it is good programming style to separate the codes for computation and postprocessing (e.g., visualization)

- `save('name')` saves all variables in the current scope to the data file `name.mat`

- `save('name','var1','var2',...)` saves only the variables `var1`, `var2`, ... to the data file `name.mat`

- `load('name')` loads the variables from the data file `name.mat` to the current scope

- `A = load('name.dat');` creates a matrix `A`
  - `name.dat` must be a text file with clear matrix structure, i.e.,
    - rows are ended by line breaks
    - all lines have the same number of entries
    - comments indicated by `%` are neglected
  - This is a very good way for data import from other programs / programming languages

Formatted writing

- Goal: Create text files that can be read by other programs / programming languages

- Open a text file `filename` for writing data by
  - `fid = fopen(filename,'w')`
  - `fid` is the so-called file identifier
    - see `help fopen` for further details

- Write data in ASCII format to the file via `fprintf`
  - `fprintf(fid,string,var1,var2,...)` like in C
    - `\n` creates new line in the output file
    - `\\` creates the backslash symbol \
    - `%%` creates the percentage symbol %
    - Use conversion specifiers to write numerical values, e.g., `%d` for integers and `%f` or `%e` for floating point numbers

- Note: Use the conversion specifier `%1.16e` to write `double` values to a file
  - Note that `double` values have about 16 digits
  - Recall that `fprintf` works only for real values

- Use `fclose(fid)` to close the file when writing is completed

- MATLAB also allows for formatted reading via `fscanf`, but it is recommended to use `load` instead
  - see `help fscanf`

Error control

- Warnings and error
- Controlled termination

- `warning`
- `lastwarn`
- `error`, `assert`
- `lasterr`
- `try-catch`

Output of warnings

- Your programs can give warnings to users, e.g., if the condition number is high and the computed solution of a linear system is possibly inaccurate
  - Warnings give information to the user without terminating the program
- `warning(string);` creates a warning
- `warning off` ensures that no warnings will be displayed to the user (not even those from other functions)
- Default: `warning on` ensures that all warnings will be displayed in the MATLAB shell
- `var = lastwarn;` returns the last warning message
  - `lastwarn('')` resets the last warning

Controlled termination

- `error(string)` displays an error message `string` and terminates the execution
- `assert(condition)` leads to termination, if `condition` fails
  - `assert(condition,string)` additionally displays the error message `string`
  - `assert(condition,string,var1,var2,...)` displays the formatted error message `string`, which is interpreted as for `fprintf`
- `var = lasterr;` returns the last error message
  - i.e., `string` from `error` or `assert`

Example: Euclidean algorithm

```
function a = euclid(a,b)

% Compute the greatest common divisor (gcd) of
% two positive integers by means of Euclidean
% algorithm which is based on
%   gcd(A,B) = gcd(B,A)
% and, for A>B,
%   gcd(A,B) = gcd(A-B,B)
%
% RESULT = EUCLID(A,B) returns the gcd of two
% positive integer scalars A and B

% ensure that input is admissible
if ~(isscalar(a) && isscalar(b))
  error('Input arguments have to be scalars');
elseif ( a~=round(a) || b~=round(b) )
  error('Input arguments have to be integers');
elseif (a<=0 || b<=0)
  error('Input arguments have to be positive');
end

% loop of the Euclidean algorithm
while (a~=b)
  if (a<b)  % guarantee a>=b
    tmp = a;
    a = b;
    b = tmp;
  end
  a = a-b;
end
end
```

- The function checks that all input is admissible, i.e., the arguments are positive integer scalars
- This is already provided by MATLAB as `gcd`

Catching errors

```
input_is_valid = false;
while (~input_is_valid)
  try
    disp('Input two positive integers:')
    a = input('a = ');
    b = input('b = ');
    ggT = euclid(a,b);
    input_is_valid = true;
  catch
    disp(lasterr)
    disp('Please repeat your input!')
  end
end
fprintf('ggT(%d,%d) = %d\n',a,b,ggT)
```

- MATLAB tries to execute the `try` block
- If an error occurs (or an error is thrown by means of `error` or `assert`), then the code continues with the execution of the `catch` block
- Recall that the function `lasterr` returns the last error message
- Usually, the `catch`-block of `try-catch-end` is used to store the current data/variables for later debugging

Functions II

- Cell Arrays
- Optional input
- Optional output

- `nargin`, `varargin`
- `nargout`
- `pwd`
- `path`, `addpath`, `rmpath`

Cell Arrays

```
A = cell(1,3);

A{1} = 2;
A{2} = 1:2:10;
A{3} = 'red';

n = length(A);
vector = A{2};
disp(A{end});
```

- Cell arrays are arrays, where the entries may have different data types
- Cell arrays are allocated via `container = cell(M,N);`
  - Dynamic allocation is possible, but should be avoided
- The entries of a cell array are `container{j,k}`
  - as for normal arrays, but with curly brackets instead of round brackets
  - e.g., linear indexing `container{j}` as for matrices,
  - e.g., `size` and `length` are applicable

Optional output of a function

- If a function would return N output values, but the calling code takes only $n \leq N$, then the remaining $N - n$ are automatically discarded
  - e.g., `[x,fx] = bisection(f,a,b)` returns the approximation `x` of a root together with the function value `fx`$= f(x)$
  - Then, the call `x = bisection(f,a,b)` assigns only the approximate root `x`, while `fx` is discarded
  - Alternatively, one can use
    - `[x,~] = bisection(f,a,b)` to discard `fx`, but only take `x`
    - `[~,fx] = bisection(f,a,b)` to discard `x`, but only take `fx`
- Any function can use the system variable `nargout` ("number of arguments out") to get the number of output arguments that are taken by the calling code (i.e., n above)
  - This information can be used to avoid unnecessary computations

Optional input to a function

- Any function in MATLAB can have obligatory and optional input
  - The system variable `nargin` ("number of arguments in") provides the information how many arguments are passed to a function
  - If the function expects n obligatory input parameters, but is called with $N \geq n$ input parameters, then the last $N - n$ are optional
- To allow for optional input, a function must have the signature

```
function [out1,out2,...] = fct(in1,in2,...,varargin)
```

  - with `out1`, `out2`, etc. being the output
  - with `in1`, `in2`, etc. being the obligatory input
  - with `varargin` ("variable arguments in") being a cell array containing the optional input
- Suppose that the function `fct` takes n obligatory input parameters
  - Then, `varargin{j}` contains the additional optional input for $j = 1, \ldots,$ `nargin` $- n$ that has been passed to `fct` by the calling code

Ex: binary search with tolerance

```
function index = binsearch(vector,query,varargin)

% Given a query Q and a tolerance TOL, seek an
% index J such that the J-th entry X(J) of a
% vector X satisfies |X(J)-Q| <= TOL. Return -1
% if no such index exists. The vector X is
% required to be sorted in ascending order
%
% J = binsearch(X,Q [,TOL]) with X being a
% numeric vector, Q being a scalar, and TOL being
% the optional tolerance which is 0 by default.

if nargin >= 3
  tolerance = varargin{1};
else
  tolerance = 0;
end

lower = 1;
upper = length(vector);
while (lower <= upper)
  index = floor(0.5*(lower + upper));
  if ( abs(vector(index)-query) <= tolerance )
    return
  elseif (vector(index) >= query)
    upper = index - 1;
  else
    lower = index + 1;
  end
end
index = -1;
end
```

- The function requires that `vector` is sorted in ascending order.

Example: secant method

```
function x0 = secantMethod(f,x,varargin)

  if nargin >= 3
    tolerance = varargin{1};
  else
    tolerance = 1e-12;
  end
  fx = f(x);
  while true
   dx = x(2)-x(1);
   assert(dx~=0,'Iteration led to x_{n} = x_{n-1}')
   df = (fx(2)-fx(1))/dx;
   assert(df~=0,'Difference quotient is zero!')
   if (abs(df) <= tol)
     warning('Diff. quotient is close to zero!')
   end
   x = [x(2), x(2)-df\fx(2)];
   fx = [fx(2), f(x(2))];
   abs_dx = abs(dx);
   max_x = max(abs(x));
   if ( abs(fx(2))<=tol && ...
       ( (abs_dx<=tol && max_x<=tol) || ...
         (abs_dx<=tol*max_x && max_x>tol) ) )
     break
   end
  end
  x0 = x(2);
end
```

- Goal: Approximate a root x_0 of $f : [a,b] \to \mathbb{R}$
- Given $x_{n-1}, x_n \in [a,b]$ with $x_{n-1} \neq x_n$, compute the root of the secant, i.e., $x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$
- If $f(x_{n+1}) \approx 0$ and $x_{n+1} \approx x_n$, then terminate

File paths

- If a function is called, then MATLAB searches certain directories to find appropriate files `name.m`
  - First, current directory, which is returned by `pwd` ("print working directory")
  - Second, all directories that are contained in MATLAB search path, which is returned by `path`
- `path` can be modified and adapted
  - `addpath('name')` adds the directory `name`
  - `rmpath('name')` removes the directory `name`
- One can overload a MATLAB command `name` by providing `name.m` in the current directory
  - MATLAB will always execute the first file that is found in the MATLAB path
- `which name` shows, which file will be used when `name` is called in MATLAB

Complexity

- Complexity of algorithms
- Landau symbol $\mathcal{O}$

Computational complexity

- The complexity of an algorithm is the amount of time, storage, and/or other resources that is necessary to execute it
  - It allows to compare different algorithms
- Recall: An algorithm is a finite sequence of unambiguous operations which specify how to solve a problem
- The computational complexity of an algorithm is the number of required elementary operations, i.e.,
  - assignments
  - comparisons
  - arithmetic operations
- Language-specific operations usually do not count, e.g.,
  - declarations & initializations
  - loops, conditional statements, etc.
  - counters
- For ease of presentation, we consider the worst-case computational complexity, i.e., the maximum number of operations required for inputs of a given size

Example: Maximum of a vector

```
1   function out = max(x)
2     out = x(1);
3     for j = 2:length(x)
4       if (out < x(j))
5         out = x(j);
6       end
7     end
8   end
```

- Complexity computation:
  - 1 assignment ⇝ Line 2
  - In each step of the `for` loop ⇝ Lines 3–7
    - 1 comparison ⇝ Line 4
    - 1 assignment (worst case!) ⇝ Line 5
- Loops always translate to a sum of operations
  - i.e., `for` in line 6 implies $\sum_{j=2}^{n}$
- Altogether:

$$1 + \sum_{j=2}^{n} 2 = 1 + 2(n-1) = 2n - 1$$

- Note: We neglect the evaluation `x(1)` in Line 2 as well as the call of `length(x)` in Line 3. We will see in the following that this is fine asymptotically, if the effort for these operations is constant

Landau symbol $\mathcal{O}$ (= big O)

- Very often, only the order of magnitude of the computational complexity is of interest
- Definition: One writes $f = \mathcal{O}(g)$ as $x \to x_0$
  - if $\limsup_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty$
  - i.e., $|f(x)| \le C\,|g(x)|$ as $x \to x_0$
  - i.e., f grows at most like g for $x \to x_0$
- Example: The determination of the maximum of a vector of length n has complexity $2n - 1 = \mathcal{O}(n)$ as $n \to \infty$
- Often, "as $n \to \infty$" is omitted, as it is clear from the context
  - Standard choice (asymptotic complexity)
- In words:
  - An algorithm has linear complexity, if its complexity is $\mathcal{O}(n)$ for problems of size n
    - e.g., determine the maximum of a vector
  - An algorithm has quasilinear complexity, if its complexity is $\mathcal{O}(n \log n)$ for problems of size n
  - An algorithm has quadratic complexity, if its complexity is $\mathcal{O}(n^2)$ for problems of size n
  - An algorithm has cubic complexity, if its complexity is $\mathcal{O}(n^3)$ for problems of size n

Matrix-vector multiplication

```
1  function b = matrixVectorProduct(A,x)
2    [m,n] = size(A);
3    b = zeros(m,1);
4    for j = 1:m
5      for k = 1:n
6        b(j) = b(j) + A(j,k)*x(k);
7      end
8    end
9  end
```

- 2 assignments for m and n
- 1 assignments for each entry of b
- In each step of the `for` loop over j $\rightsquigarrow$ Lines 4–8
  - In each step of the `for` loop over k $\rightsquigarrow$ Lines 5–7
    - 1 multiplication $\rightsquigarrow$ Line 6
    - 1 addition $\rightsquigarrow$ Line 6
    - 1 assignment $\rightsquigarrow$ Line 6
- Altogether:

$$2 + m + \sum_{j=1}^{m} \sum_{k=1}^{n} 3 = 2 + m + 3mn = \mathcal{O}(mn)$$

- Complexity $\mathcal{O}(mn)$
  - i.e., complexity $\mathcal{O}(n^2)$ for $m = n$
  - i.e., quadratic complexity for $m = n$

Linear search in a vector

```
function index = search(vector, query, tolerance)
  for index = 1:length(vector)
    if ( abs(vector(index)-query) <= tolerance )
      return
    end
  end
  index = -1;
end
```

- Task: Given a vector $x \in \mathbb{K}^n$ and a query $q \in \mathbb{K}$, seek an index j with $|x_j - q| \leq$ `tolerance`
  - Return -1 if no such index exists

- In each step of the `for` loop over j
  - 1 subtraction
  - 1 absolute value
  - 1 comparison
- Altogether:

$$\sum_{j=1}^{n} 3 = 3n$$

- Complexity $\mathcal{O}(n)$

Binary search in sorted vector

```
function index = binarySearch(vector,query,tol)
  lower = 1;
  upper = length(vector);
  while (lower <= upper)
    index = floor(0.5*(lower + upper));
    if ( abs(vector(index)-query) <= tol )
      return
    elseif (vector(index) > query)
      upper = index - 1;
    else
      lower = index + 1;
    end
  end
  index = -1;
end
```

- Task: Given a vector $x \in \mathbb{K}^n$ and a query $q \in \mathbb{K}$, seek an index j with $|x_j - q| \leq$ `tol`
  - Return -1 if no such index exists
- Assumption: Vector is sorted in ascending order
- Adapt the idea of dictionary search and consider halved vector, if $|x_j - q| >$ `tol`
- Question: How many iterations does the alg. have?
  - Each step halves the vector
  - If n is even, choose k with $n/2^k = 1$
  - Hence, at most $k = \log_2 n$ steps with each
    - 2 comparisons, 2 assignments, 1 call of `floor` and `abs`, 1 multiplication, 3 additions
- Complexity $\mathcal{O}(\log_2 n)$, i.e., logarithmic complexity
  - Sublinear complexity $\mathcal{O}(\log_2 n) \ll \mathcal{O}(n)$

Selection sort

```
function vector = selectionSort(vector)
  for j = 1:length(vector)-1
    argmin = j;
    for k = j+1:length(vector)
      if ( vector(argmin) > vector(k) )
        argmin = k;
      end
    end
    if ( argmin > j)
      vector([j argmin]) = vector([argmin j]);
    end
  end
end
```

- Selection sort is probably the most naive algorithm that sorts a vector $x \in \mathbb{R}^n$ in ascending order
- Call by value requires n assignments to copy $x \in \mathbb{R}^n$
- In each step of the `for` loop over j
  - 1 assignment
  - In each step of the `for` loop over k
    - 1 comparison
    - 1 assignment (worst case!)
  - 1 comparison
  - 2 assignments (worst case!)
- quadratic complexity $\mathcal{O}(n^2)$, because:

$$n + \sum_{j=1}^{n-1} \Big(4 + \sum_{k=j+1}^{n} 2 \Big) = n + 4(n-1) + \sum_{j=1}^{n-1} (n-j)2$$

$$= 5n - 4 + 2 \sum_{k=1}^{n-1} k = 5n - 4 + 2 \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

Cost and computational time

- Why time measurement?
  - Comparison of algorithms/implementations
  - Validation of theoretical considerations
- We suppose that all operations that have been counted for the computational complexity require the same amount of time
- Then, we can make theoretical predictions on the runtime of an algorithm
  - Linear complexity
    - Problem size $n \Rightarrow Cn$ operations
    - Problem size $kn \Rightarrow Ckn$ operations
    - i.e., $3\times$ problem size $\Rightarrow 3\times$ runtime
  - Quadratic complexity
    - Problem size $n \Rightarrow Cn^2$ operations
    - Problem size $kn \Rightarrow Ck^2n^2$ operations
    - i.e., $3\times$ problem size $\Rightarrow 9\times$ runtime
  - Cubic complexity
    - Problem size $n \Rightarrow Cn^3$ operations
    - Problem size $kn \Rightarrow Ck^3n^2$ operations
    - i.e., $3\times$ problem size $\Rightarrow 27\times$ runtime
  - etc.
- E.g., if a program takes 1 s for $n = 1.000$, then:
  - Complexity $\mathcal{O}(n) \Rightarrow$ 10 s for $n = 10.000$
  - Complexity $\mathcal{O}(n^2) \Rightarrow$ 100 s for $n = 10.000$
  - Complexity $\mathcal{O}(n^3) \Rightarrow$ 1.000 s for $n = 10.000$

Measuring the computational time

- Stopping the real time:
  - Use `tic` to start the stopwatch
  - Use `toc` to get the elapsed time in seconds
- Example:

```
>> tic
>> A = rand(10000,10000);
>> elapsed_time = toc
```

  Then, `elapsed_time` contains the time needed to create the matrix containing random entries

- Stopping the computational time:
  - `cputime` returns the CPU time of MATLAB elapsed since its start (measured in seconds)
- Example:

```
>> t = cputime;
>> A = rand(10000,10000);
>> elapsed_time = cputime-t
```

  Then, `elapsed_time` contains the CPU time needed to create the matrix containing random entries

Runtime comparison 1/2

```
clear all

Nmin = 500;
Jmax = 22;
for j = 1:Jmax
  x = 1:Nmin*2^j;
  t1(j) = cputime;
  binarySearch(x,0,0);        %*** sublinear cost
  t1(j) = cputime - t1(j);
  fprintf('binarySearch, %d: %d, %1.2f\n',
           j, length(x), t1(j));
end
n1 = Nmin*2.^(1:Jmax);

for j = 1:Jmax
  x = 1:Nmin*2^j;
  t2(j) = cputime;
  search(x,0,0);                 %*** linear cost
  t2(j) = cputime - t2(j);
  fprintf('search, %d: %d, %1.2f\n',
           j, length(x), t2(j));
end
n2 = Nmin*2.^(1:Jmax);

Jmax = 10;
for j = 1:Jmax
  x = flip(1:Nmin*2^j);
  t3(j) = cputime;
  selectionSort(x);           %*** quadratic cost
  t3(j) = cputime - t3(j);
  fprintf('selectionSort, %d: %d, %1.2f\n',
           j, length(x), t3(j));
end
n3 = Nmin*2.^(1:Jmax);

save('runtime_comparison');
```

Runtime comparison 2/2

	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log_2 n)$
n	`search`	`selectionSort`	`binarySearch`
1.000	0.00	0.01	0.00
2.000	0.00	0.04	0.00
4.000	0.00	0.10	0.00
8.000	0.00	0.10	0.00
16.000	0.00	0.33	0.00
32.000	0.00	1.26	0.00
64.000	0.00	4.87	0.00
128.000	0.00	20.83	0.00
256.000	0.00	80.29	0.00
512.000	0.00	328.20	0.00
1.024.000	0.01	$\geq$ 21min	0.00
2.048.000	0.01	$\geq$84min	0.00
4.096.000	0.01	$\geq$ 5,5h	0.00
8.192.000	0.02	$\geq$ 22h	0.00
16.384.000	0.04	$\geq$ 3,5d	0.00
32.768.000	0.07	$\geq$ 14d	0.00
65.536.000	0.14	$\geq$ 1,5m	0.00
131.072.000	0.38	$\geq$ 6m	0.00
262.144.000	0.62	$\geq$ 2y	0.00
524.288.000	1.42	$\geq$ 8y	0.00
1.048.576.000	2.41	$\geq$ 32y	0.00
2.097.152.000	11.09	$\geq$ 128y	0.00

- Logarithmic complexity is nice, as $2^{31} > 2.14 \cdot 10^9$
- Also linear complexity yields good runtime
- Quadratic complexity for large n is noticeable
  - Naive sorting of a vector of length 2.097.152.000 would require more than 128 years on my PC!
  - Probably, this could not even be solved by buying new hardware!
- Algorithms should have minimal complexity
  - This is one of the tasks of numerical analysis
  - Clearly, this is not always possible

Visualization 1/4

```
load runtime_comparison;

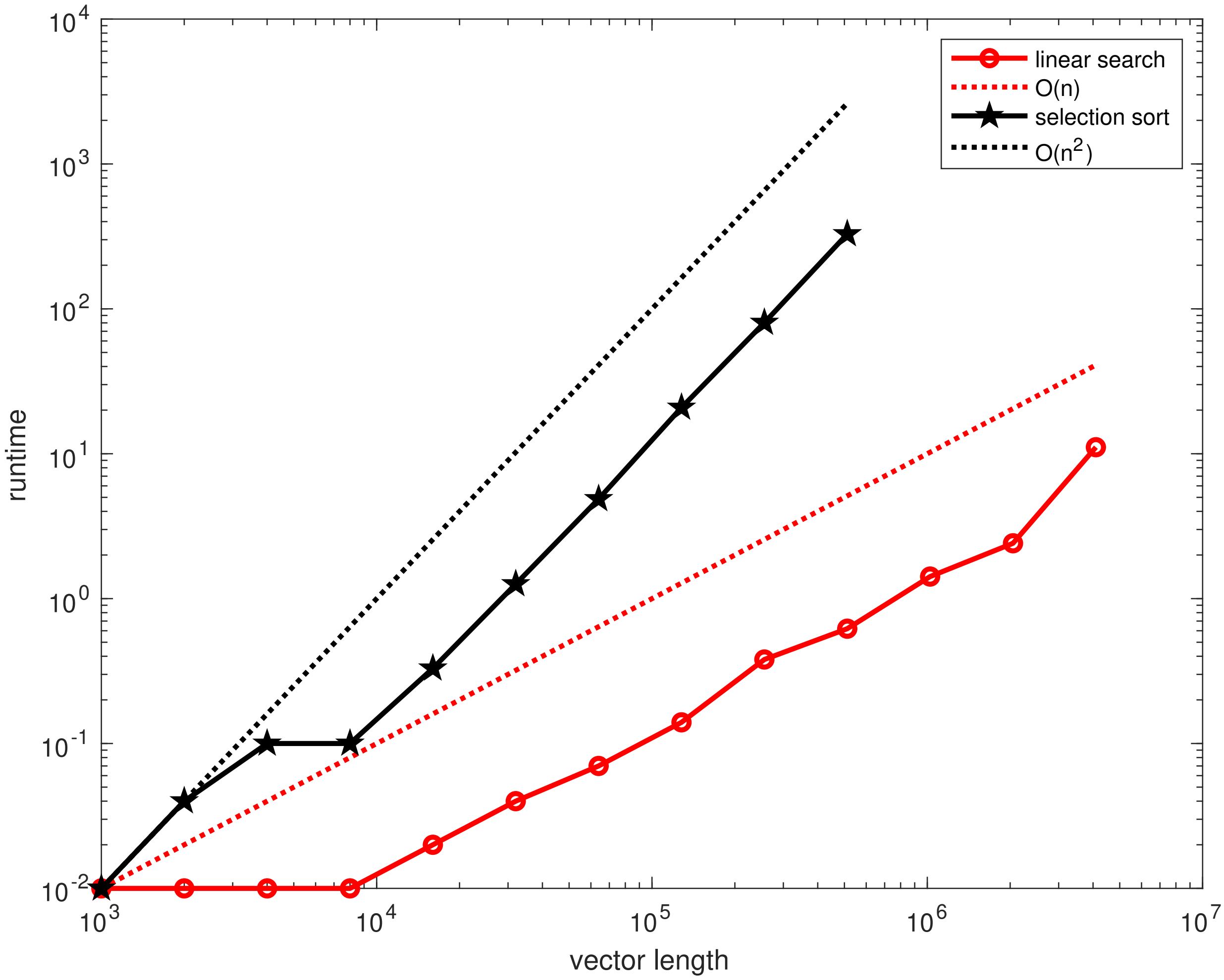
loglog(n2,t2,'r-o','LineWidth',2);
hold on;
loglog(n2,n2/n2(1)*t2(1),'r:','LineWidth',2);

loglog(n3,t3,'k-p','LineWidth',2);
loglog(n3,n3.^2/n3(1)^2*t3(1),'k:','LineWidth',2);
hold off;

ylabel('runtime')
xlabel('vector length')
legend('linear search','O(n)',
       'selection sort','O(n^2)');
print -depsc2 complexity_loglog.eps
```

- Recall that $T(n) = \mathcal{O}(n^\alpha)$ is visualized via loglog
  - $T(n)$ is the runtime for a vector $x \in \mathbb{R}^n$
  - $\alpha > 0$ is the dependence
    - $\alpha = 1$ is linear complexity
    - $\alpha = 2$ is quadratic complexity

Visualization 2/4



- Recall that $T(n) = \mathcal{O}(n^\alpha)$ is visualized via `loglog`
  - $T(n)$ is the runtime for a vector $x \in \mathbb{R}^n$
  - $\alpha > 0$ is the dependence
    - $\alpha = 1$ is linear complexity
    - $\alpha = 2$ is quadratic complexity

Visualization 3/4

```
load runtime_comparison;

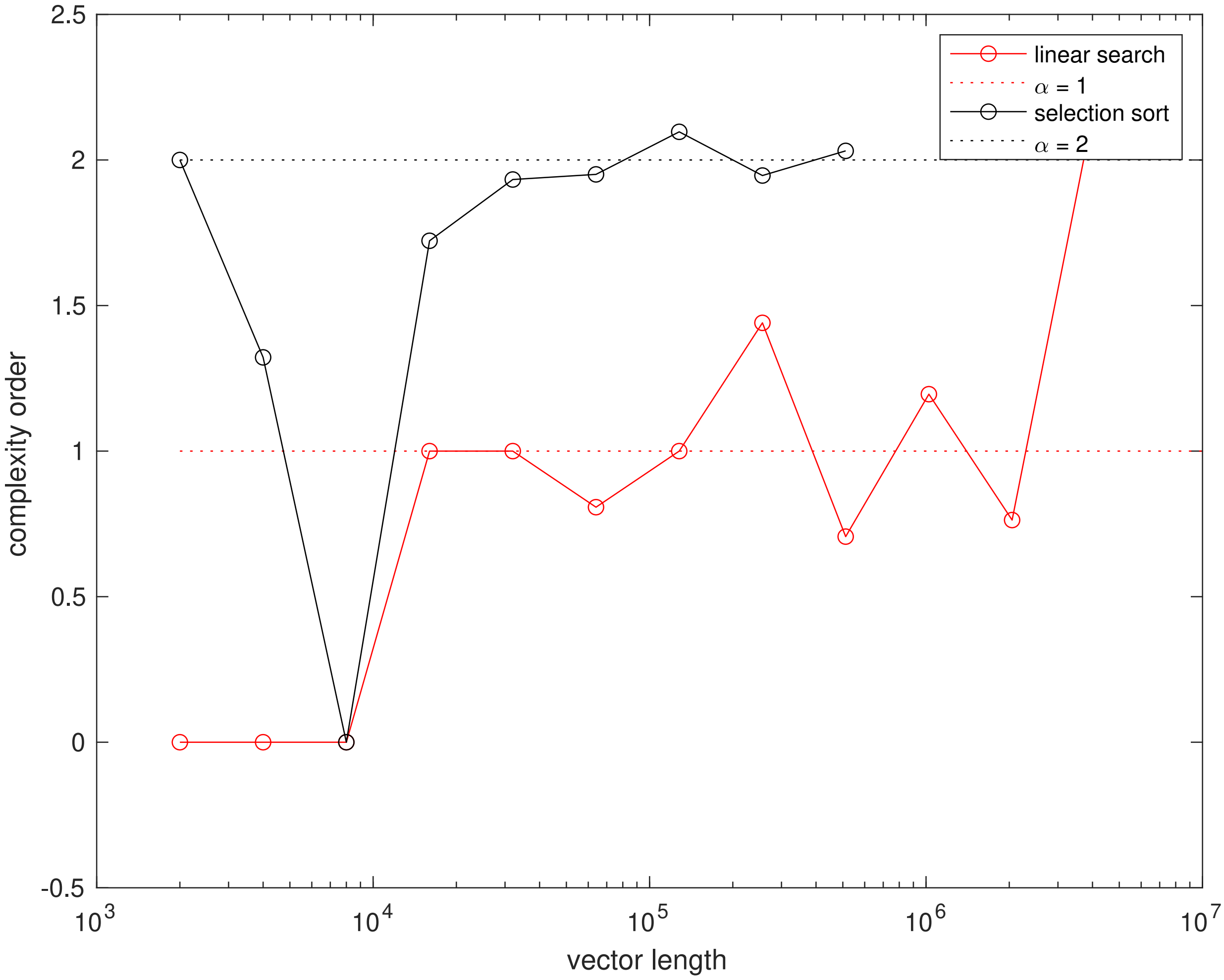
figure(2);
alpha2 = log(t2(2:end)./t2(1:end-1)) / log(2);
semilogx(n2(2:end),alpha2,'r-o');
hold on;
semilogx([n2(2),1e7],[1,1],'r:');

alpha3 = log(t3(2:end)./t3(1:end-1)) / log(2);
semilogx(n3(2:end),alpha3,'k-o');
semilogx([n3(2),1e7],[2,2],'k:');
hold off;

ylabel('complexity order');
xlabel('vector length');
legend('linear search','\alpha = 1',
       'selection sort','\alpha = 2');
print -depsc2 complexity_semilogx.eps
```

- With $T(n) = \mathcal{O}(n^\alpha)$, we suppose that $T(n) = Cn^\alpha$
  - with unknown $C, \alpha > 0$
- Then, $T(2n)/T(n) = 2^\alpha$
  - and hence $\alpha = \log_2\big(T(2n)/T(n)\big)/\log_2(2)$
- We can thus also plot the experimental α over $2n$, where the x-axis is scaled logarithmically
- The computed nodes should be "almost constant"

Visualization 4/4



- With $T(n) = \mathcal{O}(n^\alpha)$, we suppose that $T(n) = Cn^\alpha$
  - with unknown $C, \alpha > 0$
- Then, $T(2n)/T(n) = 2^\alpha$
  - and hence $\alpha = \log_2\big(T(2n)/T(n)\big)/\log_2(2)$
- We can thus also plot the experimental α over $2n$, where the x-axis is scaled logarithmically
- The computed nodes should be "almost constant"

Necessity of memory allocation 1/3

```
clear all;
N = 1e8;

%*** for loop without allocation
t = cputime;
for i = 1:N
    x(i) = i;
end
fprintf("dynamic:   %f sec\n",cputime - t);

clear x t i

%*** for loop with allocation
t = cputime;
x = zeros(1,N);
for i = 1:N
    x(i) = i;
end
fprintf("allocated: %f sec\n",cputime - t);

clear x t i

%*** MATLAB built-in arithmetics
t = cputime;
x = 1:N;
fprintf("built-in:  %f sec\n",cputime - t);
```

- Output:

```
dynamic:   8.600000 sec
allocated: 0.430000 sec
built-in:  0.350000 sec
```

Necessity of memory allocation 2/3

```
clear all;
N = 2*1e3;

%*** for loop without allocation
t = cputime;
for i = 1:N
  for j = 1:N
    x(i,j) = i*j;
  end
end
fprintf("dynamic:   %f sec\n",cputime - t);

clear x t i j

%*** for loop with allocation
t = cputime;
x = zeros(N,N);
for i = 1:N
  for j = 1:N
    x(i,j) = i*j;
  end
end
fprintf("allocated: %f sec\n",cputime - t);
```

- Output:

```
dynamic:   3.310000 sec
allocated: 0.090000 sec
```

Hidden computational time

- Since the previous runtimes do not look intimidating on a first glance, one should consider the computational complexity!
- Recall that matrices $A \in \mathbb{K}^{m\times n}$ are stored columnwise in MATLAB
- If the matrix is getting new rows and is extended to $A \in K^{M\times N}$, then all entries of A (except A_{j1} for $j = 1, \ldots, m$) must either be moved or initialized
  - This needs $\mathcal{O}(MN)$ operators
- In the last example, the matrix grows from a scalar $A \in \mathbb{R}$ over row vectors $A \in \mathbb{R}^{1\times k}$ and $A \in \mathbb{R}^{1\times N}$ to matrices $A \in \mathbb{R}^{j\times N}$ and finally $A \in \mathbb{R}^{N\times N}$
  - This amounts to $\sum_{j=2}^{N} \mathcal{O}(jN) = \mathcal{O}\big(N \sum_{j=2}^{N} j\big)$ operations
- Note that $N \sum_{j=2}^{N} j = N\big(\frac{N(N+1)}{2} - 1\big) = \mathcal{O}(N^3)$
- Overall, dynamic growth of the matrix leads to a hidden cubic complexity $\mathcal{O}(N^3)$, while the visible (algorithmic) complexity for filling the matrix is only $\mathcal{O}(N^2)$.

Sparse matrices

Sparse matrices

- A matrix $A \in \mathbb{K}^{m \times n}$ is called **sparse** if most of its entries are 0
  - i.e., number $\#\{(i,j) \mid A_{ij} \neq 0\} = \mathcal{O}(m+n)$ for $m, n \to \infty$
- Important examples are diagonal matrices, tridiagonal matrices, or more general matrices with so-called band structure
  - Such matrices appear often in applications
- Sparse matrices can be stored more efficiently with $\mathcal{O}(m+n)$ instead of $\mathcal{O}(mn)$, if only the non-zero entries are stored
- Many algorithms like matrix-vector multiplication (and also solvers) can be implemented more efficiently for sparse matrices

Coordinate format

- $N := \#\{(i,j) \mid A_{ij} \neq 0\}$ number of non-zero entries
- The so-called coordinate format relies on naively storing three vectors $I \in \mathbb{R}^N$, $J \in \mathbb{R}^N$, $a \in \mathbb{K}^N$
- Then, $1 \leq k \leq N$, $i = I(k)$, $j = J(k)$ $\Rightarrow$ $A_{ij} = a(k)$
- **Advantage:** Matrix-vector multiplication and storage are clearly $\mathcal{O}(N)$ instead of $\mathcal{O}(mn)$
- **Disadvantage:** Each access to A_{ij} may also need $\mathcal{O}(N)$ operations via linear search
- Note: For any matrix $A \in \mathbb{K}^{m\times n}$ that is dense or sparse, MATLAB provides the coordinate format by `[I,J,a] = find(A);`

Example

- $A = \begin{pmatrix} 10 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 3 \\ 0 & 7 & 8 & 0 & 0 \\ 3 & 0 & 8 & 5 & 0 \\ 0 & 8 & 0 & 9 & 13 \\ 0 & 4 & 0 & 2 & -1 \end{pmatrix}$
- $a = (10, 3, 3 \,|\, 9, 7, 8, 4 \,|\, 8, 8 \,|\, -2, 5, 9, 2 \,|\, 3, 13, -1)$
- $I = (1, 2, 4 \,|\, 2, 3, 5, 6 \,|\, 3, 4 \,|\, 1, 4, 5, 6 \,|\, 2, 5, 6)$
- $J = (1, 1, 1 \,|\, 2, 2, 2, 2 \,|\, 3, 3 \,|\, 4, 4, 4, 4 \,|\, 5, 5, 5)$

CCS-format

- MATLAB uses the coordinate format for communicating to the user / programmer
- However, it uses the CCS-format for storage
  - **Compressed Column Storage** (also: **Harwell-Boeing-Format**)
- $N := \#\{(i,j) \mid A_{ij} \neq 0\}$ number of non-zero entries
- Vectors $I \in \mathbb{R}^N$, $a \in \mathbb{K}^N$ as before
- Vector $J \in \mathbb{R}^{n+1}$ as follows:
  - $J(k)$ indicates where the k-th column starts in vector I for $1 \leq k \leq n$
  - $J(n+1) := N+1$
- **Improvement:** If only $\mathcal{O}(1)$ elements per column, then the access to A_{ij} needs only $\mathcal{O}(1)$ operations
  - However, the CCS format requires sorted data

Example

- $$A = \begin{pmatrix} 10 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 3 \\ 0 & 7 & 8 & 0 & 0 \\ 3 & 0 & 8 & 5 & 0 \\ 0 & 8 & 0 & 9 & 13 \\ 0 & 4 & 0 & 2 & -1 \end{pmatrix}$$
- $a = (10, 3, 3 \mid 9, 7, 8, 4 \mid 8, 8 \mid -2, 5, 9, 2 \mid 3, 13, -1)$
- $I = (1, 2, 4 \mid 2, 3, 5, 6 \mid 3, 4 \mid 1, 4, 5, 6 \mid 2, 5, 6)$
- $J = (1 \mid 4 \mid 8 \mid 10 \mid 14 \mid\mid 17)$, i.e., $N = 16$

Sparse matrices in MATLAB

- Sparse matrices are allocated by `sparse`
  - e.g., `A = sparse(m,n);`
  - or conversion `A = sparse(matrix);`
    - convert back by `Afull = full(A);`
- MATLAB uses optimized algorithms for sparse matrices that are substantially faster than those for full matrices
- Modification of sparse matrices is costly
  - since CCS-storage vectors are partially sorted
  - and hence memory must be copied
- Building sparse matrices can be costly
  - if one executes `A = sparse(m,n);`
  - and then assigns `A(i,j)`
  - Better:
    - first, build the naive coordinate format $I, J \in \mathbb{R}^N$ and $a \in \mathbb{K}^N$
    - then, use `A = sparse(I,J,a,m,n);` to build the matrix in the sparse format
- Recall: For any $A \in \mathbb{K}^{m \times n}$, MATLAB provides the coordinate format by `[I,J,a] = find(A);`

Sparse matrix 1/4

```
% sparse_naive.m
n = 1e4;

A = sparse( 2*eye(n) ...
          - diag(ones(n-1,1),-1) ...
          + diag(ones(n-1,1),1) );
```

- Example: Build tridiagonal matrix with

$$A = \begin{pmatrix} 2 & +1 & 0 & \cdots & 0 \\ -1 & 2 & +1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & +1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

  - 2 on main diagonal
  - ± 1 on first diagonals above and below

- Build diagonal matrices with `diag`
  - `A = diag(v,n)`
  - Parameter `v` vector for diagonal
  - Parameter `n` indicates offset from main diagonal

- This is a bad solution with runtime $\mathcal{O}(n^2)$
  - We assemble a full matrix in $\mathcal{O}(n^2)$
  - and convert it to `sparse`
    - For $n = 1e4 = 10.000$, this needs $n^2 \times 8$ Bytes ≈ 763 MB auxiliary memory!
    - compare with $(3n - 2) \times 8$ Bytes ≈ 0.23 MB!

Sparse matrix 2/4

```
% sparse_naivefor.m
n = 1e4;

A = sparse(n,n);
A(1,1) = 2;
A(1,2) = 1;
A(n-1,1) = -1;
A(n,n) = 2;

for i = 2:n-1
    A(i,i-1:i+1) = [-1 2 1];
end
```

- Example: Build tridiagonal matrix with

$$A = \begin{pmatrix} 2 & +1 & 0 & \cdots & 0 \\ -1 & 2 & +1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & +1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

- This is an even worse solution, since the runtime even exceeds $\mathcal{O}(n^2)$
  - in i-th step
    - $2 + 3\,(i-2) = \mathcal{O}(i)$ entries in matrix
    - must be sorted for CCS-format
    - Cost is $\mathcal{O}(i \log i)$ per step
  - Hence, the total cost is $\geq \mathcal{O}(\sum_{i=2}^{n-1} i) = \mathcal{O}(n^2)$

Sparse matrix 3/4

```
% sparse_tridiag.m
n = 1e4;

I = zeros(3*(n-2)+4,1);
J = zeros(3*(n-2)+4,1);
a = zeros(3*(n-2)+4,1);

I(1:2) = [1 2];
J(1:2) = [1 1];
a(1:2) = [2 -1];

for i = 2:n-1
    I(3+(i-2)*3:2+(i-1)*3) = [i-1 i i+1];
    J(3+(i-2)*3:2+(i-1)*3) = [i i i];
    a(3+(i-2)*3:2+(i-1)*3) = [1 2 -1];
end

I(end-1:end) = [n-1 n];
J(end-1:end) = [n n];
a(end-1:end) = [1 2];

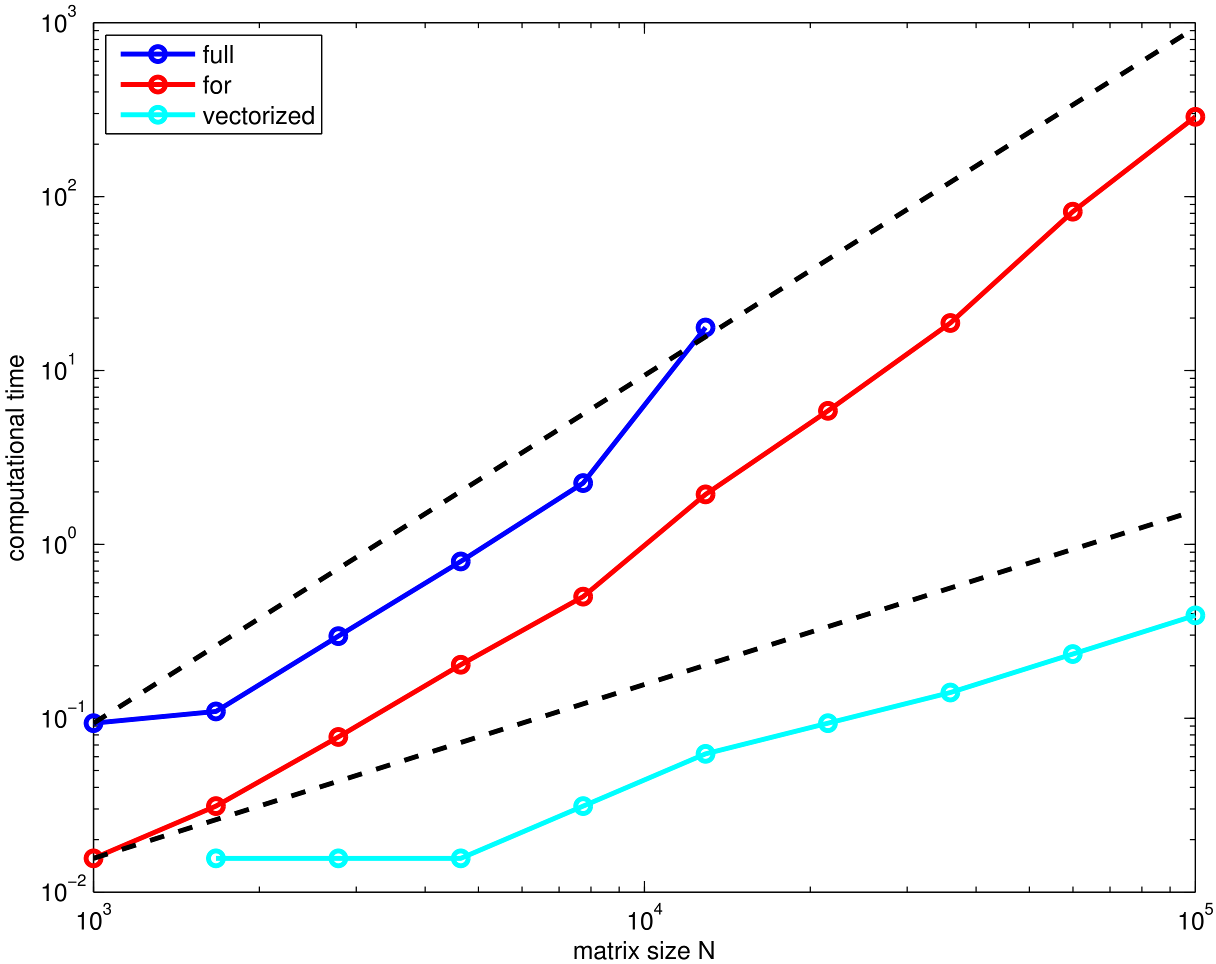
A = sparse(I,J,a,n,n);
```

- Example: Use the coordinate format to build the tridiagonal matrix with

$$A = \begin{pmatrix} 2 & +1 & 0 & \cdots & 0 \\ -1 & 2 & +1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & +1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

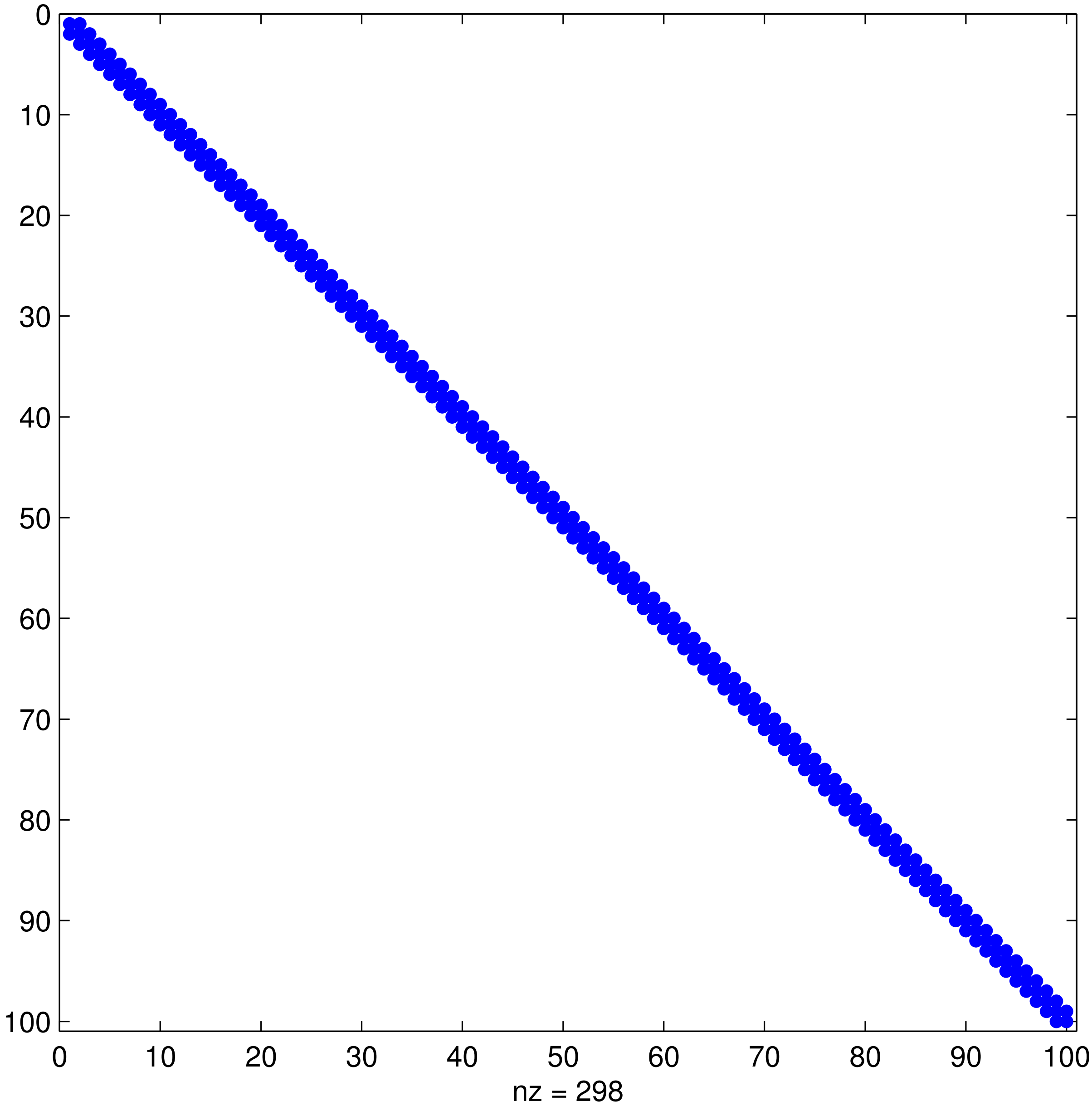
- Advantage: No temporary full matrix needed and the runtime is indeed logarithmic-linear in n, since only $1\times$ sort is required to build the CCS-format

Sparse matrix 4/4



- Visualization by `loglog`
  - time = (size of matrix $N)^{\alpha}$ with $\alpha > 0$
  - $\mathcal{O}(N^2)$ = straight line with slope 2
  - $\mathcal{O}(N)$ = straight line with slope 1

Matrix structure



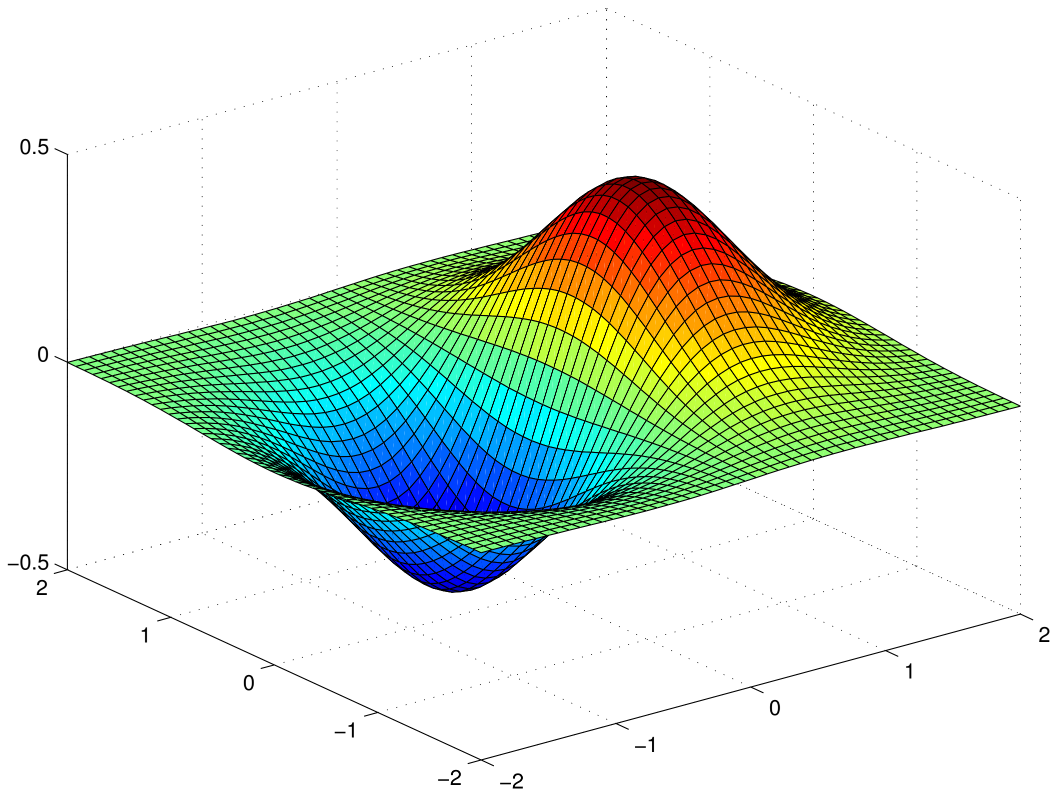
- The structure of the non-zero entries of a matrix is visualized with `spy`
  - Entries $\neq 0$ are shown in a grid
  - matrix indices on both axes
    - here: $A \in \mathbb{R}^{100\times 100}$
    - 298 entries $\neq 0$ (non-zero entries)

Visualization

- Visualization of functions $f : \mathbb{R}^2 \to \mathbb{R}$

- `meshgrid`
- `mesh`, `surf`
- `fill`
- `contour`
- `colorbar`, `colormap`

An example function



- $f(x, y) = x \cdot e^{-(x^2+y^2)}$

Tensor grid

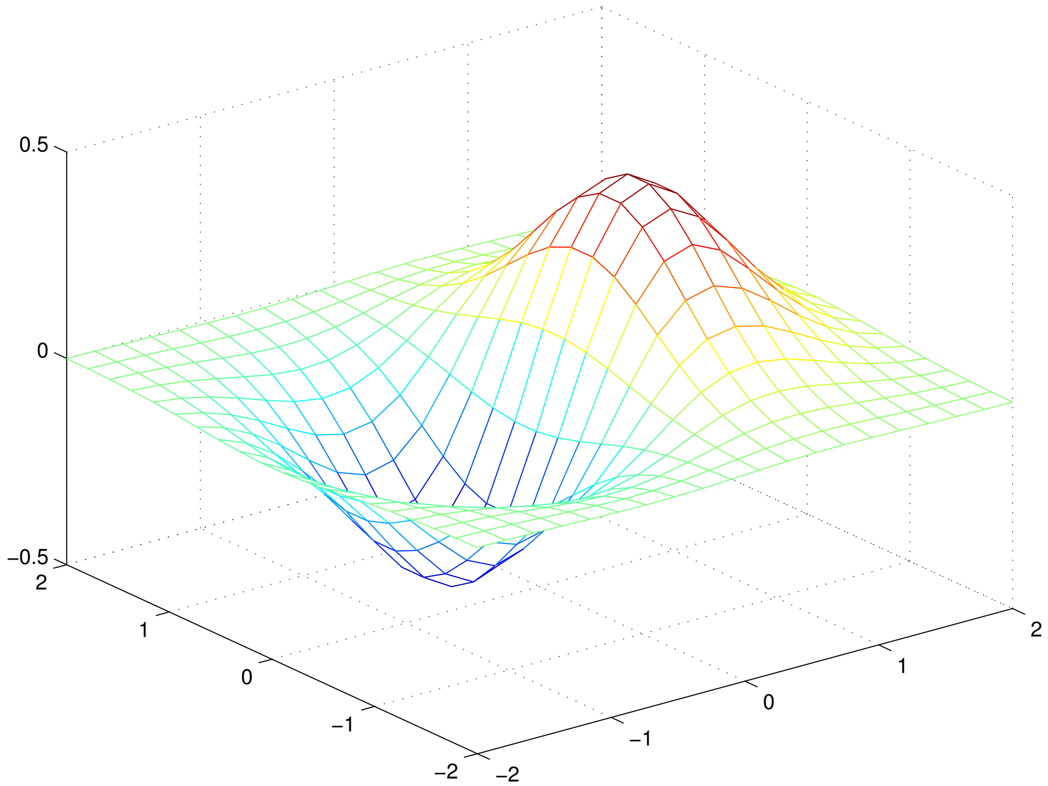
```
f = @(x,y) x.*exp(-x.^2-y.^2);
x = linspace(-2,2,20);
y = linspace(-2,2,20);
[X,Y] = meshgrid(x,y);
Z = f(X,Y);

figure(1)
mesh(X,Y,Z)

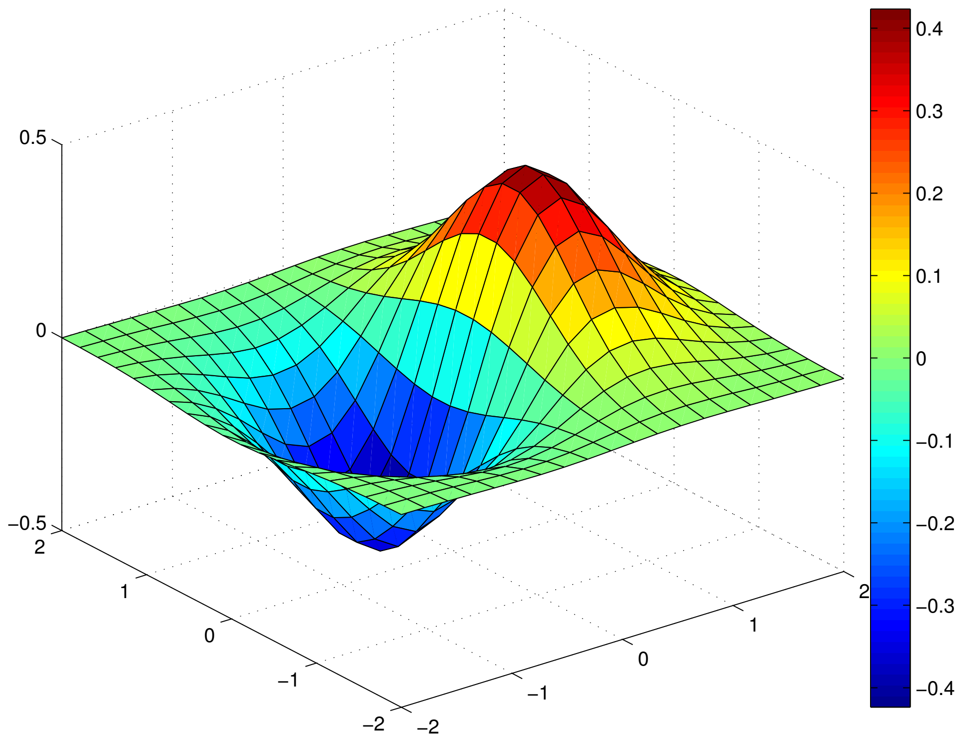
figure(2)
surf(X,Y,Z)
colorbar
```

- Subdivision $x \in \mathbb{R}^n$ of interval I, n nodes
- Subdivision $y \in \mathbb{R}^m$ of interval J, m nodes
- `[X,Y] = meshgrid(x,y)` a tensor grid for $I \times J$
  - i.e., mn nodes in $I \times J$
  - $X, Y \in \mathbb{R}^{m \times n}$
- `mesh(X,Y,Z)` plots function values over tensor grid
  - color according to function value
- `surf(X,Y,Z)` plots function values over tensor grid
  - interpolates between nodes
- `colorbar` returns color code for $z = f(x, y)$
- `colormap(rgb)` chooses RGB-map `rgb` $\in [0, 1]^{N \times 3}$
  - e.g., `jet`, `gray`, `copper`, `hot`, `cool`, `summer`, `winter`

- figure(1) ⟶ mesh



- figure(2) ⟶ surf



Contour plot

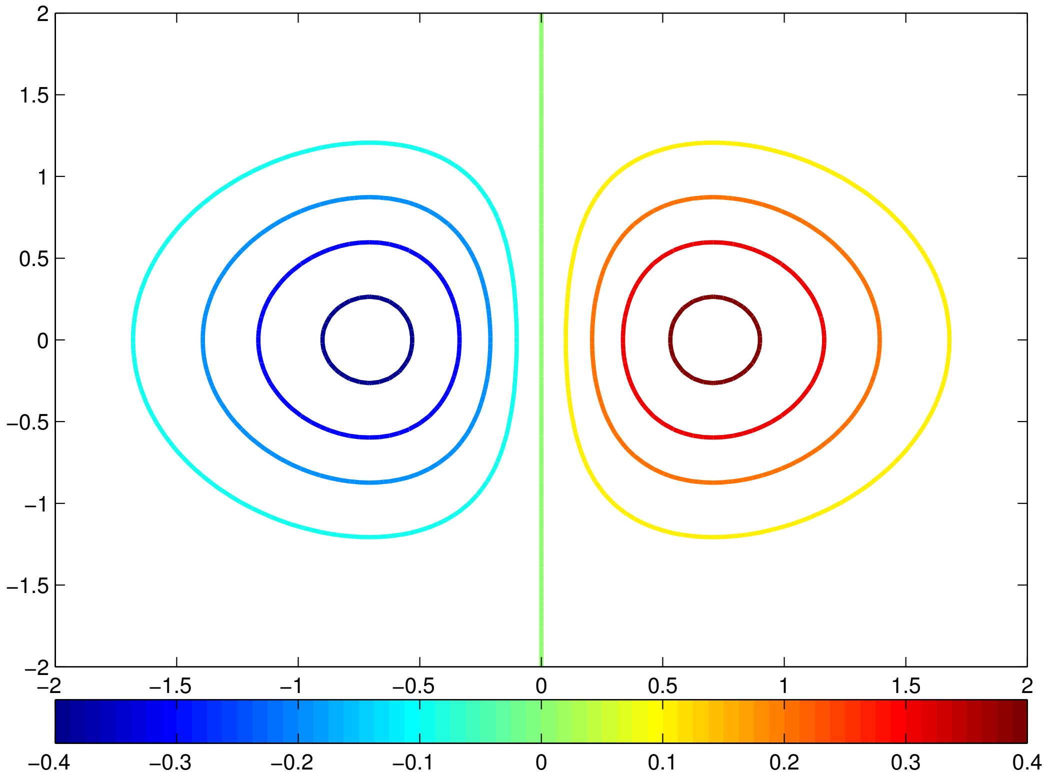
```
f = @(x,y) x.*exp(-x.^2-y.^2);
x = linspace(-2,2,100);
y = linspace(-2,2,100);
[X,Y] = meshgrid(x,y);
Z = f(X,Y);

%*** plot colored contour lines
figure(1)
contour(X,Y,Z,'LineWidth',2)
colorbar('SouthOutside')

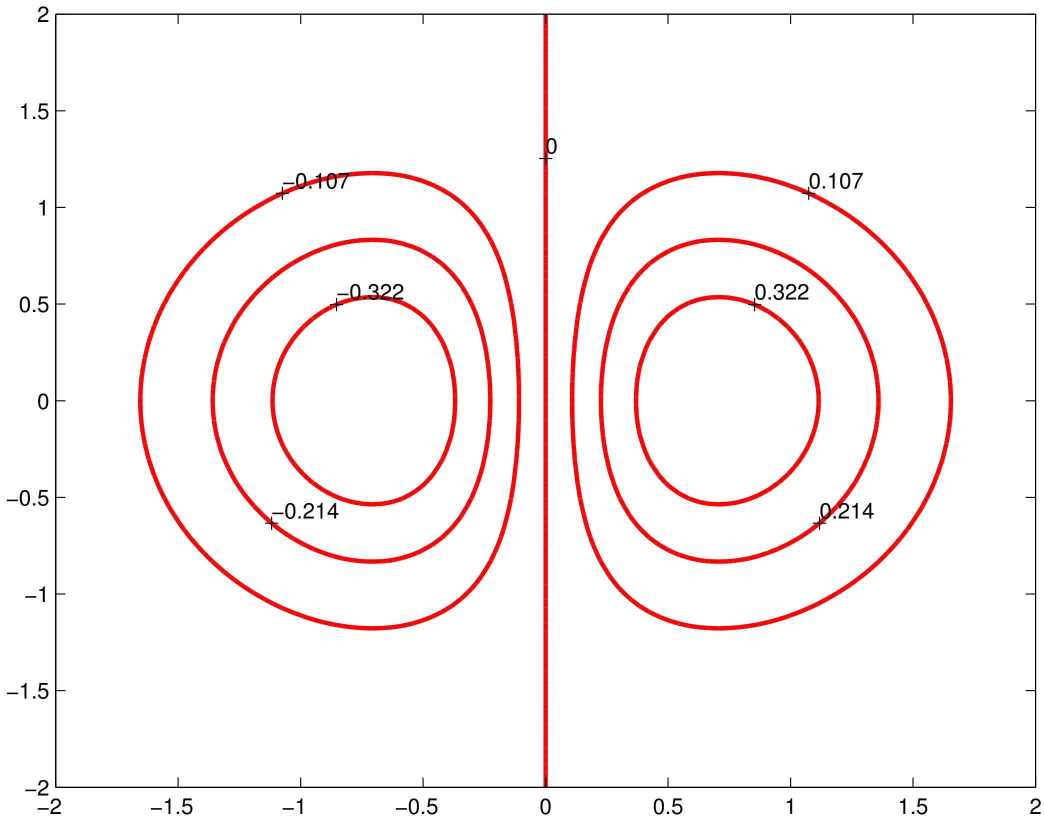
%*** contour lines red, labeled
figure(2)
C = contour(X,Y,Z,...
            7,'LineColor','r','LineWidth',2);
clabel(C)
```

- `contour(X,Y,Z)` shows colored contour lines
- Optional parameters
  - number of contour lines (default is 9)
  - further options like for `plot`
- Labeling of contour lines with `Z`-value
  - by `clabel`

▶ figure(1)



▶ figure(2)



Projection to plane

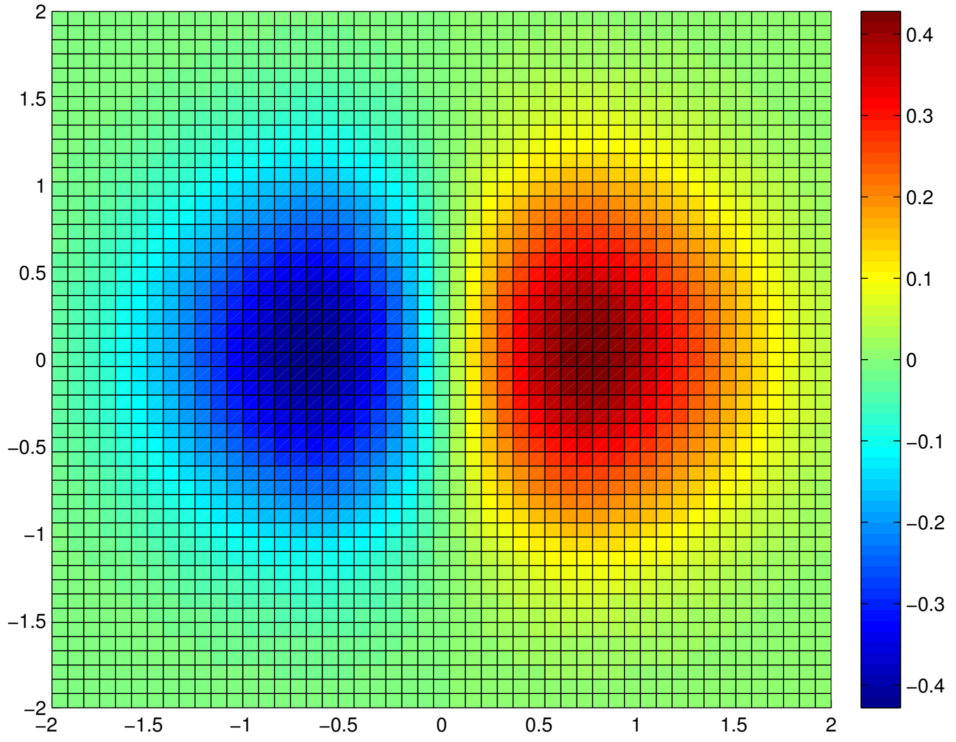
```
f = @(x,y) x.*exp(-x.^2-y.^2);
x = linspace(-2,2,50);
y = linspace(-2,2,50);
[X,Y] = meshgrid(x,y);
Z = f(X,Y);

figure(1)
surf(X,Y,Z);
view(2)
colorbar

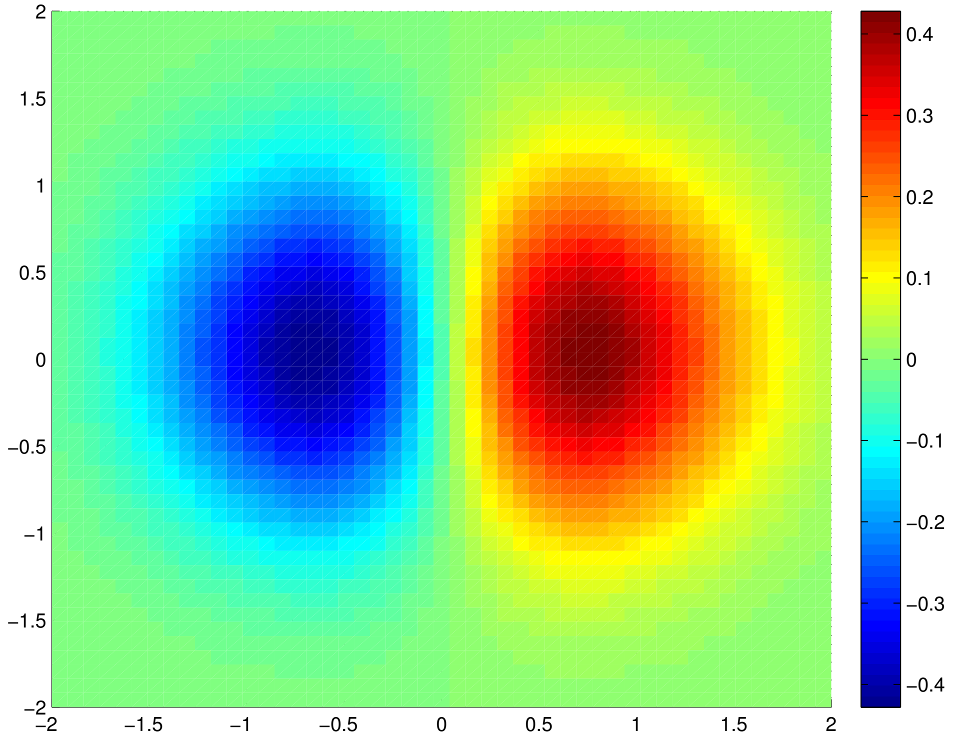
figure(2)
surf(X,Y,Z,'LineStyle','none');
colorbar
view(2)
```

- `view(azimuth,elevation)` : Location of observer
  - `elevation` = altitude angle over x-y-plane
  - `azimuth` = angle in x-y-plane
- `view(2)` = 2D from above onto x-y-plane
  - i.e., `azimuth`=0, `elevation`=90
- `view(3)` = standard 3D-settings
- `[azimuth,elevation] = view` returns current values
  - possible to rotate 3D-picture by mouse
  - read and store "good" settings by this method

▶ figure(1)



▶ figure(2)



Non-tensor grid

```
f = @(x,y) x.*exp(-x.^2-y.^2);

%*** compute known values of function
x = 4*rand(1,200)-2; % random numbers in [-2,2]
y = 4*rand(1,200)-2; % random numbers in [-2,2]
z = f(x,y);

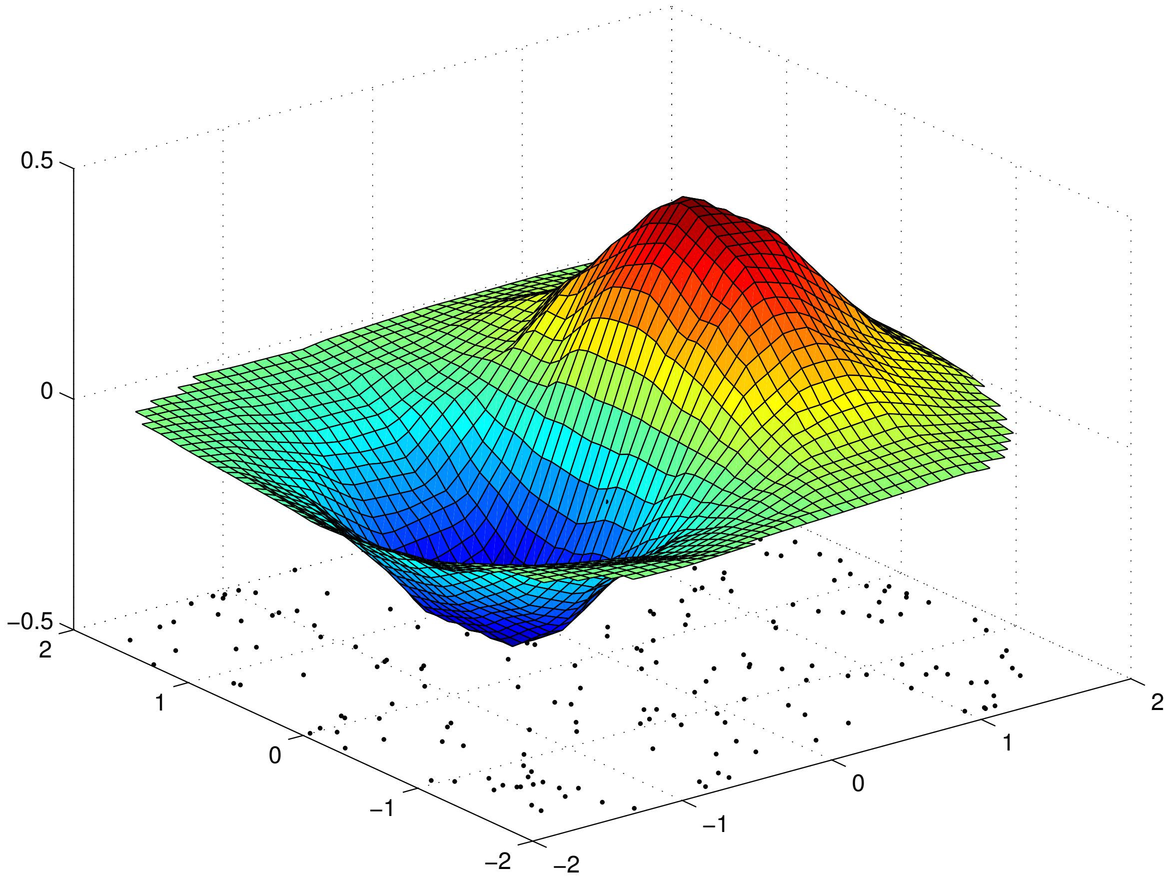
%*** build tensor grid
xx = linspace(-2,2,50);
yy = linspace(-2,2,50);
[X,Y] = meshgrid(xx,yy);

%*** approximate function values
Z = griddata(x,y,z,X,Y);

%*** plot approximated function
surf(X,Y,Z)
hold on

%*** plot random points
plot3(x,y,-.5*ones(size(x)),'k.')
hold off
```

- If data points (x, y) are not on a tensor grid
  - build tensor grid by `meshgrid`
  - approximate function values on tensor grid from known values


0.5
0
−0.5
2
1
0
−1
−2
−2
−1
0
1
2

Triangular grids

```
f = @(x,y) x.*exp(-x.^2-y.^2);

%*** compute known values of function
x = 4*rand(1,1000)-2; % random numbers in [-2,2]
y = 4*rand(1,1000)-2; % random numbers in [-2,2]
z = f(x,y);

%*** build triangulation
tri = delaunay(x,y);

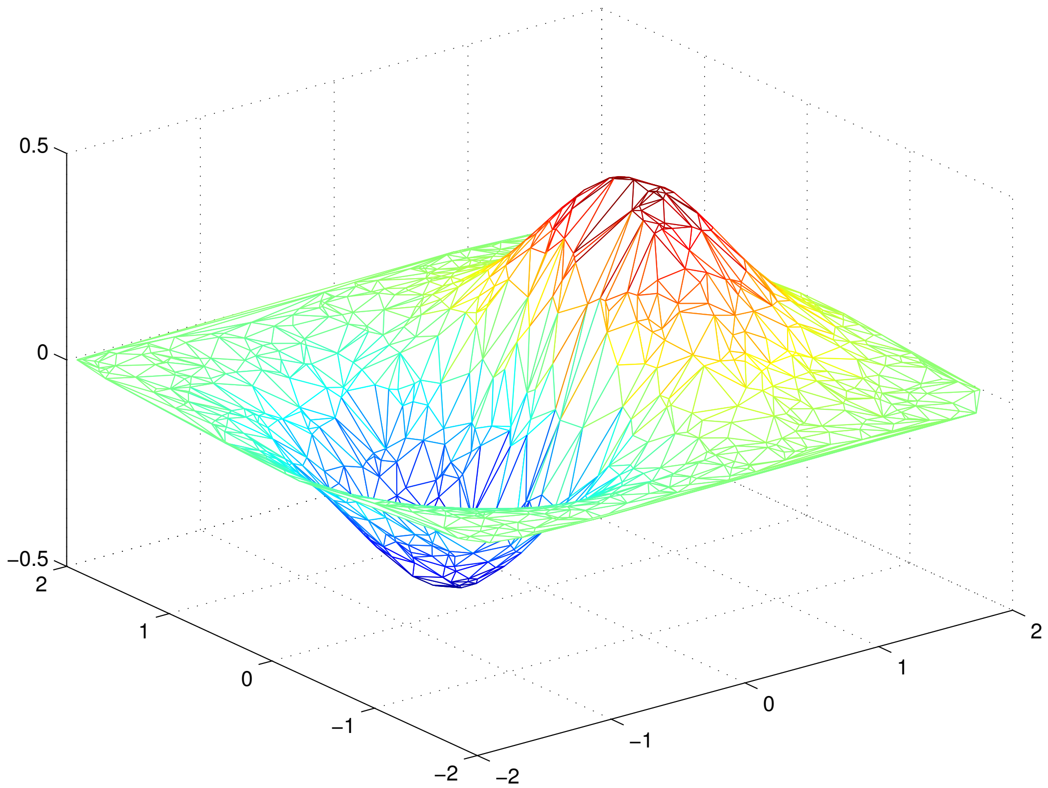
%*** plot approximated function
figure(1)
trimesh(tri,x,y,z);

figure(2)
trisurf(tri,x,y,z);

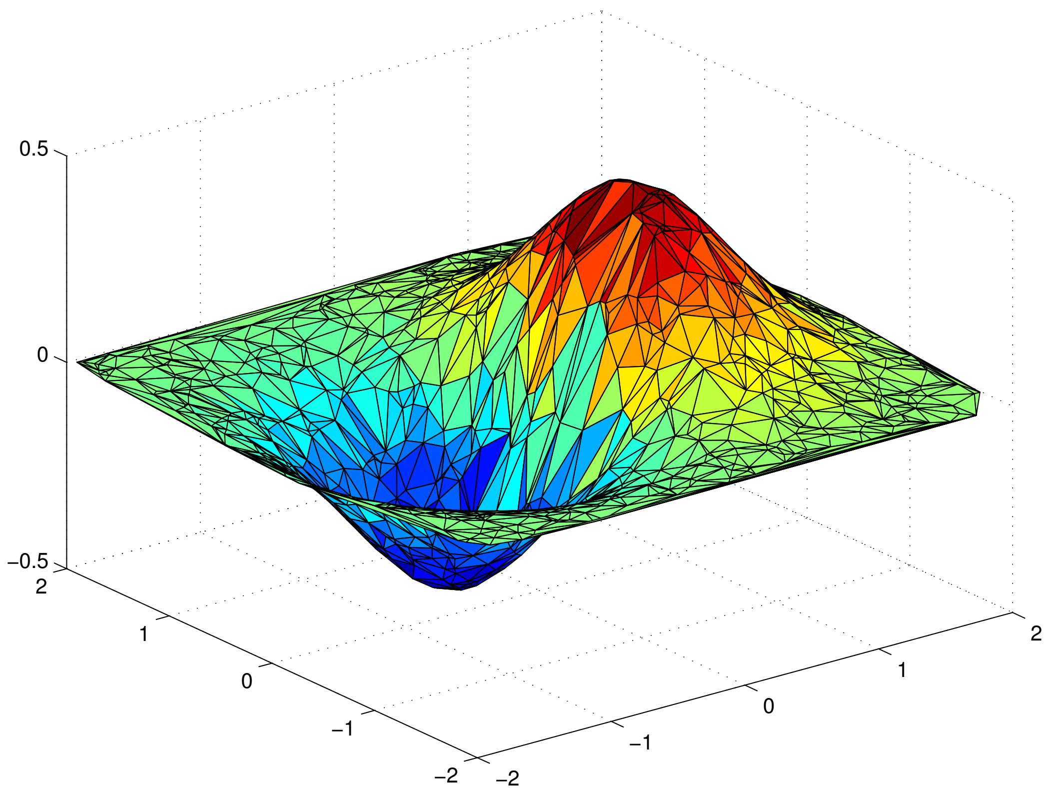
%*** show triangulation
figure(3)
trimesh(tri,x,y,zeros(size(x)),'EdgeColor','k')
view(2)
```

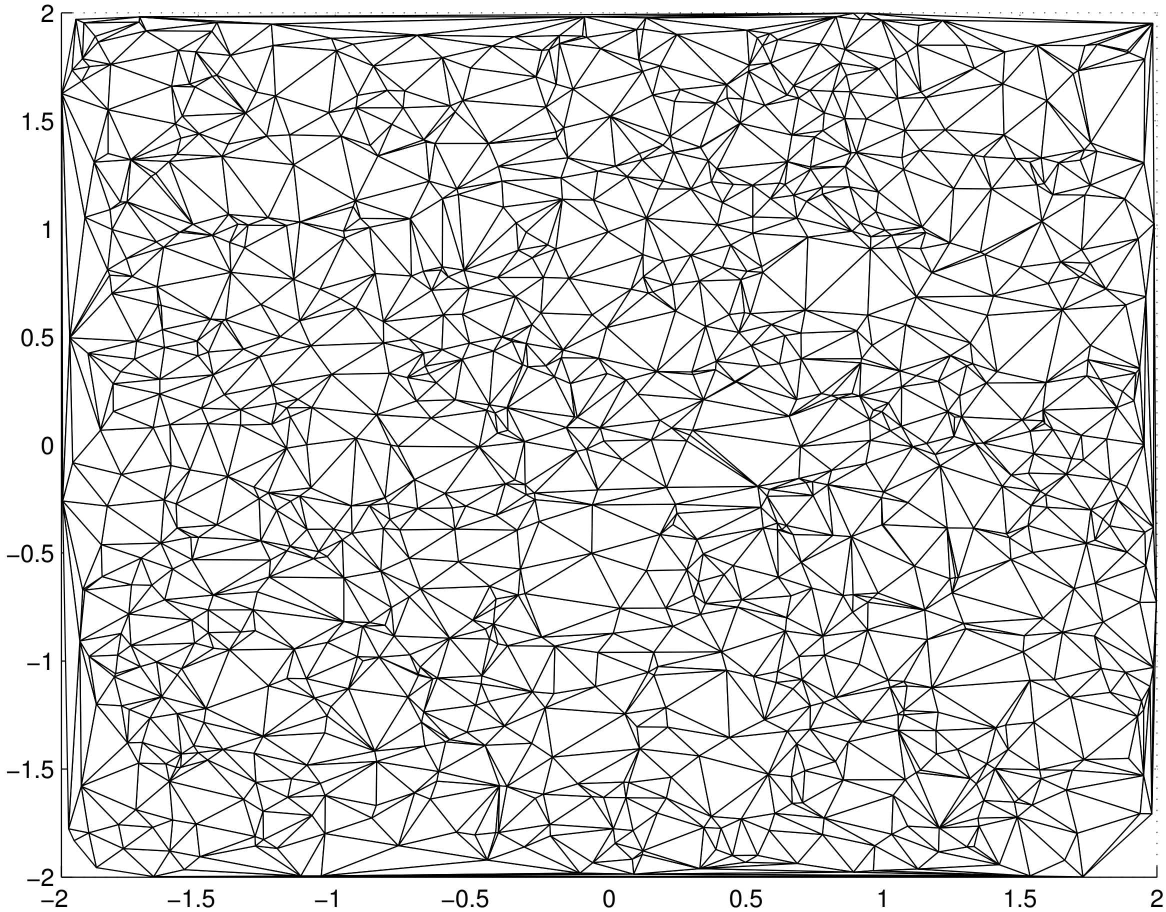
- Creates a so-called Delaunay triangulation of points into triangles
  - nodes of triangles = given points
  - nodes of each triangle determine a unique circle
    - and this circle does not contain further points
- This ensures that the angles of the triangles are as large as possible, which is numerically favorable

▶ figure(1) ⟶ trimesh



▶ figure(2) ⟶ trisurf




2
1.5
1
0.5
0
−0.5
−1
−1.5
−2
−2
−1.5
−1
−0.5
0
0.5
1
1.5
2

Some further commands

- Plots in polar coordinates : `polar`
- Bar charts : `hist`, `bar`, `barh`
- Pie charts : `pie`, `pie3`
- Fill area/volume with color : `fill`, `fill3`
- Vector fields : `compass`, `quiver`, `quiver3`
- Animations: `VideoWriter`