(eng. computational complexity). En algoritm (eng. algorithm, se Wikipedia för etymologin) är en metod för att lösa ett problem. Ett program är en implementation (realisering) av algoritmen.

En viktig del i algoritmkurser är att studera algoritmens beräkningskomplexitet, dvs. hur algoritmens minnesbehov och tidsåtgång varierar med problemstorleken.
På dessa sidor tar vi endast upp tidsaspekten.

Exempel. (Lab1) Att lösa $\boldsymbol{Ax} = \boldsymbol{b}$ där $\boldsymbol{A}$ är en $n \times n$-matris kräver ungefär $n^3/3$ additioner och multiplikationer. Om det tar tiden τ att utföra ett (+, *)-par är tidsåtgången, $\boldsymbol{T}(n)$, därför

$$\boldsymbol{T}(n) = \frac{n^3\tau}{3}$$

■

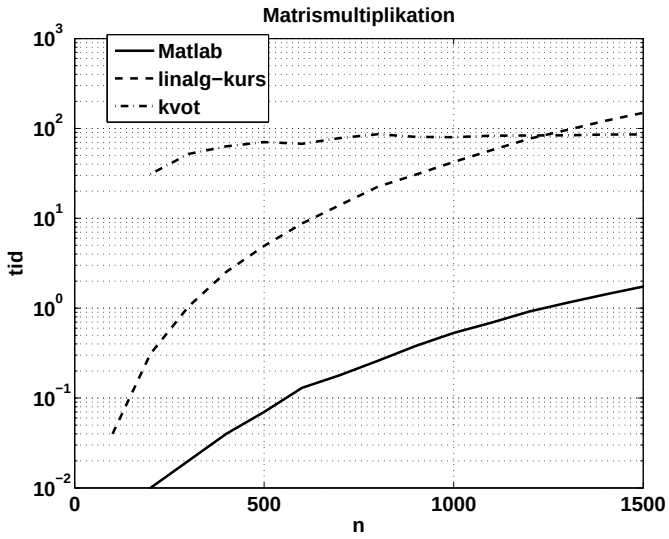
Vi kan använda sådan information för att

- Uppskatta beräkningstiden
- Se hur stort problem vi kan lösa
- Välja bland olika algoritmer

I praktiken räcker det inte att veta hur snabbt datorn utför ett (+, *)-par eftersom minnet spelar så stor roll. Två program som utför lika många operationer kan skilja sig avsevärt i tid eftersom minnet spelar in.

Exempel. Låt $\boldsymbol{A}$, $\boldsymbol{B}$ och $\boldsymbol{C}$ vara $n \times n$-matriser. Den vanliga algoritmen, från kursen i linjär algebra, tar n^3 multiplikationer och $n^2(n-1)$ additioner. Den rutin som används i MATLAB kräver lika många räkneoperationer, men den utnyttjar minnet på ett effektivare sätt.

Här en bild



MATLABs egen rutin är nästan 100 gånger snabbare än den enkla varianten. ■

Exempel. Nästlade loopar kan ta tid:

```
Program 1       Program 2         Program 3

for i = 1:n     for i = 1:n       for i = 1:n
  work            for j = 1:n       for j = 1:n
end                 work                for k = 1:n
                  end                     work
                end                     end
                                      end
                                    end
```

Antag att `work` tar tiden τ i samtliga fall. Tiderna blir då

$$\boldsymbol{T}_k(n) = n^k\tau,\ k = 1, 2, 3$$

Antag att $n = 10^6$ och att $\tau = 10^{-8}s$. Då blir

$$\boldsymbol{T}_1(n) = 0.01s,\ \boldsymbol{T}_2(n) \approx 2.8 \text{ timmar},\ \boldsymbol{T}_3(n) \approx 317 \text{ år}.$$

Antalet nästlade loopar *spelar stor roll*. Observera dock att tre loopar, som i Program 1, *efter varandra*, endast tar $0.03s$. ■

Det är vanligt (vanligare i datalogi än i numerisk analys) att man använder stort ordo (från latin, ordning, eng. big O) för ange komplexiteten. I detta sammanhang säger vi att $\boldsymbol{T}(n) = \boldsymbol{O}(n^p)$ eller $\boldsymbol{T}(n) = \mathcal{O}(n^p)$ om $\boldsymbol{T}(n) \leq \boldsymbol{M} \cdot n^p,\ \boldsymbol{M} > 0$ när $n \to \infty$.

Exempel. Säg att $\boldsymbol{T}(n) = c_3n^3 + c_2n^2 + c_1n + c_0$ där c_k är reella konstanter och $\boldsymbol{T}(n) > 0, n > 0$. Det gäller att $\boldsymbol{T}(n) = \mathcal{O}(n^3)$ ty

$$\boldsymbol{T}(n) = c_3n^3 + c_2n^2 + c_1n + c_0 = n^3\left[c_3 + \frac{c_2}{n} + \frac{c_1}{n^2} + \frac{c_0}{n^3}\right] \leq$$

$$n^3\left[c_3 + \frac{|c_2|}{n} + \frac{|c_1|}{n^2} + \frac{|c_0|}{n^3}\right] \leq \boldsymbol{M}n^3$$

för tillräckligt stort n. ■

Exempel. Den sorteringsalgoritm som används i MATLAB har (sannolikt) komplexiteten $\mathcal{O}(n \log n)$ där n är antalet element i vektorn som skall sorteras. Detta gör att det går väldigt snabbt att sortera tal.

$$n \log n = \frac{n \log_{10} n}{\log_{10} e} \Rightarrow \mathcal{O}(n \log n) = \mathcal{O}(n \log_{10} n)$$

Så, sorteringstiden växer lite snabbare än linjärt med n.
Att sortera 10^6 tal tar ungefär $0.3s$. ■

I numerisk analys är det mindre vanligt med ordo-notation i detta sammanhang. Att lösa $\boldsymbol{Ax} = \boldsymbol{b}$, när $\boldsymbol{A}$ är symmetrisk, tar ungefär $n^3/6$ (+, *)-par (halva tiden jämfört med det osymmetriska fallet), $\mathcal{O}(n^3/3) = \mathcal{O}(n^3/6) = \mathcal{O}(n^3)$ men det är skillnad på 10 minuter och 20 minuter.

I exemplet med matrismultiplikation så är båda metoderna $\mathcal{O}(n^3)$ men den ena tar hundradelen så mycket tid.

På samma sätt vill man skilja mellan två metoder med $\boldsymbol{T}_1(n) = n^3$ och $\boldsymbol{T}_2(n) = 1000n^2$. Den första metoden är $\mathcal{O}(n^3)$ och den andra $\mathcal{O}(n^2)$, men den första metoden är snabbare när $n < 1000$, fastän det är en n^3-metod.

Däremot kan man tillåta säg att approximera $n^2(n-1) \approx n^3$ när det gäller matrismultiplikation.

6 Fält - vektorer och matriser, en första bekantskap

Detta är en inledning till vektorer och matriser, så att vi kan klara av intressantare laborationer. Det kommer mer efter hand.
Ett fält eller en lista är en följd av element. I MATLAB-sammanhang talar man oftare om vektorer (endimensionellt fält) och matriser (tvådimensionellt fält, eng. one- eller two-dimensional array).
MATLAB står ju för MATrix-LABoratory och man ser MATLABs styrka vid matrisräkning.

6.1 Vektorer

Precis som i linjäralgebrakursen har vektorn ett namn och vi indexerar de individuella elementen. Först några sätt att skapa vektorer:

```
>> vek = [3 6 -8]     % skapa en radvektor
vek = 3     6    -8

>> vek = [3, 6, -8]  % komma går också bra
vek = 3     6    -8

>> v = [2+3, 4*7, -3]  % kan ha numeriska uttryck i
v = 5    28    -3

>> v = [2 +3, 4*7, -3]  % Varning!
v = 2     3    28    -3

>> v = [2 + 3, 4*7, -3] % Varning
v = 5    28    -3

>> vek(1)             % index inom ( )
ans = 3

>> vek(2)
ans = 6
```



```
>> vek(0)             % indexfel
??? Subscript indices must either be real positive
    integers or logicals.

>> vek(4)             % indexfel
??? Index exceeds matrix dimensions.

>> vek(4) = 146       % vektorn utvidgas
vek = 3     6    -8   146

>> vek(2) + 3 * sqrt(-vek(3))  % numeriska uttryck
ans =                          % som vanligt
   14.4853
```

Notera att vektorer och matriser är matematisk begrepp snarare än datalogiska objekt. Vi (jag) vill alltså ha första index ett och inte noll som i C/C++. I den numeriska språket Fortran är första index ett, som standard, men man kan välja startindex, kan tom vara negativt.

I numerisk analys är det vanligast med kolonnvektorer (kommer nedan) med de tar så stor plats att skriva ut, så i denna genomgång använder jag mest radvektorer.

```
>> vek_tr = vek'      % transponera
vek_tr =
     3
     6
    -8
   146

>> vek_tr(4)          % ett index även för kolonnvektor
ans = 146
```


Här ett annat sätt att skapa kolonner:

```
>> kol = [2; -5; 7]  % ; = radbyte
kol =
     2
    -5
     7

>> x = 1:4            % vanlig typ av vektor
x =                   % jämför  for k = 1:4
     1     2     3     4

>> x = 1:4'
x = 1     2     3     4

>> x = (1:4)'         % behövs ( )
x =
     1
     2
     3
     4

>> y = 5:-2:-4
y = 5     3     1    -1    -3

>> y = 5:-2:6
y =
   Empty matrix: 1-by-0

>> nollor = zeros(1, 3)
nollor = 0     0     0

>> ettor = ones(1, 3)
ettor = 1     1     1
```



```
>> r = rand(1, 3)   % likformig fördelning på [0, 1)
r =
   4.8626e-01   4.2040e-01   8.5471e-01

>> r = randn(1, 3)  % normalfördelning
r =                 % randn(3) blir en 3 x 3-matris
   7.2579e-01  -5.8832e-01   2.1832e+00
```

6.2 Några räkneoperationer för vektorer

De vanliga vektoroperationerna fungerar som vanligt.

```
>> a = 1:3
a = 1     2     3

>> b = 4:6
b = 4     5     6

>> c = a + 2 * b
c = 9    12    15

>> (1:4) + a               % dimensionerna måste stämma
??? Error using ==> plus
Matrix dimensions must agree.

>> a + a'                  % och orienteringen
??? Error using ==> plus
Matrix dimensions must agree.


>> a * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

```
>> a' * b                  % sk ytterprodukt. MATRIS
ans =
     4     5     6
     8    10    12
    12    15    18

>> a * b'                  % inner(skalär)-produkt
ans = 32
```

Mer om vektorer kommer senare, men nu lite om matriser.

6.3 Matriser

Matriser fungerar ungefär som vektorer,

```
>> M = [1 2; 3 4]     % ; = ny rad
M =
     1     2
     3     4

>> M(1, 2)            % matris(rad, kolonn)
ans = 2

>> M(2, 1) = 10       % ändra värde
M =
     1     2
    10     4

>> M'                 % transponat
ans =
     1    10
     2     4

>> M(2, 4) = -77      % utvidgning
M =
     1     2     0     0
    10     4     0   -77
```



```
% Uttryck
>> sqrt(abs(M(1, 2) - M(2, 1) * M(2, 2)))
ans = 3.1623

>> v = [1 2 3 0]';
>> M * v             % matris-vektor multiplikation
ans =
     5
    11

>> [1 2] * M         % rad från vänster
ans = 7    10     0  -154

>> [1 2] * M * v
ans = 27

>> [1 2]' * M        % samma regler som vanligt
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> v = rand
v = 0.5135

>> R = [cos(v), sin(v); -sin(v), cos(v)]
R =
    0.8711    0.4912
   -0.4912    0.8711

>> R' * R            % matrismultiplikation
ans =
     1     0
     0     1

>> 2 * R             % elementvis som vanligt
ans =
    1.7421    0.9824
   -0.9824    1.7421
```


Man kan få flera siffror utskrivna genom att byta utskriftsformat (det ändrar inte på det interna binära formatet dock):

```
>> pi_vek = pi * [1e-10, 1, 1e10];
>> format short
pi_vek =
   1.0e+10 *                    % <-- OBS
    0.0000    0.0000    3.1416

>> format short e               % använder jag oftast
>> pi_vek
pi_vek =
   3.1416e-10   3.1416e+00   3.1416e+10

>> format long
>> pi_vek'                      % transponat
ans =
   1.0e+10 *                    % <-- OBS
   0.000000000000000
   0.000000000314159
   3.141592653589793

>> format long e                % tar mycket plats
>> pi_vek'                      % obs transponat
ans =
     3.141592653589793e-10
     3.141592653589793e+00
     3.141592653589793e+10

>> format bank                  % kronor och ören
>> pi_vek
pi_vek =
          0.00           3.14 31415926535.90

>> format hex   % internt hexadecimalt format
>> pi_vek
pi_vek =
   3df596bf8ce7631e   400921fb54442d18   421d4223fc1f977b
```


7 Stränghantering

En teckensträng (sträng, eng. string) är en vektor av tecken. I MATLAB lagras vektorn som en vektor av motsvarande teckenkoder.

```
>> a_string = 'Matlab'
a_string = Matlab

>> double(a_string)  % teckenkoder
ans = 77    97   116   108    97    98
```

Om man tittar i manualbladet för ascii (se datoravsnittet) så framgår att M har teckenkoden 77, a har koden 97 etc. Det stämmer alltså. Man kan också gå från koder till tecken:

```
>> v = [84 104 111 109 97 115];

>> char(v)
ans = Thomas
```

Man kan också göra roliga saker som:

```
>> 'a':'z'
ans = abcdefghijklmnopqrstuvwxyz

>> 'A':'Z'
ans = ABCDEFGHIJKLMNOPQRSTUVWXYZ

>> 'A':'z'
ans =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
```

I labben kan du behöva konvertera mellan stora och små bokstäver. Så här kan man göra. När jag bildar `s - 'a' + 'A'` subtraherar jag en skalär från en vektor. Otillåtet i kursen i linjär algebra, men tillåtet i MATLAB (matematik och MATLAB-syntax är ibland olika saker). Skalären subtraheras elementvis från elementen i vektorn. Mer om sådant senare.

```
>> s = 'a':'z';
>> char(s - 'a' + 'A')   % teckenaritmetik
ans = ABCDEFGHIJKLMNOPQRSTUVWXYZ

s = 'A':'Z';
>> char(s - 'A' + 'a')
ans = abcdefghijklmnopqrstuvwxyz

% Dock
>> char(a_string  - 'a' + 'A')
ans = -ATLAB    % notera -

% Mera allmänt
>> upper(a_string)
ans = MATLAB

>> lower(a_string)
ans = matlab
```

Skilj mellan tecknet `'0'` och siffran `0`.

```
>> s = '0':'9'   % tecken
s = 0123456789

>> double(s)     % teckenkoder
ans = 48  49  50  51  52  53  54  55  56  57

>> s - '0'       % siffror
ans = 0   1   2   3   4   5   6   7   8   9
```


Här en strängmatris:

```
>> S = ['a', 'b', 'c'; 'd', 'e', 'f']
S =
abc
def

>> S(1,2)
ans = b

>> Sa = ['abc'; 'def']   % eller
Sa =
abc
def

>> Sa(1,2)
ans = b
```

Man kan utföra numeriska operationer med strängdata, men det är kanske inte så meningsfullt:

```
>> S * S'
ans =
       28814       29696
       29696       30605
```

Värdena svarar inte mot några tecken.

8 Fält och loopar

Nu till kombinationen av fält och loopar. Vi kommer att göra detta mycket kortare så småningom. Man måste dock behärska den klumpiga tekniken också, eftersom det är den enda som står till buds i C och Java t.ex.
Vi kommer nu att se på mer fullständiga program. Det är vanligt att man programmerar på engelska, med engelska variabelnamn och kommentarer. Jag kommer att använda engelska variabelnamn men svenska kommentarer. Normalt skriver jag kommentarerna på engelska också. Eftersom dessa sidor är skrivna i ett stort typsnitt, har jag ibland lite kryptiska variabelnamn för att slippa bryta raderna.
Beräkna längden (normen) av en vektor.

```
v = randn(100, 1);   % 100 värden
s = 0;
for k = 1:length(v)  % length(v) = 100
   s = s + v(k) * v(k);
 end
s = sqrt(s);
norm(v) - s          % 1.7764e-15 i detta fall
```

Beräkna antalet element, i en slumpvektor, som är mindre än en halv (simulering av slantsingling).

```
tosses = rand(100000, 1);
heads  = 0;
for k = 1:length(tosses)
  if tosses(k) < 0.5
    heads = heads + 1;
  end
end
heads   % = 50140 i detta fall
```

Beräkna det största elementet, samt motsvarande rad- och kolonnindex, i en matris.


```
M     = magic(30);  % en magisk kvadrat
max_M = M(1, 1); max_row = 1; max_col = 1; % platsbrist

for row = 1:size(M, 1)
  for col = 1:size(M, 2)
    if M(row, col) > max_M
      max_M   = M(row, col);
      max_row = row;
      max_col = col;
    end
  end
end

max_M               % 900 i detta exempel
max_row             % 30
max_col             % 8
M(max_row, max_col) % blir alltså 900
```

Sortera elementen i en vektor, i växande ordning, med den mycket enkla algoritmen "selection sort".

```
v = [7   6   2   5   8  10   3   4   9   1];
n = length(v);
for j = 1:n-1
  min_val = v(j);
  min_pos = j;
  for k = j+1:n        % för resterande element
    if v(k) < min_val  % hittat mindre?
      min_val = v(k);  % spara värde
      min_pos = k;     % och index
    end
  end
  if min_pos ~= j
    temp_v     = v(j);       % kasta om
    v(j)       = v(min_pos);
    v(min_pos) = temp_v;
  end
end
```

```
>> v
v =
   1   2   3   4   5   6   7   8   9  10
```

Givet en kvadratisk matris, beräkna diagonal-, rad- och kolonnsummor.

```
n         = 5;
M         = magic(n);      % en magisk kvadrat
row_sums  = zeros(n, 1);   % allokera minne
col_sums  = zeros(n, 1);
diag_sum1 = 0;
diag_sum2 = 0;             % andra huvud-diagonalen

for j = 1:n
  for k = 1:n
    row_sums(k) = row_sums(k) + M(k, j);
    col_sums(k) = col_sums(k) + M(j, k);
  end
  diag_sum1 = diag_sum1 + M(j, j);
  diag_sum2 = diag_sum2 + M(j, n - j + 1);
end

% Detta går också bra
%   row_sums(j) = row_sums(j) + M(j, k);
%   col_sums(j) = col_sums(j) + M(k, j);

>> row_sums'   % transponat för att spara plats
ans = 65    65    65    65    65
>> col_sums'
ans = 65    65    65    65    65
>> diag_sum1
diag_sum1 = 65
>> diag_sum2
diag_sum2 = 65
>> n^2 * (n^2 + 1) / (2 * n)
ans = 65            % det magiska värdet
```


Nu till det obligatoriska palindromexemplet.

> palindrome n [Gk palindromos running back again, fr. palin back, again + dramein to run; akin to Gk polos axis, pole - more at POLE, DROMEDARY](1629): a word, verse, or sentence (as "a man a plan a canal panama") or a number (such as 1881) that reads the same backward or forward

Sirap i Paris.
Ni talar bra latin.
Bor edra gråvita fat i vår garderob?
A man, a plan, a canal, Panama!
A dog! A panic in a pagoda!
Stressed was I ere I saw desserts.
Eva, can I stab bats in a cave?
Ma is as selfless as I am.
Al lets Della call Ed Stella.
Cigar? Toss it in a can, it is so tragic.
"Naomi, sex at noon taxes," I moan.
Saippuakivikauppias (Finska för "försäljare av kaustik soda").

Låt oss skriva ett program för den enkla varianten, när vi har avlägsnat alla skiljetecken och endast har små bokstäver. Här är några varianter, där jag dessutom introducerar några nya tekniker.

```
str   = 'cigartossitinacanitissotragic';
palin = true;                % logisk variabel
k     = 1;                   % index
n     = length(str);
ndiv2 = n / 2;
np1   = n + 1;

while palin & k <= ndiv2
  if str(k) ~= str(np1 - k)
    palin = false;
  end
  k = k + 1;
end
```



```
if palin
  disp('the string is a palindrome')
else
  disp('the string is not a palindrome')
end
```

Här är en mindre strukturerad variant (man skall inte hoppa för mycket i program):

```
str   = 'cigartossitinacanitissotragic';
palin = true;
for k = 1:length(str) / 2
  if str(k) ~= str(end + 1 - k)  % OBS: end
    palin = false;
    break;                       % hoppa ur loopen
  end
end
palin
```

Här en mer komplicerad variant som kan hantera en allmän sträng:

```
str   = 'Cigar? Toss it in a can, it is so tragic.'
left  = 1;           % pekar på vänster bokstav
right = length(str); % pekar på höger bokstav
palin = true;

while palin & left < right
  if ~isletter(str(left))       % inte en bokstav?
    left = left + 1;            % tag nästa
  elseif ~isletter(str(right))  % inte en bokstav?
    right = right - 1;          % tag nästa
  else                          % två bokstäver

    if lower(str(left)) == lower(str(right))
      left  = left + 1;         % tag nästa
      right = right - 1;        % tag nästa
    else
      palin = false;
    end  % if
```



```
  end  % if
end  % while
palin
```

9 Funktioner

Det finns en praktisk gräns för hur stora program man kan hantera ("monolithic program"). Man vill stycka upp ett program i mindre delar, i funktioner. Detta har flera fördelar:

- Kan dela upp ett program i mer hanterbara delar. Koden blir enklare att förstå, underhålla och bygga ut. Man behöver inte förstå hela koden i ett svep, utan kan koncentrera sig på de stora dragen.
- Kan återanvända kod (t.ex. `sin(x)`).
- Kan dölja variabler, behöver inte tänka på namnkonflikter (eng. information hiding).
- Kan dölja funktioner (en funktion kan vara lokal till en annan funktion).

Det vi har använt hittills är sk script-filer, som man också kan använda för att dela upp program. Alla variabler är dock globala (åtkomliga från alla andra script-filer) vilket gör det svårt att skriva stora program med script.

En MATLAB-funktion liknar en matematisk funktion, $y = f(x)$. I programmering brukar x kallas inparameter eller inargument. y kallas utparameter, utargument eller resultat. Man kan strunta i in/ut om det är uppenbart vad som avses.
I programmering finns funktioner som inte tar några inargument, t.ex. `rand()` som returnerar ett slumptal. Notera också att vi kan få olika resultat fastän vi har samma inparameter. Det finns också funktioner som inte tar några argument alls. Anropet, `figure` (eller `figure()`), skapar ett nytt plot-fönster. `figure` kan dock ta argument och returnera värden.
Liksom i matematik är det vanligt att funktioner tar mer än en inparameter. I MATLAB-programmering kan man ha flera in- och utparametrar. MATLABs `eig`-funktion kan t.ex. användas för att beräkna egenpar till det generaliserade egenvärdesproblemet:

$$Ax = \lambda Bx$$

Anropet är `[X, L] = eig(A, B)` (fyra matriser) eller `lambda = eig(A, B)` (två matriser och en vektor) om vi bara behöver egenvärdena.

Ibland anropar vi funktioner utan att tänka på det. När vi löser ett linjärt ekvationssystem skriver vi `x = A \ b` . Alternativt kan man skriva `x = mldivide(A, b)` (matrix-left-divide).

Symbolen `\` är knuten till funktionen `mldivide`. När `A` inte är kvadratisk löses problemet i minstakvadratmening.
Vi kan addera två tal genom att skriva `plus(a, b)` eller enklare `a + b`. Det är samma syntax när vi adderar vektorer och matriser. När en operators (t.ex. \) funktion bestäms av datatypen talar vi om operatoröverlagring. Detta är möjligt i t.ex. MATLAB, C++ samt Fortran90, men inte i Java.

9.1 En enkel funktion

Formen på en funktion som tar ett argument och returnerar ett argument är:

```
function ut_parameter = funktions_namn(in_parameter)
 räkna, räkna, räkna ...
ut_parameter = resultat;
```

- Man ger funktionen ett värde, man returnerar ett resultat, genom att tilldela `ut_parameter` ett värde.
- Man måste alltid returnera ett värde (om funktionen har en `ut_parameter`).
- Man sparar funktionen på filen `funktions_namn.m`
- Notera att en funktion inte kommer åt några variabler "utifrån" annat än `in_parameter` (med det vi kan nu).
- Variabler som man använder inuti funktionen är skyddade för åtkomst utifrån. Det enda (med det vi kan nu) sättet för funktionen att returnera ett resultat är via `ut_parameter` (eller via datorgrafik eller utskrifter).

Exempel. Skriv en funktion som givet en matris beräknar och returnerar skillnaden mellan matrisens största och dess minsta element.

```
function max_minus_min = max_diff(A)
max_A = A(1, 1);   min_A = A(1, 1);

for row = 1:size(A, 1)
  for col = 1:size(A, 2)
    max_A = max(max_A, A(row, col));
    min_A = min(min_A, A(row, col));
  end
end
max_minus_min = max_A - min_A;
```

```
>> M = magic(4)
M = 16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> skillnad = max_diff(M)  % kan använda andra namn
skillnad = 15

>> max_diff(1:4)           % resultat -> ans
ans = 3

>> max_diff(-4)
ans = 0

>> max_diff(2:6) + 2 * max_diff(-5:-2:-10)  % uttryck
ans = 12

>> max_diff()  % inget argument, max_diff ger samma
??? Input argument "A" is undefined.

>> max_diff(1:3, 4:5)      % två argument
??? Error using ==> max_diff
Too many input arguments.
```


Observera att namnen på in- och utaparameter är lokala till funktionen. Vi kan använda andra namn när vi anropar funktionen. Om vi inte tillhandahåller en variabel för resultatet, så lagras det i `ans`. Det är dock inte tillåtet att strunta i inparametern eller att ge för många. Vi kan använda funktionen i numeriska uttryck.

En funktion kan ha noll eller flera inparametrar och noll eller flera utparametrar, som i exemplen nedan. Dessa exempel visar enbart hur man använder ett varierande antal parametrar, de är inte tänkta att visa hur man skriver bra kod.

Här en funktion som inte har några parametrar:

```
function date_time1
datestr(now, 'yyyy-mmm-dd, HH:MM:SS')
```

```
>> date_time1()
ans = 2008-Jan-14, 15:08:58  % ans inuti date_time1
                             % ändrar inte ans utanför
>> date_time1                % funktionen
ans = 2008-Jan-14, 15:09:03
```

En funktion som har en utparameter.

```
function str = date_time2
str = datestr(now, 'yyyy-mmm-dd, HH:MM:SS');
```

```
>> tid = date_time2
tid = 2008-Jan-14, 15:09:40
>> tid
tid = 2008-Jan-14, 15:09:40
```

En funktion som har en inparameter.

```
function date_time3(form)
if form == 'date'
  datestr(now, 'yyyy-mmm-dd')
elseif form == 'time'
  datestr(now, 'HH:MM:SS')
end
```

```
>> date_time3('date')
ans = 2008-Jan-14

>> date_time3('time')
ans = 15:10:39
```

En funktion som har två utparametrar.

```
function [d, t] = date_time4
d = datestr(now, 'yyyy-mmm-dd');
t = datestr(now, 'HH:MM:SS');

>> [a, b] = date_time4
a = 2008-Jan-14
b = 15:11:37
```

En funktion som har två inparametrar och två utparametrar.

```
function [d, t] = date_time5(d_form, t_form)
if d_form == 'us'
  d = datestr(now, 'mmm-dd-yyyy');
else
  d = datestr(now, 'yyyy-mmm-dd');
end

if t_form == 'us'
  t = datestr(now, 'HH:MM:SS AM');
else
  t = datestr(now, 'HH:MM:SS');
end

>> [a, b] = date_time5('us', 'us')
a = Jan-14-2008
b = 3:18:34 PM
```

När en funktion tar mer än en inparameter är ordningen viktig; första värdet man skickar in kommer att vara värdet på första inparametern etc.

Namnen är oviktiga i detta sammanhang. Om vi har funktionen
`function r = func(a, b)`
och gör anropet `res = func(3, 19)` så kommer
`a = 3` och `b = 19` inuti funktionen. Det gäller även i följande fall:

```
a   = 19
b   = 3
res = func(b, a)
```

Namnen i anropet och inuti funktionen är inte kopplade på något sätt. Om vi ändrar på `a` eller `b` inuti funktionen så ändras *inte* variabler, med samma namn, utanför funktionen.
Det är dessutom så att parametrarna inte ändras, här ett exempel:

```
>> type func_ex     % lista en funktion

function r = func_ex(a, b)
a = a + 1;
b = 10 * b;
r = a + b;

>> par1 = 20;
>> par2 = 30;
>> res = func_ex(par1, par2)
res = 321

>> par1
par1 = 20  % har inte ändrats
>> par2
par2 = 30  % har inte ändrats
```

Detta beror på att MATLAB analyserar funktionen. Om man i funktionen ändrar på en inparameter, skapar MATLAB en kopia som funktionen arbetar på. Denna kopia avlägsnas när vi återvänder från funktionen.
Att skapa kopior tar tid och minne, så om vi *inte* ändrar på en inparameter i funktionen, så skapar MATLAB inte en kopia av denna inparameter.

Om vi vill ändra på en inparameter brukar man göra som i följande larviga exempel:

```
>> type add_one
function p = add_one(p)  % OBS: samma p
p = p + 1;

>> p = 3
p = 3

>> add_one(p)
ans = 4

>> p = add_one(p)
p = 4
```

MATLAB är lite speciellt i detta avseende. I andra språk har detta lösts på effektivare sätt, man slipper då en (i vissa fall) allokering av temporärt minne och onödiga kopieringar.

Här följer ytterligare några exempel på funktioner.
Exempel. Skriv en funktion som givet en vektor, v, och ett heltal, $m \geq 1$, som argument, beräknar och returnerar en vektor av glidande medelvärden (eng. moving averages) enligt (n är antalet element i v):

$$\frac{1}{m}\left[\sum_{k=1}^{m} v_k,\ \sum_{k=2}^{m+1} v_k, \ldots,\ \sum_{k=n-m+1}^{n} v_k\right]$$

Rutinen skall också returnera det vanliga medelvärdet av v.
Vi noterar att:

$$\sum_{k=j+1}^{j+m} v_k = v_{j+m} - v_j + \sum_{k=j}^{j+m-1} v_k$$


```
>> type mov_aver

function [vec_aver, aver] = mov_aver(v, m)
n = length(v);
if m < 1 | m > n
  error('m < 1 or m > length(v)')
end

vec_aver = zeros(n - m + 1, 1);  % allokera utrymme

s = 0;
for k = 1:m
  s = s + v(k);
end
vec_aver(1) = s;  % första summan

% beräkna resterande summan mha formeln
for j = 1:n - m
  vec_aver(j + 1) = v(j + m) - v(j) + vec_aver(j);
end

vec_aver = vec_aver / m;

% vanliga medelvärdet
aver = 0;
for k = 1:n
  aver = aver + v(k);
end
aver = aver / n;

% En enkel test
>> for m = 1:5
     [vec_aver,aver]=mov_aver(1:5, m); vec_aver', aver
   end

ans = 1     2     3     4     5
aver = 3
```

```
ans = 1.5000   2.5000   3.5000   4.5000
aver = 3

ans = 2     3     4
aver = 3

ans = 2.5000e+00   3.5000e+00
aver = 3

vec_aver = 3
aver = 3
```

■

Exempel. Skriv en funktion, `is_prime(n)`, som avgör om `n` är ett primtal.

```
function result = is_prime(n)
if n <= 0 | round(n) ~= n
  error('n must be a positive integer')
elseif n == 1           % special case
  result = false;
else
  result = true;
  for factor = 2:floor(sqrt(n))
    if rem(n, factor) == 0 % mod is another alternative
      result = false;
      return
    end
  end
end
```

Det finns en färdig rutin, `isprime`, i MATLAB som accepterar allmännare argument och som dessutom kontrollerar storleken på argumentet (tillåter bara $n \leq 2^{32}$). Skriv `type isprime` för att se koden. ■

Funktioner kan anropa andra funktioner. En funktion kan till och med anropa sig själv. Detta kallas rekursion och behandlas i slutet av kursen. Här ett exempel där en funktion anropar en annan:

Exempel. Skriv en funktion som approximerar $\int_a^b f(x)\,dx$ med trapetsmetoden (inte en speciellt bra metod).
Trapetsmetoden approximerar integralen med summan av areorna av $n-1$ parallelltrapetser. Med $n \geq 2$ och $h = (b-a)/(n-1)$, $x_k = a+(k-1)h, k = 1, \ldots, n$, så ges approximationen av formeln:

$$\int_a^b f(x)\,dx \approx h\left[\frac{f(x_1)}{2} + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

```
function I = trapeze(a, b, n)
if n < 2
  error('n must be >= 2.')
end

h  = (b - a) / (n - 1);
I  = 0.5 * (f(a) + f(b));
xk = a;

for k = 2:n-1
  xk = xk + h;
  I  = I + f(xk);
end

I = h * I;

function y = f(x)
y = sin(x) / x;
```

Svaret med 16 decimaler är 0.659329906435511833. Här några testkörningar

```
>> trapeze(1, 2, 2)
ans = 6.480598491103686e-01

>> trapeze(1, 2, 20)
```



```
ans = 6.592989203183923e-01

>> trapeze(1, 2, 200)
ans = 6.593296239739588e-01

>> trapeze(1, 2, 2000)
ans = 6.593299036362444e-01
```

■

Vår rutin vore mer användbar om funktionen inte alltid måste heta `f`. Det fixar vi i de två följande avsnitten.

9.2 Funktionshandtag

Om man har enkla funktioner (inga for-satser etc.) finns det några kortvarianter. Vi tar upp den viktigaste varianten, *anonyma funktioner* (eng. anonymous functions). Det generella utseendet är

```
funktionshandtag =
   @(kommaseparerad lista av argument) uttryck
```

Listan får vara tom eller bestå av endast ett argument. Här några exempel:

```
>> f = @(x) exp(-x^2);
>> f(1)
ans = 3.6788e-01

>> f(sqrt(-1))
ans = 2.7183e+00

>> f
f = @(x) exp(-x.^2)

>> class(f)
ans = function_handle
```



```
>> minmax = @(x) [min(x), max(x)];

>> minmax(1:4)
ans = 1     4

>> x = linspace(0, 1);
>> minmax(sin(x) .* cos(x))  % elementvis *
ans = 0   4.9999e-01

>> two_var = @(x, y) exp(-x^2 - y);
>> two_var(1, 2)
ans = 4.9787e-02

>> a = 10;
>> note = @(x) x + a;
>> note(1)
ans = 11

>> a = 10000;
>> note(1)
ans = 11      ! OBS, a = 10

>> time = @() datestr(now, 'HH:MM:SS');
>> time()
ans = 21:15:10
>> time()
ans = 21:15:11

>> time  % dock, dvs utan ()
time = @() datestr(now, 'HH:MM:SS')
```

Man kan även skapa ett funktionshandtag på följande vis:

```
funktionshandtag = @funktionsnamn
```

Ett funktionshandtag fungerar som en variabel (i flera fall) och kan lagras i en cellvektor (som vi kommer till så småningom) t.ex.

```
% Define test function

if test == 1
  test_f = @cos;  % Nytt
elseif test == 2
  test_f = @sin;
elseif test == 3
  test_f = @(x) exp(-x) * x;
end
```

Med `test = 1;` blir `test_f(pi)` lika med -1.
Snyggare är att använda cellvektorer:

```
% inga blanka
>> test_funcs = {@cos, @sin, @(x)exp(-x)*x}
test_funcs =
    @cos    @sin    @(x)exp(-x)*x

>> for k = 1:3
     test_funcs{k}(pi / 2)
   end

ans = 6.1232e-17
ans = 1
ans = 3.2654e-01
```


9.3 Funktioner som parametrar

Det är mycket vanligt i tillämpningar att skicka funktioner som parametrar. Fördelen är att användaren av t.ex. en integrationsrutin kan välja ett eget namn på integrandrutinen, namnet är inte bestämt av den som skrivit integrationsrutinen. Här följer vår förbättrade trapetsrutin, som nu även tar en funktion som parameter. Det enda vi behöver är första raden i rutinen, som nu lyder:

```
function I = trapeze(f, a, b, n)
```

Nu några tester:

```
>> trapeze(@f, 1, 2, 2000)
ans = 6.593299036362444e-01

>> trapeze(@my_integrand, 1, 2, 2000)
ans = 1.352572717657960e-01

>> type my_integrand

function y = my_integrand(x)
y = exp(-x.^2);
```

Det går bra med anonyma funktioner också:

```
>> handtag = @(x) exp(-x^2);

>> trapeze(handtag, 1, 2, 2000)
ans = 1.352572717657960e-01

>> trapeze(@(x)exp(-x^2), 1, 2, 2000)
ans = 1.352572717657960e-01
```


9.4 Globala variabler

En litet bekymmer med att använda standardiserad programvara är att funktionen vi skickar (integranden) har en given parameterlista, bestämd av den som skrivit integrationsrutinen. Det gör att man inte utan vidare kan bifoga data till sin rutin, data som kanske behövs för att beräkna funktionsvärdena.
Ett vanligt sätt att lösa denna typ av problem är att använda globala variabler. Denna teknik skall inte missbrukas, eftersom det lätt leder till oläslig kod.
Säg att vi har ett huvudprogram, en scriptfil:

```
global pressure temperature  % deklarera först

pressure    = 1.23e6;        % definiera sedan
temperature = 237.55;

I = trapeze(@a_function, -23.56, 18.54, 1000)
```

Här följer funktionen:

```
function y = a_function(x)
global pressure temperature

% här kan vi använda pressure och temperature
% för att beräkna funktionsvärdet
y = ...
```

En fördel med ovanstående lösning är att vi ändrar värdet på t.ex. `pressure` på *ett* ställe. Om man skriver in värdet, `1.23e6`, både i huvudprogram och funktion glömmer man kanske att ändra i funktionen när man senare ändrar i huvudprogrammet.

9.5 Flera funktioner i samma fil

När man skriver större program blir det lätt många funktioner och därmed filer att hålla reda på. I vissa fall kan man lagra flera funktioner i samma fil. Detta exempel visar på strukturen. Funktionerna är lagrade på filen `prime.m` (i detta fall).

```
function y = prime(x)  % primary function
                       % file name  prime.m
 ...
function [z, y] = another_function(x)  % a sub function
 ...
function out = a_third_function(x)  % a sub function
 ...
```

Endast den primära funktionen är synlig utifrån (kan anropas av funktioner som *inte* ligger i filen). Funktionerna i filen kan anropa varandra. Om det finns en funktion på en egen, separat, fil med samma namn som en subfunktion (`sub` säg), och en funktion i filen anropar `sub`, så väljs subfunktionen.

Det går att driva detta ett steg till. Man kan ha funktioner som är lokala till andra funktioner (eng. nested functions). Det finns också "private functions" (som ligger i en katalog `private`). Dessa funktioner kan anropas från en funktion en nivå ovanför den privata katalogen. Slutligen har vi överlagrade funktioner (MATLAB har visst stöd för objektorienterad programmering). Vi tar inte upp någon av dessa varianter i denna kurs.

9.6 Ett exempel med tärningskast

Vi vill simulera kast med en tärning och räkna frekvenserna av ettor, tvåor etc. Här följer en sekvens av funktioner. Antalet kast bestäms av en inparameter **num_throws**.
Här är den första lösningen, som är rätt opraktisk.

```
function die_freq1(num_throws)

a = 0; b = 0; c = 0; d = 0; e = 0; f = 0;
for k = 1:num_throws
  throw = rand;
  if throw < 1/6
    a = a + 1;
  elseif throw < 2/6
    b = b + 1;
  elseif throw < 3/6
    c = c + 1;
  elseif throw < 4/6
    d = d + 1;
  elseif throw < 5/6
    e = e + 1;
  else
    f = f + 1;
  end
end

a, b, c, d, e, f
```

```
>> die_freq1(10000)
a = 1656
b = 1674
c = 1660
d = 1638
e = 1701
f = 1671
```


Lösningen är opraktiskt att programmera och opraktiskt för användare (människor och program) av funktion. Man kan inte enkelt efterbehandla, t.ex. plotta, frekvenserna. Tänk om sinus-funktionen skrev ut resultatet, då skulle *inte* följande fungera

```
>> r = 1.23;
>> x = r * cos(0.1);
>> y = r * sin(0.1);
```

Följande lösning, **die_freq2** är bättre, men ändå lite opraktisk. Jag har kopierat **die_freq1** men tagit bort utskriftsraden.
Första raden i **die_freq2** lyder:

```
function [a, b, c, d, e, f] = die_freq2(num_throws)
```

och så här används funktionen:

```
>> [a, b, c, d, e, f] = die_freq2(10000)
a = 1614
b = 1706
c = 1667
d = 1623
e = 1682
f = 1708
```

Anropet är lite jobbigt: Notera att följande inte gör det man kanske tror:

```
>> frekvens_vektor = die_freq2(10000)
frekvens_vektor = 1636
```

Man får bara första frekvensen **a**. Analogt ger

```
>> [a, b] = die_freq2(10000)
```

de två första frekvenserna.

Det är enklare för alla parter om man förpackar frekvenserna i en vektor, **freqs**, så här:

```
function freqs = die_freq3(num_throws)

freqs  = zeros(6, 1);

for k = 1:num_throws
  throw = rand;
  if throw < 1/6
    freqs(1) = freqs(1) + 1;
  elseif throw < 2/6
    freqs(2) = freqs(2) + 1;
  elseif throw < 3/6
    freqs(3) = freqs(3) + 1;
  elseif throw < 4/6
    freqs(4) = freqs(4) + 1;
  elseif throw < 5/6
    freqs(5) = freqs(5) + 1;
  else
    freqs(6) = freqs(6) + 1;
  end
end
```

man kan nu enkelt hantera resultatet:

```
>> frekvens_vektor = die_freq3(10000)
frekvens_vektor =
        1675
        1655
        1628
        1758
        1645
        1639
>> (frekvens_vektor - 10000 / 6)'
ans = 8.3333 -11.6667 -38.6667  91.3333 -21.6667 -27.6667
```


Med vektorer kan man också skriva enklare kod:

```
function freqs = die_freq4(num_throws)
freqs  = zeros(6, 1);

for k = 1:num_throws
  throw = rand;
  for j = 0:5
    if j/6 <= throw & throw < (j+1)/6
      freqs(j+1) = freqs(j+1) + 1;
      break
    end
  end
end
```

Detta går att göra *ännu kortare* (man kan bli av med if-satsen och for-j-loopen), men det visar jag inte (det skall vara något kvar till labben också).

När man skall redovisa en sådan uppgift i en laboration, skriver man lämpligen ett huvudprogram, **die_main** säg, som kan vara en script-fil (behöver inte vara en funktion). Kanske något i stil med.

```
n_throws = 1000; % antalet tärningsskast

% Här testar vi ...
dice_freq1(n_throws)
% och här
[a, b, c, d, e, f] = dice_freq2(n_throws)
freqs_3 = dice_freq3(n_throws)
freqs_4 = dice_freq3(n_throws)

% Lite analys av...
freqs_4 - n_throws / 6

% Nu plottar vi ...
plot(...
```

En annan fråga var hur man presenterar resultatet. Vi kommer så småningom att se på hur man kan göra snygga tabeller. Här är en primitiv variant.

```
>> frekvens_vektor = die_freq3(10000);
% skapa en matris med två kolonner
>> [(1:6)', frekvens_vektor]
ans =
           1        1679
           2        1665
           3        1627
           4        1661
           5        1684
           6        1684

% skriver inte ut ans
>> disp([(1:6)', frekvens_vektor])
           1        1679
           2        1665
           3        1627
           4        1661
           5        1684
           6        1684
```

Så, i huvudprogrammet kan man ha något i stil med:

```
disp('Resultat av tärningsskast.')
disp('         Ögon     Frekvens')
disp([(1:6)', frekvens_vektor])
```

som ger

```
Resultat av tärningsskast.
         Ögon     Frekvens
           1        1682
           2        1733
           3        1599
           4        1640
           5        1704
           6        1642
```


10 Avlusning

En stor del av en programmerares tid brukar upptas att hitta och korrigera fel i program. Denna process kallas avlusning (eng. debugging). Felen beror oftast på programmeraren, men kan även orsakas av fel i kompilatorer, programsystem som MATLAB och kanske av hårdvaran. I större programmeringsprojekt skriver en programmerare 10-20 korrekta rader per dag. Antalet beror givetvis på kodkvalitet, tillämpning, språk etc.

MATLAB är rätt tacksamt när det gäller att hitta fel, C++-program är *mycket* svårare att felsöka (index- och pekarfel fångas normalt inte). Det finns speciella program, avlusare (eng. debuggers) som kan underlätta felsökandet (för erfarna programmerare).

Det vanligaste felet för nybörjaren är "språkfel" (felaktig syntax).

```
>> si = sin[0.1]    % fel typ av parenteser
?? si = sin[0.1]
            |
Error: Unbalanced or misused parentheses or brackets.
```

Dessa är normalt lätta att åtgärda. Mycket svårare är logiska fel och då speciellt sådana som inte orsakar något felmeddelande utan bara ett felaktigt resultat (om man nu får ett felaktigt resultat när man testar). Programmet kan ju vara felaktigt, i stora delar, men producera korrekta resultat i några enkla testfall. *Om* man får ett felmeddelande, är man (kanske) i en bättre situation.

```
>> x = a(n);    % indexfel, varför?
??? Index exceeds matrix dimensions.
```

Arbeta baklänges i programmet. Varför fick vi indexfel. Är vektorn för kort eller är `n` för stort? Kanske har vi fel variabelnamn.

Om programmet avslutar normalt, men med fel resultat: Kontrollera indata, GIGO, "Garbage In Garbage Out".
Skriv ut resultat under exekveringen (tag bort semikolon).

Ibland har vi facitvärden och kan jämföra, om inte, kan vi kanske göra en rimlighetskontroll. Har vi ett testproblem där vi känner resultatet?
Lägg in kontroller. Använd gärna MATLABs, **`warning`**, som ger en "traceback". Finns även **`error`**, som avbryter körningen. Kan även använda **`return`** för att avbryta körningen, och bara testa en del av koden.

```
>> type warn_ex

function warn_ex
% code ...
  another_func(2, 3)
% code ...

function another_func(x, y)
% code ...
if x^2 + y^2 > 1   % should not happen
  warning('x^2 + y^2 > 1, x^2 + y^2 = %e', x^2 + y^2)
end
% code ...

>> warn_ex
Warning: x^2 + y^2 > 1, x^2 + y^2 = 1.300000e+01
> In warn_ex>another_func at 9
  In warn_ex at 3
```

Man kan lägga in sådana kontroller när man skriver programmet, "defensive programming". Om en kontroll är tidskrävande slår man på den med hjälp av en debug-flagga.

```
  if debug_expensive
    % Time consuming computation

    if cannot_happen
      warning( ...
    end

  end
```


Murphys lag:

> If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.

Programmera för "alla" eventualiteter (åtminstone om någon annan än du själv skall använda koden).
Ibland undrar man om en viss funktion ens exekveras. När det gäller if-satser undrar man ibland om något/vilket alternativ som utförs. Man kan ha formulerat något villkor fel eller glömt något. Man kan göra en egen trace, t.ex.:

```
function func1(a)
disp('*** in func1')

if a < 0
  disp('*** func1: a < 0')
  ... code
else
  disp('*** func1: a >= 0')
  ... code
end
```

Man kan också lägga in utskrifter som talar om var man befinner sig i koden:

```
disp('x before solve')
x = A \ b
 ...
disp('x before correction')
x = x + corr
```

Det vanligaste felet (åtminstone i traditionella språk) är nog indexfelet (i vektorer och matriser). Index kan vara otillåtet (då får vi ett felmeddelande, i MATLAB i alla fall) eller tillåtet men felaktigt.

Exempel. Tag ut de sista 7 elementen ur vektorn `vec`.

```
last_elem = vec(end-7:end); % tar ut 8 element
```

`vec(m:n)` med $\texttt{m} \leq \texttt{n}$ tar ut `m - n + 1` element, inte `m - n` element. ■

Ett mycket vanligt fel i traditionella språk att utföra en loop en gång för mycket/litet. Det är inte lika vanligt i MATLAB pga av MATLABs kraftfulla matrishantering.
Felet har ett eget namn: "off-by-one error (oboe)". Ett typiskt fall är räknare i samband med while-loopar:

```
ind = 0;        % eller 1
while ind < n  % eller ind <= n
  ind = ind + 1;
  vec(ind) ...

% eller

  vec(ind) ...
  ind = ind + 1;
end
```

Testa första samt sista iterationen. Om en loop skall gå ett givet antal varv brukar det vara mindre risk för fel med for-loopar.

Ett vanligt fel i numerisk analys (t.ex. i trapetsformeln) är följande:

Exempel. Vi delar in intervallet, $[a, b]$, i en uppsättning lika långa delintervall, (x_j, x_{j+1}). Intervallernas ändpunkter är $a = x_1 < x_2 < x_3, \ldots < x_n = b$. Hur långt är varje delintervall? Vanliga svar: $(b-a)/(n-1), (b-a)/n, (b-a)/(n+1)$.
Lösning: tänk specialfall (som är tillräckligt allmänt, så att det täcker upp alla fall). Med $n = 3$ har vi $a = x_1 < x_2 < x_3 = b$. Tre x-värden ger två intervall (ett mindre), så rätt svar är $(b-a)/(n-1)$.

Eftersom man har olika numrering i olika sammanhang kan det bli ännu mer förvirrat: $a = x_0 < x_2 < x_3, \ldots < x_{n-1} = b$, $a = x_0 < x_2 < x_3, \ldots < x_{n+1} = b$ etc. ■

Avlusning av funktioner. Kontrollera in- och utparametrar vad avser ordningsföljd, antal och typ. Variabler som är lokala till en funktion är lite besvärliga att inspektera. `keyboard`-kommandot kan vara praktiskt.

```
  function ...

    keyboard
% återvänd med att skriva return (sex tecken)
```

Man stannar på `keyboard`-raden och får tillgång till variablerna i funktionen. Prompten byts till `K»`. Man kan utföra beräkningar, plotta etc. på vanligt sätt. För att fortsätta exekveringen skriver man `return` (man skriver verkligen strängen). Om man har en loop som anropar `keyboard` och vill avsluta i förtid, skriver man `dbquit`, exekveringen av m-filen avslutas då.

11 Testning av program

> A bug-free program is an abstract. theoretical concept.
> Dennie Van Tassel
>
> All sufficiently complex programs have bugs. Anonymous
>
> Program testing can be used to show the presence of bugs, but never to show their absence! Edsger Dijkstra
>
> An effective way to test code is to exercise it at its natural boundaries. Brian Kernighan

Man har givetvis testat sitt program under avlusningsfasen, men större system kräver en speciell testningsfas. Här följer några saker att tänka på. En bra och utförligare beskrivning finns hos Wikipedia, sök efter "Software testing".

Vi tänker oss ett relativt litet MATLAB-program som utför numeriska beräkningar (att testa ett program som Firefox är ett helt annat företag). Man brukar välja att testa slumpdata, men effektivare är att förfara som Brian Kernighan föreslår. Man måste t.ex. kunna sortera en vektor som har *ett* element, kunna lösa ett linjärt ekvationssystem, $\boldsymbol{Ax} = \boldsymbol{b}$, där $\boldsymbol{A}$ är en 1×1-matris. Kontrollera resultat av loopar som går noll eller ett varv. Blir alla variabler definierade fastän villkoret i en while-loop aldrig blir uppfyllt?

```
for k = m:n  % problem om m > n?
 ...
end

while villkor
  x = ...
end

här använder vi x
```


När man testar numeriska rutiner kan man i viss utsträckning använda slumpdata, men slump-problem behöver inte vara typiska ($\boldsymbol{Ax} = \boldsymbol{b}$ från PDE) och de brukar inte vara speciellt svåra. Man kan normalt tänka ut svårare testproblem.

Exempel. Vi vill testa norm-rutinen (för vektorer) i MATLAB och några MATLAB-cloner.
Octave, `http://www.gnu.org/software/octave`.
Scilab, `http://www.scilab.org`.

```
>> x = rand(2, 1)           % I Matlab
x =
   9.2217e-01
   5.1609e-01
>> norm(x) - sqrt(sum(x.^2))
ans = 2.2204e-16

octave:8> x = rand(2, 1)    % I Octave
x =
  0.94248
  0.78092

octave:9> norm(x) - sqrt(sum(x.^2))
ans = 0

--> x = rand(2, 1)          % I Scilab
 x  =
    0.2113249
    0.7560439

-->norm(x) - sqrt(sum(x.^2))
 ans  = 0.
```

Så inga problem. Problem kan det dock bli om man känner till programmens svaga punkter.

```
>> norm(1e200)
ans = 1.0000e+200
>> norm(1e-200)
ans = 1.0000e-200

octave:11> norm(1e200)
ans = Inf
octave:12> norm(1e-200)
ans = 0

-->norm(1e200)
 ans  = Inf
-->norm(1e-200)
 ans  = 0.
```

Så varken Octave eller Scilab klarar av att räkna ut normen av vektorer med stora eller små tal, fastän det inte är några egentliga problem. I exemplet har dessutom vektorn bara ett element. Både Octave och Scilab använder sannolikt en algoritm enligt

```
norm(x): sqrt(sum(x.^2))
```

Problemet är att `1e200^2` ger overflow och `1e-200^2` underflow, så MATLAB använder en listigare algoritm som skalar indata. ■

Exempel. Sinusberäkning för stora argument. Facit med Maple:

$$\sin 10^{20} \approx -0.64525128526578084420581171131 3$$

```
>> sin(1e20) + 0.64525128526578084420581171131 3
ans = 0

octave:25> sin(1e20) + 0.64525128526578084420581171131 3
ans =  -1.01670605993712e-01

-->sin(1e20) + 0.64525128526578084420581171131 3
 ans  = - 1.016706059937121E-01
```

Slumpargument ger sannolikt inga problem alls. ■

Exempel. Här ett sista test med determinanter:

```
>> det(diag([1e-200 1e-200 1e200 1e200]))
ans = 1.0000e-00

octave:41> det(diag([1e-200 1e-200 1e200 1e200]))
ans =  1.0000e+00

-->det(diag([1e-200 1e-200 1e200 1e200]))
 ans  = 0.000E+00     % OBS, noll
```

Om man kastar om ordning med de stora talen först ger Scilab `INF`. ■

Man bör försöka testa alla delar av ett program. I en enkel test utnyttjar man kanske bara en delmängd av de funktioner man skrivit och en del av alternativen i if-satser.

12 Dokumentation

Större program och sådana som används av andra än programmeraren måste dokumenteras (dock brukar även programmeraren glömma detaljer). Det skall sägas att min föreläsningsexempel inte är så väldokumenterade, eftersom typsnittet gör att jag för begränsa antalet tecken. Det blir också lite svårläst när har begränsat utrymme. Dessutom fungerar jag, förhoppningsvis, som en levande kommentar under föreläsningen.

Man dokumenterar av flera anledningar.

- När man utvecklar program, man glömmer snabbt.
- Program som har en livslängd på flera år kommer att modifieras av någon (dig eller andra).
- För att förstå vad program gör. För att kunna återanvända kod.

Typ av dokumentation:

- Kommentarer i koden, ett minimum.
- Manualblad (i en unix-miljö).
- I större programsystem, on-line-hjälp, html-sidor, manualer.

Vi koncentrerar oss på kommentarer i koden.

- Inledande kommentarer. Vad gör funktionen. Beskrivning av in- och ut-variabler. Begränsningar. Felmeddelanden. Programmerare, version, datum. Utnyttja funktionen hos MATLABs `help`-kommando.
- Kommentarer i koden.

`help my_func` listar de inledande kommentarerna i koden (fram till första ickekommentar).

```
function res = my_func(arg)
% This ....    listas
%              listas
n = length(arg);          % listas EJ
%                           listas EJ
```


Här ett hoptryckt exempel på kommentarer i början av en Fortran-funktion:

```
NAME
  DPOTRF - compute the Cholesky factorization of a real
           symmetric positive definite matrix A

SYNOPSIS
  SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
    CHARACTER      UPLO
    INTEGER        INFO, LDA, N
    DOUBLE         PRECISION A( LDA, * )

PURPOSE
  DPOTRF computes the Cholesky factorization of a real
  symmetric positive definite matrix A.
  The factorization has the form
     A = U**T * U,   if UPLO = 'U', or
     A = L  * L**T,  if UPLO = 'L',
  where U is an upper triangular matrix and L is
  lower triangular.

  This is the block version of the algorithm, calling
  Level 3 BLAS.


ARGUMENTS
  UPLO  (input) CHARACTER*1
        = 'U':  Upper triangle of A is stored;
        = 'L':  Lower triangle of A is stored.

  N     (input) INTEGER
        The order of the matrix A.  N >= 0.

  A     (input/output) DOUBLE PRECISION array, dimension
        (LDA,N). On entry, the symmetric matrix A. If
        UPLO = 'U', the leading N-by-N upper triangular
        part of A contains the upper triangular part
```

```
        of the matrix A, and the strictly lower triangula
        part of A is not referenced. If UPLO = 'L', the
        leading N-by-N lower triangular part of A
        contains the lower triangular part of the matrix
        A, and the strictly upper triangular part
        of A is not referenced.

        On exit, if INFO = 0, the factor U or L from the
        Cholesky factorization A = U**T*U or A = L*L**T.

  LDA   (input) INTEGER
        The leading dimension of the array A.
        LDA >= max(1,N).

  INFO  (output) INTEGER
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an
              illegal value
        > 0:  if INFO = i, the leading minor of order
              i is not positive definite, and the
              factorization could not be completed.
VERSION
  LAPACK version 3.0, 15 June 2000
```

I vissa fall lägger man namnet på programmeraren och en versionshistorik.
När man dokumenterar kod skall man inte bara översätta koden till naturligt språk.

```
% Increase k by 2
k = k + 2;
```

är en meningslös kommentar, en person som läser koden kan ju normalt programmera. En bättre kommentar kan se ut så här (där `...` är en lämplig fortsättning).

```
% Take care of the even case. We don't have to
% consider odd numbers, since ...
k = k + 2;
```


Förklara vad ett programsegment gör. Enklare algoritmer och datastrukturer beskrivs i koden, mer komplicerade kräver separat dokumentation.
Man kan få gratisdokumentation genom att använda vettiga variabelnamn, t.ex. `begin_word`, `end_word`, `BeginWord`, `EndWord`, `WordBegin`, `WordEnd`,

```
function words = SentenceToWords(sentence)
 ...
function words = sentence_to_words(sentence)
 ...
function words = sentence2words(sentence)
 ...
function words = FindWords(sentence)
 ...
function words = GetWords(sentence)
```

Alltför långa variabelnamn blir svårlästa. Konsonanter är viktigare än vokaler (i slutet av namn).

```
function words = Extrct_Wrds(sentence)
 ...
function words = strng2wrds(string)
 ...
```

Det är bra om man är konsekvent vid namngivning. Jag brukar använda stora bokstäver för matriser och små för vektorer och skalärer t.ex.

Var *väldigt* försiktig med att använda tecken som kan förväxlas.

```
0, o, O, Q    (noll, bokstäver)
1, l, i, I    (ett, bokstäver)
```

Tänk också på att programmet kanske läses med ett annat typsnitt, där det är mindre (ingen) skillnad mellan dessa bokstäver. Om man använder ett typsnitt utan serifer (sans-serif) brukar det vara liten skillnad mellan tecknen.
Här ett exempel i Helvetica: 0, o, O, Q, 1, l, i, I.

13 Mer om vektorer och matriser, vektorisering

Vi har redan sett en del vektorer och matriser, men MATLAB kan mycket mer. En del av följande operationer definieras inte i kursen i linjär algebra. Skilj på matematik och programspråk. I ett programspråk kan man implementera bekväm notation som inte har en direkt motsvarighet i matematik. Använd *inte* syntax från MATLAB i matematik. I MATLAB kan man t.ex. skriva `x = A \ b`, vänsterdivision med matris, men man får *inte* skriva

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

i en matematiskt text.

Vi börjar med några *elementvisa* operationer.

```
>> a = 1:3;   % lite data
>> b = 4:6;

>> a .* b
ans = 4    10    18

>> a ./ b
ans = 2.5000e-01   4.0000e-01   5.0000e-01

>> a .\ b
ans = 4.0000e+00   2.5000e+00   2.0000e+00
```

Man skall lära sig att använda punkter där de behövs. Skriv *inte* `c = a + 2 .* b`. Varför?

1. Den som läser koden får intrycket av programmeraren inte vet att multiplikation med skalär *definitionsmässigt* utförs elementvis. Det ser amatörmässigt ut.

2. Om det hade stått `c = a + v .* b;` hade jag tänkt "`v` måste vara en vektor, eftersom man använder elementvis multiplikation". Koden "ljuger" alltså.
3. Det kan introducera fel. Uttryck kan bli definierade som med korrekt syntax hade varit odefinierade. Se exempel nedan.

Om man avlägsnar punkterna från divisionerna får man fortfarande resultat. Den första kan tolkas i termer av minstakvadratproblem $\min_\xi ||b\,\xi - a||_2$.

```
>> a / b
ans =
   4.1558e-01

>> a \ b            % ingen uppenbar tolkning
ans =
            0            0            0
            0            0            0
   1.3333e+00   1.6667e+00   2.0000e+00
```

Nu till några intressantare operationer:

```
>> a .^ b
ans = 1    32   729

>> ones(size(a)) ./ a   % invertera
ans =
   1.0000e+00   5.0000e-01   3.3333e-01

>> size(a)
ans = 1     3

>> 1 ./ a               % kortare
ans =
   1.0000e+00   5.0000e-01   3.3333e-01
```