## Block 3: Programmering

Beräkningsvetenskap I

## Sanningar om programmering

- Ett *program* är ett antal kommandon och särskilda kontrollstrukturer lagrade i en eller flera filer
- Att utveckla och skriva program kallas att *programmera*
- Att programmera – att se till att datorn löser det problem man vill lösa. Det handlar om *problemlösning*
- Att skriva ett program kräver förarbete med t ex papper och penna innan man sätter sig vid datorn. Viktiga delar är att formulera *algoritm* och välja *datastrukturer* (hur data ska lagras)



## Ett vardagsexempel

Algoritmer använder vi även till vardags.
Ett exempel:

**Indata:**

**Ingredienser 4 portioner:**
1 stor hackad gul lök
2 msk olja
+ 1 msk smör
1 tråg (350 g) strimlat kycklingkött
1 tsk mald kanel
1 tsk mald ingefära
1/2 tsk grovmalen peppar
1/2 g saffran
2 tsk salt
2 stora tomater
8 strimlade aprikoser
ca 2 dl vatten
citronklyftor till servering



## Ett vardagsexempel (forts.)

Algoritm exempel, forts

**Algoritmen:**

**Så här gör du:**
Fräs löken mjuk i panna med hälften av matfettet. Lägg över på en tallrik. Lättbryn köttet i panna med resten av matfettet. Krydda med kanel, ingefära, peppar, saffran och salt. Krama kärnorna ur tomaterna, grovhacka och blanda i köttet tillsammans med aprikoserna.
Häll på vatten och sjud 4 minuter under lock. Smaka av. Servera kycklingen på couscous (koka enligt råd på förpackningen), som blandats med 500 gram kålrot i små tärningar, kokta i 5 minuter och servera avsilad

**Utdata**: Kycklingtagine och couscous



## Programmet som recept

- Detaljerad steg-för-steg-beskrivning av hur uppgiften ska lösas
- Utgår från en repertoar av grundläggande operationer
- Inget får underförstås.
  Det enda som får förutsättas är att den som ska följa receptet har kännedom om de grundläggande operationerna



## Grundläggande byggstenar i algoritmer / program

- Tilldelning (av värden till variabler)
- Grundläggande operationer (räkna, hantera text, läsa indata, skriva utdata, etc)
- Kontrollstrukturer:
  Alternativ
  Upprepning
- Uppdelning av algoritmen i mindre komponenter (funktioner, procedurer, objekt)

## Programmeringsprocessen

Programmeringsprocessen innehåller mycket mer än att skriva programtexten

- **Formulering av problemet**
- **Analys av problemet**
- **Design av programmet**
- **Implementering**
- **Testkörning och felsökning**
- **Dokumentering**



## Exempel 1: Formulering

*Skriv ett program som läser in två tal , beräknar medelvärdet av dessa och skriver ut resultatet*

I detta exempel tränar vi på tilldelning och räkneoperationer



## Exempel 1: Analys

Analys av problemet:

a. Vilka indata är nödvändiga?
   **Svar**: Två tal
b. Av vilken typ är indata? [Begreppet datatyp]
   **Svar**: Reella tal
c. Uppkommer det några specialfall?
   **Svar**: Inte i det här exemplet.
d. Vad är utdata? Av vilken datatyp?
   **Svar**: Ett reellt tal



## Exempel 1: Algoritm

- Grov algoritmformulering
  Bryt ner problemet i delproblem

  Med hjälp av naturligt språk
  1. Läs in två tal.
  2. Beräkna medelvärdet.
  3. Skriv ut medelvärdet.



## Exempel 1: Algoritm

- *Pseudokod*
  Naturligt språk + variabler, d v s bryt ned i mindre delar.
  Eftersträva samma detaljnivå som i det slutliga programmet.
  1. Skriv ledtext: **`Ge två heltal...`**
  2. Läs in de två talen, namnge variablerna, t ex **`tal1`** och **`tal2`**
  3. Beräkna medel och tilldela variabel **`medel`**
     **`medel = (tal1 + tal2)/2`**
     eller
     använd MATLAB-funktionen **`mean`**.
  4. Skriv ut resultatet (värdet av variabeln **`medel`**)



## Exempel 1: Implementering

- För att få fram själva programmet måste algoritmformuleringen översättas till programtext/kod, d v s
  algoritm → programspråk
- Kan vara vilket programspråk som helst, t ex C, C++, Matlab eller något annat
- Måste då känna till vilka element i språket som motsvarar de olika instruktionerna i algoritmen

## Att skriva program i Matlab

- I Matlab läggs koden alltid i m-filer, t ex i MittProgr.m
- Två varianter
  - kommandofiler (skript)
  - funktioner
- Kommandofiler har du redan använt, funktioner senare
- Vanligen använder man en kombination av båda



## Exempel 1: Implementering

```
% prog1
% Program som beräknar medelvärdet
% av två tal

disp('Ge två tal:');
tal1 = input('Tal 1: ');    % inläsning
tal2 = input('Tal 2: ');
medel = mean([tal1, tal2]); % beräkna medel
% utskrift
texten = ['Medelvärdet av talen är ', …
num2str(medel)];
disp(texten);
```



## Exempel 1

- *Exekvering,* d v s körning, av programmet ger följande

```
>> prog1
Ge två tal:
Tal 1: 3
Tal 2: 6
Medelvärdet av talen är 4.5
```

Här har *användaren*, d v s den som kör programmet, matat in talen 3 och 6.



## Exempel 1

- Några lärdomar
  - Matlab-program lagras i m-filer,
  - Programmet utförs sekventiellt, rad för rad, uppifrån och ner
  - Använd kommentarer (efter %-tecken)
    - Inledande kommentar som kort förklarar var programmet gör (skrivs ut vid **`help prog1`**)
    - Kommentarer inne i koden som förklarar vad som görs i olika avsnitt
  - Några inbyggda funktioner som används
    - input, disp, mean, num2str



## Exempel 2: Formulering

*Skriv ett program som läser in två tal, beräknar kvoten mellan talen och skriver ut denna.*

I detta exempel tränar vi på kontrollstrukturen alternativ



## Exempel 2: Analys

- Analys av problemet

a. Vilka indata är nödvändiga?
   **Svar:** Två tal
b. Av vilken typ är indata?
   **Svar:** Reella tal
c. Uppkommer det några specialfall eller komplikationer?
   **Svar:** Ja, om nämnaren är lika med noll. Programmet bör kolla om nämnaren är noll och i så fall ge felutskrift
d. Vad är utdata? Av vilken datatyp?
   **Svar:** Kvoten, reellt tal

## Exempel 2: Algoritm

- Algoritmformulering, pseudokod
  1. Skriv ledtext: **Ge två tal**,....
  2. Läs in värden till variablerna **taljare** och **namnare**.
  3. **Om namnaren är 0 så**
     skriv ut ett felmeddelande
     **annars**
     sätt **kvot = taljare/namnare**
     skriv ut resultatet, dvs värdet av **kvot**



## Exempel 2: Implementering

```
% prog2
% Program som beräknar kvoten av två tal.

disp('Ge två tal'); % Inläsning av tal
taljare = input('Ge täljare: ');
namnare = input('Ge nämnare: ');

% Beräkna kvot
if namnare == 0
   disp('Division med noll!');
 else
   kvot = taljare/namnare;
   disp(['Kvoten blir: ', num2str(kvot)]);
end
```



## Exempel 2

- Exekvering, körning ger

```
>> prog2
Ge två tal
Ge täljare: 1
Ge nämnare: 2
Kvoten blir: 0.5

>> prog2
Ge två tal
Ge täljare: 2
Ge nämnare: 0
Division med noll!
```



## Exempel 2

- Några lärdomar
  - If-satser är en grundläggande struktur som finns i programmering
  - Används för att hantera olika specialfall, *alternativ*
  - Vilken gren i if-satsen som kommer att användas styrs av logiska operatorer - **namnare==0** är antingen sant eller falskt
  - Observera skillnad mellan = och ==
    - = ger tilldelning, t ex **a=3** medför att **a** tilldelas värdet 3
    - == kontroll om likheten sann, antingen sann eller falsk
  - Avsluta if-sats med **end**



## Exempel 2

- ”Lek” med logisk operator

```
>> 3 == 3
  ans = 1
>> t = (3 == 3)
  t = 1
>> t = (3 == 2)
  t = 0
>> a = [1 2 2 0];
>> a == 0
  ans = 0  0  0  1
>> a == 2
  ans = 0  1  1  0
```



## Exempel 3: Formulering

*Skriv ett program som läser in ett visst antal tal och sedan beräknar summan samt medelvärdet av dessa.*

*Resultaten skall skrivas ut.*

*Låt användaren först tala om hur många tal som ska behandlas.*

I detta fall tränar vi på kontrollstrukturen *upprepning ett visst antal gånger*

## Exempel 3: Analys

- **Analys av problemet**
  a. Vilka indata är nödvändiga?
     **Svar:** Antalet tal (**`antal`**) och vektor där talen lagras (**`tal`**)
  b. Av vilken typ är indata?
     **Svar:** **`antal`** heltal, **`tal`** reella tal, lagrade i en datastruktur i form av en vektor
  c. Uppkommer det några specialfall eller komplikationer?
     **Svar:** Ja, om **`antal`** är **`0`**.
  d. Vad är utdata? Av vilken datatyp?
     **Svar:** Summan är ett reellt tal, medel är ett reellt tal. Inga vektorer.



## Exempel 3: Algoritm

1. Ledtext: **`'Ge antalet tal:´`**
2. Läs in **`antal`**.
3. Upprepa: **För `i=1,2,3,...,antal`**
   Läs in ett tal och lagra som element nummer **`i`** i vektor **`tal`**
4. **Om `antal` ≤ 0 så**
   Skriv: 'Inga tal har bearbetats.'
   **annars**
   Beräkna summa:
   **`summa = sum(tal)`** (Matlab-funktion)
   Beräkna medelvärde:
   **`medel = summa/antal`**
   Skriv ut resultatet, dvs **`summa`** och **`medel`**



## Exempel 3: Implementering

```
% Program som beräknar medelvärde och summa
% av tal. Antalet tal läses in initialt.
antal = input('Ge antalet tal: ');
tal = zeros(1,antal); % Initiera tal-vektor
for i = 1:antal
  tal(i) = input(['Ge tal nr ', …
                  num2str(i), ': ']);
end
% Beräkna summa och medel och skriv ut
if antal <= 0
  disp('Inga tal har bearbetats.');
else
  summa = sum(tal);
  medel = summa/antal;
  disp(['Summa = ', num2str(summa),...
        ' och medel = ', num2str(medel)]);
end
```



## Exempel 3

- Körning ger

```
>> prog3
Ge antalet tal: 5
Ge tal nr 1: 4
Ge tal nr 2: 6
Ge tal nr 3: 7
Ge tal nr 4: 1
Ge tal nr 5: 3
Summa = 21 och medel = 4.2

>> prog3
Ge antalet tal: 0
Inga tal har bearbetats.
```



## Exempel 3

- Några lärdomar
  - "Upprepning ett visst antal gånger" är en grundläggande kontrollstruktur i programmering
  - Kallas i många programspråk för *for-loop*
  - **`for i=1:antal`** (Matlab-syntax) betyder på svenska *för i-värdena 1, 2,.., antal* och innebär att variabeln **`i`** kommer att starta med värdet 1 och successivt räknas upp
    i=1
    i=2
    ... osv
    tills man nått upp till *antal*
  - Obs att man måste veta antalet varv på förhand (**`antal`** måste vara definierad)
  - Avsluta for-loop med **end**



## Exempel 4: Formulering

- Modifiera exempel 3 så man inte vet innan hur många tal som skall läsas in, utan stanna inläsningen då <enter> ges.

*Skriv ett program som läser in tal tills <enter> ges. Beräkna summan samt medel av talen och skriv ut dessa.*

I det här exemplet tränar vi på kontrollstrukturen *upprepa så länge ett visst villkor är uppfyllt*

## Exempel 4: Analys

- Inläsningen av tal ska stoppas då man trycker på <enter>-tangenten. Hur kan man testa detta?
- <enter>-tangenten vid inläsning ger som resultat att ”tomma mängden” läses in, en tom variabel.
  Om en variabel, t ex variabeln **`t`** är tom, kan i Matlab testas med **`isempty(t)`**



## Exempel 4: Analys

- **Analys av problemet**
  a. Vilka indata är nödvändiga?
     **Svar:** Ett antal tal.
  b. Av vilken typ är indata? Heltal, reella tal?
     **Svar:** Reella tal, lagras i vektor **`tal`**.
  c. Uppkommer det några specialfall eller komplikationer?
     **Svar:** Ja, om inga tal läses in utan <enter> ges direkt – antal blir då noll
  d. Vad är utdata? Av vilken datatyp?
     **Svar:** Summan och medel är reella tal



## Exempel 4: Algoritm

1. Ledtext: Ge tal, avsluta med **`return`**
2. Initiera räknaren: **`antal = 0`**
3. Initiera tal-vektorn till tom vektor
4. Läs in 1:a talet innan loopen, placera i **`t`**.
5. Upprepa: **Så länge som `t`** är icke-tomt
   Placera **`t`** sist i talvektorn **`tal`**
   Räkna upp räknaren: **`antal = antal + 1`**
   Läs ett nytt tal och lagra i **`t`**
6. **Om `antal`** är 0 **så**
   se uppgift 3



## Exempel 4: Implementering

```
% prog4
% Program som beräknar medelvärde och summa av tal
% Inläsningen avslutas med <enter>

disp('Ge tal, avsluta med <enter>');
antal = 0;  % räknare initieras
tal = [];   % tal initieras till tom vektor
t = input('Ge första talet: ');

while ~isempty(t) % Loopa så länge som t icke-tom
  tal = [tal t];   % Placera talet i tal-vektorn
  antal = antal + 1;     % utöka antal
  t = input('Ge nytt tal: '); % läs nytt tal
end
if antal == 0
  se exempel 3
end
```



## Exempel 4

- Körning ger

```
>> prog4
Ge tal, avsluta med <enter>
Ge första talet: 41
Ge nytt tal: 50
Ge nytt tal: -4
Ge nytt tal: 0
Ge nytt tal:
Summa = 87 och medel = 21.75
```



## Exempel 4

- Några lärdomar
  - ”Upprepning så länge ett visst villkor är uppfyllt” är en grundläggande kontrollstruktur i programmering
  - Kallas i allmänhet för *while-loop*
  - I Matlab: avsluta while-loop med **end**

## Stora problem och m-filer

- Hittills har vi haft *en* m-fil (kommandofil) för varje program när vi löst ett problem eller skrivit Matlab-kod direkt i kommandofönstret
- Större problem måste delas upp i *delproblem*, var och en med egen algoritm
- Varje delproblem löses för sig oberoende av det större problemet
- Man behöver på det sättet bara ha kontroll över ett delproblem i taget



## Huvudprogram och underprogram

- Delproblemen blir vanligen en egen Matlab-funktion, eller flera funktioner (delproblemen kan innehålla egna underprogram)
- Större program blir uppdelade i fler *delprogram* (underprogram), dvs flera m-filer per problem (*funktionsfiler* och *kommandofiler*)
- En av m-filerna (vanligen kommandofil) utgör själva *huvudprogrammet* – detta styr problemlösningen och *anropar* funktionerna.



## Kommandofiler och funktioner

- En *kommandofil* är ett sätt att ”lagra” kommandon som annars skulle skrivits interaktivt. Genom att köra filen så exekveras alla kommandon i den. Inga parametrar.
- *Funktioner* kan liknas vid en ”svart låda”. Man stoppar in *indata (inparametrar)* och får som resultat *utdata (utparametrar)*.


- Den svarta lådan har **egna variabler**, för mellanlagring av data, som vi utanför ej behöver bry oss om (eller kan se)



## Kommandofiler och funktioner

- Skriv i filen **`cossin.m`** följande funktion:

```
function [x, y] = cossin(x0, x1, n)
% [x, y]=cossin(x,y,n)
% Beräknar cos(x)+sin(x) på intervallet
% [x0 x1] i n punkter
x = linspace(x0,x1,n);
y = cos(x)+sin(x);
```

- Funktionen anropas t ex genom:

```
>> xstart=0; xslut=pi; n=100;
>> [x1,y1] = cossin(xstart,xslut,n);
```



## Kommandofiler och funktioner

```
>> [x1, y1] = cossin(xstart,xslut,n);
```

```
function[x, y] = cossin(x0, x1, n)
x = linspace(x0,x1,n);
y = cos(x)+sin(x);
```

| Utanför funktionen | | i funktionen | | utanför funkt |
|---|---|---|---|---|
| xstart | → | x0 | | |
| xslut | → | x1 | | |
| n | → | n | | |
| | | x | → | x1 |
| | | y | → | y1 |



## Kommandofiler och funktioner

Olika anrop till samma funktion

```
>> [t, u] = cossin(0,2*pi,50);
```

eller...

```
>> [a, b] = cossin(0,2*pi,50)
```

Medför att **a** och **b** skrivs ut (inget semikolon)

eller...

```
>> a1 = 0; a2 = 2*pi;
>> cossin(a1,a2,50);
```

Utdata lagras inte i någon variabel

## Kommandofiler och funktioner:

Observera också …

```
>> [t, u] = cossin(0,2*pi,50);
```

Variablerna x och y, som bildas i **cossin**, är bara tillgängliga ”internt” i funktionen.

Variablerna t och u, som bildas i huvudprogrammet eller kommandofönstret är bara tillgängliga där.

Man säger att variablerna är *lokala*

All kommunikation av data mellan de olika programkomponenterna sker genom *parameteröverföring*



## Programmeringsprocessen

- **Formulering av problemet**
  - Vilka resultat vill vi ha?
  - Vad ska göras?
  - Vilka indata behövs?
  - Vilka utdata (grafer etc)

Specifikation av problemet kräver ofta kontakt med de som ska arbeta med programmet, t ex sjukvårdspersonal för system inom sjukvården



## Programmeringsprocessen

- **Strukturering, analys av problemet**
  - Dela upp i mindre delproblem
    Färdiga kommandon för vanliga delproblem, övriga får man lösa själv
  - Vilken information, dvs *indata* behövs för delproblemen?
  - Vad har indata för *typ*?
    Är det matris, vektor, sträng, reellt tal, heltal… ?
  - Hur löser man problemet i det mest *generella fallet*? Lös inte ett speciellt fall utan alla.
  - Finns det *specialfall*? Hur löser man dessa?
  - Vilken information, *utdata,* skall meddelas?



## Programmeringsprocessen

- **Algoritm**
  Bestäm lösningsmetod och formulera *algoritm*, dvs en följd av instruktioner och kommandon som i detalj beskriver vad som skall göras i programmet för att lösa delproblemet.

  Definition av *algoritm:*
  *"En systematisk procedur som i ett ändligt antal steg utför en beräkning eller löser ett givet problem."*



## Programmeringsprocessen

- **Kodning / implementering**
  Algoritmen omformas till programspråk (i vårt fall Matlab)
  - I princip att översätta algoritmen till kod
  - Instruktionerna måste vara *språkligt korrekta*, annars protesterar datorn.
  - Däremot *logiska fel* hittar inte datorn - datorn utför alla programinstruktioner exakt som du definierar dem!

  ***Tips!***
  *undvik onödiga fel genom att tänka och skissa algoritmer först, innan du sätter dig och kodar vid datorn.*
  *Starta med miniprogram och utöka efter hand. Lägg in ”flashigheter” sist när allt fungerar.*



## Programmeringsprocessen

- **Felsökning och testning**
  Ovanligt att program fungerar korrekt från början. Måste testköra för olika indata, försöka hitta felen för att sedan rätta till dem och köra igen. Kallas *avlusning* (*debugging*).

- **Dokumentering**
  För att ett program skall vara användbart för andra, måste det dokumenteras och kommenteras.
  - Se till att koden innehåller "lagom med kommentarer” på vettiga ställen (dvs %-rader i MATLAB)
  - I professionell programmering brukar man ha separata dokument där strukturen beskrivs

## Kontrollstrukturer

- I programmering används några få grundläggande s k *kontrollstrukturer*
  - Alternativ: if
  - Upprepning: for och while
- Strukturerna avslutas alltid med **end** i Matlab
- Kod indenteras (indrag) så att man tydligt ser var olika satser börjar och slutar, t ex

```
if t~=0
    disp('...');
    kvot = a/t;
       ...
end
```



## Kontrollstrukturer

- *Alternativ*, **if** (och **switch**)
  För alternativ kan man använda **if** eller **switch**. Vanligast med **if**

```
if logiskt uttryck
    satsgrupp
end
```

- **Ex)**

```
x = input('Give a number x: ');
if x > 0
    disp('x is greater than zero!')
end
```



## Kontrollstrukturer

- Finns varianter med "else":

```
if logiskt uttryck 1
    satsgrupp 1
else
    satsgrupp 2
end
```

eller t ex

```
if logiskt uttryck 1
    satsgrupp 1
elseif logiskt uttryck 2
    satsgrupp 2
else
    satsgrupp 3
end
```



## Kontrollstrukturer

- Viktigt att man i if-satser täcker alla möjligheter och de olika grenarna inte logiskt överlappar varandra. Ofta bra med sista **else** som täcker ”resten”
- **Ex)**

```
x = input('Give a number x: ');
if x > 0
    disp('x is greater than zero!')
elseif x >= 0
    disp('x is greater than or equal to
    zero!')
end
```

...inte så bra! Varför?



## Kontrollstrukturer

- **Ex)** Vi antar att **result** har tilldelats ett värde som ska representera resultat av ett tärningskast

```
if (result==1 | result==3 | result==5)
    disp('odd number of eyes')
elseif (result==2 | result==4 | result==6)
    disp('even number of eyes')
else
    disp('What kind of dice is this?')
end
```

Tecknet **|** betyder ”eller” (logiskt eller)



## Kontrollstrukturer

- Detta kan även lösas med strukturen **switch**

```
switch result
  case {1,3,5}
    disp('odd number of eyes')
  case {2,4,6}
    disp('even number of eyes')
  otherwise
    disp('What kind of dice is
          this?')
end
```

## Kontrollstrukturer

- Allmänt **switch**

```
switch uttryck
   case värde 1
        satsgrupp 1
   case värde 2
        satsgrupp 2
             ...
   case värde n
        satsgrupp n
   otherwise
        satsgrupp
end
```

**switch** kan ses som en variant av **if**. Är ibland enklare att använda



## Kontrollstrukturer

- Upprepning, **for** och **while**
  - Upprepning med **for** innebär att en satsgrupp blir utförd ett förutbestämt antal gånger.

```
for variabel = uttryck
     satsgrupp
end
```

```
Ex)
 % Bygg upp en vektor med for-loop.
 x = [];     % Starta med tom vektor
 for k = 1:5
   x(k) = input('Ge element i x:');
 end
```



## Kontrollstrukturer

- Upprepning med **while** används när man på förhand inte vet hur många gånger en satsgrupp ska utföras

```
while logiskt uttryck
     satsgrupp
end
```

```
Ex)
 x = [];  % Tom vektor vid start
 a = 1;
 while a > 0  % stopp om <= noll
    x = [x a];
    a = input('Enter value: ');
 end
```



## Relationsoperatorer

- Har nu även sett exempel på s k *relationsoperatorer*

| | |
|---|---|
| **<** | Mindre än |
| **<=** | Mindre än eller lika med |
| **>** | Större än |
| **>=** | Större än eller lika med |
| **==** | Lika med |
| **~=** | Skilt från |

- Ger värdet **true** (*sant*) eller **false** (*falskt*)
- I MATLAB gäller att om något är sant så har det värdet 1 och om något är falskt så har det värdet 0.



## Logiska operatorer

- Följande *logiska operatorer* finns i MATLAB:

| | | | |
|---|---|---|---|
| **&** | och | **~** | negation (inte) |
| **\|** | eller | **xor** | exklusivt eller |

| | |
|---|---|
| **A & B** | sant om både A och B är sanna |
| **A \| B** | sant om antingen A, B, eller båda är sanna |
| **~A** | sant om A är falskt och falskt om A är sant |
| **xor(A,B)** | sant om A är sant eller om B är sant (ej sant om både A och B är sanna) |



## Relationsoperatorer

- "Lek" med relationsoperatorer

```
>> t=true
t =
     1
>> t=false
t =
     0
>> x = -2:2
x =
   -2  1  0  1  2
>> x == 0
ans =
    0  0  1  0  0
>> x ~= 0
ans =
    1  1  0  1  1
```

```
>> x > 0
ans =
    0  0  0  1  1
>> x >= 0
ans =
    0  0  1  1  1
>> (x==0) | (x==1)
ans =
    0  0  1  1  0
>> (x==0) & (x==1)
ans =
    0  0  0  0  0
```

## Kommandofiler och funktioner

- En funktion måste alltid börja med funktionshuvud, t ex

```
function [x, y] = cossin(x0, x1, n)
```

utparametrar inparametrar

Obs! inled alltid med **function**

- Alla variabler inne i funktionen är lokala, dvs existerar inte utanför funktionen
- Filnamn måste överensstämma med funktionsnamn, dvs här cossin.m



## Kommandofiler och funktioner

- Kommandofiler
  är ofta specifika för ett visst problem som skall lösas. Kan vara svåra att återvända
- Funktioner
  är lättare att återanvända i flera program för olika problem av liknande sort (jfr Matlabs inbyggda funktioner)
- Att skriva funktioner
  - tänk först på vad som ska in och vad som ska ut ur funktioner, dvs in- och utparametrar
  - Ha inga onödiga utskrifter, vanligen bara varningsutskrifter – användaren bestämmer om något ska skrivas ut eller ej



## Kommandofiler och funktioner

- Det finns många inbyggda (fördefinierade) funktioner i MATLAB

Exempel:

```
if abs(sum(x)) > 10
    plot(x,y)
end
```

Egentligen är **abs**, **sum**, **plot** funktioner i filerna **abs.m**, **sum.m**, **plot.m**



## Funktioner exempel

Funktion som beräknar medelvärdet av två tal.

```
function z = medelfunk(x, y)
% z = medelfunk(x,y)
% Beräknar medelvärdet av två tal, x, y
z = ( x + y )./2;
```

Anrop från kommandofilen **main.m**:

```
a=2; b=6; c = medelfunk(a,b);
disp(['medel = ', num2str(c)]);
```

Ger vid körning resultatet

```
>> main
 medel = 4
```

Här: två in och en utparameter



## Funktioner exempel

- Inledande kommentarer i funktioner visas då man använder **help**-kommandot (precis som i kommandofiler)

```
>> help medelfunk

  z = medelfunk(x,y)
  Beräknar medelvärdet av två tal, x, y
```

- Kommentaren bör beskriva hur man anropar funktionen och kort vad den gör – man ska kunna använda funktionen bara genom hjälp-texten



## Okänt antal parametrar

- En funktion kan ha egenskapen att den anropas med olika antal parametrar olika gånger.
  - **nargin** = antal inparametrar vid anrop
  - **nargout** = antal utparametrar vid anrop

**Ex)**

```
function [x, y] = cossin(x0, x1, n)
% [x, y]=cossin(x,y,n)
% Beräknar cos(x)+sin(x) på [x0 x1] i
% n punkter
if (nargin==2), n=100; end
x = linspace(x0,x1,n);
y = cos(x)+sin(x);
```

## Okänt antal parametrar

- Kan nu anropas med t ex

```
>> [x, y] = cossin(0, 2*pi);
```

  då sätts n till 100 i funktionen

- …eller t ex

```
>> [x, y] = cossin(0, 2*pi, 200);
```

- Ett sätt att ha s k defaultvärden i funktioner



## Olika sorters fel

Det finns tre typer av fel, s k buggar som kan inträffa i program

- Syntaxfel
  Ett grammatiskt fel. Kan ej översätta Matlab-koden till "datorkod". Felmeddelanden från Matlab inte alltid så lätta att tolka.
- Exekveringsfel
  Fel som uppkommer under körningen och medför att programmet "kraschar", dvs slutar exekvera, kallas exekveringsfel eller runtime-fel.
- Logiska fel
  Programmeraren har tänkt fel, men programmet kör. Svårupptäckt.



## Vanliga syntaxfel i MATLAB

- Skrivfel, t ex **`plott(x,y);`**
- Utelämnade tecken
  t ex multiplikationstecknet, *: **`3(x+5)`**
  eller **`if ~( (x > 0) & (y < pengar*17)`**
  eller **`if ( x = 1 )…`**
- Felstavat variabelnamn

```
skyldigMig = 50;
if SkyldigMig > 100
```

- Oidentifierade variabler
  t ex glömt initiera **`sum`** före loop

```
for i=1:5, sum=sum+i; end, sum
```



## Vanliga syntaxfel i MATLAB

- Glömd punkt vid elementvisa operationer, ger

```
??? Error using ==> ^ Matrix must
be square.
```

- Skrivit **`else if`**, istället för **`elseif`**
- Slarv med kolon, komma, semikolon



## Goda råd vid programmering

1. Ha enbart en sats per rad
2. Indentera (flytta in text) på vettigt sätt.
3. Använd blanka rader för att dela upp koden i segment, t ex före/efter block
4. Kommentera varje block
5. Använd vettiga variabelnamn och funktionsnamn
6. Dokumentera gärna variabler och förklara
7. Ha ett "dokumentationshuvud" i början av filen

Kanske fel kan undvikas om koden uppfyller detta ☺



## Goda råd…

- Effektiv programmering:
  - I största möjliga grad bör *vektoroperationer* användas
    Ex) Antag att $f = (f_1, f_2, \ldots, f_n)$ skapats och vi vill beräkna $f_1 + 2f_2 + \ldots + 2f_{n-1} + f_n$

```
% Metod 1
tmp = f(1);
for k = 2:n-1
     tmp = tmp + 2*f(k);
end
sum = tmp + f(n)
```

## Goda råd…

- Effektiv programmering:
  - Ex forts) Utnyttja att **sum** är en s.k. skalärprodukt mellan *(1,2,…,2,1) och* $(f_1, f_2,\ldots,f_n)^T$

```
% Metod 2
sum = [1 2*ones(1,n-2) 1]*f';
```

Metod 2 är mycket snabbare!

> Tecknet ' i **f'** betyder transponat, dvs $f^T$



Informationsteknologi

## Goda råd…

- Effektiv programmering:
  - Behandla vektorer och matriser som "vilka variabler som helst"
  - Skriv generellt. Undvik att skriva funktioner som bara fungerar för ett problem.
  - Undvik siffror inne i loopar och inne i program. Använd bokstäver och sätt konstanter och parametrar initialt i koden.



Informationsteknologi

## Goda råd…

- Effektiv programmering

```
[m,n] = size(A);
for i = 1:m
  for j = 1:n
    A(i,j) = A(i,j)-…
  end
end
```

är bättre än t ex

```
for i = 1:500
  for j = 1:400
    A(i,j) = A(i,j)-…
  end
end
```