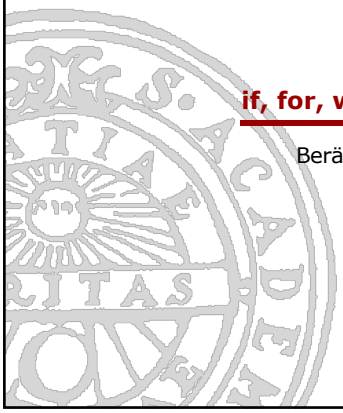


Programmering
if, for, while, funktioner

Beräkningsvetenskap I/KF

Från labben: if, for och while

- Det finns tre **grundläggande strukturer** i programmering, s k *kontrollstrukturer*
  - Alternativ, *if*

```
if logiskt uttryck
     :
end
```

```
if logiskt uttryck
     :
else
     :
end
```

```
if logiskt uttryck 1
     :
elseif logiskt uttryck 2
     :
else
     :
end
```

Informationsteknologi

Från labben: if, for och while

- Repetition, två varianter: *for* och *while*

```
for i = startindex:slutindex
     :
end
```

```
for i = startindex:steg:slutindex
     :
end
```

```
while logiskt uttryck
     :
end
```

Man kallar detta för loopar, for-loop respektive while-loop

Informationsteknologi

Från labben: if, for och while

- Exempel

```
x = input('Ge ett tal: ');
if x > 4
   disp('x är större än 4');
else
   disp('x är mindre eller lika med 4');
end
```

```
x = input('Ge ett tal: ');
if x > 4
   disp('x är större än 4');
end
```

- Gör indrag, s k *indentering*, så man enkelt ser var if-satsen börjar och slutar

Informationsteknologi

Från labben: if, for och while

- 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”

```
x = input('Ge ett tal: ');
if x > 0
   disp('x är positivt');
elseif x <= 0
   disp('x är negativt eller noll');
elseif x == 0
   disp('x är lika med noll');
end
```

…inte så bra! Varför?

Informationsteknologi

Från labben: if, for och while

- Exempel

```
n = input('Ge antal Fibbonaccital: ');
fib = [1;1];
for i = 3:n
   f = fib(i-1) + fib(i-2);
   fib = [fib;f];
End
fib              % eller disp(fib)
```

- Utskrift efter loopen (inte inne i loopen)
- Gör indrag, s k *indentering*, så man enkelt ser var loopen börjar och slutar
- För att förstå koden – torrexekvera och/eller testa kommandon i kommandofönstret

Informationsteknologi

Från labben: if, for och while

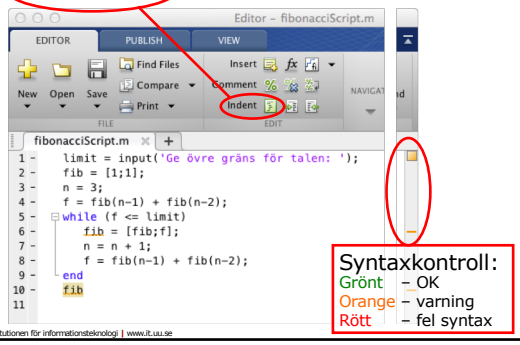
- Exempel

```
limit = input('Ge övre gräns för talen: ');
fib = [1;1];
f = fib(1) + fib(2);
n = 3;
while (f <= limit)
   n = n + 1;
   fib = [fib;f];
   f = fib(n-1) + fib(n-2);
End
fib
```

- Gör *indentering*, precis som tidigare
- För att förstå koden – torrexekvera och/eller testa kommandon i kommandofönstret

Tips

- Matlabs editor indenterar automatiskt
- Kan också markera koden och använd knappen **Smart Indent** under Edit i editor-fönstret



Ytterligare en kontrollstruktur

- **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)

Ytterligare en kontrollstruktur

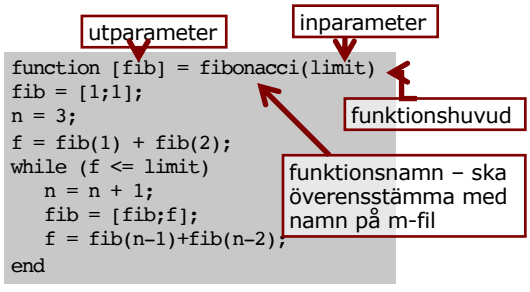
- Detta kan även lösas med strukturen **`switch`**

```
result = ceil(6*rand()); % slumpar tärning
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
```

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

Från labben: egna funktioner

- Hittills har alla program varit i form av *scriptfiler*
- Nytt i labben: egna *funktioner* istället för script. Exempel:



funktioner

Uppgift 1
Newtons 2:a lag: $F = ma$

- Skriv en Matlabfunktion med namn **`kraft`** som beräknar F givet m (kg) och a=9.81 m/s^2.
- Testa att funktionen fungerar genom att anropa den med olika värden på m, t ex din egen vikt.
  Testa även för flera vikter samtidigt, t ex din + din kompis vikt

funktioner

Uppgift 2
Massan m kan inte vara negativ

- Lägg in en kontroll av detta i din funktion **`kraft`**, så att funktionen avbryts om m<0 och gör en utskrift "**`massan < 0`**", men annars beräknar kraften.
- Testa att funktionen fungerar genom att anropa med positivt m och med negativt m . Fungerar det även för många värden på massan samtidigt (m som vektor)? Testa.

funktioner

Uppgift 3
Skriv ett *script* som beräknar kraften F tills F=1000, då m = 1, 2, 3, ...
Använd din funktion för att beräkna F (men *ändra INTE* i funktionen). Skriv ut den sist beräknade massan och kraften (dvs då $F\approx1000$).

Testa att det fungerar!
När det fungerar, hur skulle man göra om alla m och motsvarande F-värden ska lagras i vektorer?

funktioner

Uppgiften du just löst kan uttryckas

För att beräkna kraften F givet massa och accelerationen kan man använda Newtons 2:a lag $F = ma$.
Skriv ett program som beräknar kraften F för massan m = 1, 2, 3, etc upp till en viss kraft F^*, där F^* är ett specifikt värde på F. Programmet ska skriva ut beräknade värden och det ska inte tillåta negativ massa.

funktioner

Lärdom
Problemet har lösts stegvis

- Först funktions som definierade modellen
- Sedan förbättring/uppsnyggning av funktionen (test av m<0)
- Sedan ett script som använder funktionen

I varje steg testas respektive del av programmet.

Viktig strategi för problemlösning:
Försök inte lösa hela problemet på en gång, lös stegvis!
Typiskt ett antal funktioner som löser delproblem på ett generellt sätt, och en scriptfil som sätter ihop delarna, löser det specifika problemet.

Programmeringsprocessen

- Att programmera handlar om *problemlösning*
- Att skriva ett program kräver vanligen förarbete med t ex papper och penna innan man sätter sig vid datorn. Skissa, rita, gör exempel på papper
- Viktigt att veta vad man vill ha fram, vad som skrivas ut, vad som ska plottas (vad ska vara på axlarna etc). Ofta tydligare när man skissar först.
- Viktigt att förstå problemet och att formulera någon typ av lösningsskiss - *algoritm*

Script respektive funktioner

- Script
  är ofta specifika för ett visst problem som skall lösas
- Funktioner
  är typiskt mer generella och kan återanvändas 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
  - Vanligen heller ingen inläsning med **`input`**, värden ges istället via inparametrar

Script respektive funktioner

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

- Har egna **lokala variabler**, för mellanlagring av data, som vi utanför ej behöver bry oss om
- Funktioner *anropas* (man kan kan inte köra funktioner)

Script respektive 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 lagrade i filerna **abs.m**, **sum.m**, **plot.m**

Detsamma gäller **polyfit**, **polyval**, **input**, **linspace**, **norm**, **cond**, etc etc etc etc etc etc …

Funktioner igen

- Skriv i filen **cossin.m** funktionen **cossin**:

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


Funktioner igen

```
>> [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

Funktioner igen

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);
```

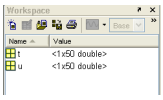
Utdata lagras inte i någon variabel

Funktioner igen

Observera också att…

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

Leder till att variablerna **t** och **u** bildas, se t ex workspace. Däremot bildas inte **x** och **y** - alla variabler som enbart skapas inne i funktionen är ”interna”.



Man säger att variablerna är *lokala*

Funktioner igen

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

```
>> help cossin

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

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

Olika 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

Exempel

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


Olika 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

Programmeringsprocessen – ett exempel

- För det s k gyllene snittet gäller att $a + b$ förhåller sig till a som a förhåller sig till b:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi.$$

Detta har den positiva roten

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887...$$

Enligt labben så går kvoten mellan två på varandra följande tal i Fibonaccis talföljd mot Gyllene snittet. Är det sant?
Skriv ett program som testar detta och visar att det stämmer.

Programmeringsprocessen – ett exempel

- **Problemlösning**
  Uppenbarligen är Fibonaccis talföljd ett delproblem här. ”Glöm” resten av problemet och lös detta delproblem för sig.
  Lösning av Fibonaccis talföljd är gjord på labben.

  En variant av funktionen på labben:

```
function fib = fibonacci(antalF)
% fib = fibonacci(n)
% Funktion som beräknar n st Fibonaccital och
% lagrar talen i vektorn fib
fib = zeros(antalF,1);
fib(1:2) = [1; 1];
for i=3:antalF
   fib(i) = fib(i-1)+fib(i-2);
end
```


Programmeringsprocessen – ett exempel

- **Problemlösning, forts**
  Delproblemet med Fibonaccis talföljd nu löst.
  Återgå till det stora problemet.
  Möjlig **algoritm** (skiss) för lösning:
  - Läs in (**input**) hur många Fibonaccital som ska skapas, lagra i variabeln **n**
  - Skapa **n** st Fibbonaccital med den färdiga funktionen, lagra i variabel (vektor) **Ftal**
  - Beräkna kvoten mellan sista och näst sista talet, **kvot = Ftal(n)/Ftal(n-1)**
  - Beräkna exakt värde med formeln och absolut fel
  - Skriv ut kvoten och absoluta felet

Programmeringsprocessen – ett exempel

- Program
  Överför algoritm till kod. Vi bestämmer oss t ex för en scriptfil med namn **`gsTestScript`**

```
% Läs in n och beräkna n st Fibonaccital
n = input('Ge antal Fibbonaccital: ');
Ftal = fibonacci(n);

% Beräkna kvot och fel
kvot = Ftal(n)/Ftal(n-1);
g_snitt = (1+sqrt(5))/2;
fel = abs(kvot-g_snitt);

% Gör utskrift
disp(['Kvoten blir: ',num2str(kvot)]);
disp(['Skillnad mot exakt värde: ',num2str(fel)]);
```


Programmeringsprocessen – ett exempel

- Testning av programmet

```
>> gsTestScript
Ge antal Fibbonaccital: 5
Kvoten blir: 1.6667
Skillnad mot exakt värde: 0.048633

>> gsTestScript
Ge antal Fibbonaccital: 10
Kvoten blir: 1.6176
Skillnad mot exakt värde: 0.00038693

>> gsTestScript
Ge antal Fibbonaccital: 15
Kvoten blir: 1.618
Skillnad mot exakt värde: 3.1465e-06
```


Programmeringsprocessen – ett exempel

- Resultaten verkar rimliga – problemet löst
- I det här läget är det vanligt att man vill snygga till, förbättra och kanske bygga ut koden
  - Kanske intressant att se i en plot hur talen konvergerar mot gyllene snittet?
  - Kanske skriva om **`gsTestScript`** som en funktion istället?
  - etc, etc…

Programmeringsprocessen

- **Formulera algoritm (lösningsskiss)**
  Förstå problemet, dela upp i delproblem som är enklare att lösa. Ofta klarar man inte av att lösa hela problemet men däremot delproblemen. Gör sedan en struktur för lösningen. Skriv i någon typ av blandning av vanligt språk och programspråk (pseudokod).

  Kodning/implementering
  Algoritmen omformas till programspråk (i vårt fall Matlab). Starta med ”miniprogram”, testa att det fungerar och utöka sedan efter hand. Dela upp koden i avdelningar med kommentarer emellan.
  Lägg in ”flashigheter” sist när allt fungerar.

Programmeringsprocessen

- **Felsökning och testning**
  Ovanligt att program fungerar korrekt från början. Måste testköra för olika indata och olika specialfall. Rätta till fel och testkör 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

Stora problem och underprogram

- Vi har kombinerat scriptfiler och funktioner (anrop av funktionen i scriptfilen) – ett exempel på ett *huvudprogram* och ett *underprogram* (funktionen)
- Större problem måste delas upp i *delproblem*, var och en med egen algoritm och eget under-program
- Varje delproblem löses för sig oberoende av det större problemet
- Man behöver på det sättet bara ha kontroll över ett delproblem i taget

Olika sorters fel

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

- Syntaxfel
  Ett grammatiskt fel, koden följer ej syntaxregler. I Matlab ser man detta i högerlisten i editorn Felmeddelanden 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. Programmet kan köras, med ger felaktiga resultat (vissa fall bara ibland). 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
```


Vanliga syntaxfel i MATLAB

- Glömd punkt vid elementvisa operationer, ger

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

- Skrivit `else if`, istället för `elseif`
- Någon inbyggd funktion har blivit omdefinierad, t ex genom

```
input = 5;
```

  Om sedan input-kommandot ska användas så uppfattar Matlab det som en variabel.
  Lösning: skriv `clear all` (eller `clear input`) i kommandofönstret
- Slarv med kolon, komma, semikolon

Goda råd vid programmering

1. Ha enbart en sats per rad
2. Indentera (indrag vid if, loopar etc)
3. Använd blanka rader för att dela upp koden i segment
4. Kommentera varje block av kod
5. Använd vettiga variabelnamn och funktionsnamn. God idé att använda namn som stämmer överens med problemformuleringen
6. Kommentera gärna variabler och förklara
7. Ha ett "dokumentationshuvud", en hjälptext, i början av filen

Kanske fel kan undvikas om koden uppfyller detta

Goda råd…

- Effektiv programmering:
  - Försök använda vektorer och kommandon för vektorer så mycket som möjligt, t ex

```
b = zeros(n,1);
```

bättre än

```
for i=1:n
   b(i) = 0;
end
```

  - Skriv generellt. Undvik att skriva funktioner som bara fungerar för ett problem
  - Undvik siffror inne i loopar och i program. Använd variabler och sätt konstanter och parametrar initialt i koden istället

Goda råd…

- Effektiv programmering
  Exempel. Vi vill beräkna någon typ av matris-operation för en 500×400-matris

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

klarar av alla matriser

klarar bara av detta specifika fall

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