Funktionen

gespeichert in Datei *Funktionsname.m*

Funktionen

gespeichert in Datei *Funktionsname.m*

`function` [*Rückgabevariable*, . . .]=*Funktionsname*(*Parameter*, . . .)
  % Kurzbeschreibung für Stichwortsuche
  % . . . Beschreibung . . .

  . . . *Befehle*
end

lokale Funktionen

Funktionen

gespeichert in Datei *Funktionsname.m*

`function` [*Rückgabevariable*, . . .]=*Funktionsname*(*Parameter*, . . .)
  % Kurzbeschreibung für Stichwortsuche
  % . . . Beschreibung . . .

  . . . *Befehle*
end

lokale Funktionen

Verlassen der Funktion durch `return`

Beispiel

Fibonacci-Zahlen z_n mit Rekursion

$$z_1 = z_2 = 1, \quad z_n = z_{n-1} + z_{n-2},\ n > 2$$

Beispiel

Fibonacci-Zahlen z_n mit Rekursion

$$z_1 = z_2 = 1, \quad z_n = z_{n-1} + z_{n-2},\ n > 2$$

```
function z = fibonacci(n)
% n-te Fibonacci-Zahl z

z_last = 0; z = 1;
for k=2:n
   z_save = z_last;
   z_last = z;
   z = z + z_save;
end
```

Beispiel

Fibonacci-Zahlen z_n mit Rekursion

$$z_1 = z_2 = 1, \quad z_n = z_{n-1} + z_{n-2},\ n > 2$$

```
function z = fibonacci(n)
% n-te Fibonacci-Zahl z

z_last = 0; z = 1;
for k=2:n
   z_save = z_last;
   z_last = z;
   z = z + z_save;
end
```

```
  >> z = fibonacci(10)
        z = 55
```

Speicherung aller berechneten Fibonacci-Zahlen

$\rightsquigarrow$ elementares Programm

```
z(k) = z(k-1) + z(k-2)
```

Speicherung aller berechneten Fibonacci-Zahlen
$\rightsquigarrow$ elementares Programm

```
z(k) = z(k-1) + z(k-2)
```

einzeilige Programm-Version

```
z = [1; 0] * ([0 1; 1 1]^n) * [0; 1];
```

Rekursive Programmversion:

Rekursive Programmversion:

```
function z = fibonacci(n)
% FIBONACCI n-te Fibonacci-Zahl z

if n<3
   z = 1;
else
   z = fibonacci(n-1) + fibonacci(n-2);
end
```

Beispiel

Binomialkoeffizient

Beispiel

Binomialkoeffizient

```
function c = binomial(n,k)
% BINOMIAL binomial coefficient
% c = binomial(n,k)
% n,k: nonnegative integers with k <= n
% c: n choose k

if k < 0 | k > n | n < 0
   disp('invalid input'); return;
end

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

Aufruf der Funktion

```
function c = binomial(n,k)
```

Aufruf der Funktion

```
function c = binomial(n,k)

>> c = binomial(5,2)
>> c
   10
>> binomial(3,4)
>> invalid input
>> binomial(4,3)
>> ans =
   4
```

Darstellung der Kommentare im Funktionskopf mittels `lookfor` und `help`:

```
>> lookfor binomial
 BINOMIAL binomial coefficient

>> help binomial
 BINOMIAL binomial coefficient
 c = binomial(n,k)
 n,k: nonnegative integers with k <= n
 c: n choose k
```

Programmierung mit Hilfe einer Unterfunktion:

```
function c = binomial(n,k)
c = product(n)/(product(n-k)*product(k));
end
function p = product(m)
p = 1;
for k = 2:m, p = p*k; end
end
```

Programmierung mit Hilfe einer Unterfunktion:

```
function c = binomial(n,k)
c = product(n)/(product(n-k)*product(k));
end
function p = product(m)
p = 1;
for k = 2:m, p = p*k; end
end
```

Programmierung mit Rekursion:

```
function c = binomial(n,k)
if n == 0 | k == 0 | k == n
   c = 1;
else
   c = binomial(n-1,k-1) + binomial(n-1,k);
end
```

Beispiel

Newton-Verfahren:

```
function x = newton(f, df, x)
% Newtonverfahren zur Bestimmung einer Nullstelle
% einer Funktion f mit Ableitung df nahe bei x
max_iter = 100; tol = 1.0e-10;
for k=1:max_iter
   fx = f(x); dfx = df(x);
   if abs(fx) < tol
      display('Nullstelle bestimmt'); return;
   elseif abs(dfx) < tol
      display('waagrechte Tangente'); return;
   end;
   x = x - fx/dfx;
end
display('keine Konvergenz');
```

Übergabe der Funktion und ihrer Ableitung als function-handles

```
>> f = @(x) x^3-x; df = @(x) 3*x^2-1;
>> x = newton(f, df, 4)
x =
   1.0000
```

Übergabe der Funktion und ihrer Ableitung als function-handles

```
>> f = @(x) x^3-x; df = @(x) 3*x^2-1;
>> x = newton(f, df, 4)
x =
  1.0000
```

kurze Programmvariante bei gesicherter Konvergenz

```
...

while abs(f(x)) > eps
   x = x - f(x)/df(x);
end

...
```