Technische Universität München
M2 - Numerische Mathematik
Dr. Laura Scarabosio

2.2. Übung. Einführung in die Programmierung (MA 8003)

Hinweis: Ab jetzt werden Schleifen benötigt.

Aufgabe 2.2.1: Verändern Sie die Funktion `isprime2` aus der Vorlesung so, dass sie auch für Vektoren und Matrizen als Eingabeparameter funktioniert.

Fügen Sie am Anfang der Funktion Abfragen hinzu, die überprüfen, ob die Einträge ganzzahlig und positiv sind. Falls eine Bedingung verletzt ist, brechen Sie mit einer geeigneten Fehlermeldung ab (`error('...')`).
Tipp: Um die Ganzzahligkeit zu überprüfen, können Sie z. B. die Funktion `floor` oder `round` geeignet verwenden.

Lösung 2.2.1:

```
function l = isprime2(x)
  if (any(x(:) < 0))
    error('Eingabewerte m\"ussen positiv sein');
  elseif (any((x(:) - floor(x(:))) ~= 0))
    error('Eingabewerte m\"ussen ganzzahlig sein');
  end
  l = logical(x);
  for i = 1:length(x(:))
    d = [2, 3:2:sqrt(x(i))];
    rest = rem(x(i), d);
    l(i) = all((rest ~= 0 & x(i) > 1) | x(i) == 2);
  end
```

Aufgabe 2.2.2: Das Collatz-Problem, auch als $(3n+1)$-Vermutung bezeichnet, ist ein ungelöstes mathematisches Problem, das 1937 von Lothar Collatz entdeckt wurde.

Bei dem Problem geht es um Zahlenfolgen, die nach einem einfachen Bildungsgesetz iterativ konstruiert werden:

a) Beginne mit irgendeiner natürlichen Zahl n.

b) Ist n gerade, so nimm als nächstes $n/2$.

c) Ist n ungerade, so nimm als nächstes $3n + 1$.

Implementieren Sie eine Funktion `collatz`, die diese Zahlenfolge konstruiert und auf dem Bildschirm ausgibt. Brechen Sie ab, wenn die Zahl 1 ist. Der Rückgabewert soll die Anzahl der Durchläufe sein, bis man bei 1 landet. **Tipp:** Verwenden Sie `mod` oder `rem`.

Können Sie die folgende Vermutung von Collatz an Beispielen bestätigen?

> Jede so konstruierte Zahlenfolge endet im Zykel 4,2,1 egal, mit welcher natürlichen Zahl man beginnt.

Diese Vermutung ist übrigens bis heute noch unbewiesen. Paul Erdös sagte zu der Beweisbarkeit:

> *„Mathematics is not yet ready for such problems.“*

Lösung 2.2.2:

```
function k = collatz( n )
k = 0;
while (n ~= 1)
    k = k + 1;
    if (mod(n,2) == 0)
        n = n./2;
    else
        n = 3*n + 1;
    end
    disp(n);
end
```

Aufgabe 2.2.3: Erstellen Sie eine anonyme Funktion `m`, die für die Eingabe $x \in \mathbb{R}^+$ das arithmetische Mittel von x und $2/x$ berechnet.

Speichern Sie in `y` eine beliebige positive Zahl. Konvergiert die Iteration `y = m(y)`? Implementieren Sie eine `while`-Schleife, die bei Konvergenz abbricht ($|y - m(y)| \leq 10^{-16}$).

Haben Sie eine Idee, was das Ergebnis ist?

Lösung 2.2.3: `m = @(x) (x+2./x)./2;`
`while abs(y-m(y))>1e-16, y=m(y); end`
Das Verfahren konvergiert gegen $\sqrt{2}$ (Heron Verfahren).

Aufgabe 2.2.4: Es soll eine Funktion `peval(p,x)` geschrieben werden, die ein Polynom an den Stellen `x` auswertet, wobei `p` der Koeffizientenvektor des Polynoms ist. Verwenden Sie eine Schleife, um die Matrix der Monome zu erzeugen.

Versuchen Sie die Funktion so zu schreiben, dass sie sowohl für Zeilen- als auch Spaltenvektoren `x` und `p` funktioniert.

Lösung 2.2.4:

```
function y = peval(p, x)
% PEVAL Wertet Polynom an Stellen aus x aus
  n = length(p);
  X = 0;
  for i = 1:n
    X(:, i) = x(:).^(i-1);
  end
  y = X * p(:);
```

Aufgabe 2.2.5: `x` sei ein Zeilenvektor (z.B. `x = 1:10`). Eliminieren Sie jeweils die `for`-Schleife:

a)
```
n=0;
for i=1:length(x), n = n + x(i)^2; end
```

b)
```
y = zeros(1,length(x));
y(1) = x(1) + 2*x(end);
for i=2:length(x);
    y(i) = x(i) + 2*x(i-1);
end
```

* c)
```
max_err = 0; n=100;
f0 = sin(pi/4); f1 = sin(1);
for x = linspace(0,1,n),
    p=x*f1+(1-x)*f0;
    err=abs(sin(x)-p);
    max_err=max(max_err,err);
end
```

d)
```
n = round(rand(1, 100)*100);
s = 0;
for i=n
   if (rem(i,2) == 0)
       s = s + i;
   else
       s = s + 2*i;
   end
end
```

Überzeugen Sie sich, dass Ihre Ergebnisse stimmen, indem Sie die Rückgabe Ihrer Lösung mit dem angegeben Programmcode vergleichen.

Lösung 2.2.5:

a) `n = x*x'` Alternativ `n = sum(x.^2)`

b) `y = [2*x(end)+x(1), 2*x(1:end-1)+x(2:end)]` Alternativ `y(2:end) = x(2:end) + 2*x(1:end-1)`

c)
```
x = linspace(0,1,n); p = x*sin(1) + (1-x)*sin(pi/4);
max_err = max(abs(sin(x)-p));
```

d) `l = rem(n, 2) == 0; s = sum(n(l)) + 2*sum(n(~l))`

Aufgabe 2.2.6 (*): Implementieren Sie eine eigene Variante der MATLAB-Funktion `primes`. Diese Funktion berechnet die Primzahlen von 2 bis n.

```
>> myprimes(11)
ans =
    2    3    5    7   11
```

Dabei soll weder die Matlab-Funtion `primes` noch die Funktion `isprime` verwendet werden.

Verwenden Sie das *Sieb des Eratosthenes*:
Legen Sie einen Vektor mit allen Zahlen von 2 bis n an. Streichen bzw. markieren (z.B. durch -1) Sie nun alle Vielfachen der ersten Zahl des Vektors. Wiederholen Sie das Vorgehen mit der nächsten nicht markierten Zahl im Vektor und so fort. Brechen Sie ab, wenn Sie die letzte Zahl erreicht haben. Kann man auch früher abbrechen?

Lösung 2.2.6:

```
function [ p ] = myprimes( n )
p=1:n; p(1) = -1;
for pp=[2, 3:2:sqrt(n)]
  if (p(pp) == -1)
    continue;
  end
```

```
  p(2*pp:pp:end) = -1;
end
p = p(p ~= -1);
```

Alternativ:

```
function [ p ] = myprimes( n )
p=2:n;
for k=1:sqrt(n)
  r = p(k);
  if (r ~= 0)
    p(k+r:r:end) = 0;
  end
end
p = p(p > 0);
```

Aufgabe 2.2.7: Erzeugen Sie einen Vektor `x` mit den Quadratzahlen von `1` bis `10000` auf folgende Weisen:

- Setzen Sie `x = 0`. Verwenden Sie eine `for`-Schleife um die Einträge einzeln an `x` anzuhängen.
- Setzen Sie `x = zeros(10000, 1);`. Verwenden Sie Ihre Schleife aus dem vorherigen Teil.
- Direkt, ohne Schleife.

Vergleichen Sie die Laufzeiten der verschiedenen Ansätze mit `tic` und `toc`.

Lösung 2.2.7:

```
>> tic; x = 0; for i=1:10000, x(i) = i.^2; end; toc;
Elapsed time is 0.059646 seconds.

>> tic; x = zeros(10000, 1); for i=1:10000, x(i) = i.^2; end; toc;
Elapsed time is 0.000695 seconds.

>> tic; x = (1:10000).^2; toc;
Elapsed time is 0.000086 seconds.
```

An dieser Aufgabe sieht man, wie viel Zeit man in Matlab sparen kann, indem man das Anhängen von Elementen an einen Vektor (Faktor 85) bzw. Schleifen (Faktor 8) vermeidet.

Aufgabe 2.2.8 (*): Als Beispiel in der Vorlesung haben wir das Jacobi-Verfahren gesehen. Ein alternatives iteratives Verfahren zum Lösen eines linearen Gleichungssystems $Ax = b$, mit $A \in \mathbb{R}^{n \times n}$, ist das Gauss-Seidel-Verfahren.

Sei $x^{(0)}$ ein beliebiger Startvektor. Die $(k+1)$-te Iteration des Gauss-Seidel-Verfahrens lautet:

$$x_i^{(k+1)} := \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, \ldots, n,$$

wobei $\sum_{j=1}^{0}$ und $\sum_{j=n+1}^{n}$ leere Summen darstellen und a_{ij} der (i,j)-te Eintrag der Matrix A ist. Eine hinreichende Bedingung für Konvergenz ist, dass A symmetrisch positiv definiert ist.

Implementieren Sie eine Matlab-Funktion

```
x = gauss_seidel(A, b, x0, tol, maxiter)
```

die das Gleichungssystem $Ax = b$ mit dem Gauss-Seidel-Verfahren löst. Die Argumente `x0` (Startvektor), `tol` (Toleranz) und `maxiter` (höchste Anzahl von Iterationen) sollen beliebige Argumente sein (wie im Jacobi-Verfahren in der Vorlesung gesehen). Die Abbruchbedingung soll das Erreichen der Toleranz (als Norm des Residuums gemessen) oder der maximalen Iterationsanzahl sein. **Tipp**: Benutzen Sie eine Schleife um die Elemente von `x` in jeder Iteration. Eine Vektorisierung ist in diesem Fall nicht einfach möglich. **Tipp**: Nehmen Sie an, dass die Matrix A keine Nullwerte auf der Diagonalen hat (d.h. Division durch diagonale Elemente von A möglich ist).

Lösung 2.2.8:

```
function x = gauss_seidel(A, b, x0, tol, maxiter)
%Gauss Seidel Verfahren

d = size(A);
if (nargin < 2)
  error('Mindestens A und b muessen uebergeben werden');
elseif (d(1) ~= d(2))
  error('A muss quadratische Matrix sein');
elseif (~isvector(b))
  error('b muss ein Vektor sein');
elseif (length(b) ~= d(1))
  error('Vektor b muss Laenge length(A) haben');
elseif (~all(diag(A)))
  error('Diese Funktion wirkt nur wenn A keine Nullwerte auf der Diagonalen hat');
end
if (nargin < 5), maxiter = 1000; end
if (nargin < 4), tol = 1e-6; end
if (nargin < 3)
  x = zeros(length(b),1);
elseif (length(x0) ~= length(A))
  error('Vektor x0 muss Laenge length(A) haben');
else
  x = x0(:);
end
r = b - A*x;
k = 1;
while (norm(r) >= tol && k<maxiter)
    x(1) = (b(1) - A(1,2:end)*x(2:end))/A(1,1);
    for i=2:length(x)-1
        x(i) = (b(i)-A(i,1:i-1)*x(1:i-1) - A(i,i+1:length(x))*x(i+1:end))/A(i,i);
    end
    x(end) = (b(end)-A(end,1:end-1)*x(1:end-1))/A(end,end);
  r = b - A*x;
  k = k + 1;
end
```

Eine Vektorisierung ist möglich, wenn die Matrix als $A = LU$ geschrieben wird, wobei L eine untere Dreiecksmatrix und U eine obere Dreiecksmatrix sind. Mit einer solchen Zerlegung kann man das Gauss-Seidel-Verfahren als $Lx^{(k+1)} = b - Ux^{(k)}$ schreiben.