OTH Regensburg

Studiengang
IN, IT B. Sc.
Prof. Dr. Martin Weiß
Sommersemester 2019

Computerarithmetik und Rechenverfahren

3. Praktikum: Gleitpunktarithmetik, Stabilität, MATLAB

3.1. Kondition von skalaren Funktionen

Berechnen Sie mit MATLAB die relativen Fehler

$$\frac{f(\tilde{x}) - f(x)}{f(x)}, \qquad \tilde{x} = x + \Delta$$

für $x = 1$ und $\Delta = \frac{1}{100}$, $\Delta = \frac{1}{1000}$, und die vom Arbeitsblatt bekannten Funktionen, mit $c = 7$, $n = 4$:

$$f_1(x) = x^2,\ f_2(x) = c \cdot x^3,\ f_3(x) = x^2 + 5x^3,\ f_4(x) = c \cdot x^n,\ f_5(x) = \sin(x).$$

Das könnten Sie ganz elementar für jede Funktion in ein Skript tippen, oder eine Funktion schreiben, die eine Funktion als Argument erhält. In C wäre das ein Funktionszeiger, in MATLAB heißt das Konzept *function handle*:

```
function deltaF = relativeError(f, xs, x)
fx = f(x);
deltaF = abs((f(xs) - fx) / fx);
end
```

a) Schreiben Sie 5 Funktionen `f1`, …, `f5` in eigenen Files und berechnen Sie die relativen Fehler mit Aufrufen, bei denen das function handle mit einem `@` gekennzeichnet wird:

   ```
   relativeError(@f1, xs, x)
   ```

b) Vergleichen die Ergebnisse jeweils mit dem Wert $\kappa_{\text{rel}}(x) \cdot \frac{\tilde{x}-x}{x}$

c) Viele kleine Dateien für die Funktionen sind lästig. Kurze Funktionen kann man als sog. *anonyme Funktionen* inline definieren:

   ```
   f1Anonym = @(x) x.^2;
   ```

   Diese können Sie fast genauso wie Funktionen verwenden, die in eigenen Dateien definiert sind, etwa `f1(4)` aufrufen. Als Argument für Funktionen mit function handles unterscheidet sich aber die Syntax, man braucht *kein* `@`. Seltsam; die Gründe dafür sind mir schleierhaft.

   ```
   relativeError(f1Anonym, xs, x)
   ```

d) (***) Wenn Sie ein Funktion für die Kondition $\kappa_f(x)$

   ```
   function kappaF = conditionRelative(f, x)
   ```

implementieren wollen, brauchen Sie auch die Ableitung f'. Dazu können Sie ein zweites Argument `fPrime` ergänzen, oder sich tiefer in die Symbolic Toolbox einarbeiten, die ableiten kann.

e) (***) Lesen Sie die Dokumentation zu `function` für weitere Feinheiten.

3.2. Stabilität: Punktoperationen sind (meist) besser als Strichoperationen

Gegeben sei die Polynomfunktionen 7. Grades:

$$\begin{aligned} p(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\ q(x) &= (x-1)^7 \end{aligned}$$

Beachten Sie dabei die Identität $p(x) = q(x)$. (Sie erinnern sich an den allgemeinen binomischen Satz und die Binomialkoeffizienten vom früheren Blatt: $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot a^k \cdot b^{n-k}$.)

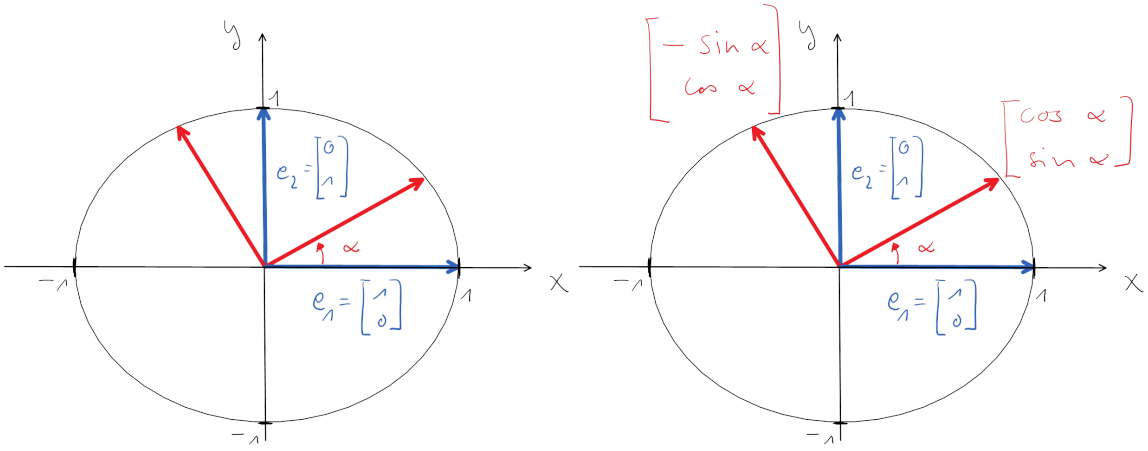
Zeichnen Sie mit Matlab eine graphische Darstellung von $p(x)$ für $x \in [0.988, 1.012]$ mit einer Schrittweite von $h = 0.0001$. Interpretieren Sie das Aussehen der Kurve. Vergleichen Sie dazu mit der ensprechenden Zeichnung, die bei der Darstellung von $q(x) = (x-1)^7$ entsteht.

3.3. 2d-Plots: Liniengraphiken, Rotation

Analysieren Sie den Code im Skript `mymovie.m`. Verstehen Sie vor allem, warum die Matrix mit den sin und cos-Einträgen eine Drehung bewirkt, wie die Koordinaten an die Matrix multipliziert werden, und was die Argumente von `line` bedeuten.

Schöner wohnen: Ergänzen Sie weitere Punkte am Häuschen: Türe, Kamin, Garage, ...

Erinnerung: In der Abbildungsmatrix einer linearen Abbildung stehen spaltenweise die Bilder der Basisvektoren:



3.4. Vektorisierung, Zeitmessung, Graphik

a) Schreiben Sie eine MATLAB-Funktion zur Multiplikation von Matrizen $A \in \mathrm{Mat}_{l,m}(\mathbb{R})$, $B \in \mathrm{Mat}_{m,n}(\mathbb{R})$ "von Hand", d.h. ohne Verwendung des MATLAB-Operators $*$ für Matrizen, sondern mit geschachtelten Schleifen gemäß der Definition von $C := A \cdot B$:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj} \qquad \text{für } i = 1,\ldots,l, j = 1,\ldots n$$

Für die skalare Multiplikation $A_{ik} \cdot B_{kj}$ dürfen Sie den Multiplikationsoperator `*` natürlich einsetzen. Verwenden Sie folgenden Funktionskopf:

```
function C = MatrixMultiplikation(A, B)
```

Verwenden Sie folgende MATLAB Funktionen:

i) `size()` zum Ermittlen der Größe einer Matrix

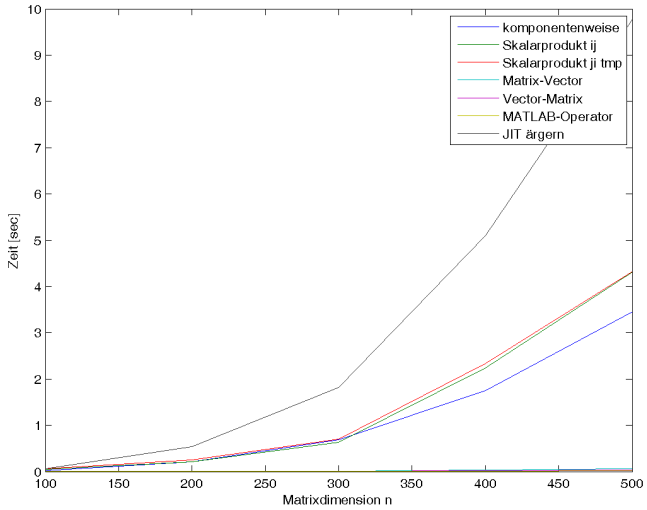
ii) `zeros()` zum Anlegen einer Matrix einer bestimmten Größe

b) Testen Sie Ihre Implementierung mit verschiedenen Matrizen und vergleichen Sie das Ergebnis mit dem, das MATLAB liefert.

c) Verwenden Sie den `:`-Operator zur Multiplikation von Zeilen und Spalten und sparen Sie so Schleifen ein.

d) Vergleichen Sie durch geeignete Messungen mit Matrizen unterschiedlicher Größen die Laufzeit Ihrer Funktionen mit der der eingebauten MATLAB-Funktion. Verwenden Sie dazu `tic` und `toc`.

e) Stellen Sie Ihre Ergebnisse graphisch dar: Laufzeit als Funktion der Dimension, etwa in folgender Form:



3.5. call by value, call by reference

Legen Sie eine Funktion `callby` an:

```
function AOut = callby( AIn )
AIn(1,2) = 9;
AOut = AIn;
AIn(1,1) = 5;
AOut(1,1) = 7;
end
```

[numbers=none] Was erwarten Sie als Werte von `A` und `B` nach den Anweisungen:

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
>> B = callby(A)
```

3.6. Datentyp `logical`: Logischer Zugriff auf Daten

Logische Operatoren wie etwa Vergleiche liefern den Datentyp `logical` mit 0 für false und 1 für true:

```
A = [ 1  2 -3
      4 -5  6
     -7 -8  9]
flags = A > 0
class(A)
class(flags)
A(flags)
flags2=2*flags
class(flags2)
```

Mit `logical` können Daten aus einer Matrix oder einem Vektor extrahiert werden; das Ergebnis wird als Vektor geliefert und ist dann keine Matrix mehr.

MATLAB macht aus vielen kuriosen Operationen einen Sinn. Auf `logical` können arithmetische Operationen durchgeführt werden (was manchmal durchaus sinnvoll und praktisch ist, etwa `sum(sum(A>0))`; das Ergebnis ist dann ein `double`-Datentyp.

3.7. Variable Anzahl von input- und output-Argumenten

polar2cart

cart2polar

3.8. *** Variable Anzahl von input- und output-Argumenten: `nargin`, `nargout`

Eine variable Anzahl von input-Argumenten, und damit eine Art variabler Argumente, sowie eine variable Anzahl von input- und output-Argumenten kann man mithilfe der Systemvariablen

`nargin` und `nargout` realisieren. Auch unterschiedliche Typen von Argumenten sind möglich, also eine Art Polymorphismus, vgl. die MATLAB-Funktionen `xlim` oder `meshgrid`.

Im Beispiel kann der Anwender die Funktionen wahlweise mit einem Argument als 2d-Vektor oder mit zwei skalaren Argumenten arbeiten.

```
r = 10; phi = pi/3;
[x, y] = polar2cart(r, phi)
v = polar2cart(r, phi)

[r,phi] = cart2polar(x, y)
[r,phi] = cart2polar(v)

function [r, phi] = cart2polar(x, y)
if nargin == 1
    % Argument als Vektor x = [x,y]
    % in Komponenten aufspalten
    y = x(2);
    x = x(1);
end
r = sqrt(x*x + y*y);    % oder: r = norm([x, y])
phi = atan2(y, x);
end

function [x, y] = polar2cart(r, phi)
if nargout == 2
    x = r*cos(phi);
    y = r*sin(phi);
else % nargout == 1
    x = [r*cos(phi); r*sin(phi)];
end
```

Für kompliziertere Varianten wie Strings als Schlüsselwörtern für Optionen, etwa in

```
plot(x,y,'LineWidth',3, 'color', 'g')
```

siehe `https://de.mathworks.com/help/matlab/matlab_prog/parse-function-inputs.html`.