WiMa-Praktikum 1

Universität Ulm, Sommersemester 2019

Woche 6

Lernziele

In diesem Praktikum sollen Sie üben und lernen:

- Erzeugung von Function Handles
- Umgang mit Function Handles

Am Anfang geben wir Ihnen eine kurze Einführung in Function Handles in MATLAB.

Beantworten Sie danach bitte erst einige Fragen, bevor Sie sich an den Rechner setzen!

Inhalt

Einführung

Bereits in der zweiten Woche des WiMa-Praktikums haben Sie sich ausgiebig mit dem Begriff der *Funktion* auseinandergesetzt. Hierbei wurde eine Funktion, welche dann in einem Hauptprogramm ausgeführt werden konnte, in einem gesonderten *m-File* gespeichert. Dabei enthielt die Funktion selber lediglich elementare Datentypen als Argumente (Vektoren, Matrizen, Skalare).

Allerdings ist es auch möglich, Funktionen zu programmieren, deren Argumente selber wieder Funktionen sind. Für einen solchen Fall werden *Function Handles* verwendet. Dies kann zum Beispiel der Fall sein, wenn eine numerische Approximation einer Ableitung programmiert werden soll. Sind nämlich mehrere Funktionen, z. B. $f_1(x) = \sin(x)$ und $f_2(x) = x^2$, gegeben, und wir wollen die Ableitungen dieser Funktionen in einem festen Punkt $x = 1$ mit Hilfe der Approximation

$$f'(x) \approx D^+ f(x) := \frac{f(x+h) - f(x)}{h} \tag{1}$$

für ein festes $h > 0$ bestimmen, so könnte man für die Funktionen f_1 und f_2 den Wert $D^+ f(x)$ mit Hilfe der folgenden Zeilen berechnen:

```
>> x = 1;
>> h = 0.001;
>> Df1 = (sin(x+h) - sin(x)) / h
Df1 =
    0.5399
>> Df2 = ((x+h)^2 - x^2) / h
Df2 =
    2.0010
```

Offensichtlich kann dies auf Dauer sehr unübersichtlich werden, da wir die Formel (1) der Approximation für jede einzelne Funktion neu eintippen müssen. Viel eleganter wäre es doch, wenn man diese Formel nur einmal festlegen müsste und dabei die eigentliche Funktion, deren Ableitung approximiert werden soll, gewissermaßen variabel lassen könnte. Wir werden an späterer Stelle noch einmal auf dieses konkrete Beispiel eingehen und sehen, wie dies mit Hilfe von Function Handles möglich sein wird.

Man kann also bereits jetzt festhalten, dass Function Handles u. a. dazu dienen, Informationen über eine Funktion an eine weitere Funktion zu übergeben. Desweiteren können Function Handles aber auch dazu verwendet werden, die Anzahl der benötigten Files zu reduzieren. Sie können bei wiederholtem Zugriff somit die Performance steigern.

Allgemein gilt die Empfehlung, Function Handles immer dann einzusetzen, wenn dies möglich ist, und sie gegenüber der Übergabe des Funktionsnamen als String zu bevorzugen. Man erkennt also relativ schnell, dass Function Handles für einen effizienten und eleganten Programmierstil unter MATLAB zunehmend wichtiger werden.

Erzeugen eines Function Handle - Der @-Operator

Um ein Function Handle in MATLAB zu erzeugen, verwendet man den sogenannten `@`-Operator. Hierbei wird dem eigentlichen Funktionsnamen einfach ein „`@`“ vorangestellt. Dabei ist es vollkommen irrelevant, ob die Funktion selber geschrieben wurde oder durch MATLAB bereits vorhanden ist. Man beachte, dass bei der Erzeugung eines Function Handle nicht der komplette Pfad der Funktion angegeben werden muss, solange sich die Funktion im jeweiligen Verzeichnis befindet. Das Handle lässt sich dann wie die Originalfunktion benutzen und auswerten (in älteren MATLAB-Versionen war dazu der Befehl `feval` notwendig):

```
>> f = @sin;          % Erzeuge Function Handle
>> f(pi)              % Auswertung Sinus-Fkt. bei x = pi
ans =
   1.2246e-16
```

Will man die gleiche Operation mit $f(x) = x^2$ wiederholen, so sind folgende Eingaben notwendig:

```
>> f = @(x) x^2;      % Erzeuge Function Handle
>> f(pi)              % Auswertung x^2 bei x = pi
ans =
    9.8696
```

Man beachte, dass im Gegensatz zur Sinusfunktion die Funktion $f(x) = x^2$ keine Funktion ist, die bereits von MATLAB vorgegeben ist. Daher muss man MATLAB mitteilen, dass es sich um eine Funktion handelt, die von einem Parameter abhängt, und dass x dieser Parameter ist. Dieses geschieht durch das Symbol `@(x)`. Durch `@(x) x^2` wird eine sogenannte *anonyme Funktion* mit einem Parameter `x` erzeugt, deren Function Handle dann der Variablen `f` zugewiesen wird.

1. Anwendungsbeispiel

Wir wollen uns nun noch weitere Anwendungsbeispiele anschauen, in denen ein Function Handle erzeugt werden soll. Zunächst wollen wir einen Funktionsplotter erstellen, der ein Handle einer Funktion erhält und die Funktion graphisch auf dem Intervall $[0, 5]$ darstellt.

Man beachte, dass diese Funktion als Parameter ein Function Handle verlangt und auch überprüft, ob ein Function Handle übergeben wurde:

```
1  function funcPlotter(func_handle)
2    if ~isequal(class(func_handle), 'function_handle')
3      error('Argument muss ein Function Handle sein!');
4    end
5
6    x = linspace(0, 5);
7    fx = func_handle(x);
8
9    plot(x, fx, 'r-.');
10   xlabel('x');
11   ylabel('y');
12   title('Plot der Funktion');
13 end
```

Will man nun beispielsweise die Sinusfunktion plotten lassen, so muss das durch den `@`-Operator erzeugte Function Handle der Sinus-Funktion `sin` übergeben werden. Damit wird verhindert, dass MATLAB versucht, die Funktion `sin` schon bei der Parameter-Übergabe auszuwerten.

Der Funktionsaufruf

```
funcPlotter(@sin)
```

erzeugt folgenden Plot:

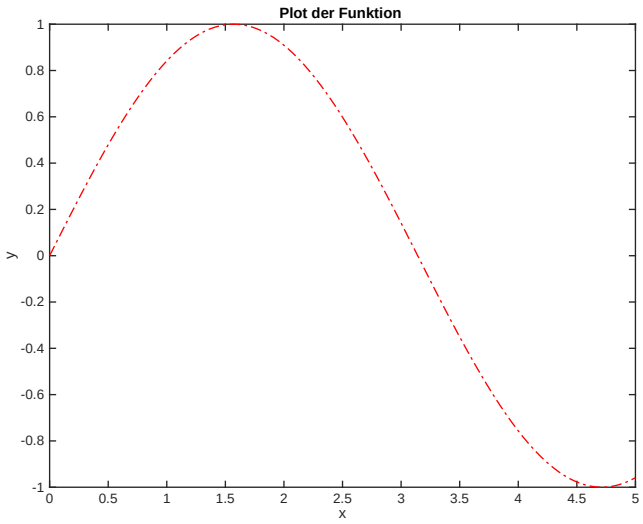

Abbildung 1: Plot der Sinusfunktion durch die Funktion `funcPlotter`

2. Anwendungsbeispiel

Als zweites Anwendungsbeispiel wollen wir noch einmal auf das Beispiel aus der Einführung zurückgreifen. Wir erinnern uns daran, dass wir mit Hilfe von (1) die Ableitung einer Funktion an einer vorgegebenen Stelle numerisch approximieren wollen. Desweiteren haben wir festgestellt, dass das Vorgehen aus der Einleitung auf Dauer sehr unübersichtlich wird, da man den Differenzenquotienten (1) für jede neue Funktion auch neu eintippen müsste.

Übersichtlicher ist es, wenn man eine Routine `numAbleitung` programmiert, die eine vorgegebene Funktion f, einen Wert x und ein festes $h > 0$ als Parameter übergeben bekommt und damit die näherungsweise Ableitung $f'(x)$ nach Gleichung (1) berechnet. Die Routine müsste dann nur einmal programmiert werden und könnte für alle Funktionen genutzt werden. Dies bedeutet aber, dass man die Funktion f als Argument der Funktion `numAbleitung` übergeben muss, welches mit Hilfe von Function Handles realisiert werden kann. Die Routine `numAbleitung` könnte beispielsweise folgende Gestalt haben:

```
1  function Df = numAbleitung(f, x, h)
2    Df = (f(x+h) - f(x)) / h;
3  end
```

Näherungen (mit Schrittweite $h = 0.001$) für die Ableitungen der Funktionen $f_1(x) = \sin(x)$ und $f_2(x) = x^2$ an der Stelle $x = 1$ erhält man durch:

```
>> format long
>> numAbleitung(@sin, 1, 0.001)
ans =
   0.539881480360327
>> numAbleitung(@(x) x^2, 1, 0.001)
ans =
   2.000999999999697
```

Operationen mit Function Handles

MATLAB beinhaltet zwei Funktionen, welche es ermöglichen, Function Handles in Strings zu konvertieren und umgekehrt.

Function Handles in Strings konvertieren und umgekehrt

Mit dem Befehl `str = func2str(fhandle)` lässt sich aus einem Function Handle `fhandle` ein String `str` mit dem Namen der Funktion hinter dem Handle erzeugen. Das Gegenstück zu diesem Befehl lautet `fhandle = str2func(str)`, welcher aus einem String `str` ein Function Handle `fhandle` erstellt.

Wir greifen noch einmal unser erstes Anwendungsbeispiel auf, in dem wir einen Funktions-Plotter geschrieben haben. Dieser soll an dieser Stelle etwas modifiziert werden. Ziel ist es, dass der Name der Funktion, die geplottet wird, sowohl auf der y-Achse als auch im Titel erscheint. Wir erinnern daran, dass die zu plottende Funktion als Function Handle an die Funktion `funcPlotter`

übergeben wird. Wir wollen also ein Function Handle in einen String umwandeln. Dazu führen wir zwischen den Zeilen 5 und 6 noch folgende Zeile ein

```
1  func_name = func2str(func_handle);
```

und modifizieren die Zeilen 11 und 12 entsprechend. Insgesamt erhalten wir also:

```
1  function funcPlotter(func_handle)
2    if ~isequal(class(func_handle), 'function_handle')
3      error('Argument muss eine Function Handle sein!');
4    end
5
6    func_name = func2str(func_handle);
7    x = linspace(0, 5);
8    fx = func_handle(x);
9
10   plot(x, fx, 'r-.');
11   xlabel('x');
12   ylabel(func_name);
13   title(sprintf('Plot der Funktion %s', func_name));
14 end
```

Als Ausgabe entsteht dann folgender Plot:

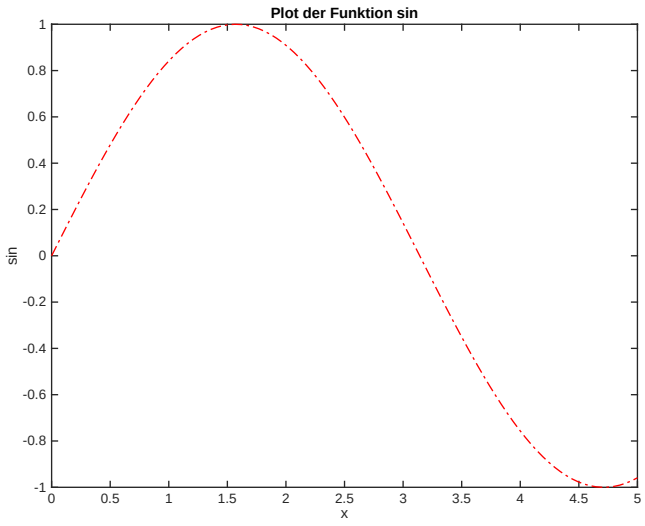

Abbildung 2: Plot der Sinusfunktion

Offline Aktivitäten

Übereinstimmen

Schreiben Sie vor jeden Begriff auf der linken Seite den passenden Buchstaben der Beschreibung, die am besten mit der aus der rechten Spalte übereinstimmt.

_____	1. `str2func`	a.	Erzeugt ein Function Handle.
_____	2. `feval`	b.	Wertet eine Funktion nach dem Function Handle oder Namen der Funktion aus.
_____	3. `func2str`	c.	Wandelt einen String in ein Function Handle um.
_____	4. `@functionname`	d.	Wandelt ein Function Handle in ein String um.
_____	5. `@(arglist)function`	e.	Erzeugt eine anonyme Funktion.

Ihre Antwort:

Fragen und Antworten

Beantworten Sie die folgenden Fragen.

6. Sei `foo` der Name einer Funktion. Geben Sie zwei verschiedene Befehle an, mit denen man ein Function Handle für `foo` erzeugen kann.

   Ihre Antwort:

7. Ihre Funktion `foo` hat als Parameter ein Function Handle `bar`. Innerhalb Ihrer Funktion `foo` wird geprüft, ob die Funktion, die hinter dem Handle `bar` steckt, einen gültigen Rückgabewert liefert. Falls dies nicht der Fall ist, soll eine Fehlermeldung ausgegeben werden. Mit welchem Befehl könnte eine Fehlermeldung erzeugt werden, in der der Name der ungültigen Funktion vorkommt?

   Ihre Antwort:

8. Wir definieren `x = ones` und `y = @ones`. Was ist die Ausgabe von:

   a) x(1),
   b) y(1),
   c) x(2) und
   d) y(2)?

   Ihre Antwort:

Programmausgaben

Für jedes der folgenden Programmsegmente lesen Sie zuerst die Zeilen und schreiben Sie die Ausgabe an die dafür vorgesehene Stelle.

9. Wie lautet die Ausgabe des folgenden Skriptes:

```
1  handle = @sin;
2  x = pi/2;
3  fprintf('%s(%f) = %f\n', func2str(handle), x, handle(x))
```

Ihre Antwort:

10. Es sei die Funktion `maxGrid` wie folgt definiert:

```
1  function [xmax, fmax] = maxGrid(fh, x)
2
3  xtemp = x;
4  fmax  = fh(x(1));
5  count = 1;
6
7  for i=2:length(x)
8    temp = fh(x(i));
9    if temp > fmax
10     xtemp(1)     = x(i);
11     fmax         = fh(x(i));
12     count        = 1;
13   elseif temp == fmax
14     count        = count + 1;
15     xtemp(count)= x(i);
16   end
17 end
18
19 xmax = xtemp(1:count);
20 end
```

Wie lautet die Ausgabe des folgenden Aufrufs?

```
1  x = [0 pi/2 pi 3/2*pi];
2  [xmax, ymax] = maxGrid(@(x)abs(sin(x)), x)
```

Ihre Antwort:

Praktikumsaufgaben

11. a) Schreiben Sie die Funktionen `res = axpy(alpha, x, y)` und `res = myeval(f, x, y)`, die folgende Zuweisungen durchführen

$$\texttt{axpy}(\alpha, x, y) = \alpha\, x + y$$
$$\texttt{myeval}(f, x, y) = f(x, y).$$

b) Benutzen Sie nun die Funktion `myeval`, um die Funktion `axpy` an der Stelle $\alpha = 2, x = 5, y = 10$ auszuwerten.

12. Betrachten Sie die Funktion

$$w_n(x) = A_n \sin(\frac{2\pi\, n}{P} x).$$

Die Funktion ist eine sinusförmige Welle, wobei die Amplitude durch A_n und die Frequenz durch n bestimmt werden. In dieser Aufgabe wollen wir eine Funktion schreiben, die Function Handles für verschiedene Wellen erzeugt, und eine weitere Funktion, die diese Wellen graphisch darstellt.

a) Schreiben Sie eine Funktion `sinusoids.m`, die den Parameter P sowie die Parametervektoren n und A_n erhält. Zurück gegeben werden soll ein Cell Array, der für jedes Parameterpaar $(n(i), A_n(i))$ ein Function Handle der entsprechenden Wellenfunktion w_n enthält.

b) Schreiben Sie anschließend eine Funktion `plotwaves.m`, die als Parameter den vorher erzeugten Cell Array und Intervallgrenzen a, b erhält. Die Funktion soll dann die vorher erzeugten Wellen im Bereich $[a, b]$ graphisch darstellen.

c) Testen Sie anschließend Ihre Funktionen, indem Sie beispielsweise für Parametervektoren n bzw. A_n mit jeweils 5 Komponenten Wellen erzeugen und diese dann plotten.

13. Zur Lösung von nichtlinearen (skalaren) Gleichungen verwendet man in der Numerik häufig Algorithmen, die sich zur Nullstellenbestimmung eignen. Der Grund dafür ist, dass sich nichtlineare Gleichungen in ein Nullstellenproblem umformulieren lassen. Eine der Methoden zur Nullstellenbestimmung ist die sogenannte *Bisektionsmethode*, welche im Wesentlichen nichts anderes als eine Intervallschachtelung ist. Um die Existenz der Nullstelle x^* im Intervall $I := (a, b)$ zu garantieren, setzen wir im folgenden voraus, dass die Funktion $f : [a, b] \to \mathbb{R}$ stetig ist und dass $f(a) \cdot f(b) < 0$ gilt (Zwischenwertsatz).

    Das Verfahren funktioniert wie folgt: Zunächst wird im Mittelpunkt $m = \frac{1}{2}(a + b)$ des Intervalls I der Funktionswert $f(m)$ bestimmt. Ist $f(m) \neq 0$ (d. h. die Nullstelle ist noch nicht gefunden), entscheidet das Vorzeichen von $f(m)$, in welchem der Teilintervalle $[a, m]$ bzw. $[m, b]$ die gesuchte Lösung x^* liegt. Dieses wird über die Bedingung $f(a) \cdot f(m) < 0$ bzw. $f(m) \cdot f(b) < 0$ untersucht. Nun passt man die Intervallgrenzen entsprechend an, so dass x^* im neuen halbierten Intervall liegt, und beginnt von vorne. Diese Schritte führt man solange aus, bis die Länge des Intervalls eine gegebene Toleranz unterschreitet. Der letzte berechnete Wert des Mittelpunktes ist dann die approximierte Nullstelle.

    a) Schreiben Sie nun eine Funktion

    ```
    val = bisection(f, a, b, tol),
    ```

    welche für eine gegebene Funktion f, die als Function Handle zu übergeben ist, ein gegebenes Intervall $I = (a, b)$ und eine gegebene Toleranz `tol` näherungsweise die Nullstelle bestimmt. Verwenden Sie als Abbruchkriterium, dass die Länge des Intervalls kleiner oder gleich der gegebenen Toleranz ist.

    b) Bestimmen Sie mit Hilfe Ihres Programmes jeweils näherungsweise die Nullstelle der folgenden Funktionen:

        i) $f(x) = \sin(x)$ auf $I = [2, 4]$ mit `tol` $= 0.001$.

        ii) $f(x) = x^3 - 1$ auf $I = [-1, 3]$ mit `tol` $= 0.001$.