Programmstrukturierung, Debugging und weitere Tipps für das Programmieren mit Matlab

Jascha Knepper

Mathematisches Institut, Universität zu Köln

Universität zu Köln

Übersicht

- Diverses: `surf`- und `mesh`-Plots, Funktionsparameter, Matrixindizierung und Vektorisierung
- Programmstrukturierung
- Debugging
- Performance

Diverses

`surf`- und `mesh`-Plots, Funktionsparameter, Matrixindizierung und Vektorisierung

Einige grundlegende Tipps

- In der Kommandozeile eingeben:

  `doc BEFEHL` z.B. `doc reshape`

- Matlab-Editor: Markierungen beachten und Probleme beheben (oft sinnvoll).
- `cell arrays`: Array mit beliebigem Inhalt (kann z.B. Matrizen, Vektoren, Strings etc. enthalten).

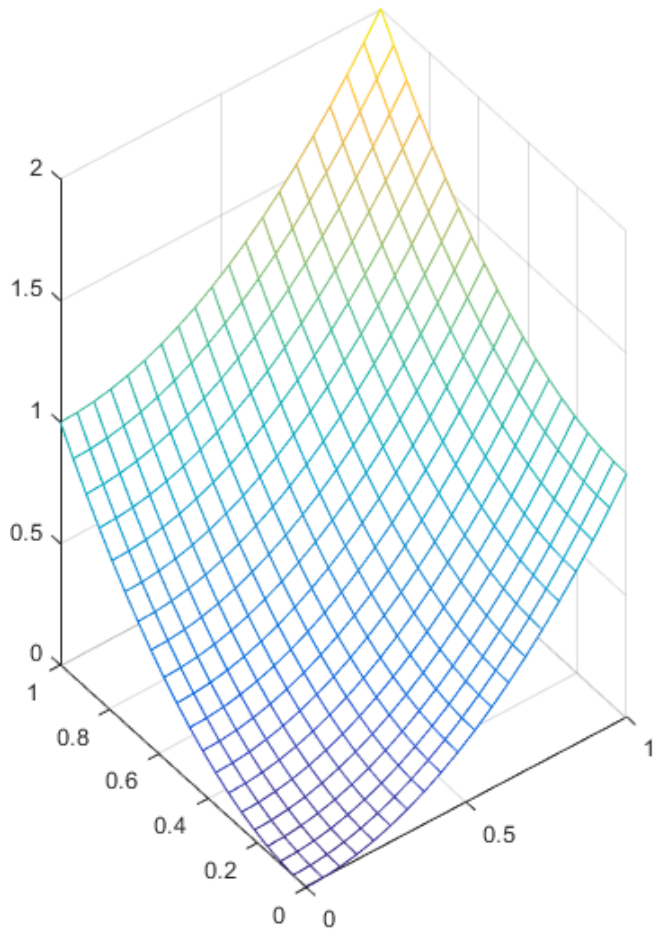
```
1  clear;clc
2
3  a = randi(25,3);    % Vektor
4  B = randn(10,11);   % 2D-Matrix
5  C = rand(4,5,6);    % 3D-Matrix
6  method = 'test';
7  x = [];
8  y = pi;
9
10 daten = {a,B,C,method,x,y};
11
12 teilDaten = daten(2:4);
13
14 einElement = teilDaten{3}
15
16 teilDaten{3} = [];
```

surf- und mesh-Plots und Downsampling

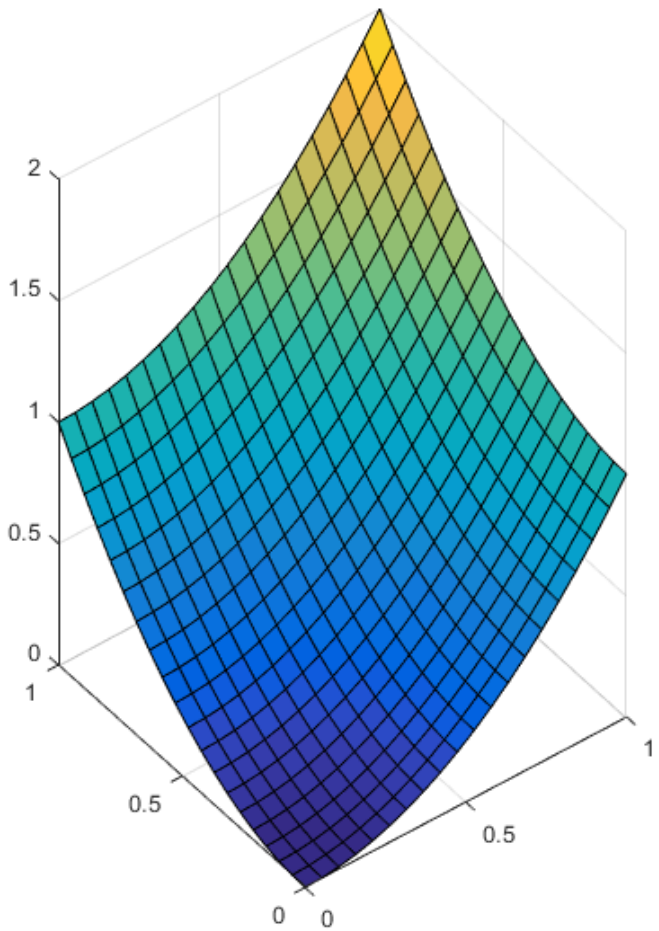
- Plotte die Funktion $x^2 + y^2$ auf einem Gitter.

Niedrige Auflösung

```
1  clear;clc
2  n = 20;
3  x = linspace(0,1,n);
4  y = x;
5  [x,y] = meshgrid(x,y);
6  z = x.^2 + y.^2;
7
8  figure
9  subplot(1,2,1)
10 mesh(x,y,z)
11 subplot(1,2,2)
12 surf(x,y,z)
```



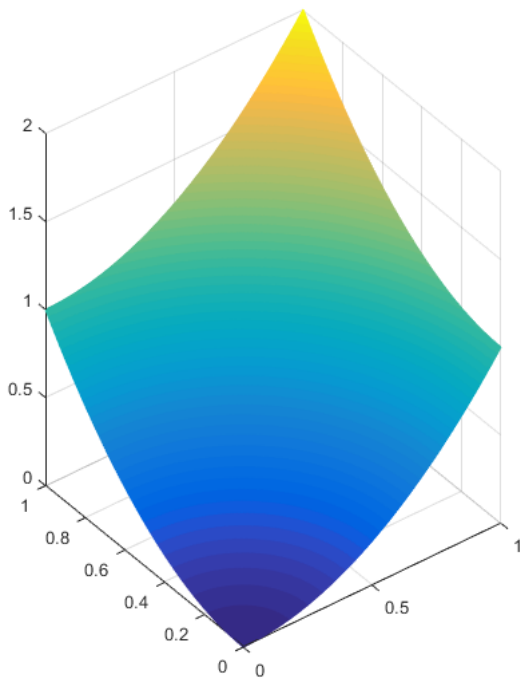
`mesh`-Plot



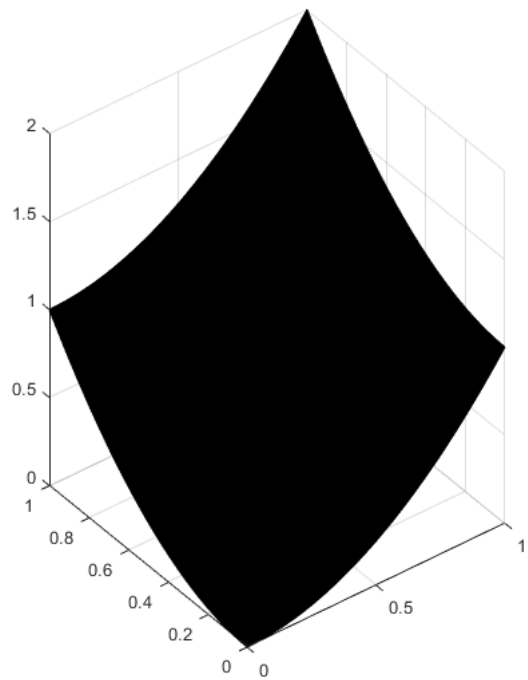
`surf`-Plot

Hohe Auflösung

```
1  clear;clc
2  n = 500;
3  x = linspace(0,1,n);
4  y = x;
5  [x,y] = meshgrid(x,y);
6  z = x.^2 + y.^2;
7
8  figure
9  subplot(1,2,1)
10 mesh(x,y,z)
11 subplot(1,2,2)
12 surf(x,y,z)
```



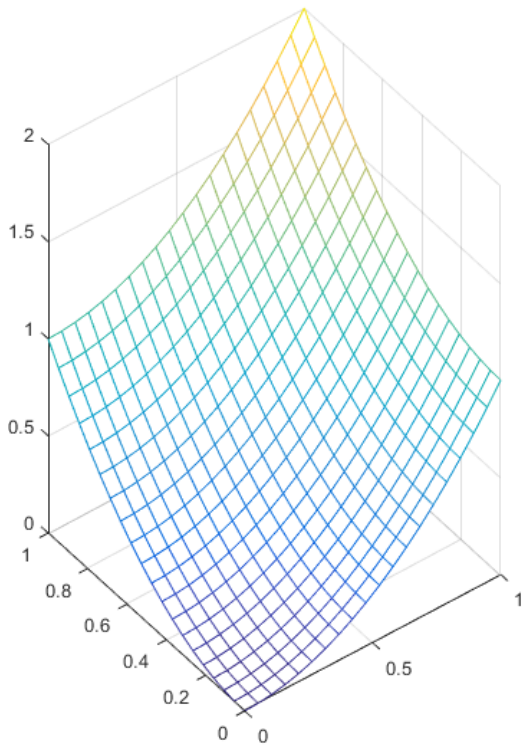
`mesh`-Plot



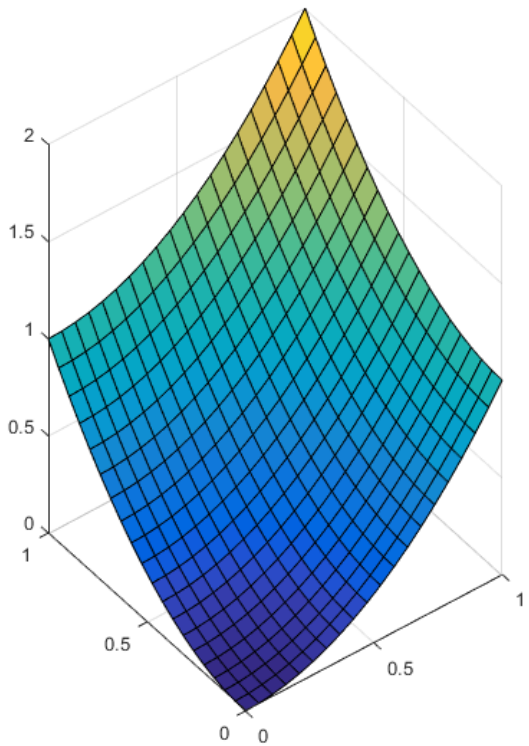
`surf`-Plot

Downsampling

```
14 ind = 1:10:n;
15 figure
16 subplot(1,2,1)
17 mesh(x(ind,ind),y(ind,ind),z(ind,ind))
18 subplot(1,2,2)
19 surf(x(ind,ind),y(ind,ind),z(ind,ind))
```



`mesh`-Plot



`surf`-Plot

Funktionsparameter

Bei der Parameterübergabe ist man relativ frei. Es bieten sich, je nach Situation, oft folgende Typen an:

- **string:** z.B. Methodenwahl

```
function y = param_string(x,method)
    if strcmpi(method,'square')
        y = x.^2;
    elseif strcmpi(method,'ident')
        y = x;
    else
        error('Unbekannte Methode %s.',method)
    end
end
```

- **boolean:** z.B. um etwas ein- oder auszuschalten

```
function y = param_boolean(x,isAlwaysZero)
    if isAlwaysZero
        y = zeros(size(x));
    else
        y = x;
    end
end
```

- **integer:** z.B. als flag

  Achtung: Man kann zwar in Matlab auch Integertypen verwenden (z.B. mit `int32(wert)`), meist reicht es jedoch eine ganze Zahl im Standardtyp `double` von Matlab zu speichern.

```
1  function y = param_integer(x,iflag)
2      switch iflag
3          case 1 % Methode 1
4              y = x.^2;
5          case 2 % Methode 2
6              y = x;
7          otherwise
8               error('Unbekannter flag %i.',iflag)
9      end
10 end
```

Sollen mehrere (vorzugsweise konstante) Parameter übergeben werden, so kann man diese in einem Struct[1] übergeben. Ein Struct kann dann einfach an Subroutinen weitergegeben werden, was das Vertauschen von Parametern verhindert.

Struct

```
1  function XT = param_struct()
2      settings.T = 1;
3      settings.dt = 0.1;
4      settings.h = 0.2;
5      settings.method = 'VR';
6
7      XT = finite_difference(settings);
8  end
9
10 function XT = finite_difference(settings)
11     XT = [];
12     fprintf('T  = %g\n',settings.T)
13     fprintf('dt = %g\n',settings.dt)
14     fprintf('h  = %g\n',settings.h)
15     fprintf('method: %s\n',settings.method)
16 end
```

[1] Performance-Tipp: man sollte vermeiden einen Array aus Structs zu benutzen und einen Struct aus Arrays vorziehen, d.h. vermeide `array(i).x` und verwende `x.array(i)`.

Matrixindizierung, logische Indizierung und Vektorisierung

- Extrahiere Teilmatrix.

Schleife

```
1  clear;clc
2  A = randn(10,10);
3  jVon = 3; jBis = 7;
4  iVon = 2; iBis = 4;
5
6  B = zeros(iBis-iVon+1,jBis-jVon+1);
7  for i = iVon:iBis
8      for j = jVon:jBis
9          B(i-iVon+1,j-jVon+1) = A(i,j);
10     end
11 end
```

Vektorisiert

```
1  clear;clc
2  A = randn(10,10);
3  jVon = 3; jBis = 7;
4  iVon = 2; iBis = 4;
5
6  B = A(iVon:iBis,jVon:jBis);
```

- Werte $y(2\pi x)$ auf einem Gitter $G \subseteq [0,1]$ aus.

Schleife

```
1 clear;clc
2 n = 100;
3 h = 1/n;
4 y = zeros(n+1,1);
5
6 for i = 0:n
7     y(i+1) = sin(2*pi*i*h);
8 end
```

Vektorisiert

```
1 clear;clc
2 n = 100;
3 y = sin(2*pi*linspace(0,1,n+1));
```

- Plotte alle Gitterpunkte x^h: $y(x^h) \leq -0.5$.

Schleife

```
1  clear;clc
2  n = 100;
3  x = linspace(0,1,n+1);
4  y = sin(2*pi*x);
5
6  figure
7  plot(x,y)
8  hold on
9
10 for i = 1:length(y)
11     if (y(i) <= -0.5)
12         plot(x(i),y(i),'ro')
13     end
14 end
```

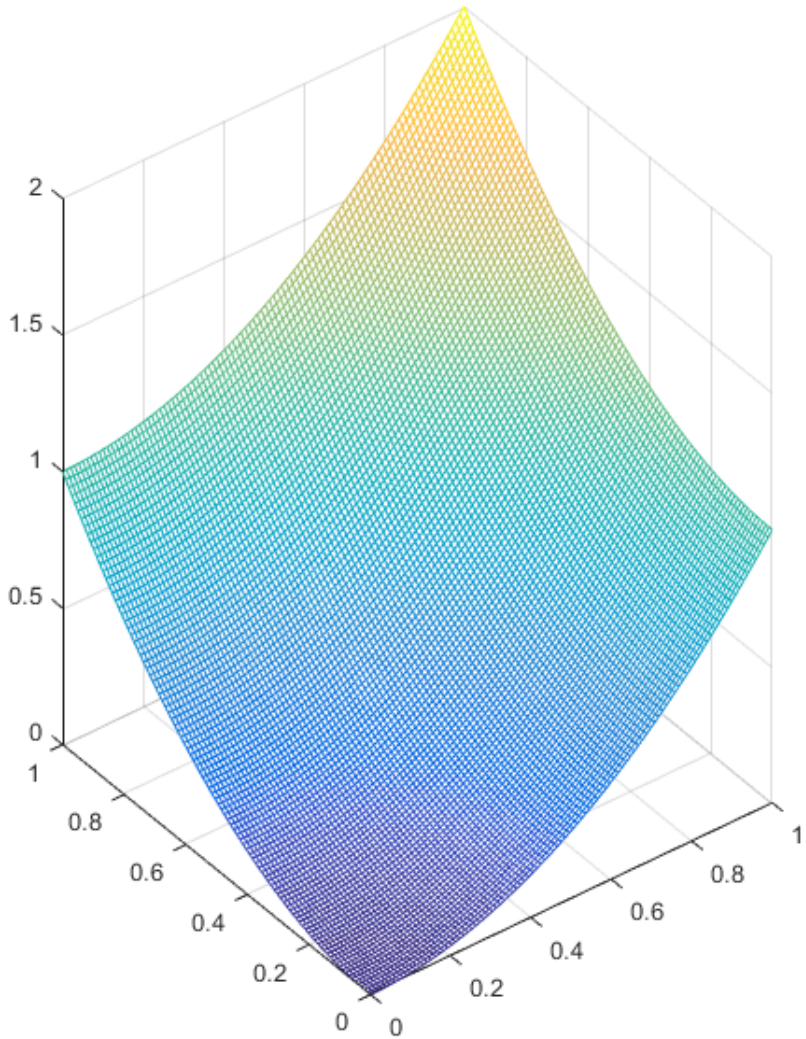
Vektorisiert

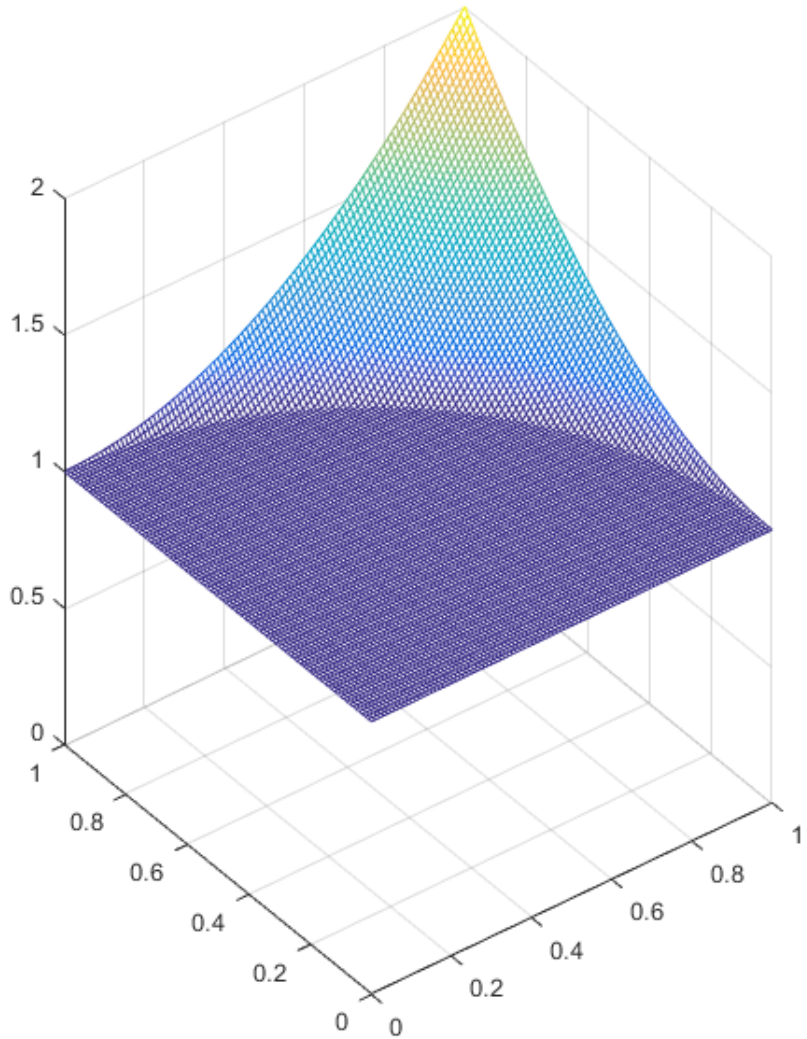
```
1  clear;clc
2  n = 100;
3  x = linspace(0,1,n+1);
4  y = sin(2*pi*x);
5
6  figure
7  plot(x,y)
8  hold on
9
10 ind = (y <= -0.5);
11 plot(x(ind),y(ind),'ro')
```

- Werte die Funktion $x^2 + y^2$ auf einem Gitter $G \subseteq [0,1]^2$ aus und setze alle Werte auf 1, in denen $x^2 + y^2 < 1$ gilt.

Logische Indizierung

```
1  clear;clc
2  n = 100;
3  x = linspace(0,1,n);
4  y = x;
5  [x,y] = meshgrid(x,y);
6  z = x.^2 + y.^2;
7
8  figure
9  subplot(1,2,1)
10 mesh(x,y,z)
11 zlim([0,2])
12
13 z(z < 1)=1;
14 subplot(1,2,2)
15 mesh(x,y,z)
16 zlim([0,2])
```


2
1.5
1
0.5
0
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


2
1.5
1
0.5
0
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

Zwei hilfreiche Befehle:

- `reshape`: Forme z.B. eine Vektor $a \in \mathbb{R}^{1\times 2n}$ zu einer Matrix $A \in \mathbb{R}^{2\times n}$ um: `A=reshape(a,2,n);`.

- `repmat`: Dupliziere Matrix oder Vektor, z.B. $A = (B, B, B)$, mit $B \in \mathbb{R}^{m\times n}$; `A=repmat(B,1,3);`.

Programmstrukturierung

Funktionstypen, Mathematische Notation und Diverses

- Ein strukturiertes Programm hilft die Zeit, die zum Debugging benötigt wird, zu verringern,
- das Programm kann leichter erweitert oder wiederverwendet werden und
- der Code ist leichter verständlich.

Es gibt eine Reihe von Möglichkeiten ein Programm zu strukturieren. Die wichtigste Maßnahme ist dabei die Modularisierung, in Matlab mit Funktionen[2].

[2] In Matlab kann man auch objektorientiert schreiben: `https://de.mathworks.com/help/matlab/object-oriented-programming.html`. Wir werden diesen Aspekt hier jedoch nicht betrachten.

Funktionstypen

Die wichtigste Art Programme in Matlab zu strukturieren ist das Zusammenfassen von Codeteilen in Funktionen. Dazu stehen uns verschiedene Typen zur Verfügung.

```
1   function x = funktionstypen(n,v)
2
3       x = zeros(n,n);
4
5       anonymousFunction = @(j) subFunction(j);
6
7       nestedFunction(2)
8
9       function nestedFunction(j)
10          x(j,j) = anonymousFunction(v);
11      end
12  end
13
14  function ind = subFunction(j)
15      ind = 2*j;
16  end
```

- **main function**
- **local/sub function**

  Eine Unterfunktion wird nur von der Hauptfunktion (Typ `function`) oder von anderen Unterfunktionen der Hauptfunktion aufgerufen. Der Code steht unterhalb des Codes der Hauptfunktion. Ist die Unterfunktion zu umfangreich, sollte sie als Hauptfunktion in eine eigene Datei ausgelagert werden.
- **nested function**

  Eingebettete Funktionen stehen innerhalb einer Hauptfunktion (vorzugsweise am Ende) und haben direkten Zugriff auf die Variablen der Hauptfunktion, wodurch bei Schreiboperationen keine Kopien erstellt werden müssen. Außerdem müssen diese Variablen nicht beim Aufruf übergeben werden.

  Aber Vorsicht: Da die Variablen der Hauptfunktion in der eingebetteten Funktion zur Verfügung stehen (und überschrieben werden können), verringert sich die Übersicht und man baut leichter Fehler ein.

  Möchte man eingebettete Funktionen nutzen, sollte man genau darauf achten,

  1. alle lokalen Variablen vorher zu initialisieren,
  2. gewünschte Variablen der Hauptfunktion tatsächlich zu überschreiben,
  3. Variablen der Hauptfunktion nicht ungewünscht zu überschreiben.
- **anonymous function**

  Mit anonymen Funktionen kann man schnell simple Funktionen implementieren, für die eine eigene Datei Overkill wäre, z.B. `f=@(x)x.^2`.

Struktur durch mathematische Notation

Wird ein Problem aus der Mathematik implementiert, so kann es sinnvoll sein, sich so nah wir möglich an die mathematische Notation im Programm zu halten. Oftmals ist dadurch auch schnell klar, wie sich das Programm modular aufbauen lässt. Dadurch bekommt das Programm eine sinnvolle und verständliche Struktur.

unstrukturiert

```
39 function y = RK_explicit(A,y0,tIter,dt,ButcherTableau)
40     % alpha = ButcherTableau(1:end-1,1);
41     beta =  ButcherTableau(1:end-1,2:end);
42     gamma = ButcherTableau(end,2:end);
43     N = length(gamma);  % Anzahl Stufen des Verfahrens
44     %
45     y = zeros(length(y0)+2,tIter+1); % init
46     y(2:end-1,1) = y0;
47     yn = y0;
48     %
49     for n = 1:tIter
50         k = zeros(length(yn),N);
51         ynp1 = yn;
52         for i = 1:N
53             k(:,i) = A*yn;
54             sum = zeros(size(yn));
55             for j = 1:i-1
56                 sum = sum + dt*k(:,j)*beta(i,j);
57             end
58             k(:,i) = k(:,i) + A*sum;
59             %
60             ynp1 = ynp1 + dt*k(:,i)*gamma(i);
61         end
62         %
63         y(2:end-1,n+1) = ynp1;
64         yn = ynp1;
65     end
66 end
```

strukturiert

```
39 function k = compute_ki(A,yn,dt,beta,N)
40     k = zeros(length(yn),N);
41     for i = 1:N
42         k(:,i) = A*(yn + dt*k*beta(i,:)');
43     end
44 end
45
46 function y = RK_explicit(A,y0,tIter,dt,ButcherTableau)
47     % alpha = ButcherTableau(1:end-1,1);
48     beta =  ButcherTableau(1:end-1,2:end);
49     gamma = ButcherTableau(end,2:end);
50     N = length(gamma);  % Anzahl Stufen des Verfahrens
51     %
52     y = zeros(length(y0)+2,tIter+1); % init
53     y(2:end-1,1) = y0;
54     yn = y0;
55     %
56     for n = 1:tIter
57         % Werte Inkrementfunktion aus.
58         k = compute_ki(A,yn,dt,beta,N);
59         phi = k*gamma';
60         %
61         yn = yn + dt*phi;  % Iteriere
62         %
63         y(2:end-1,n+1) = yn;
64     end
65 end
```

Diverses

Allgemeiner Tipp zum Programmieren

1. Zuerst eine einfache Methode implementieren (z.B. explizites Euler-Verfahren) und testen.
2. Anschließend das allgemeine Verfahren (z.B. explizites RK-Verfahren) implementieren und testen. Notfalls auf die einfache Methode (in allgemeiner Formulierung) zurückfallen bis der Fehler behoben wurde.
3. Nach jedem abstraktem Schritt sollte das Programm auf Korrektheit getestet werden.
4. Wähle für die Tests zunächst ein einfaches Modellproblem (z.B. $y' = \lambda y$).

- Die Komplexität eines Programmes kann u.a. anhand der *McCabe-Metrik* (zyklomatische Komplexität) bestimmt werden. Diese ermittelt die Anzahl unabhängiger Pfade, die durch eine Funktion genommen werden können; je mehr Abzweigungen es gibt (z.B. durch geschachtelte `if`-Abfragen), desto unübersichtlicher wird das Programm. Manchmal wird empfohlen bereits ab einem Wert von 10 die Funktion in Teilfunktionen aufzubrechen. In Matlab kann die Komplexität mit

  ```
  mlint('dateiname.m','-cyc')
  ```

  bestimmt werden.

- Läuft ein Programm nach der Implementierung korrekt oder wurde ein altes Programm erweitert, so ist es oft notwendig dieses Programm neu zu strukturieren, um es in Zukunft einfacher wiederzuverwenden bzw. erweitern zu können. Dies nennt sich *code refactoring*.
- *interfaces*: Hat man z.B. eine Reihe verschiedener Methoden implementiert, so kann es sinnvoll sein ein Interface zu schreiben, um in Zukunft einfacher Zugriff auf die Methoden zu haben. So ist z.B. die `interp1`-Funktion in Matlab ein Interface zu verschiedenen Interpolationsmethoden, welche mittels eines Parameters ausgewählt werden können.

  Dies verringert zudem den Zugriff von außerhalb auf "Low-Level"-Funktionen. Werden an diesen Änderungen vorgenommen, so ist es oftmals einfacher im Interface die Daten des Users aufzubereiten, anstatt den User für jeden Aufruf einer Low-Level-Funktion dazu zu zwingen die Daten selbst aufzubereiten (z.B. Transponieren eines Vektors). Auch lassen sich so einfacher Legacy-Versionen anbieten.

  Interfaces können somit die Codestruktur und Wiederverwendbarkeit verbessern.

Top-Down- und Bottom-Up-Programmierung

- **Top-Down:** Definiere zunächst die wichtigsten Funktionen (von oben nach unten). Low-Level-Funktionen werden durch Dummy-Funktionen ersetzt. Dadurch liegt die Struktur des Programmes fest und es können nach und nach (von oben nach unten) die Details ergänzt werden.
- **Bottom-Up:** Schreibe zunächst die Low-Level-Funktionen (z.B. explizite RK-Methode, implizite RK-Methode) und füge diese anschließend zu etwas Größerem zusammen (schreibe zB. ein Interface).

Je komplexer das Programm ist (und insb. bei Teamarbeit), desto mehr Gedanken muss man sich a priori über die Programmstruktur machen. Man tendiert in diesem Fall zum Top-Down-Ansatz. Bei kleineren Programmen bzw., falls man noch nicht weiß, wie das Programm am Schluss aussehen soll, tendiert man zum Bottom-Up-Ansatz. Oftmals wendet man jedoch eine Mischung aus beiden Prinzipien an.

Debugging

Debugging-Hilfen, Prinzipien und der Matlab-Debugger

Ein gut strukturiertes, modulares Programm ist essentiell, um die einzelnen Programmteile gezielt testen zu können. Dies erleichtert den Debugging-Prozess und wird umso wichtiger, je komplexer das Programm ist.

Debugging-Hilfen

Punkte, die man im Voraus beachten kann, um den Debugging-Prozess zu verkürzen:

- Die Reduzierung der Anzahl an Übergabeparameter (z.B. durch Nutzung von Structs) hilft oft im Voraus Fehler zu vermeiden.
- Standard-Debugging-Ausgaben (`fprintf` für wichtige Informationen) sowie `error`, `assert` und `warning` verwenden.

```
1  function debug_p1()
2      info = struct('T',0.0,'dt',0.1,'h',0.05,'a',-pi,'b',pi);
3      test(info);
4  end
5
6  function test(info)
7      assert(info.T > 0.0,'T muss positiv sein.')
8      assert((info.dt > 0.0) && (info.h > 0.0),'dt und h muessen positiv sein.')
9      if info.dt > info.h
10         warning('CFL-Bedingung nicht erfuellt.')
11     end
12     fprintf('Parameter:\n\tT = %g, dt = %g, h = %g, a = %g, b = %g\n', ...
13         info.T,info.dt,info.h,info.a,info.b)
14 end
```

- Globale Variablen (und *nested functions*) vermeiden.
- `clear` und `clc` am Anfang eines Skriptes verwenden.
- Sinnvolle Variablennamen
- Sinnvolle Kommentare (beschreibe eher **was** passiert als **wie**)
- Code einrücken.
- *Code cells* zum Strukturieren nutzen (anders als ein normaler Kommentar werden diese mit zwei Prozentzeichen eingeleitet).
- Lange Zeilen umbrechen (nutze `...` für Zeilenumbruch).
- Codestil und Formatierung sollten konsistent sein.
- Während der Implementierung muss oft ein Kompromiss zwischen einer allgemeinen Umsetzung eines Problems und der harten Kodierung gefunden werden (Bsp.: klassisches RK-Verfahren (4 Stufen): explizit aufschreiben oder allgemein in Schleifenform mit Butcher-Tableau?). Hier kann man sich die Frage stellen, ob eine allgemeine Umsetzung den Debugging-Aufwand verringert und, ob man die allgemeine Implementierung in Zukunft für etwas anderes (z.B. ein anderes RK-Verfahren) wiederverwenden kann.
- Hängt ein Programm von unbekanntem, externen Input ab, (z.B., wenn eine Datei geöffnet wird), so kann es sinnvoll sein mit `try` und `catch` Fehler abzufangen.

Debugging-Prinzipien

Wie beim Programmieren kann man beim Debugging die Top-Down- und Bottom-Up-Prinzipien anwenden.

Beim Top-Down-Debugging ersetzt man Funktionen durch Dummy-Funktionen und verifiziert den Code von oben nach unten. Die Dummy-Funktionen werden so nach und nach wieder entfernt.

Beim Bottom-Up-Debugging überprüft man die Korrektheit von Low-Level-Funktionen mit Hilfe von Tests und arbeitet sich so von unten nach oben vor. Dies ist generell die robustere Methode, da keine Dummy-Funktionen getestet werden.

Matlab-Debugger

Genereller Hinweis zum Debugging mit Matlab:

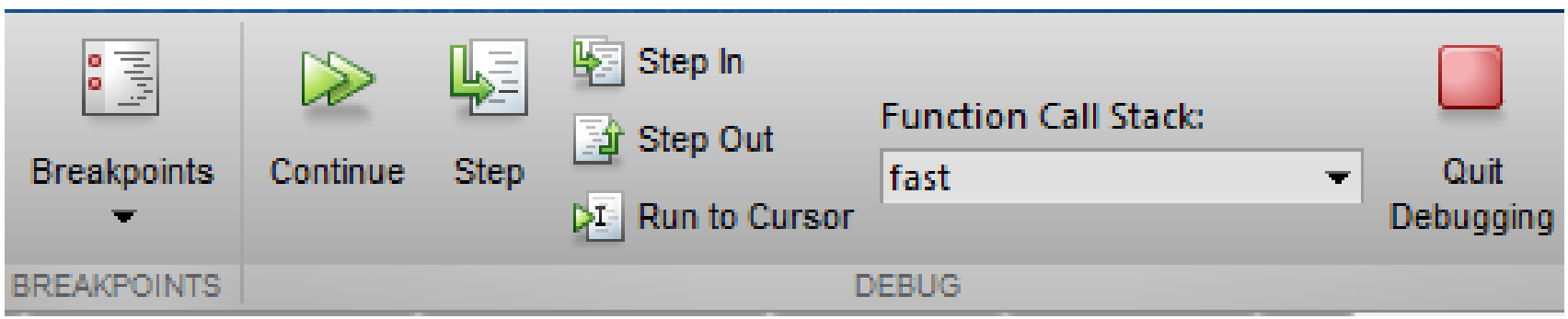
Drückt man `STRG+C` in der Konsole, kann ein Programm abgebrochen werden. Manchmal kann dies etwas dauern, da einerseits Matlab nur periodisch überprüft, ob der Nutzer `STRG+C` gedrückt hat, und anderseits, da Matlab ggfs. in einer vorkompilierten Funktion steckt und diese nicht abbrechen kann. Dauert dies zu lange, hilft es nur Matlab zu beenden.

- Eine simple Form des Debuggings besteht in der Ausgabe von Informationen (z.B. mit `disp` oder `fprintf` oder durch Entfernung des Semikolons) sowie ggfs. der Verwendung des `pause`- oder `keyboard`-Befehls. Letzterer erlaubt es die Variablen im aktuellen Workspace anzuschauen und zu manipulieren (man setzt mit `keyboard` einen **Breakpoint**).

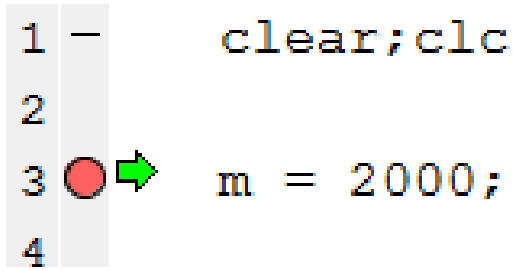
  Weiterhin kann man Sicherheitstests mit `assert` oder `error` einbauen.

  Tipp: Manchmal ist es hilfreich vorzeitig eine Funktion oder ein Skript zu verlassen. Dies kann mit `return` geschehen.

- Für eine fortgeschrittene Fehlersuche enthält Matlab einen Debugger, welcher erlaubt
  1. (conditional) Breakpoints zu setzen,
  2. ein Programm Schritt-für-Schritt oder abschnittsweise durchlaufen zu lassen,
  3. beim Auftreten von Fehlern, Warnungen, `NaN` oder `Inf` in den Debugging-Modus zu schalten.



Breakpoints

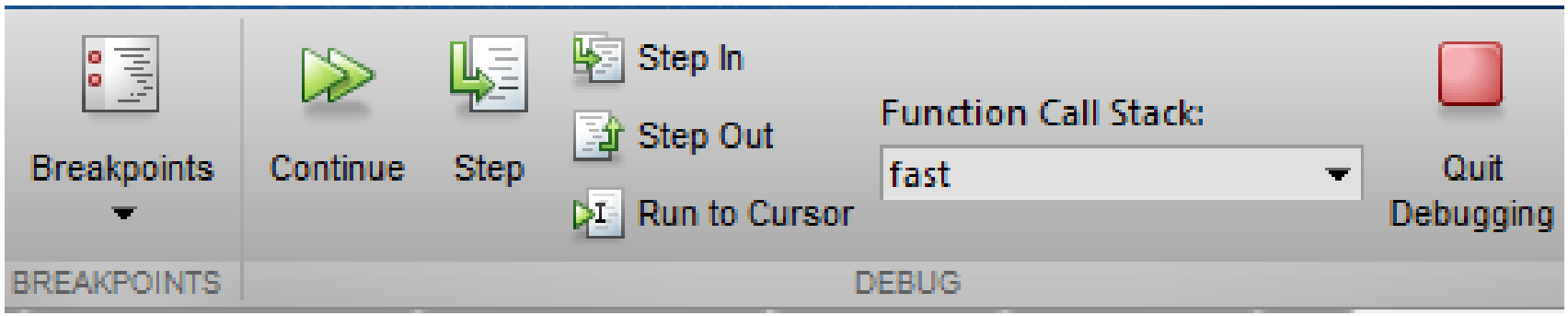


Breakpoints werden entweder an der entsprechenden Stelle im Code mit dem `keyboard`-Befehl gesetzt oder am linken Rand einer Codezeile in der GUI von Matlab. Führt man das Programm aus, so stoppt Matlab an der entsprechenden Stelle und man kann die lokalen Variablen auswerten oder modifizieren. Änderungen am Code werden jedoch erst übernommen, wenn das Programm beendet wurde.

Komplexe Bedingungen an einen Breakpoint (*conditional breakpoint*) können einfach im Code zusammen mit dem `keyboard`-Befehl umgesetzt werden. Simple Bedingungen können auch in der GUI (Rechtsklick auf einen Breakpoint) gesetzt werden (hilfreich in Schleifen, wenn z.B. erst zum i-ten Schritt der Debugging-Modus aktiviert werden soll).

Ist man im Debugging-Modus an einem Breakpoint, so hat man für den weiteren Verlauf verschiedene Optionen:

- Das Programm normal weiterlaufen lassen.
- Nur einen Schritt ausführen.
  - Sofern im nächsten Schritt eine Funktion ausgewertet wird, so kann man in diese hereinspringen.
  - Ist man bereits in einer Funktion, so kann man aus dieser herausspringen.
- Das Programm bis zum Mauscursor ausführen.



Performance

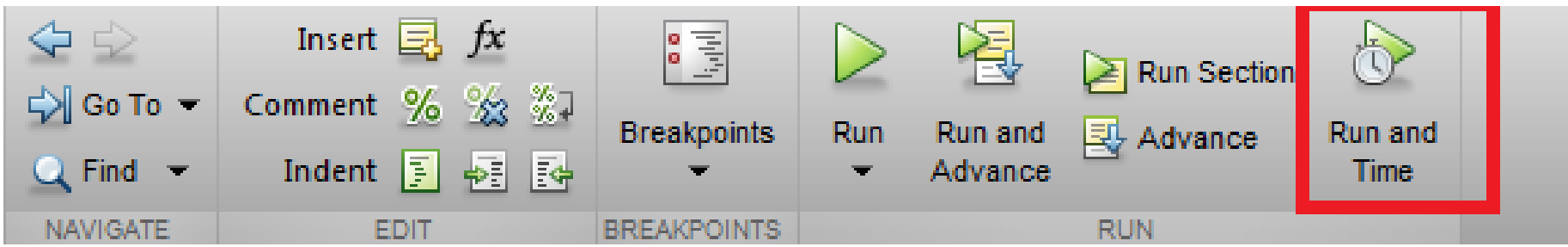
Speicherverbrauch von Variablen und Matlab-Profiler

Zunächst ist es wichtig ein korrektes und gut strukturiertes Programm zu schreiben. Man kümmert sich erst ganz zum Schluss um die Performance.

Ausnahme: Unzulänglichkeiten, wie z.B. fehlende Speicherreservierung, können es u.U. schwierig machen das Programm sinnvoll zu testen.

Tipps

- Mit `whos` kann man sich von Variablen im aktuellen Workspace den Typ und den benötigten Speicher ausgeben lassen. Dies ist im Debugmodus auch im Workspace-Bereich der GUI möglich.
- Mit den Befehlen `tic` und `toc` (im Code verteilt) bekommt man schnell einen Überblick über die Laufzeiten einzelner Bereiche.
- Vorsicht ist beim Vergleich von Zeitmessungen unter 1 Sekunden geboten. Man sollte dann ein Programm mehrfach ausführen und den Mittelwert bilden, denn andere Programme und das System können die Zeitmessungen verfälschen.
- Grafische Ausgaben sollten bei Zeitmessungen deaktivert werden.
- Matlab hat zudem einen *Profiler* eingebaut, mit dem Bottlenecks meist schnell aufgespürt werden können.



Matlab-Profiler

fehlende Initialisierung

```
1  clear;clc
2
3  m = 2000;
4
5
6  for i = 1:m
7      for j = 1:m
8          A(i,j) = i*j;
9      end
10 end
```

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
8	`A(i,j) = i*j;`	4000000	3.737 s	95.7%	
9	`end`	4000000	0.161 s	4.1%	
1	`clear;clc`	1	0.003 s	0.1%	
7	`for j = 1:m`	2000	0.001 s	0.0%	
10	`end`	2000	0.001 s	0.0%	
All other lines			0.000 s	0.0%	
Totals			3.904 s	100%	

mit Initialisierung

```
1  clear;clc
2
3  m = 2000;
4
5  A = zeros(m,m);
6  for i = 1:m
7      for j = 1:m
8          A(i,j) = i*j;
9      end
10 end
```

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
8	`A(i,j) = i*j;`	4000000	0.210 s	56.2%	
9	`end`	4000000	0.157 s	42.1%	
1	`clear;clc`	1	0.004 s	1.1%	
5	`A = zeros(m,m);`	1	0.001 s	0.2%	
7	`for j = 1:m`	2000	0.001 s	0.2%	
All other lines			0.001 s	0.3%	
Totals			0.374 s	100%	