Programmieren in Fortran

Burkhard Bunk

6.3.2013

1 Fortran

Fortran ist die älteste höhere Programmiersprache für numerische Anwendungen: die erste Version entstand 1954 bei IBM. Schon frühzeitig wurde ein ANSI-Standard formuliert, wichtige Stufen waren Fortran 66 und 77. Die letzte große Revision hat zu Fortran 90 geführt, seitdem hat es nur noch kleinere Modifikationen gegeben.
Die Sprache ist gut lesbar und auf Probleme der numerischen Mathematik zugeschnitten. Als Einführung wird im folgenden ein Beispielprogramm Schritt für Schritt besprochen. Eine systematische Darstellung von Fortran 90 findet sich in [1], es gibt auch eine Reihe von Einführungen im Internet [2].

2 Beispielprogramm

Das folgende Programm steht in einer Datei `wurzeln.f90`, die von

`http://poolinfo.physik.hu-berlin.de`

heruntergeladen werden kann. Es berechnet Quadrat– und Kubikwurzeln mit Hilfe der Iterationen

$$\begin{aligned} x_{n+1} &= \frac{1}{2}\left(x_n + \frac{z}{x_n}\right) \to \sqrt{z} \\ y_{n+1} &= \frac{1}{3}\left(2y_n + \frac{z}{y_n^2}\right) \to z^{1/3} \end{aligned}$$

2.1 Listing

```
1 !                                                    B Bunk 4/2001
2 ! Tabelle von Quadrat- und Kubikwurzeln berechnen    rev    3/2012
3 !     und in File schreiben
4
5 module global
6
7   real(4), parameter            :: tol = 1.e-6    ! Fehlertoleranz
```

```
 8
 9 end module global
10
11 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
12
13 program wurzeln
14
15   integer, parameter                  :: nmax = 100
16   real(4), dimension(nmax,3)          :: tabelle
17
18   print*,'x0, dx, Anzahl?'            ! Daten eingeben und pruefen
19   read*, x0, dx, n
20   if (n > nmax) then
21      print*,'Fehler: Anzahl zu gross fuer Tabelle'
22      stop
23   endif
24
25   do i=1,n          ! Schleife ueber Tabellenzeilen
26
27      x = x0 + (i-1)*dx
28
29      tabelle(i,1) = x        ! Tabellenzeile berechnen
30      tabelle(i,2) = w2(x)    ! Funktionsaufruf
31
32      call sub3(x,w3)         ! Aufruf von Subroutine
33      tabelle(i,3) = w3
34
35      print*, tabelle(i,1:3)  ! Kontrollausdruck
36
37   enddo             ! Ende Tabellenzeilen
38
39   open(10,file='wurzeln.tab')         ! File oeffnen
40   do i=1,n
41      write(10,*) tabelle(i,1:3)       ! Zeile Schreiben
42   enddo
43   close(10)                           ! File schliessen
44   stop
45 end program wurzeln
46
47 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
48
49 function w2(z)
50
51   ! w2 <- sqrt(z) berechnen
52
53   use global
54
```

```
55   if (z < 0.) then                    ! Argument pruefen
56      print*,'Fehler in w2: negatives Argument'
57      stop
58   endif
59
60   w2 = 1.                                ! Startwert
61   do
62      w2alt = w2
63      w2 = .5 * (w2 + z/w2)               ! Iteration
64      if ( abs(w2-w2alt) <= tol ) exit    ! Konvergenztest
65   enddo
66 end function w2
67
68 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
69
70 subroutine sub3(z,w3)
71
72   ! w3 <- z**(1/3) berechnen
73
74   use global
75
76   if (z < 0.) then                    ! Argument pruefen
77      print*,'Fehler in w3: negatives Argument'
78      stop
79   endif
80
81   w3 = 1.                                ! Startwert
82   do
83      w3alt = w3
84      w3 = (2*w3 + z/w3**2)/3             ! Iteration
85      if ( abs(w3-w3alt) <= tol ) exit    ! Konvergenztest
86   enddo
87 end subroutine sub3
```

3 Zeilenkommentar

3.1 Struktur

1-3: Kommentare (jeweils von ! bis Zeilenende)
5-9: Modul `global`
13-45: Hauptprogramm `wurzeln`
49-66: Funktion `w2` für Quadratwurzel
70-87: Routine `sub3` für Kubikwurzel

3.2 Modul `global`

5 `module` - 9 `end` : Modul, definiert globalen Parameter `tol`

3.3 Hauptprogramm `wurzeln`

13 `program` - 45 `end` : Begrenzung des Hauptprogramms
15: Deklaration und Definition einer ganzzahligen Konstanten
16: Deklaration einer Matrix (für die Tabellierung der Werte) mit `nmax` Zeilen und drei Spalten
18: Bildschirmausgabe der Frage nach Parametern (* bedeutet "automatisches" Format)
19: Einlesen der Paramter von der Tastatureingabe. Es sollen n Werte tabelliert werden, angefangen mit x_0 und in Schritten von dx
20-23: Programm wird (mit Fehlermeldung) gestoppt, wenn Tabelle zu klein
25-37: Schleife über x-Werte, d.h. Tabellenzeilen
27-29: aktuellen x-Wert berechnen und in die erste Tabellenspalte schreiben
30: Wurzelfunktion (s.u.) aufrufen, Resultat in die zweite Spalte
32-33: Routine für Kubikwurzel aufrufen, Resultat `w3` in die dritte Spalte
35: aktuelle Tabellenzeile (Spalten `1:3`) auf den Schirm schreiben
39: Datei öffnen als Unit 10
40-42: Tabelle zeilenweise in Datei schreiben
43: Datei schließen
44: Programm stoppen

3.4 Funktion `w2`

49: Kopfzeile der Funktion `w2` mit Dummy–Argument `z`; der Funktionsname dient als Variable, um das Resultat zurückzugeben
53: Modul `global` einbinden (für Parameter `tol`)
55-58: Fehlerbehandlung
60: Iteration initialisieren
61-65: Iterationsschleife
64: Schleife beenden, wenn Genauigkeit erreicht ist
66: implizites `return` am Ende des Unterprogramms

3.5 Routine `sub3`

70: Kopfzeile der Routine mit Dummy–Argumenten `z` als Eingabe und `w3` zur Rückgabe des Resultats; der Name der Routine dient nur zum Aufruf
74: Module `global` einbinden (für Parameter `tol`)
76-79: Fehlerbehandlung
81: Iteration initialisieren
82-86: Iterationsschleife
85: Schleife beenden, wenn Genauigkeit erreicht ist
87: implizites `return` am Ende des Unterprogramms

4 Übersetzen und Aufrufen

Das Quellprogramm muss mit einem Fortran–Compiler übersetzt werden.
Die GNU–Compiler–Suite (gcc etc, ab Version 4) stellt `gfortran` zur Verfügung:

```
unix> gfortran wurzeln.f90
```

Alternativ kann man auch die Compiler von Intel oder Portland benutzen:

```
      unix> ifort wurzeln.f90
oder
      unix> pgfortran wurzeln.f90
```

In allen Fällen entsteht eine ausführbare Binärdatei mit dem Standardnamen `a.out`, deren Aufruf das Programm ablaufen lässt

```
      unix> a.out
```

Meist gibt man der Binärdatei gleich beim Übersetzen einen besseren Namen, z.B. einfach den Basisnamen `wurzeln` ohne Erweiterung:

```
      unix> gfortran -o wurzeln wurzeln.f90
      unix> wurzeln
```

5 Einige Punkte im Überblick

5.1 Quellformat

Hier wird das mit Fortran90 eingeführte Quellformat (*free source format*) als selbstverständlich vorausgesetzt, die alte, noch an Lochkarten orientierte Schreibweise (*fixed source format*) wollen wir vergessen.
Grundregel: jede Zeile enthält *eine* Anweisung.
Will man mehrere Anweisungen hintereinander in eine Zeile schreiben, dann trennt man sie durch ; (Semikolon) voneinander. Soll eine Anweisung in der nächsten Zeile fortgesetzt werden, dann setzt man ein `&` ans Zeilenende.
Zeilen können bis zu 132 Zeichen lang sein, es sind bis zu 39 Fortsetzungszeilen erlaubt.
Von einem `!` an ignoriert der Compiler alles bis zum Zeilenende, so kann man Kommentare einfügen. Leerzeilen sind auch erlaubt.
Namen bestehen aus Buchstaben, Ziffern und '_' (Unterstrich), beginnend mit einem Buchstaben. Laut Standard werden Groß- und Kleinbuchstaben in Variablen und Schlüsselworten *nicht* unterschieden (ein Erbe aus der Lochkartenzeit). Die Unterscheidung kann man zwar durch Compileroptionen oft erzwingen, aber darauf ist kein Verlass. Am besten: man schreibt alles klein (Zeichenketten und Kommentare ausgenommen).

5.2 Datentypen

Die wichtigsten Datentypen:

```
      integer           = integer(4)
      integer(4)        4 Bytes, Bereich +/- 2^31 = 2x10^9 (ca)
      integer(8)        8 Bytes, nur auf 64-Bit-Systemen

      real              = real(4)
      double precision  = real(8)
      real(4)           4 Bytes, ca 7 Dezimalst., Bereich 10^(+/-38)
      real(8)           8 Bytes, ca 16 Dezimalst., Bereich 10^(+/-308)
```

```
complex           = complex(4)
complex(4)        2 x real(4)
complex(8)        2 x real(8)

character         Zeichen, 1 Byte
character(n)      Zeichenkette, n Bytes
```

Variablen (und benannte Konstanten) werden i.a. im Programmkopf (oberhalb der ausführbaren Anweisungen) deklariert und vielleicht auch initialisiert.
Die (alte) **implizite Typdeklaration** (*implicit typing*) von Fortran nimmt, wenn nichts anderes deklariert ist, folgendes an:

- `real` für Namen, die mit a-h, o-z beginnen;
- `integer` für Namen, die mit i-n beginnen.

Diese Automatik spart Schreibarbeit, macht aber aus einem Tippfehler womöglich eine neue Variable. Deshalb kann man dieses Verhalten *abschalten* mit der Deklaration

```
implicit none
```

im Kopf *jedes* Programmteils. Es gibt auch Compilerschalter, die das pauschal erledigen:

```
gfortran -fimplicit-none ...
ifort -implicitnone ...
pgf90 -Mdclchk ...
```

Bei **arithmetischen Operationen** zwischen verschiedenen Zahlentypen wird zunächst auf den höheren (umfassenderen) Typ konvertiert und dann gerechnet, das führt i.a. zum erwarteten Ergebnis. Überraschend ist eher, wenn es *nicht* geschieht: Die Division zweier `integer`–Zahlen wird ganzzahlig ausgeführt, ein Rest fällt einfach weg. So wird z.B. `1/3 -> 0`, das ist selten beabsichtigt – eine böse Falle.

5.3 Mathematischen Funktionen

Fortran kennt ohne weiteres eine Reihe von Funktionen, u.a.

```
abs  sqrt  exp  log  log10  sin  cos  tan  asin  acos  atan
```

Sie sind in der Regel definiert für alle Gleitkomma–Typen (einschließlich der komplexen), ihre Auswertung richtet sich nach dem Typ des Arguments. Das entspricht meistens der Erwartung, man muss aber bei konstanten Argumenten aufpassen: `sqrt(2.)` ist nur einfach genau, erst `sqrt(2.d0)` gibt doppelte Genauigkeit.
Für die Potenz gibt es das Rechenzeichen $**$ (*nicht* $\wedge$).

5.4 Kontrollstrukturen

Schleife mit unbegrenzter Zahl von Durchläufen und Abbruchbedingung:

```
do
      Anweisungen

      if ( Bedingung ) exit

      Anweisungen
enddo
```

Schleife mit Kontrollvariable:

```
do i=2,6          ! durchlaeuft i=2,3,4,5,6
      Anweisungen
enddo
```

Andere Schrittweiten:

```
do i=3,10,2       ! durchlaeuft i=3,5,7,9 (Schrittweite 2)
do i=8,3,-2       ! durchlaeuft i=8,6,4   (rueckwaerts)
do i=8,3          ! leer
```

Die letzte Schleife wird überhaupt nicht ausgeführt (abweisendes `do`), weil schon der Anfangswert über dem Endwert liegt (analog bei Rückwärtsschleifen).
Die Kontrollvariable (hier `i`) soll/muss vom Typ `integer` sein – bei `real` wäre der "letzte" Durchlauf durch Rundungsfehler unsicher.
Einfaches `if` mit *einer* Anweisung:

```
if ( Bedingung ) Anweisung
```

`if` mit Block von Anweisungen:

```
if ( Bedingung ) then
      Anweisungen
endif
```

Allgemeine Form:

```
if ( Bedingung1 ) then
      Anweisungen
else if ( Bedingung2 ) then
      Anweisungen
else if ( Bedingung3 ) then
      Anweisungen
...
else
      Anweisungen
endif
```

Die Bedingungen werden nacheinander geprüft, bis eine zutrifft. Der `else`–Block wird durchlaufen, wenn alle `if`–Zweige negativ getestet wurden.
Vergleichsoperatoren für Zahlen (und Zeichenketten):

```
==   /=   >   <   >=   <=
```

Hier noch eine Übersicht über Befehle, die etwas beenden:

Befehl..	beendet..
cycle	Schleifendurchlauf
exit	Schleife
return	Unterprogramm
stop	Programm

Das `end` in einem Unterprogramm impliziert ein `return`, das `end` im Hauptprogramm ein `stop`.

5.5 Felder

Beispiele für die Deklaration von Feldern:

```
integer, dimension(6)                :: v
real(8), dimension(0:3,3)            :: M
real(8), dimension(:,:), allocatable :: A
```

`v` bezeichnet also ein einfach indiziertes Feld (Liste, Vektor) mit (Integer–)Komponenten `v(1)..v(6)`.
`M` ist ein real(8)-Feld mit zwei Indizes (Tabelle, Matrix) und Elementen `M(i,j), i=0..3, j=1..3`. Mit der Doppelpunkt–Notation kann man also Indizierungen vorgeben, die nicht bei 1 beginnen. Dieselbe Notation beim Zugriff auf Feldelemente spezifiziert Teilfelder, z.B.

```
v(2:4)      => v(2) v(3) v(4)       Feld mit 3 Elementen

M(1:2,1:2)  => M(1,1) M(1,2)        2x2 Untermatrix
               M(2,1) M(2,2)

M(1,:)      => M(1,1) M(1,2) M(1,3) Zeile aus M
```

Wie man am letzten Beispiel sieht, bezeichnet ":" den gesamten Indexbereich (laut Deklaration), hier also alle Spalten, so dass eine vollständige Zeile angesprochen ist. Da man diese Teilfelder auch wieder passend zuweisen kann, ist der Umgang mit Untermatrizen denkbar einfach.
Der Umgang mit Vektoren und Matrizen wird durch die kompakte Notation sehr erleichtert: ähnlich wie in Matlab kann man ganze Felder addieren, mit Skalaren multiplizieren, einer Feldvariablen zuweisen usw. Die eingebauten mathematischen Funktionen wirken in der Regel elementweise, es gibt aber auch besondere Funktionen für Summation über Feldelemente, Extremwerte etc. Im Gegensatz zu Matlab wirken auch die Rechenoperationen `* / **` *zwischen* Feldern *elementweise*. Für Vektor– und Matrixoperationen gibt es besondere Funktionen wie `dot_product`, `matmul` usw.

Das letzte Beispiel oben deklariert ein Feld `A(i,j)` von noch nicht festgelegter Größe. Den Speicherbereich kann man zur Laufzeit anlegen und wieder freigeben (dynamische Speicherverwaltung), z.B.

```
allocate(A(m,n))
...
deallocate(A)
```

Danach kann das Feld neu angelegt werden usw.

Literatur

[1] M. Metcalf, J. Reid, Fortran 90 Explained, Oxford University Press, New York 1990

[2] `http://wwwasdoc.web.cern.ch/wwwasdoc/f90.html`
`http://www.rz.uni-bayreuth.de/lehre/fortran90/vorlesung/index.html`
`http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html`