B Notizen zu Fortran 90/95

B.1 Numerische Datentypen

Für numerische Rechnungen bevorzugt man

- `integer` alias `integer(4)`
- `real(8)` alias `double precision`
- `complex(8)` (d.h. `real(8)` für Real– und Imaginärteil)

Die (alte) implizite Typdeklaration von Fortran nimmt automatisch an:

- `real` alias `real(4)` für Namen, die mit `a-h, o-z` beginnen,
- `integer` für Namen, die mit `i-n` beginnen.

Diese Automatik kann man mit

```
implicit none
```

(ganz oben in *jedem* Programmteil) abschalten, es gibt dafür aber auch Compilerschalter:

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

Wer schreibfaul und/oder ganz mutig ist, kann die implizite Typdeklaration aber auch auf `real(8)` umschalten mit

```
implicit real(8) (a-h,o-z)
```

Schließlich kann man die Standardgröße von `real` und `complex` auch von der Kommandozeile aus auf 8 Bytes umstellen:

```
gfortran -fdefault-real-8 ..
ifort -r8 ..
pgf90 -r8 ..
```

Das betrifft Variablen, numerische und symbolische Konstanten und die Resultatwerte der Funktionen `real(), float(), cmplx()`.

B.2 Komplexe Größen

Beispiele für Deklaration und Anweisungen:

```
real(8)     :: a, b, r
complex(8)  :: z, w

z = (2., 3.)                  ! z = 2 + 3i  (Konstante)
z = cmplx(a, b, kind(a))      ! z = a + b i, als complex(8)

a = real(z)       ! Realteil, als real(8)
b = aimag(z)      ! Imaginaerteil, als real(8)
```

```
r = abs(z)          ! Betrag, als real(8)
w = conjg(z)        ! konjugiert komplex, als complex(8)

w = 2 * z**2        ! komplexe Arithmetik

w = sqrt(z)         ! Wurzel, als complex(8)
```

Die arithmetischen Operationen werden, wenn komplexe Größen beteiligt sind, automatisch komplex ausgeführt. Dasselbe gilt für die Fortran–eigenen Funktionen (z.B. `sqrt, log, exp`), wenn sie mit komplexen Argumenten aufgerufen werden. Dabei bleibt auch die Genauigkeit erhalten. Ausnahme ist der Zusammenbau von reellen Variablen zu einer komplexen Zahl mit `cmplx`: dort muss man die doppelte Genauigkeit mit dem dritten (optionalen) `kind`–Argument erzwingen, sonst verliert man durch `complex` alias `complex(4)` im Zwischenschritt an Genauigkeit.

B.3 Felder

Beispiele für die Deklaration von Feldern:

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

`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.
Die arithmetischen Operationen (`+ - * / **`) wirken elementweise (bei passenden Feldern). Wenn ein Operand ein Skalar ist, wird er auf alle Feldelemente angewandt. Viele Fortran–Funktionen wirken ebenfalls elementweise, z.B. `abs, sqrt, exp, sin`.
Es gibt auch eine Reihe von Funktionen speziell für Felder, u.a.

```
sum(array)        prod(array)    ! Summe, Produkt der Elemente

minval(array)     minloc(array)  ! Wert, Position des Minimums
maxval(array)     maxloc(array)  ! Wert, Position des Maximums
```

```
dot_product(vec1,vec2)          ! Skalarprodukt
matmul(mat1,mat2)               ! Matrixprodukte
matmul(vec,mat)   matmul(mat,vec)
transpose(mat)                  ! transponierte Matrix
```

Die folgenden Beispiele zeigen, wie der Umgang mit Vektoren und Matrizen durch die kompakte Notation erleichtert wird (ähnlich wie in Matlab):

```
real(8) :: s
real(8), dimension(10)    :: u, v, w
real(8), dimension(10,10) :: mat, mat1, mat2

w = u + s*v       ! Vektor + Skalar x Vektor

s = sum(u**2)     ! Norm-Quadrat (Summe der Quadrate)

s = dot_product(u,v)    ! Skalarprodukt
v = matmul(mat,u)       ! Matrix x Vektor
mat = matmul(mat1,mat2) ! Matrix x Matrix
```

Mit komplexwertigen Feldern geht alles analog. Zu beachten ist, dass die Funktion `dot_product(u,v)` bei komplexen Vektoren `u,v` tatsächlich $(u^\dagger v)$ berechnet.

B.4 Dynamischer Speicher

B.4.1 Allocate

Allgemein kann man ein Feld mit dem Attribut `allocatable` deklarieren und dann zur Laufzeit mit `allocate` den Speicher anfordern. So legt man z.B. eine Matrix dynamisch an:

```
real(8), dimension(:,:), allocatable :: mat
...
m = 5
n = 7
if (allocated(mat)) deallocate(mat)       ! vorsichtshalber
allocate(mat(m,n))
```

B.4.2 Automatische Arrays

In Unterprogrammen (`subroutine` oder `function`) gibt es eine einfachere Möglichkeit, Arbeitsvektoren dynamisch anzulegen: man deklariert sie mit variabler Größe, die zur Zeit des Aufrufs bekannt sein muss, z.B.

```
subroutine sub(n, ...)

real(8), dimension(n) :: work
...
```

Hier stammt die Variable `n` aus der Aufrufliste (und muss als Input gesetzt sein), sie kann aber auch als globale Variable definiert sein.
Einen automatischen Array kann man bei der Deklaration *nicht* auch noch initialisieren!

B.5 Globale Variablen

Prinzipiell sind Programmteile (`program, subroutine, function, module`) vollkommen unabhängig voneinander, auch wenn sie in derselben Datei stehen. Wenn verschiedene Programmteile auf gemeinsame Daten zugreifen sollen, deklariert man sie in einem Modul, z.B.

```
module common
      integer, parameter            :: maxdim = 3, maxvol = 100000

      integer ndim, lsize(maxdim), nvol
      integer nn(maxvol,-maxdim:maxdim)
      real(8) mass
end module common
```

und macht sie mit
`use common`
in verschiedenen Programmteilen bekannt. Wenn das Hauptprogramm diesen Modul benutzt, dann ist die Lebensdauer der Daten für die gesamte Laufzeit gesichert, andernfalls kann man das durch das Attribut `save` in der Deklaration erzwingen. Eine Initialisierung in der Deklaration impliziert übrigens das `save`–Attribut.

B.6 Funktionsname als Argument

Der tatsächlich im *Aufruf* stehende Programmname muss als `external` deklariert werden:

```
      external laplace
      ...
      call cg(nvol, phi, b, laplace, relerr, maxiter, 0)
```

Bei der *Definition* des Unterprogramms erkennt der Compiler die Routine an ihrem Aufruf:

```
      subroutine cg(n, x, b, matvec, relerr, maxiter, istart)
      ...
      call matvec(n, x, s)
```

B.7 Compilieren und Linken

Im PC–Pool sind folgende Compiler für Fortran 90/95 installiert:

```
gfortran - GNU Fortran (seit gcc-4.0)
ifort    - Intel Fortran 77/90/95
pgf90    - Portland Group Fortran 90/95
```

Im einfachsten Fall compiliert man eine Quelldatei `prog.f90` mit

```
gfortran prog.f90
```

und erzeugt so (wenn alles gut geht) ein lauffähiges Binärfile `a.out`
Tatsächlich besteht die Erzeugung des Binärfiles aus mehreren Schritten, die man auch einzeln aufrufen kann:

```
gfortran -c prog.f90          # compile prog.f90 -> prog.o
gfortran -o prog.x prog.o     # link prog.o -> prog.x
```

Das wird deutlicher, wenn man den Quellcode in mehreren Dateien `prog1.f90` `prog2.f90 ..` vorliegen hat:

```
gfortran -o prog.x *.f90
```

erzeugt wiederum ein lauffähiges Binärfile `prog.x` in einem Schritt, während

```
gfortran -c *.f90
gfortran -o prog.x *.o
```

erstmal jedes Quellfile `progn.f90` zu einem Objectfile `progn.o` compiliert und diese dann im zweiten Schritt zum Binärfile `prog.x` verlinkt.
Wenn in einem Quellfile (z.B. `prog.f90`) **Module** definiert werden (z.B. `global, data`), dann entsteht beim Compilieren neben `prog.o` zu jedem Modul eine vorkompilierte Datei mit Endung `mod`, z.B. `global.mod, data.mod`. Der *Compiler* braucht diese Moduldatei, wenn ein Modul in einem anderen Programmteil mit `use` eingebunden wird. Falls das in derselben Datei geschieht, muss die Definition des Moduls *vor* dem ersten `use` stehen.
Die Optionen sind jeweils entweder dem Compilier- der dem Linkschritt zuzuordnen.
Compileroptionen sind u.a.

```
-O              Optimierung einschalten (GNU!)
-Wall           alle Warnungen anzeigen (Fehlersuche!)
-c              nur compilieren, nicht linken
```

Einige Linkeroptionen:

```
-o <name>       Binaerfile umbenennen (statt a.out)
```

Die bedingte Compilierung eines über mehrere Quelldateien `*.f90` verteilten Projekts kann man mit Hilfe eines Makefiles organisieren, wie unten beschrieben. Hier entfallen die Header–Dateien (`*.h` in C), man hat aber auf Abhängigkeiten zu achten, die durch Bezug auf Module (`use ..`) entstehen. So begründet ein

```
use common_data
```

eine Abhängigkeit von `common.o`, wenn der `module common_data` in der Quelldatei `common.f90` definiert ist.

B.8 Fehlersuche

Fortran–Compiler unterstützen die Fehlersuche durch einige nützliche Optionen.

B.8.1 Undeklarierte Variablen

Wenn man `implicit none` *nicht* gesetzt hat, werden undeklarierte Variablen je nach Name als `real` oder `integer` interpretiert. Um Tippfehler aufzudecken, könnte man überall `implicit none` eintragen und sehen, was passiert. Der Compiler hat eine Option, um das global einzuschalten (ohne das Programm zu editieren.)

B.8.2 Nicht initialisierte Variablen

Der Zugriff auf nicht gesetzte Variablen führt zu obskurem Verhalten, denn es wird der zufällig vorgefundene Speicherinhalt als Wert genommen. Der Compiler kann eine entsprechende Warnung geben.

B.8.3 Überschreiten der Feldgrenzen

Wenn ein Index die deklarierten Feldgrenzen überschreitet, wird auf einen willkürlichen benachbarten Speicherplatz zugegriffen (oder es gibt einen *segmentation fault*). Der Compiler kann Kode erzeugen, der die Einhaltung der deklarierten Feldgrenzen zur Laufzeit überwacht.

B.8.4 Übersicht

	gfortran	ifort	pgf90
undeklarierte Variablen	-fimplicit-none	-implicitnone	-Mdclchk
uninitialisierte Variablen	-Wuninitialized -Wall	-check uninit -check all	
Feldgrenzen	-fbounds-check	-check bounds -check all	-Mbounds

B.9 Einhalten des Standards

Der Compiler kann **warnen**, wenn das Quellprogramm nicht–standardkonforme Elemente enthält:

```
gfortran -std=f95 -pedantic ..
ifort -stand f95 ..
pgf90 -Mstandard ..
```

B.10 Einige spezielle Aufrufe

B.10.1 Uhrzeit

Fortran 90 definiert einen Zugang zur Uhrzeit:

```
integer(8) :: count, count_rate, count_max
real :: clocktime

call system_clock(count, count_rate, count_max)

clocktime = real(count)/count_rate     ! in sec
```

Die Argumente von `system_clock` sind optional, das Resultat `clocktime` ist die (Uhr–)Zeit in Sekunden, mit willkürlichem Nullpunkt (z.B. Sekunden seit 1.1.1970 00:00 UTC) und einer typischen Genauigkeit im Bereich von Mikrosekunden oder darunter.

B.10.2 CPU-Zeit

Fortran 95 erlaubt das Auslesen der CPU-Zeit (in Sekunden, mit willkürlichem Nullpunkt), ggf. summiert über alle *threads* des Prozesses:

```
real :: cputime

call cpu_time(cputime)
```

Typische Auflösung: 1..10 microsec.

B.10.3 Kommandozeile

Seit Fortran 2003 unterstützt der Standard das Einlesen von Argumenten aus der Kommandozeile:

```
integer :: num_args, i
character(50) :: arg

num_args = command_argument_count()
call get_command_argument(i, arg)   ! i = 0..num_args
```

Für `i=0` bekommt man den Namen des aufgerufenen Programms, mit `i=1..num_args` die Argumente als Zeichenketten. Wenn dort tatsächlich Zahlen eingeben wurden, kann man sie daraus formatiert lesen, z.B. eine `integer n`:

```
read(arg,*) n
```