

EDV2

Wissenschaftliches Programmieren

(Crashkurs: Fortran)

SS 2022

Organisatorisches

- Arbeitsgruppen (max. 2 Personen / Gruppe (=PC))
- Login: `edv2di01`, `edv2di02` ... `edv2mi01`, `edv2mi02` ...
- Passwort: an der Tafel. NACH der Gruppenzuweisung ÄNDERN mit `passwd`
- home Verzeichnis z.B. `/home/EDV2/edv2di01/` • Unterlagen: `/home/EDV2/Unterlagen/01Ue-2022`
- Unterverzeichnisse für Übungen z.B. `/home/EDV2/edv2di01/01Ue-2021-03-02/`
Übungen 01-04: wissenschaftl. Prog.; 05-10: NMS. Verzeichnisse NICHT ändern!
- Bis zur Folgewoche müssen in den betreffenden Verzeichnissen Ihre Programme (kompilierbar und lauffähig) abgelegt werden, plus eine Textdatei `PROTOKOLL.txt` (Anleitung siehe Übungsblatt)
- Kommentare der Tutoren dann in `/home/EDV2/edv2di01/01Ue-2021-03-02/TUTOREN/KOMMENTARE.txt`

Programmieren: Allgemeines

1. Verstehen des Problems und seiner Lösungsmöglichkeiten
2. Planung
 1. Suchen nach vorhandenen Programmen, Bibliotheken, Algorithmen
 2. Auswahl der Programmiersprache
 3. Entwicklung der eigenen Daten- und Programmstruktur
 1. Modularer Aufbau, Zerlegung in Teilprobleme oder –aufgaben
 2. Lösungsweg skizzieren: Flussdiagramm
3. Programmieren
4. Testen
5. **Verwenden**
6. Warten und Pflege (Erweiterungen, Dokumentation)

Programmieren: Allgemeines

- Problem verstehen → Programmstruktur überlegen → programmieren
- prozedural / Zerlegung der Aufgabe in Teilprobleme
- aus der Programmstruktur sollte transparent sein, was geschieht
- ausführlich kommentieren und dokumentieren
- Schleifen etc einrücken (bei Python zwingend)

FORTRAN „Formula Translator“

- *prozedural* (=Zerlegung in Teilprobleme)
seit 2003 Objektorientierung möglich
- *kompilierte* Sprache: ausführbare Programmdatei
(Skripte (z.B. Python) *interpretiert* Code jedes Mal neu)
- seit Jahrzehnten Programmiersprache in der Physik
→ es existieren umfangreiche Programmbibliotheken
→ viele Programme existieren in FORTRAN.

Diese müssen verstanden, adaptiert und erweitert werden

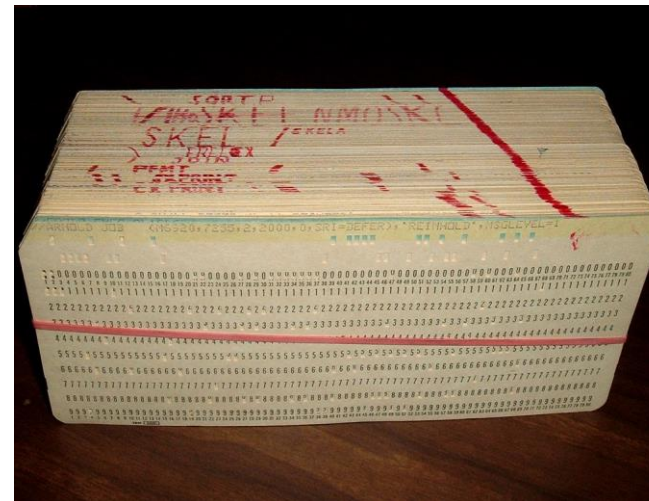
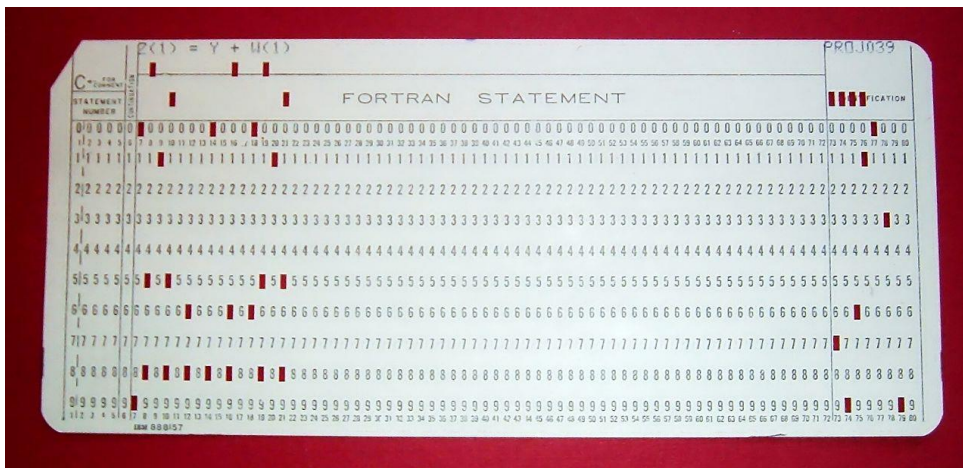
Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

FORTRAN once upon a time

```
C---- THIS PROGRAM READS INPUT FROM THE CARD READER,  
C---- 3 INTEGERS IN EACH CARD, CALCULATE AND OUTPUT  
C---- THE SUM OF THEM.  
100 READ(5,10) I1, I2, I3  
10 FORMAT(3I5)  
IF (I1.EQ.0 .AND. I2.EQ.0 .AND. I3.EQ.0) GOTO 200  
ISUM = I1 + I2 + I3  
WRITE(6,20) I1, I2, I3, ISUM  
20 FORMAT(7HSUM OF , I5, 2H, , I5, 5H AND , I5,  
* 4H IS , I6)  
GOTO 100  
200 STOP  
END
```

- Spalten 7-72 enthalten die Anweisungen
- Spalte 1: „C“ oder „!“ kennzeichnet eine Kommentarzeile
- Spalten 2-5: Anweisungsnummern (Labels, Sprungmarken)
- Spalte 6: markiert Fortsetzungszeile
- Spalten 73-80: Lochkartennummern

1 Programmzeile → 1 Lochkarte



Programmaufbau FORTRAN

program1.f90

```
! Programm-Kopf
program hello
implicit none ! optional, aber dringend empfohlen
```

```
! Variablen Definitionen
integer :: n,i ! oder: integer n,i
```

```
! Anweisungen
n=0
do i=1,10
    n=n+i
enddo
write(*,*) "Ergebnis: ", n
```

```
end
```

- Dateiendung: .f90
für alle Versionen ab FORTRAN90
.f oder .f77: für altes Format
- Gross- und Kleinschreibung
wird NICHT unterschieden!
- ohne `implicit none`:
i...n: ganze Zahlen
o...z: nicht-ganzzahlig

Variablentypen in FORTRAN

FORTRAN hat 5 eingebaute Datentypen:

- `integer i, j` ganze Zahlen (mit Vorzeichen)
- `real x` nicht-ganzzahlige Zahl (mit Vorzeichen)
- `complex z` Paar $z=(x,y)$ mit x,y real: $z = x + i y$
- `character ch` ein Zeichen
- `logical isOkay` Boolean (true oder false)

```
i = -1
x = 3.1415
z = (x, 0.5)
ch = 'A'
isOkay = .true.
isOkay = .false.
```

Datentypen haben verschiedene Varianten

Typ	Maximaler Wert <code>huge(i)</code>	signifikante Stellen <code>precision(x)</code>
• <code>integer i</code>	2147483647 (4 Byte = 32 Bit)	
• <code>integer(kind=8) i</code>	9223372036854775807 (8 Byte)	
• <code>real x</code>	3.40282347E+38	6 ("1=1.0000004")
• <code>real(kind=8) x</code> <code>double precision</code>	1.7976931348623157E+308	15

Variablentypen und Operationen

ACHTUNG: (implizite) Konversion von Variablentypen

```
integer :: i,j  
real :: r  
double precision :: rdp
```

```
j=3  
i=1  
r=1.0  
rdp=1.d0
```

```
write(*,*) i    / j    → 1 / 3    = 0  
write(*,*) r    / j    → 1. / 3    = 0.3333333343  
write(*,*) rdp  / j    → 1.d0 / 3  = 0.33333333333333333331
```

*	Multiplikation	x * y
/	Division	x / y
+	Addition	i + j
-	Subtraktion	a - b
**	Exponentierung	x**2

explizite Konversionen

```
i=3  
r = real(i)  
rdp = real(i,kind=8) → real(i,kind=8) / 3 = 0.33333333333333333331  
i = int(r) ! rundet AB: "5.4 = 5.6 = 5"
```

Bedingte Anweisungen (if then else)

```
if (x.eq.0.) y = 1.          ! falls nur eine Anweisung im if-Block
```

```
if (x.eq.0.) then          ! mehrere Anweisungen
  y = 1.
  z = -1.
endif
```

```
if (x.eq.0.) then
  y = 0.
else if (x.lt.0.) then
  y = -1.
else
  y = 1.
end if
```

Vergleichsoperatoren

== **.EQ.** gleich "equal"

/= **.NE.** ungleich "not equal"

< **.LT.** kleiner "less than"

<= **.LE.** kleiner gleich "less than or equal"

> **.GT.** größer "greater than"

>= **.GE.** größer gleich "greater than or equal"

Schleifen

```
do i = 1,3      ! d.h. i=1,2,3
  x = x + real(i)
enddo
```

```
do i = N,1,-2   ! d.h. N, N-2, ..., 1
  ...
enddo
```


```
i=0  ! Variable initialisieren !!
do while (i.le.N)
  ...
  i=i+1
enddo
```

alte Schreibweise (Fortran77):

```
do 10 i = 1,3
  ...
10 continue
```

Felder

unveränderliche “Variable”. z.B. `double precision, parameter :: pi = acos(-1.d0)`



```
integer, parameter :: n=100
integer :: i(10) ! d.h. i(1) ... i(10)

double precision :: x(n) ! x(1)...x(100)

double precision :: M(-2:2,100)

! M(-2,1) ... M(-2,100)
! ...
! M(2,1) ... M(2,100)

character(len=20) ch
```

```
do i = 1,n
    x(i)= real(i,kind=8)
enddo
```

```
integer :: n
double precision, allocatable :: x(:)
```

```
...
n=100
allocate(x(n))
...
deallocate(x)
...
```

Ein- und Ausgabe am Bildschirm

unit Format



```
write(*,*) 'Hello world'
write(*,*) 'x = ', x
write(*,'(1F12.6,1I5)') x, i

write(6,*) x,i ! unit=6 ist der Bildschirm
write(*,'(5F10.6)') (x(i), i=1,5) ! implizite Schleife
```

```
read(*,*) x,y
read(5,*) x,y ! unit=5 ist das Terminal (Tastatur)

read(5,110) x,y
110 format(2F12.6)
```

Ein- und Ausgabe in Dateien

unit Format



```
write(10,*) 'x =', x    ! schreibt in die Datei: fort.10

open(10, file = 'test.dat')
  write(10,*) 'x = ', x
close(10)
```

integer ierr

```
open(10, file='data.dat', status='old', iostat=ierr)
if (ierr.ne.0) STOP 'error opening data.dat'

do
  read(10,*,iostat=ierr)x,y
  if (ierr.ne.0) exit
  ...
enddo
close(10)
```

exit = verlasse Schleife! ansonsten: endlos loop.

Funktionen

```
program neck
implicit none
integer n
double precision r, flaeche, fneck
n=3
r=1.d0
flaeche = fneck(n,r)
write(*,*) flaeche
end
```

```
double precision function fneck(n,r)
implicit none
integer n
double precision r,pi
! Rueckgabewert := Name der Funktion !
pi=acos(-1.d0)
fneck = n * r**2 * sin(pi/n)
return
end
```


Unterprogramme: Subroutinen

```
call einlesen('input.dat',x,n)
...
call ausgabe('output.dat',x,n)
```

```
subroutine einlesen(datei,x,n)
implicit none
character(len=*) datei
integer i,n
double precision x(n)

open(10, file = datei, status='old')
do i=1,n
    read(10,*)x(i)
enddo
close(10)

end subroutine
```

Kompilierung

...im "terminal": CTRL+ALT+T

```
$ gfortran program.f90
```

```
$ ./a.out
```

```
$ gfortran -O3 -o program program.f90
```

```
$ ./program
```

optional → Muss

mehr Warnungen

z.B. vector(n+1) abgefragt
fuer Feld vector(n)

Fehler in Zeile x



```
$ gfortran -Og -Wall -fimplicit-none -fcheck=all -fbacktrace program.f90
```

Kommando in `compile.sh` schreiben, dann: `$ sh compile.sh`

Terminal

öffnen mit: CTRL+ALT+T

shortcuts im Terminal:

“Pfeil nach oben”: zeige letzten ausgeführten Befehl

“tab”: auto-complete, z.B. cd 01Ue “tab” → cd 01Ue-2022...

```
$ pwd    # zeigt derzeitiges Verzeichnis an

$ cd 01Ue-2022-...    # change directory in Unterverzeichnis.

$ ls      # zeigt Inhalt des Verzeichnisses an

$ gedit program.f90 &    # öffnet einen Editor und erstellt
                        (so noch nicht vorhanden) die Datei „program.f90“
                        Das „&“ schickt den Editor in den Hintergrund →
                        man kann das Terminal weiter verwenden

$ gfortran program.f90

$ ./a.out

$ cp /home/EDV2/Unterlagen/01Ue.../Gpsies_austria.csv . #kopiert ins aktuelle Verzeichnis
```