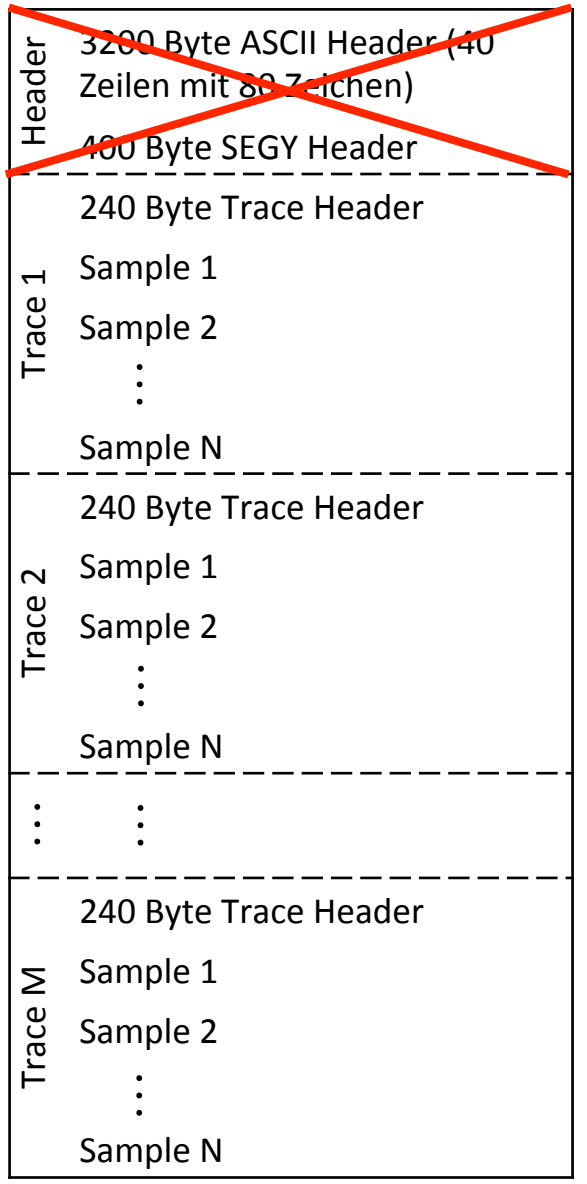
Einführung

Seismic Unix (SU) ist ein Seismik Open-Source Software Paket welches durch das Center of Wave Phenomena (CWP) an der Colorado School of Mines (CSM) bereitgestellt wird.
SU läuft auf allen Unix, Linux, Solaris und Mac OS X Betriebssystemen. Die Befehle für SU werden von der Kommandozeile aufgerufen (ähnlich wie GMT). Dabei können Aus- und Eingabe von Kommandos wie bei GMT mittels | miteinander verbunden werden. Die Befehle können auch wieder über Flags gesteuert werden.
SU verwendet als Datenformat auch das SEGY-Format jedoch OHNE den 3200 Byte ASCII Header und den 400 Byte SEGY Header. Der Zugriff auf die Header erfolgt nicht über die Byte-Position sondern über Kürzel. Hier die Wichtigsten:

Key	Bytes	Beschreibung
tracl	1-4	Spurnummer im gesamten Datensatz
tracr	5-8	Spurnummer innerhalb einer Auslage
fldr	9-12	Schussnummer im Feld (FFID)
tracf	13-16	Kanalnummer
ep	17-20	Schussnummer
offset	37-40	Schuss-Empfänger Offset
gelev	41-44	Empfängerhöhe
selev	45-48	Schusshöhe
scalel	69-70	Skalar f. Höhe (-10 = / 10; 10 = * 10)
scalco	71-72	Skalar f. Koord. (-10 = / 10; 10 = * 10)

Key	Bytes	Beschreibung
sx	73-76	Schuss X-Koordiante
sy	77-80	Schuss Y-Koordinate
gx	81-84	Empfänger X-Koordinate
gy	85-88	Empfänger Y-Koordinate
sstat	99-100	Schuss-Statik (ms)
gstat	101-102	Empfänger-Statik (ms)
tstat	103-104	Gesamte Statik (ms)
delrt	109-110	Verzögerungszeit der Spur (Shift)
ns	115-116	Anzahl an Sample der Spur
dt	117-118	Sampleintervall in Microsekunden



Befehlsübersicht

SU

Eine gute Übersicht aller Befehle mit Man-page gibt es unter: sepwww.stanford.edu/oldsep/cliner/files/suhelp/suhelp.html

1. Data Compression
2. Editing, Sorting and Manipulation

   `sushw, suchw, sugethw, suwind, susort, suhtmath`
3. Filtering, Transforms and Attributes

   1D: `subfilt, suconv, suxcor`

   2D: `sudipfilt`

   `sufft, suifft, suamp, suspecfk, suspecfx, sureduce`
4. Gain, NMO, Stack and Standard Processing

   `sugain, sunmo, sumute, sustack, sustatic, suacor, suvibro, suvlength`
5. Graphics

   `suxgraph, suximage, suxwigb, suxmovie, suxpicker`
6. Import/Export

   `segyread, segywrite, segyhdrs, b2a, a2b, suaddhead, suascii`
7. Migration and Dip Moveout

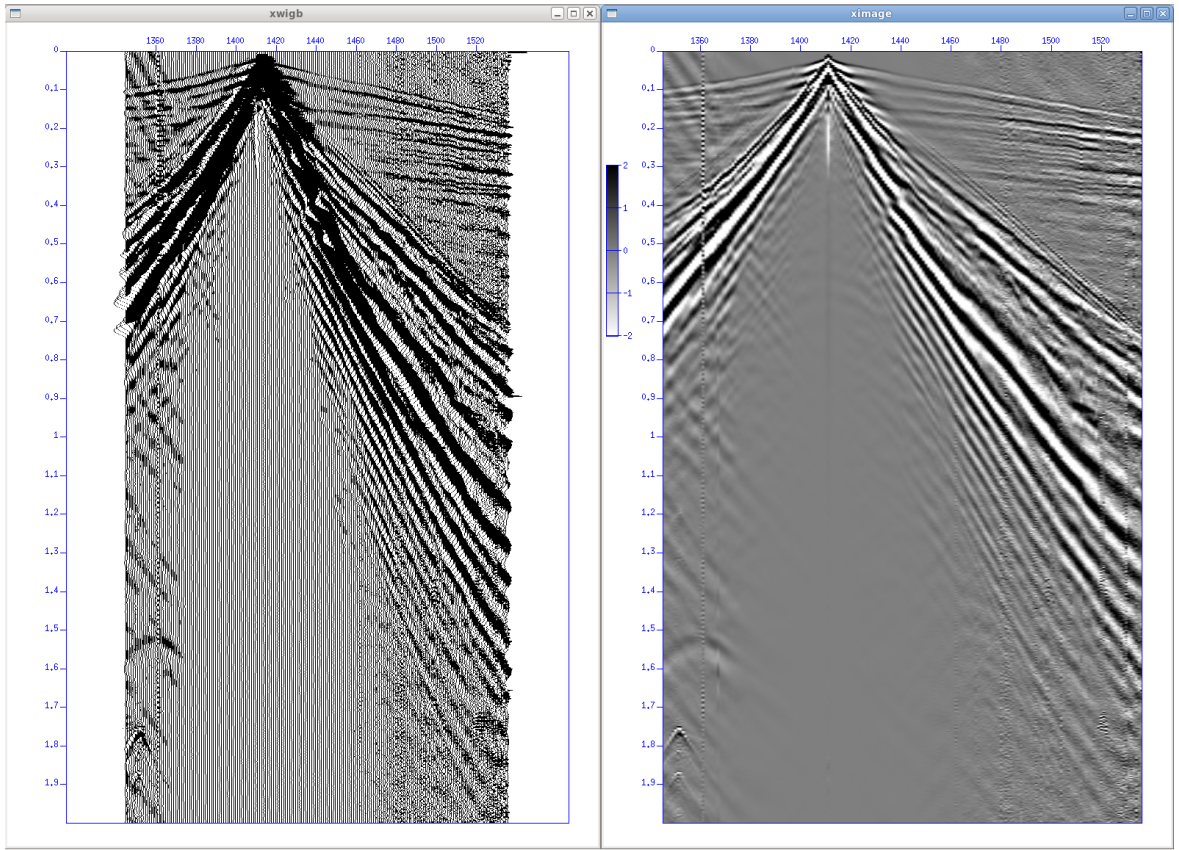
   `sumigfd, sumigprefd, sustolt, sudmovz`
8. Simulation and Model Building
9. Utilities

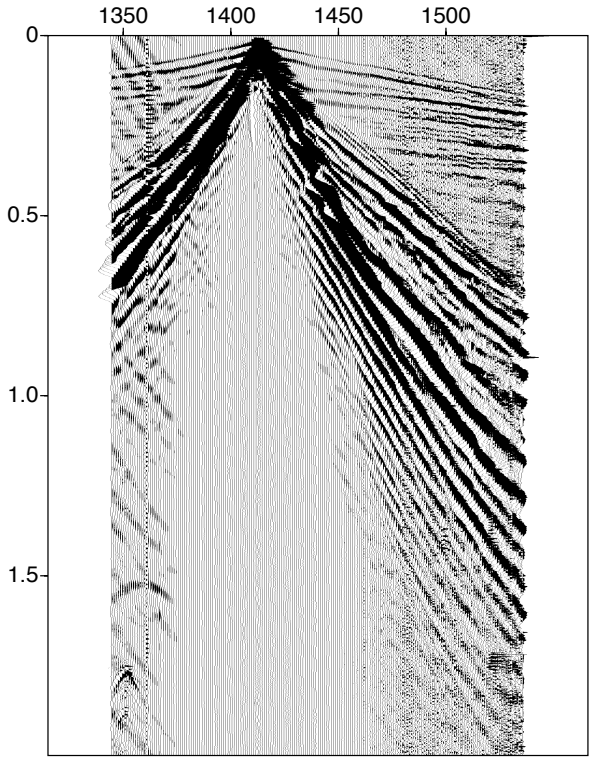
   `sunormalize, sushift, sumax, sumean`

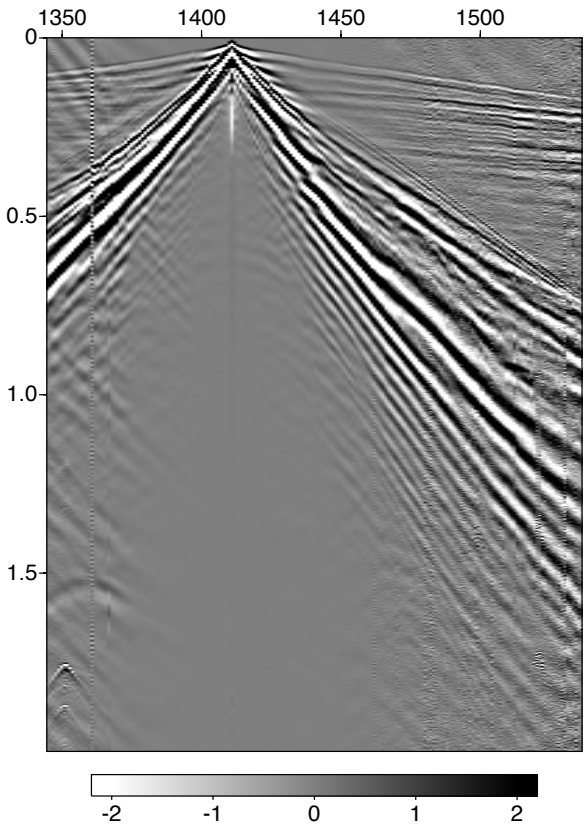
GRAPHISCHE AUSGABE

Zum Visualisieren der Daten gibt es mehrere Möglichkeiten in Abhängigkeit von Input (SU-Files oder Binär) und Output (X-Window on screen oder PS-File)

	X-Window	**PS-File**
SU-Input	z.B. suximage, suxwigb	z.B. supsimage, supswigb
BIN-Input	z.B. ximage, xwigb	z.B. psimage,pswigb

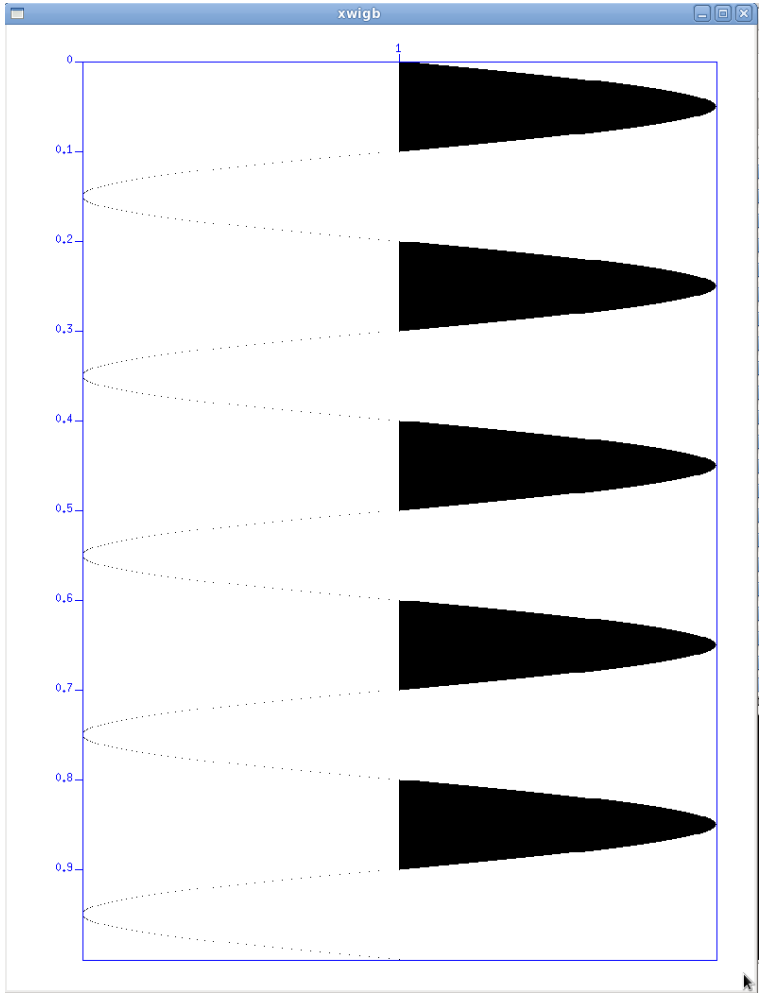






Beispiel1

Es soll mittels AWK eine Sinusfunktion als Zeitreihe erstellt und dann mittels SU in SEGY-Format umgewandelt und anschließend visualisiert werden.



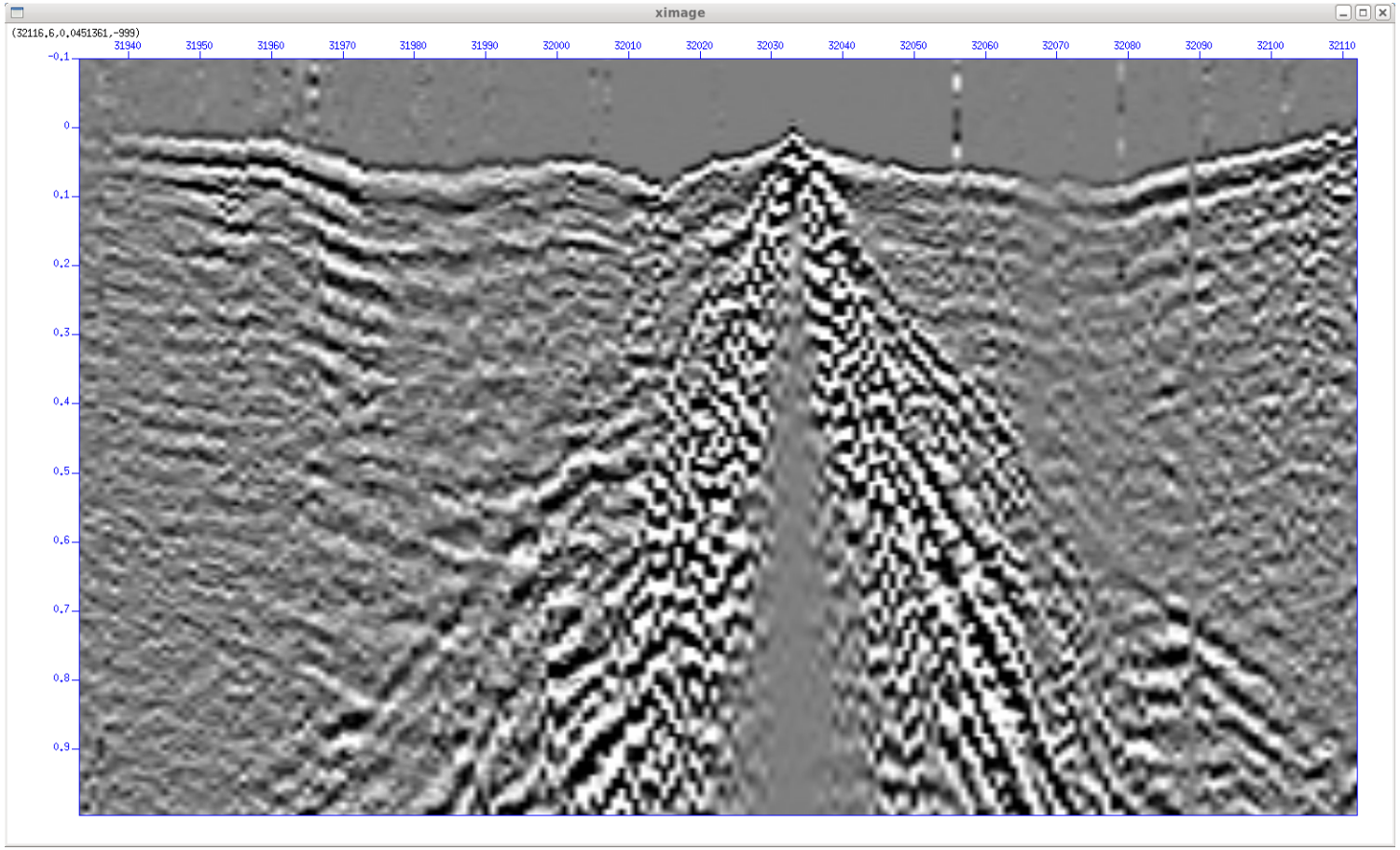

```
#!/bin/tcsh

set dt=0.001
set f=5
set tmax=1
set ns=`echo $tmax $dt | awk '{print int($1/$2)+1}'`

echo $ns $dt $f | \
awk '{pi=atan2(0,-1);for(i=0;i<$1;i++)print sin(i*$2*$3*2*pi)}' | \
a2b n1=1 > sin.dat
suaddhead ns=$ns < sin.dat | sushw key=dt a=`echo $dt | awk '{print $1*1000000}'` > sin.su
segyhdrs < sin.su | segywrite tape=sin.segy endian=0
suxwigb < sin.su
```

Beispiel2

Reduzierte Darstellung des gefilterten Records 43 (5-30 Hz) mit vred=6km/s zwischen -0.1 und 1 Sekunden .




```
#!/bin/tcsh

set shot=43
set clip=2
segyread tape=/homelocal/marco/simba_real/segy/simba_756rec.sgy endian=0 | \
segyclean  | suwind max=0 accept=$shot key=ep | suwind max=0 accept=1 key=trid  | \
sugain pbal=1  > tmp.su
subfilt zerophase=0 fstoplo=4 fpasslo=5 fpasshi=30 fstophi=40 astoplo=0 apasslo=1 apasshi=1
astophi=0  < tmp.su | sushift tmin=-0.1 | sureduce rv=6 | suvlength ns=220 | suximage clip=2
```

- Unix – Befehle
  - Verzeichniswechsel
  - Erstellen, löschen, kopieren oder verschieben von Dateien/Verzeichnissen
  - Grobe Bearbeitung von Dateien
  - Ändern von Benutzerrechten

- AWK / SED
  - Manipulieren von Dateien und Datensätzen

- SHELL – Skripte
  - Zusammenfassen verschiedenster Befehle in ein Skript
- FORTRAN 95
  - Programme zum Lösen gestellter Probleme

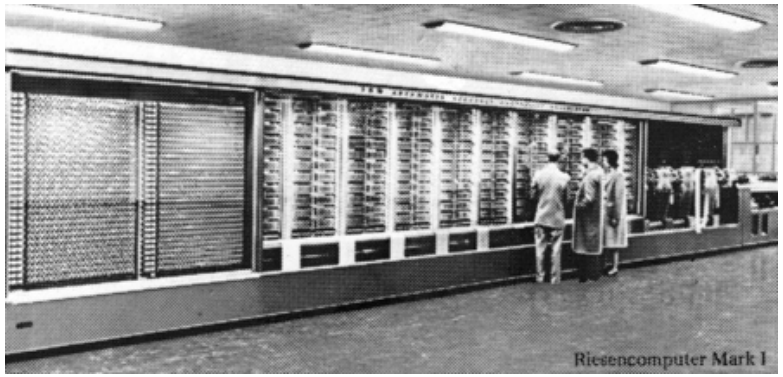
- GMT
  - Plotten von Daten, Karten,…
  - Manipulieren von Daten (z.B. interpolieren)
- LATEX
  - Zusammenfassen von Ergebnissen in Textform (z.B. Bericht)
  - Erstellen von Tabellen, Formeln
  - Einbinden von Grafiken

Warum gerade diese Programme??

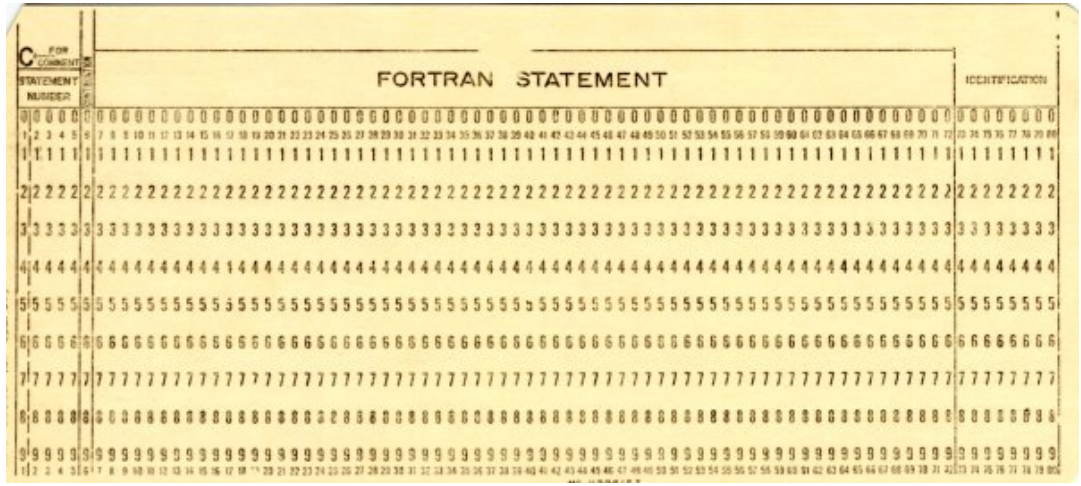
Alle sind kostenlos und universell einsetzbar !!!!

Fortran (**For**mula **Tran**slation) ist eine alte Programmiersprache die ursprünglich dafür benutzt wurde wissenschaftliche Formeln in Computercode umzusetzen. Seit der Einführung 1954 wurde Fortran ständig weiter entwickelt. (FORTRAN I, FORTRAN II, FORTRAN IV, FORTRAN-66, FORTRAN-77, Fortran 90, Fortran 95, Fortran 2000, Fortran 2003, Fortran 2008, Fortran 2010).

Im Gegensatz zu AWK, wird bei Fortran (wie auch z.B. bei C/C++) der Source-Code in Maschinensprache übersetzt (kompiliert) und damit ein Executable erzeugt. Bis Fortran 77 mußte der Source-Code als .f Datei abgespeichert werden. Ab Fortran 90 wird der Source Code als .f mit angehängter Version abgespeichert (z.B. .f90, .f95).



Das Übersetzen in die Maschinensprache übernimmt dann der Compiler. Heutzutage gibt es eigentlich nur noch Zwei: ifort (Intel-Compiler, nicht kostenfrei) und gfortran (GNU-Compiler, kostenfrei). Die Compiler können auch mittels Flags gesteuert werden um z.B. gleich den Namen des Executable festzulegen. Der gfortran-Compiler ist für Mac, Linux, Sun,... kostenlos verfügbar.



Eine Lochkarte=80 Byte
1 Mio Lochkarten = 80 MB
entspricht einen Stapel von ca. 170 Metern Höhe, einer Masse von ca. 2.5 t und einem Volumen von ca. 6 m^3

Fortran 77

Spalte 1 c oder * für Kommentare
Spalte 1-6 für Statement Labels
Spalte 7-72 Statements

```
      program myfirstprogram
      implicit none
C Variablendeklaration
      real a,b

C Einlesen der Zahlen
      write(6,*)'Bitte erste Zahl eingeben: '
      read(5,*)a
      write(6,*)'Bitte zweite Zahl eingeben: '
      read(5,*)b

C Berechnen und Ausgeben
      write(6,*)'Die Multiplikation',
     +' beider Zahlen ergibt: ',a*b
      end
```

Fortran 90

```
program myfirstprogram
implicit none
! Variablendeklaration
real :: a,b

!Einlesen der Zahlen
write(6,*)'Bitte erste Zahl eingeben: '
read(5,*)a
write(6,*)'Bitte zweite Zahl eingeben: '
read(5,*)b

!Berechnen und Ausgeben
write(6,*)'Die Multiplikation beider',&
          ' Zahlen ergibt: ',a*b
end program myfirstprogram
```

Kompiliert werden die Source Codes mit

```
gfortran -o myfirstprogram myfirstprogram.f
```

```
gfortran -o myfirstprogram myfirstprogram.f95
```

Ausführen und Ausgabe des Programs:

```
$ ./myfirstprogram
 Bitte erste Zahl eingeben:
10
 Bitte zweite Zahl eingeben:
20
 Die Multiplikation beider Zahlen ergibt:    200.000000
```

Datentypen

Datentyp	Beschreibung	Wertebereich
Logical	logische Werte	.true. .false.
Integer	Ganze Zahlen	
integer*2	2 Byte Integer	-32768 bis 32767
integer*4	4 Byte Integer (default)	-2147483648 bis -2147483647
Real	Fließkommazahlen (4 Byte)	$-10^{38,53}$ bis $10^{38,53}$
Double Precision	Fließkommazahlen (8 Byte) (=real*8)	$-10^{308,25}$ bis $10^{308,25}$
Complex	Komplexe Zahlen (4 Byte)	$-10^{38,53}$ bis $10^{38,53}$
Double Complex	Komplexe Zahlen (8 Byte) (=complex+8)	$-10^{308,25}$ bis $10^{308,25}$
Character(len=1)	String	Alle möglichen Zeichen mit der Anzahl gleich der Länge hinter len=. In diesem Fall ein beliebiges Zeichen.

Hinweis: Die Wertzuweisung bei komplexen Zahlen erfolgt mit folgender Syntax: `complexnumber=(realval,imagval).` Der Zugriff auf den Real- oder Imaginärteil einer komplexen Zahl wiederum erfolgt über `real(complexnumer)` bzw. `aimag(complexnumber).`

Mittels vordefinierter Funktionen können auch Datentypen in andere temporär konvertiert werden (cast):

Konvertieren nach	**Funktionsname**	**Beispiel**	
Integer	`int()`	`x=int(10.8)`	➔ x=10
Nearest Integer	`nint()`	`x=nint(10.8)`	➔ x=11
Real	`real()`	`x=real(10)`	➔ x=10.0000
Double precision	`dble()`	`x=dble(10)`	➔ x=10.0000
Complex	`cmplx()`	`x=cmplx(10,20)`	➔ x=(10,20)
Double Complex	`dcmplx()`	`x=dcmplx(10,20)`	➔ x=(10,20)
Character	`write`	`write(x,'(I2)')10`	➔ x=10

Character-Funktionen

Es stehen diverse Funktionen zur Manipulation von Charactern (=Strings) zur Verfügung. Demonstriert sollen Diese an folgenden Beispiel:

```
charcter(len=15) :: cc
cc=' ABC12345ABC '  !Character der Länge 15 mit 13 Zeichen (davon 2 Space)
```

Funktion	Beschreibung	Beispiel & Ausgabe (. in der Ausgabe repräsetiert Space)
`()`	Zugriff auf Teile des Strings	`cc(3:5)` ➔ `BC1`
`len`	Stringlänge	`len(cc)` ➔ `15`
`len_trim`	Stringlänge ohne ABSCHLIESSENDE Leerzeichen	`len_trim(cc)` ➔ `12`
`trim`	Gibt String ohne ABSCHLIESSENDE Leerzeichen zurück	`trim(cc)` ➔ `.ABC12345ABC`
`adjustl`	Linksbündiges Ausrichten eines Strings	`adjustl(cc)` ➔ `ABC12345ABC....`
`adjustr`	Rechtsbündiges Ausrichten eines Strings	`adjustr(cc)` ➔ `....ABC12345ABC`
`repeat`	Wiederholen eines Strings	`repeat(cc,2)` ➔ `.ABC12345ABC....ABC12345ABC...`
`index`	Gibt die Position einer bestimmen Zeichenkette im String zurück	`index(cc,'ABC')` ➔ `2` `index(cc,'ABC',.true. )` ➔ `10`
`scan`	Prüft, ob ein String EIN Zeichen aus einem Zeichensatz enthält	`scan(cc,'ABC')` ➔ `2` `scan(cc,'ABC',.true. )` ➔ `12`
`verify`	Prüft, ob ein String EIN Zeichen aus einem Zeichensatz nicht enthält	`scan(cc,'ABC')` ➔ `1` `scan(cc,'ABC',.true. )` ➔ `15`

Mit // können zwei Character Verbunden werden. Hier ein abschliessendes Beispiel für die Character-Manipulation:

```
! To compile with 'gfortran -o charman charman.f95'
program charman
implicit none
integer :: i1
real    :: r1
character(len=35) :: cc,c1,c2
i1=100
r1=10.5
write(c1,'(I5)')i1
write(c2,'(F8.3)')r1
cc='Snapshot_Shot'//trim(adjustl(c1))//'_F'//trim(adjustl(c2))//'Hz.bin'
print*,'>',c1,'<'
print*,'>',trim(adjustl(c1)),'<'
print*,'>',c2,'<'
print*,'>',trim(adjustl(c2)),'<'
print*,'>',cc,'<'
print*,'>',trim(adjustl(cc)),'<'
end program charman
```

Ausgabe:

```
>  100                              <
>100<
>  10.500                           <
>10.500<
>Snapshot_Shot100_F10.500Hz.bin     <
>Snapshot_Shot100_F10.500Hz.bin<
```

Math. & Trig. Funktionen

Funktion	Beschreibung	# Argumente	Beispiele
`max`	Maximumwert	2 oder mehr	`max(1,5,9,2)` ➔ `9`
`min`	Minimumwert	2 oder mehr	`min(1,5,9,2)` ➔ `1`
`abs`	Absolutwert	1	`abs(-10.5)` ➔ `10.5;` `abs(10.5)` ➔ `10.5;`
`dim`	Positive Differenz	2	`dim(10,5)` ➔ `5;` `dim(5,10)` ➔ `0;` `dim(-10,-5)` ➔ `0;` `dim(-5,-10)` ➔ `5;`

`mod`	Modulus	2	`mod(10,3)` ➔ `1;` `mod(10,5)` ➔ `0;`
			`mod(5,10)` ➔ `5;` `mod(-10,3)` ➔ `-1;`
			`mod(-10,-3)` ➔ `1;` `mod(10,-3)` ➔ `1;`

`sign`	Vorzeichenänderung	2	`sign(10,5)` ➔ `10;` `sign(-10,-5)` ➔ `-10;` `sign(10,-5)` ➔ `-10;` `sign(-10,5)` ➔ `10;`
`dprod`	Double Precision Produkt zweier Real-Variablen	2	
`sqrt`	Quadratwurzel	1	`sqrt(2025)` ➔ `45;`
`exp`	Exponentialfunktion	1	`exp(0.0)` ➔ `1;`
`log`	Natürlicher Logarithmus	1	`log(148.413162)` ➔ `5;`
`log10`	Logarithmus zur Basis 10	1	`log10(1000.0)` ➔ `3;`

Math. & Trig. Funktionen

Die Argumente der Trig. Funktionen sind immer in Radians. Pi läßt sich über folgende Beziehung berechnen: `pi=4*atan(1.0)`

Funktion	Beschreibung	Beispiele
`sin`	Sinus	`sin(pi/2)` ➔ `1`
`asin`	Arc-Sinus	`asin(1)` ➔ `1.57079637`
`cos`	Kosinus	`cos(1)` ➔ `0`
`acos`	Arc-Kosinus	`acos(0)` ➔ `1`
`tan`	Tangens	`tan(0)` ➔ `0`
`atan`	Arc-Tangens	`atan(0)` ➔ `0`
`sinh`	Sinus Hyperbolicus	`sinh(0)` ➔ `0`
`cosh`	Kosinus Hyperbolicus	`cosh(1)` ➔ `1`
`tanh`	Tangens Hyperbolicus (=sinh/cosh)	`tanh(1)` ➔ `0`

Verzweigungen

Wie auch bei der TCSH gibt es in Fortran die einfache Verzweigung (if-then-else) und die Mehrfachverzweigung (select-case). Dabei gilt folgende Syntax:

```
if(i==1)print*,i
```

```
if(i==1)then
  print*,i
endif
```

```
if(i==1)then
  print*,i
else
  print*,'keine 1'
endif
```

```
i=3
select case (i)
       case (1)
            print*,"Es ist eine 1"
       case (2)
            print*,"Es ist eine 2"
       case (3)
            print*,"Es ist eine 3"
       case default
            print*,"Es ist keine 1"
            print*,"Und keine 2"
            print*,"Und auch keine 3"
end select
```

Die möglichen Operatoren sind:

==	Gleich	<	Kleiner	<=	Kleinergleich	.or.	Oder
/=	Ungleich	>	Grösser	>=	Grössergleich	.and.	Und

Schleifen

In Fortran gibt es zwei Arten von Schleifen: Endliche und Endlose.

Endliche Schleife

```
do i=1,10
    print*,i
end do

do i=1,10
    print*,i
    if(i==5)exit
end do

Schleife1: do i=1,10
   Schleife2: do j=1,10
         print*,i,j
         if(j==5)exit Schleife1
   end do Schleife2
end do Schleife1
```

Endlose Schleife

```
i=1
do
if(i==10)exit
i=i+1
end do


i=1
j=1
Schleife1: do
   Schleife2: do
         print*,i,j
         if(j==5)exit Schleife1
         j=j+1
   end do Schleife2
i=i+1
end do Schleife1
```

Um nicht wirklich endlose Schleifen zu erzeugen, sollte die "Endlosschleife" einen Zähler und eine IF-Bedingung zum Abbruch beinhalten. Diese Art von Schleife ist nützlich z.B. für die Ermittlung der Anzahl ein Zeilen innerhalb einer Datei.

Der Input (read) und Output(write) bei Fortran-Programmen kann über das Terminal oder Dateien erfolgen. Die Syntax bei read und write ist die Gleiche: `read(KANAL,FORMAT)var1,var2,... ;write(KANAL,FORMAT)var1,var2,....` Bei KANAL kann eine beliebige Nummer für eine Datei angegeben werden die vorher mit "open" geöffnet wurde oder die 5 für die Standardeingabe und die 6 für die Standardausgabe (ein * setzt hier die bei read die 5 als default und bei write die 6). Wird bei FORMAT kein Format gewünscht muss hier ein * gesetzt werden, sonst gilt folgende Syntax: '(FORMATBESCHREIBER)', wobei FORMATBESCHREIBER in Abhängigkeit vom Datentyp wie folgt aussehen kann:

Datentyp	**Format**
Integer	`I5, I30`
Real, Double Precision	`F10.5, E20.10`
Character	`A, A10`
Platzhalter	`X,10X`

```
program format_beispiel
implicit none
integer :: i1=12345
real    :: r1=12.345
character(len=14) :: fstring='(I5,3X,F6.3)'

write(*,'(I5,3X,F6.3)'),i1,r1

write(*,100)i1,r1
100 FORMAT(I5,3X,F6.3)

write(*,fstring)i1,r1
end program format_beispiel
```

Ausgabe:

```
12345   12.345
12345   12.345
12345   12.345
```

Um Dateien zu erstellen oder existierende Dateien einzulesen, müssen Diese vorher mit OPEN geöffnet werden. Dabei wird zwischen ASCII und BINÄR - Dateien unterschieden. Zuerst ASCII

```
OPEN(UNIT=30,file='Datei_ASCII.in',status='old',FORM='FORMATTED',iostat=status_open)
OPEN(UNIT=31,file='Datei_ASCII.out',status='unknown',FORM='FORMATTED',iostat=status_open)
```

Und nun Binär. Hier gibt es mehrere Möglichkeiten:

```
OPEN(UNIT=32,file='Datei_BIN.in',status='old', ACCESS='STREAM',iostat=status_open)
OPEN(UNIT=33,file='Datei_BIN.out',status='unknown', ACCESS='STREAM',iostat=status_open)
OPEN(UNIT=34,file='Datei2_BIN.in',status='old', ACCESS='DIRECT',recl=4,iostat=status_open)
OPEN(UNIT=35,file='Datei2_BIN.out',status='unknown', ACCESS='DIRECT',recl=4,iostat=status_open)
```

Beim Öffnen wird jeder Datei eine Zahl zugeordnet (=KANAL), die dann im READ oder WRITE Statement angegeben werden muss. Optional können noch mit iostat mögliche Fehlercodes (Integer) in Variablen geschrieben werden.

`read(32)var1`	liest die erste Anzahl an Bytes gleich der Länge var1 (meist 4 Bytes - als Bytes 1-4)
`read(32,pos=20)var1`	liest ab Byte 20 die Anzahl an Bytes gleich der Länge var1 (meist 4 Bytes - als Bytes 20-24)
`write(33)var1`	schreibt die erste Anzahl an Bytes gleich der Länge var1 (meist 4 Bytes - als Bytes 1-4)
`write(33,pos=20)var1`	schreibt ab Byte 20 die Anzahl an Bytes gleich der Länge var1 (meist 4 Bytes - als Bytes 1-4)
`read(34,rec=5)var1`	liest record 5; entspricht Bytes 17-20 in der Datei, weil diese mit recl=4 geöffnet wurde
`write(35,rec=5)var1`	schreibt record 5; entspricht Bytes 17-20 in der Datei, weil diese mit recl=4 geöffnet wurde

Da aber nicht immer gesichert ist, ob eine Datei existiert, empfiehlt es sich, vor OPEN die Existenz mit INQUIRE zu checken. Dazu ein kleines Beispielprogramm:

```
program readfile
implicit none
logical :: exist_file
integer :: count,status_read
real    :: dummy

inquire(file='datei.in',exist=exist_file)
if(exist_file)then
open(unit=10,file='datei.in',form='formatted',status='old')
count=0
  do
    read(10,*,iostat=status_read)dummy
    if(status_read/=0)exit
    count=count+1
  end do
  print*,'Number of line:',count
end if

end program readfile
```

Namelist

Bis jetzt musste der Code jedes mal neu kompiliert werden, wenn man Parameter ändern will. Dies kann man umgehen, indem man die Werte für die Parameter mittels "NAMELIST" von Parameterdateien einliest. Die Parameterliste wird mit `namelist /title/var1,var2,var3,...` im Programm definiert. Die Variablen in der Parameterliste müssen natürlich vorher deklariert werden. Die Parameterdatei hat dann folgende Syntax: Mit &title und &end wird eine Parameterliste erstellt. Der Titel der Parameterliste (hier title) muss den gleichen Namen haben, wie die Liste im Programm unter der die Variablen aufgelistet sind. Innerhalb der Parameterliste können nun die Werte der Variablen angegeben werden: `var=...` .Dabei muß nicht allen Variablen, welche im Programm unter `namelist` aufgelistet sind, ein Wert zugewiesen werden. Es dürfen hier nur nicht Variablennamen erscheinen, die im Programm unter namelist NICHT definiert sind. Geöffnet wird die Parameterdatei wie eine normale ASCII-Datei. Eine Besonder-heit ist jetzt nur das read-Statement: Wo sonst der Formatbe-schreiber eingetragen werden kann, muss hier jetzt `nml=title` stehen. Es besteht auch die Möglichkeit mehrere Listen in einer Parameterdatei abzuspeichern. Es müssen dafür nur dementsprechend die Listen im Programm definiert werden. Abgerufen können die Listen über mehrere Read-Statements, wobei hier die Zahl für die Datei natürlich gleich bleibt und sich nur der Listenname hinter `nml=` ändert.

```
program readparameters
implicit none
integer             :: i1
real                :: r1
character(len=80)   :: c1
namelist /parameter/i1,r1,c1

i1=10
r1=3.14
c1='Vorher'
write(*,'(A,A,I3,X,F5.3)')trim(adjustl(c1)),':',i1,r1
open(unit=10,file='parameter.in',form='formatted',status='old')
read(10,nml=parameter)
write(*,'(A,A,I3,X,F5.3)')trim(adjustl(c1)),':',i1,r1
end program readparameters
```

Ausgabe

```
Vorher: 10 3.140
Nachher: 20 6.280
```

Inhalt von parameter.in:

```
&parameter
i1=20
r1=6.28
c1="Nachher"
&end
```

Bisher konnten Variablen nur ein Datenwert zugeordnet werden. Es gibt aber auch Situation (z.B. bei Schleifen), wo einer Variable mehrere Werte zugeordnet werden sollen, die über einen Index abgerufen werden sollen. Dafür gibt es Arrays. Arrays sind aneinandergehängte Variablen gleichen Namens, auf deren Inhalte über einen Index zugegriffen werden können. Arrays können alle Datentypen haben, wie "normale" Variablen (logical, integer, real, complex, character). Zur Deklaration von Arras und der Ausgabe deren Inhalts ein kleines Beispielprogramm:

```
program firstarray
implicit none
integer,dimension(3)            :: alter=(/32,34,45/)
integer,dimension(3)            :: nummer
real,dimension(3)               :: groesse=(/1.83,1.77,1.65/)
character(len=7),dimension(3)   :: name=(/"Mueller","  Meier","Schulze"/)
character(len=7),dimension(3)   :: vorname=(/" Stefan","Manfred"," Ingrid"/)
integer                         :: i

write(*,'(A6,X,A7,X,A7,X,A5,X,A7)')"Nummer","Name","Vorname","Alter","Groesse"
do i=1,3
nummer(i)=i
   write(*,'(I6,X,A7,X,A7,X,I5,X,F7.2)')nummer(i),name(i),vorname(i),alter(i),groesse(i)
end do
end program firstarray
```

Characters innerhalb eines Arrays müssen gleich lang sein!!!

Ausgabe:

```
Nummer    Name Vorname Alter Groesse
     1 Mueller  Stefan    32    1.83
     2   Meier Manfred    34    1.77
     3 Schulze  Ingrid    45    1.65
```

Es ist auch möglich mehrdimensionale Arrays zu definieren. Da Fortran jedoch immer von 1D-Arrays (Vektoren) ausgeht, muß die Wertzuweisung mit Hilfe von `shape` und `reshape` erfolgen:

```
program arraytest
implicit none
integer,dimension(3,3) :: ar
integer                :: i,j
print*,shape(ar)
print*,"************************************"
ar=reshape((/1,2,3,4,5,6,7,8,9/),shape(ar))
do i=1,3
    print*,ar(i,1),ar(i,2),ar(i,3)
end do
print*,"************************************"
ar=transpose(reshape((/1,2,3,4,5,6,7,8,9/),shape(ar)))
do i=1,3
    print*,ar(i,1),ar(i,2),ar(i,3)
end do
end program arraytest
```

Ausgabe:

```
           3           3
************************************
           1           4           7
           2           5           8
           3           6           9
************************************
           1           2           3
           4           5           6
           7           8           9
```

Mit Shape wird die Dimension von dem Array ausgegeben (Zeilen, Spalten) und dementsprechend die Matrix aufgefüllt (Spaltenweise). Möchte man die Matrix aber Zeilenweise aufgefüllt haben, muss man die Matrix einfach anschließend transponieren.

Das funktioniert ja soweit ganz gut. Doch was passiert, wenn man keine MxM Matrizen hat? Dann benötigen wir 2 Arrays! Ist es außerdem möglich, die Matrix "dynamisch" auszugeben? Ja, die Anzahl an Zeilen/Spalten kann man sich mit `size` ausgeben lassen. Damit nicht nach jedem write-statement ein automatischer Zeilenumbruch erfolgt, muss hinter dem Formatbeschreiber noch advance='no' folgen.

```
program arraytest
implicit none
integer,dimension(4,3) :: ar
integer,dimension(3,4) :: ar2
integer                :: i,j
write(*,'(2(I2,X))')shape(ar)
print*,"***********"
ar=reshape((/1,2,3,4,5,6,7,8,9,10,11,12/),shape(ar))
do i=1,size(ar,1)   !Schleife ueber Zeilen
   do j=1,size(ar,2)    !Schleife ueber Spalten
     write(*,'(X,I2)',advance='no')ar(i,j)
   end do
   write(*,*)
end do
print*,"***********"
ar2=transpose(reshape((/1,2,3,4,5,6,7,8,9,10,11,12/),shape(ar)))
do i=1,size(ar2,1)   !Schleife ueber Zeilen
   do j=1,size(ar2,2)    !Schleife ueber Spalten
     write(*,'(X,I2)',advance='no')ar2(i,j)
   end do
   write(*,*)
end do
end program arraytest
```

Ausgabe:

```
4  3
***********
 1  5  9
 2  6 10
 3  7 11
 4  8 12
***********
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

```
program arrayalloc
implicit none
integer,allocatable,dimension(:,:) :: ar
integer                 :: i,j,nx,ny
write(*,*)"Nx,Ny?"
read(*,*)nx,ny
allocate(ar(ny,nx))
!Berechnung
!Sollte wegen Speicherzugriff immer
!Spaltenweise verlaufen
do j=1,nx
   do i=1,ny
     ar(i,j)=i+j
   end do
end do
!Ausgabe Zeilenweise
do i=1,ny
   do j=1,nx
     write(*,'(X,I2)',advance='no')ar(i,j)
   end do
   write(*,*)
end do
deallocate(ar)
end program arrayalloc
```

Es kommt oft vor, dass die Anzahl an Zeilen und Spalten einer Matrix vorher nicht bekannt sind und sich erst durch einlesen vom Parametern ergeben. Wie können nun die Anzahl an Zeilen und Spalten dynamisch gesetzt werden? Dafür gibt es die Befehle `allocate` und `deallocate`. Diese Befehle sind eine unglaubliche Verbesserung im Vergleich zu Fortran 77.

Eingabe/Ausgabe:

```
 Nx,Ny?
5,4
  2  3  4  5  6
  3  4  5  6  7
  4  5  6  7  8
  5  6  7  8  9
```

Abschließend noch eine zusammenfassende Auflistung der wichtigsten Funktionen für Arrays:

Funktion	**Beschreibung**
`dot_product(vector1,vector2)`	Skalarprodukt zweier Vektoren
`matmul(matrix1,matrix2), matmul(matrix1,vektor1)`	Matrixmultiplikation
`transpose(matrix)`	Transponierte
`maxloc(vektor1), maxloc(matrix1)`	Index des Feldes mit dem höchsten Wert
`minloc(vektor1), minloc(matrix1)`	Index des Feldes mit dem kleinsten Wert
`maxval(vektor1), maxval(matrix1)`	Höchste Wert des Arrays
`minval(vektor1), minval(matrix1)`	Kleinste Wert des Arrays
`sum(vektor1), sum(vektor1)`	Summe aller Komponenten
`product(vektor1),product(vektor1)`	Produkt aller Komponenten
`cshift(matrix,anz,dim)`	Zyklische Verschiebung

`where(bedingung)`
`  anweisung1`
`else where`
`  anweisung2`
`end`
	Anweisung in Abhängigkeit von Werten eines Datenfeldes

ARRAYS

FORTRAN95

```
program arrayfunc
implicit none
integer,dimension(3,3) :: ar
integer,dimension(3,3) :: ar2
integer                :: i,j
print*,"***********"
ar=transpose(reshape((/1,2,3,4,5,6,7,8,9,10,11,12/),shape(ar)))
call print_array(ar,size(ar,2),size(ar,1))
print*,"***********"
write(*,*)maxloc(ar)
write(*,*)minloc(ar)
write(*,*)maxval(ar)
write(*,*)minval(ar)
write(*,*)sum(ar)
write(*,*)product(ar)
print*,"***********"
ar2=cshift(ar,1,1)
call print_array(ar2,size(ar,2),size(ar,1))
print*,"***********"
ar2=cshift(ar,-1,1)
call print_array(ar2,size(ar,2),size(ar,1))
print*,"***********"
ar2=cshift(ar,1,2)
call print_array(ar2,size(ar,2),size(ar,1))
print*,"***********"
ar2=cshift(ar,-1,2)
call print_array(ar2,size(ar,2),size(ar,1))
print*,"***********"
ar2=0
where (ar>4)
ar2=1
else where
ar2=-1
end where
call print_array(ar2,size(ar,2),size(ar,1))
end program arrayfunc

subroutine print_array(ar,nx,ny)
integer,intent(in) :: nx,ny
integer,intent(in) :: ar(nx,ny)
integer            :: i,j
do i=1,ny
   do j=1,nx
       write(*,'(X,I2)',advance='no')ar(i,j)
   end do
   write(*,*)
end do
end subroutine print_array
```

Ausgabe:

```
***********
 1  2  3
 4  5  6
 7  8  9
***********
          3           3
          1           1
          9
          1
         45
     362880
***********
 4  5  6
 7  8  9
 1  2  3
***********
 7  8  9
 1  2  3
 4  5  6
***********
 2  3  1
 5  6  4
 8  9  7
***********
 3  1  2
 6  4  5
 9  7  8
***********
-1 -1 -1
-1  1  1
 1  1  1
```

Für ein Anwendungsbeispiel für ein Fortran-Programm in Kombination mit AWK, SHELL und GMT soll die Diskrete Fouriertransformation (DFT) implementiert werden. Dabei soll ein Zeitsignal mittels AWK erstellt und mit GMT geplottet werden. Die Fouriertransformation soll dann durch ein Fortran-Programm realisiert werden. Anschließend soll das Amplitudenspektrum mittels AWK und GMT geplottet werden.

Die DFT für eine diskrete Zeitreihe f ist gegeben durch

$$F_k = \sum_{j=0}^{N-1} f_j e^{i\omega_k j\Delta t}$$

Die iDFT für eine diskrete Fourierreihe F ist gegeben durch

$$f_j = \frac{1}{N} \sum_{k=0}^{N-1} F_k e^{-i\omega_k j\Delta t}$$

Als Zeitsignal soll ein Sinus aus 3 verschiedenen Frequenzen und Amplituden verwendet werden:

```
f(t)=sin(2*2π*t)+0.5*sin(5*2π*t)+0.2*sin(20*2π*t)
```