# Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi/

## 4. Formattiertes Input/Output, Formatbeschreiber

# Input /Output Befehle

**Quelle/Ziel der Eingabe/Ausgabe**

- Standard Input/Output (Konsole, Pipe, etc.)
- Interne Datei (Schreiben in eine / Lesen von einer Zeichenkette)
- Externe Datei (eine Datei im Filesystem)

**Form der Eingabe/Ausgabe**

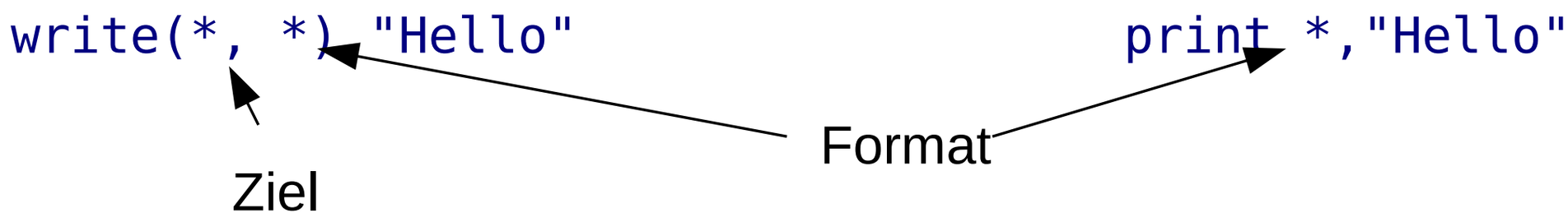
- Formattierte E/A (ASCII-Daten, auch für Menschen lesbar, portabel)
- Unformattierte E/A (Binärdaten, nur maschinenlesbar, nicht portabel)

**Behandlung von Datensätzen (records)**

- E/A mit Vorschub (advancing IO)
- E/A ohne Vorschub (non-advancing IO)
- Streamartig I/O (ähnlich zu C)

# Formattierte Ausgabe

```
write(dest, form) io-list   ! Formatted output to spec. destination
print form, io-list         ! Formatted output to stdout
```




- Wenn als Ziel „*“ spezifiziert wird, geht Ausgabe auf stdout (Konsole)
- Wenn als Format * spezifiziert wird, entscheidet das System, wie die Einträge dargestellt werden.
- Soll ein spezifisches Format für die Darstellung verwendet werden, muss die entsprechende Formatbeschreibung (z.B. als String) angegeben werden.

<u>Formatbeschreibung bestehen aus:</u>

- Typbeschreiber
- Kontrolldirektive(n)
- Zeichenkette(n)

# Typbeschreibung – Integers

| | | | |
|---|---|---|---|
| **Integer (dezimal):** | **I***w* | *w* | Breite des Ausgabefeldes |
| | **I***w*.*m* | *m* | Minimale Anzahl von Stellen (Zahlen) |

- Wenn Feldgröße größer als tatschächliches Output, wird es von links mit Leerzeichen aufgefüllt.

```
write(*, "(I5)") 123          ! -> |__123|
```

- Wenn minimale Anzahl von Stellen angegeben, werden Leerzeichen bei Bedarf mit Nullen ersetzt

```
write(*, "(I5.4)") 123      ! -> |_0123|
```

- Vorzeichen wird bei der Ermittlung der Outputgröße mitgezählt

```
write(*, "(I5.4)") -123       ! -> |-0123|
```

- Wenn Feldgröße kleiner als Outputgröße, wird kein Output erzeugt (nur Sterne)

```
write(*, "(I3)") -123         ! -> |***|
```

- Wenn Feldgröße 0 ist, wird sie dynamisch angepasst (keine Auffüllung)

```
write(*, "(I0)") -1234    ! -> |-1234|
write(*, "(I0)") 23       ! -> |23|
```

# Datentypbeschreibung – Integers (#2)

| | | | |
|---|---|---|---|
| **Integer (hexadezimal):** | **Z***w* | *w* | Breite des Ausgabefeldes |
| | **Z***w*.*m* | *m* | Minimale Anzahl von Stellen (Zahlen) |

```
write(*, "(Z6.4)") 12        ! -> |__000C|
```

| | | | |
|---|---|---|---|
| **Integer (oktal):** | **O***w* | *w* | Breite des Ausgabefeldes |
| | **O***w*.*m* | *m* | Minimale Anzahl von Stellen (Zahlen) |

```
write(*, "(O6.4)") 12        ! -> |__0014|
```

| | | | |
|---|---|---|---|
| **Integer (binär):** | **B***w* | *w* | Breite des Ausgabefeldes |
| | **B***w*.*m* | *m* | Minimale Anzahl von Stellen (Zahlen) |

```
write(*, "(B6.4)") 12        ! -> |__1100|
```

- Verwendung und Einschränkungen bei **Z**, **O** und **B** analog zum **I.**

# Datentypbeschreibung – Reele Zahlen

| **Fixkommanotation:** | `F`*`w`*`.`*`d`* | *w* | Breite des Feldes |
|---|---|---|---|
| | | *d* | Anzahl der Dezimalstellen |

- Dezimalpunkt und Vorzeichen (wenn vorhanden) bei der Outputgröße mitgezählt

```
write(*, "(F8.4)") -9.3_dp      ! -> |_-9.3000|
write(*, "(F6.4)") -9.3_dp      ! -> |******|
```

- Bei nicht angezeigten signifikanten Dezimalstellen mathematisch korrekt gerundet:

```
write(*, "(F8.4)") -9.56789_dp     ! -> |_-9.5679|
write(*, "(F8.4)") 123.456789_dp! -> |123.4568|
```

- Feldbreite kann 0 gesetzt werden, um dynamisch angepasst zu werden:

```
write(*, "(F0.2)") -32.128_dp   ! -> |-32.13|
```

# Datentypbeschreibung – Reele Zahlen (#2)

| | | | |
|---|---|---|---|
| **Exponentialnotation** | **E***w*.*d* | *w* | Breite des Feldes |
| | **E***w*.*d***E***e* | *d* | Anzahl der Dezimalstellen |
| | | e | Anzahl der Stellen im Exponenten |

- Vorzeichen, Dezimalpunkt, Exponentcharakter (E), Vorzeichen des Exponenten werden bei Bestimmung der Outputgröße mitgezählt

```
write(*, "(E16.8)") -3.5e+32_dp      ! -> |_-0.35000000E+33|
write(*, "(E16.8)") -3.5e-32_dp      ! -> |_-0.35000000E+33|
```

- Wenn Anzahl der Stellen im Exponenten nicht festgelegt ist, wird bei Exponenten mit Absolutwert über 99 das Exponentenzeiche „E“ entfernt

```
write(*, "(E10.3E3)") 12.3456e103_dp   ! -> |0.123E+105|
write(*, "(E10.3)") 12.3456e103_dp   ! -> |_0.123+105|
```

- Wenn Absolutwert des Exponenten über 999 liegt, muss die Anzahl der Exponentenstellen  im Formatbeschreiber spezifiziert werden. (Kommt nur bei 16-byte Fließkommazahlen vor.)

# Datentypbeschreibung – Reele Zahlen (#3)

| | | | |
|---|---|---|---|
| **Ingenieurnotation:** | **EN***w*.*d* | *w* | Breite des Feldes |
| | **EN***w*.*d***E***e* | *d* | Anzahl der Dezimalstellen |
| | | e | Anzahl der Stellen im Exponenten |

- Das selbe wie Exponentialnotation, Exponent jedoch durch drei teilbar:

```
write(*, "(EN12.4)") 23456.0_dp ! -> |_23.4560E+03|
```

| | | | |
|---|---|---|---|
| **Wissenschaftliche Notation**: | **ES***w*.*d* | *w* | Breite des Feldes |
| | **ES***w*.*d***E***e* | *d* | Anzahl der Dezimalstellen |
| | | e | Anzahl der Stellen im Exponenten |

- Das selbe wie Exponentialnotation, Ganzteil jedoch im Intervall [1,10):

```
write(*, "(ES12.4)") 23456.0_dp ! -> |__2.3456E+04|
```

- Bei komplexen Zahlen müssen für reellen und imaginären Teil separate Beschreiber (für jeweils eine reelle Zahl) angegeben werden:

```
complex(dp) :: zz
zz = (12.0_dp, -3.0_dp)
write(*, "(F6.2,E14.6)") zz     ! -> |_12.00_-0.300000E+01|
```

# Datentypbeschreibung – Logische Werte, Charaktere, Allgemein

| **Logical:** | ***Lw*** | *w* | Breite des Feldes |
|---|---|---|---|

```
write(*, "(L3)") .true.          ! -> |__T|
write(*, "(L3)") .false.         ! -> |__F|
```

| **Character:** | ***A*** | *w* | Breite des Feldes |
|---|---|---|---|
| | ***Aw*** | | |

```
write(*, "(A)") "Hello!"         ! -> |Hello!|
write(*, "(A10)") "Hello!"       ! -> |____Hello!|
```

| **Allgemeiner Typ:** | **G***w*.*d* | *w* | Breite des Feldes |
|---|---|---|---|
| | **G***w*.*d***E***e* | *d* | Anzahl der Dezimalstellen |
| | | e | Anzahl der Stellen im Exponenten |

- Wendet an Fließkommazahlen abhängig vom Wert F oder E Formattierung an:

```
write(*, "(G10.4)") 3e-13_dp     ! -> |0.3000E-12|
write(*, "(G10.4)") -12.5_dp     ! -> |-12.5000__|
```

| **Allgemeiner Typ:** | **G***w* | *w* | Breite des Feldes |
|---|---|---|---|

- Wendet I, L oder A abhängig vom Typ ab

# Allgemeine Regel

- Nacheinander spezifizierte Feldformate müssen mit Komma (,) getrennt werden:

```
write(*, "(I3, F12.6)") 12, 93.45_dp   ! -> | 12   93.450000|
```

- Wiederholte Feldformate (bzw. -gruppen) durch vorangestellte Anzahlangabe:

```
write(*, "(3I4)") ...          =   write(*, "(I4,I4,I4)") ...
write(*, "(2(I3,F6.2))") ...   =   write(*, "(I3,F6.2,I3,F6.2)") ...
write(*, "(I3,2(L2,A4))")      =   write(*, "(I3,L2,A4,L2,A4)") ...
```

- Arrays in einer I/O-Liste werden elementenweise formattiert:

```
integer :: array1(4), ii
real(dp) :: array2(4)
array1 = [ -1, 3, -4, 5 ]
array2 = [ -3.0_dp, 6.1_dp, -5.8_dp, 13.0_dp ]

write(*, "(4I3)") array1
write(*, "(4F12.6)") array2
```

- Implizite Schleifen in I/O-Listen möglich:

write (*, "**(4(I3,F12.6))**") **(array1(ii), array2(ii), ii = 1, 4)**

# Bearbeitungsreihenfolge der Formatbeschreiber

- Formatbeschreiber solange berücksichtig, bis I/O-Liste abgearbeitet ist:

```
write(*, "(3I4)") 1, 2   ! -> |   1   2|
                         ! 3rd I4 field format ignored
```

- Wenn mehr Elemente in der I/O-Liste als Formatbeschreiber:
  - Liste der Formatbeschreiber so oft wiederverwendet, wie nötig
  - Bei jeder Wiederverwendung wird ein Zeilenschub eingefügt (neuer Datensatz)

```
write(*, "(I3,F6.2)") 1, 4.0_dp, 2, 8.0_dp     ! -> | 1  4.00|
                                                !    | 2  8.00|
```

```
integer :: array1(4)
array1 = [ -1, 3, -4, 5 ]

write(*, "(I3)") array1          ! -> | -1|
                                 !    |  3|
                                 !    | -4|
                                 !    |  5|
```

# Bemerkungen

- Formatbeschreiber können auch Strings enthalten.

```
write(*, "('The value is: ', I3)") 123  ! -> |The value is: 123|
write(*, "(A,I3)") "The value is: ", 123
```

- Groß- und Kleinschreibung sind äquivalent bei den Datentypbeschreibungen:

```
write(*, "(I3)") 123
write(*, "(i3)") 123
```

- Groß- und Kleinschreibung sind nicht äquivalent bei Strings in Formatbeschreiber:

```
write(*, "('A NUMBER: ', I3)") 12       ! -> |A NUMBER:  12|
write(*, "('a number: ', I3)") 12       ! -> |a number:  12|
```

- Falsche Datentypbeschreiber verursachen Laufzeitfehler:

```
write(*, "(I3)") "Try this!"            ! -> (run-time error)
```

# Kontrollfelder

- Kontrollfelder beinflussen nur I/O-Format, bearbeiten keinen Eintrag in der I/O-Liste

| | |
|---|---|
| **X** | Fügt Leerzeichen ein |
| **/** | Fängt neuen Datensatz (neue Zeile) an |
| **:** | Rest nur angewendet, wenn zum nächsten Formatbeschreiber ein entsprechender Eintrag in der I/O-Liste vorhanden. |

```
write(*, "(I2,I2)") 13, 42      ! -> |1342|
write(*, "(I2,3X,I2)") 13, 42   ! -> |13___42|

write(*, "(I2,2/,I2)") 13, 42   ! -> |13|
                                !    ||
                                !    ||
                                !    |42|


nn = 1
write(*, "('v1= ',I2,:,' v2=',I2)") array1(1:nn)  ! -> |v1= -1|
write(*, "('v1= ',I2,' v2=',I2)") array1(1:nn)    ! -> |v1= -1 v2=|
```

# Formatspezifikation

- * (automatische Formattierung):

```
write(*, *) 12.6
```

- Konstante Zeichenkette:

```
write(*, "(F10.3)") 12.6
```

- Charaktervariable

```
character(len=*), parameter :: formStr = '(F10.3)'
write(*, formStr) 12.6
```

```
character(len=7) :: cc
cc = "(F10.3)"
write (*, cc) 12.6
```

- Expliziter Format-Befehl über Anweisungsnummer (*statement label*)

```
      write (*, 112) 12.6, 22
      write (*, 112) 21.5, 14
      :
112 format(F10.3,I3)
```

- Anweisungsnummer müssen eindeutig sein
- Mehrere write-Befehle können auf selbe Anweisungsnummer hinweisen

# Formattierung beim Lesen

- Read-Befehl akzeptiert die selben Formatbeschreiber, wie write
- Formatbeschreiber beim Lesen können zu unerwünschten Effekten führen:

```
read(*, "(I3)") ii  ! <- |1234|
write(*,*) ii       ! -> |123|

read(*, "(F4.2)") xx  ! <- |-123|
write(*,*) xx         ! -> |-1.23|
```

```
read(*,*) ii      ! <- |1234|
write(*,*) ii     ! -> |1234|

read(*,*) xx      ! <- |-123|
write(*,*) xx     ! -> |-123.0|
```

Beim Lesen immer (wenn möglich) automatische Formatierung verwenden!

- Mit Formatbeschreiber A können Zeichenketten mit Leerzeichen eingelesen werden:

```
character(20) :: str

read(*,*) str           ! <- |test1 test2|
write(*,*) trim(str)    ! -> |test1|
read(*, "(A)") str      ! <- |test1 test2|
write(*,*) trim(str)    ! -> |test1 test2|
```

# Fehlerbehandlung beim Lesen

- Fehlerhafte Eingabe (z.B. falscher Datentyp) führt zum Programmabbruch

```
real(dp) :: rr

read(*,*) rr   ! <- |test| (run-time error)
write(*,*) rr
```

- Eingabestatus kann in einer Integer-Variable gespeichert werden.
  - 0 signalisiert Erfolg, alles andere Fehler
  - kein Abbruch bei Fehler

```
integer :: status
real(dp) :: rr

read(*,*, iostat=status) rr
if (status /= 0) then
  write(*,*) "Provided input is not a number"
else
  write(*,*) "You provided: ", rr
end if
```

# Interne Dateien (Zeichenketten)

Zeichenketten können Ziel eines write-Befehls oder Quelle einer read-Befehls sein:

```
character(20) :: str
write (str, "(I3,F8.2)") 12, 124.96_dp
write (*,*) trim(str)
```

```
character(20) :: str
integer :: ii
str = "128"
read(str, *) ii
write(*, *) ii
```

- Schreiben in interne Dateien kann u.a. dazu verwendet werden, Formatbeschreiber (oder andere Zeichenketten, die Nummer enthalten sollen) dynamisch zu erstellen:

```
character(20) :: str
integer :: width, nDecimal

width = 10
nDecimal = 6
write(str, "(A,I0,A,I0,A)") '(F', width, '.', nDecimal, ')'
write(*, str) 25.2_dp
```

```
str="(F10.6)"
! -> | 25.200000|
```

# Externe Dateien

- Beim Öffnen einer externen Datei muss der Datei eine Nummer (unit) zugeordnet werden
- Möglicher Intervall für Unit hängt vom Compiler ab (10-99 ist meistens sicher)
- Die Datei wird während der I/O Operationen durch diese Zahl identifiziert
- I/O Befehle ganz analog wie bei I/O auf Konsole
- Beim Schließen der Datei wird das Unit wieder freigegeben
  (kann z.B. einer anderer Datei wieder zugeordnet werden)

```
open(12, file="test.txt", status="new", form="formatted")
write (12, *) "Hello File!"
close(12)
```

**Wichtigste Optionen für open:**

```
action= "read" | "write" | "readwrite"
iostat= Statusvariable
file=   Dateiname
status= "old" | "new" | "replace" | "scratch"
access= "sequential" | "direct"
form=   "formatted" | "unformatted"
position= "asis" | "rewind" | "append"
```

**Wichtigste Optionen für close:**

```
iostat= status_variable
status= "keep" | "delete"
```

# Wichtige Optionen für open

- **file=** Dateiname
- **iostat=** Statusvariable (0 – OK, ungleich Null – Fehler)
- **form=**
  - **"formatted"** Formattiertes (ASCII) I/O
  - "unformatted" Nicht formattiertes (binäres) I/O
- **status=**
  - **"old"** Existierende Datei soll geöffet werden (wenn nicht vorhanden: Fehler)
  - "new" Nicht existierende Datei soll erzeugt werden (wenn vorhanden: Fehler)
  - **"replace"** Wenn Datei existiert, Datei ersetzen, ansonsten neu erzeugen.
  - "scratch" Temporäre Datei erzeugen (nach close wird automatisch gelöscht)
- **action=**
  - **"read"** Nur lesen
  - **"write"** Nur schreiben
  - **"readwrite"** Lesen und schreiben
- **position=**
  - "rewind" Datei-I/O nach dem Öffnen vom Anfang der Datei
  - "append" Datei-I/O nach dem Öffnen vom Ende der Datei

# Optionen für close

- **`iostat=`** Statusvariable (0 – OK, ungleich Null – Fehler)
- **`status=`**
  - `"keep"` Datei nach dem Schließen beibehalten (außer scratch-Datei)
  - `"delete"` Datei nach dem Schließen löschen

# Unformattierte E/A

- Daten werden in binärem Format gespeichert (wie im Arbeitsspeicher)

Vorteile:

- Kein Overhead wegen Konversion
- Verlustfreies Speichern
- Sehr kompakt
  (geignet für größere Datenmengen)

Nachteile:

- Nicht übertragbar zwischen Architekturen
- Datei kann nicht vom Benutzer editiert werden
- Datei kann von anderen (nicht Fortran) Programmen nicht gelesen werden

- Datei muss beim Öffnen explizit als „unformatted“ definiert werden
- I/O-Befehle genauso, wie beim formattierten I/O, aber ohne Formatangabe

```
! Writing the content of some variable to the disc.
open(18, file="test.bin", form="unformatted", status="replace", &
    &action="write")
write(18) ii, dummy
close(18)

! Reading back variables from the disc.
open(18, file="test.bin", form="unformatted", status="old", action="read")
read(18) ii,  dummy
close(18)
```

# Unformattierte E/A – Records

- Der write-Befehl schreibt den gesammten Inhalt der angegebenen I/O-Liste als ein Record.
- Der read-Befehl springt zum Anfang des nächsten Records, und liest von diesem Record, bis die angegebene I/O-Liste aufgefüllt ist.

**Falsch!**

```
write(18) i1, i2  ! |ii|
:
read(18) test     ! |i|
read(18) dummy    ! |i|
```

**Richtig**

```
write(18) i1, i2      ! |ii|
:
read(18) test, dummy  ! |ii|
```

**Falsch!**

```
write(12) dummy, dummy  ! |ii|
write(12) dummy, dummy  ! |ii|
:
read(12) (array(ii), ii=1,4)
! |iiii|
```

**Richtig**

```
write(12) dummy, dummy, &
    &dummy, dummy              ! |iiii|
:
read(12) (array(ii), ii=1,4)   ! |iiii|
```

Zusammengehörende read und write Befehle sollten äquivalente I/O-Listen haben.

# Aufgabe

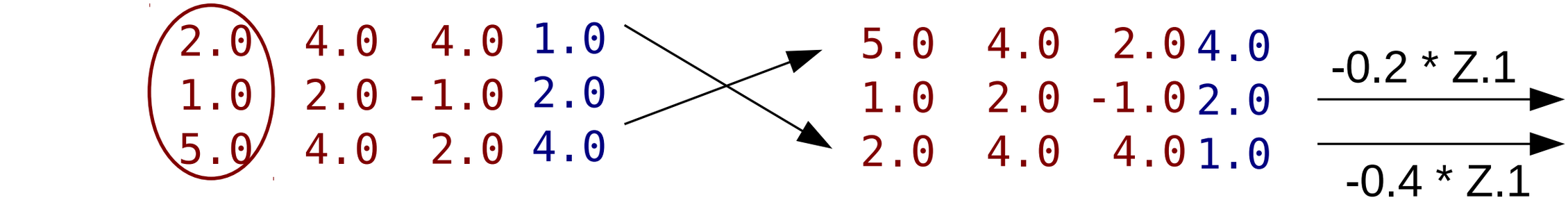
- Schreiben Sie ein Programm zur Lösung eines linearen Gleichungssystems (Ax=b) mittels Gauss-Elimination mit folgendem Programmablauf:
  1. Größe des Problems und Komponenten von A und b von der Konsole einlesen
  2. Gleichungssystem mittels Gauss-Elimination auf Dreieckform bringen. Dabei Zeilenpivot anwenden. (Aktuelle Zeile mit derjenigen Zeile vertauschen, die das Element mit dem größten Absolutwert in der aktuellen Spalte hat.)
  3. Es soll während der Umformung geprüft werden, dass keine Elemente mit Absolutwert kleiner als 1e-10 auf der Diagonale entstehen. Ist das der Fall (linear abhängiges Gleichungssystem), soll das Programm mit entsprechender Fehlermeldung abbrechen.
  4. Durch Zurücksubstitution die Komponenten von x berechnen (und speichern).
  5. Lösung auf die Konsole schreiben.
- Hinweis: Hilfreiche Funktionen (Für genaue Spezifikationen siehe Referenzhandbuch)

| | |
|---|---|
| `abs(xx)` | Absolutwert von `xx` (`xx` kann Skalar oder Array sein) |
| `maxloc(array, dim=iDim)` | Index des Maximalelements eines Arrays |

```
real(dp), allocatable :: tmp(:)
integer :: pos
:
pos = maxloc(tmp, dim=1)
```

# Gausselimination mit Zeilenpivot

- Matrixgleichung: A x = b
- Matrix mit Zeilenpivot in eine Obere-Dreiecksmatrix transformieren
  (rechte Seite muss mittransformiert werden)




- Zurücksubstituieren

**X3** = 1.5 /-3.0 = -0.5

**X2** = (-0.6 - 3.2 * **X3**) / 2.4 = 0.4166666...

**X1** = (4.0 - 2.0 * **X3** - 4.0 * **X2**) / 5.0 = 0.666666666...