Wissenschaftliches Programmieren

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

3. Felder, Speicherallokierung, Zeiger

Fibonacci-Reihe

```
f1 = 1
f2 = 1
write(*,*) 1, f1
if (limit >= 2) then
  write (*,*) 2, f2
end if
if (limit >= 3) then
  do ii = 3, limit
    ! Prevent overflow
    if (huge(tmp) - f1 < f2) then
      write(*,*)
      write(*,*) "Calculation stopped to prevent overflow!"
      exit
    end if
    tmp = f1 + f2
    f1 = f2
    f2 = tmp
    write(*,*) ii, f2
  end do
end if
```

Alternative Lösung:

```
if (huge(tmp) / 2 < f2) then
  :
  exit
end if
```

Überlaufüberprüfung

Allgemeine Bemerkungen

- Zwecks Portabilität immer sicherstellen, dass Quellcode dem Fortran 2003 Standard entspricht, und keine Spracherweiterungen benutzt! (z. B. Schleifen über reelle Variablen). Es sollten auch alle Warnungen eingeschaltet werden, damit einige schlechte Programmiertechniken angezeigt werden.

```
gfortran -std=f2003 -pedantic -Wall -o test test.f90
```

- Auf Lesbarkeit achten, insbesondere bezüglich Leerzeichen bei Operatoren

```
aa = aa+2.0_dp*bb+cc**(-32.0_dp-sqrt(dd))
```

```
aa = aa + 2.0_dp * bb + cc**(-32.0_dp - sqrt(dd))
```

- Zeilenumbruch sollte an „logischer“ Stelle Erfolgen, möglich außerhalb von Klammern. Fortsetzungszeile sollte möglich mit einem Operator anfangen.

```
aa = aa + 2.0_dp * bb + cc**(-32.0_dp -&
    &sqrt(dd))
aa = aa + 2.0_dp * bb &
    & + cc*(-32.0_dp - sqrt(dd))
```

- Größere Projekte haben meistens einen „style guide“, woran man sich halten sollte.

Felder

Datenfelder (indizierte Variablen) oder Arrays:

- Vektoren, Matrizen, logisch zusammengehörende Werte (Tabellen)
- Alle Einträge in einem Array haben den gleichen Typ
- Feldtyp kann jeder intrinsische oder abgeleiteter Typ sein (aber kein Pointer)
- Maximaler **Rang** (Anzahl der Dimensionen): **7**

Deklaration:

```
real(dp), dimension(3) :: vector
real(dp), dimension(3,3) :: matrix
integer, dimension(2,4,6) :: test
```

oder

```
real(dp) :: vector(3)
real(dp) :: matrix(3,3)
integer :: test(2,4,6)
```

Elemente schreiben/auslesen:

```
vector(1) = 12.0_dp
vector(2) = 2.0_dp
matrix(2,1) = 3.0_dp
test(1,2,3) = -5
```

```
vector(1) = matrix(1,1) * vector(1)  &
    & + matrix(1,2) * vector(2) &
    & + matrix(1,3) * vector(3)
test(2,1,1) = test(1,1,1) + 1
```

Indizes fangen mit **1** an (außer explizit anders deklariert)!

Felddeklarationen

Indexgrenzen können bei Deklaration festgelegt werden

(**Beide Indexgrenzen sind im Indexbereich enthalten!**)

```
real(dp) :: vec1(3)                  ! vec1(ii)  ii = 1..3
real(dp) :: vec2(0:9)                ! vec2(ii)  ii = 0..9
real(dp) :: mtx(-5:12,-3:1,0:10)     ! mtx(kk,jj,ii)  kk = -5..12,
                                     ! jj = -3..1, ii = 0..10
```

Feldformabfragen:

- **shape(*array*)**
  Feldform
- **size(*array* [,*dim*])**
  Größe des Feldes (gesammten oder entlang einer Dimension)
- **lbound(*array* [,*dim*])**
  untere Indexgrenze(n)
- **ubound(*array* [,*dim*])**
  obere Indexgrenze(n)

```
write(*,*) shape(vec2)          ! 10
write(*,*) shape(mtx)           ! 18 5 11
write(*,*) size(vec2)           ! 10
write(*,*) size(mtx)            ! 990 (18*5*11)
write(*,*) size(mtx, dim=1)     ! 18
write(*,*) size(mtx, 2)         ! 5

write(*,*) lbound(vec2)         ! 0 (1D Array)
write(*,*) lbound(mtx)          ! -5 -3 0 (1D)
write(*,*) lbound(mtx, dim=1)   ! -5 (scalar)

write(*,*) ubound(mtx)          ! 12 1 10 (1D)
write(*,*) ubound(mtx, dim=3)   ! 10 (scalar)
```

Anordnung der Felder im Speicher

- Elemente eines Arrays werden in Fortran (im Gegensatz zu C, C++, Python etc.) im Speicher **Spaltenweise** (***column-major***) abgelegt:

erster Index wächst **am schnellsten**
letzter Index wächst **am langsamsten**

```
real(dp) :: array1(3,2)
```

Speicher ↓

```
#ffffff00   array1(1,1)
#ffffff08   array1(2,1)
#ffffff10   array1(3,1)
#ffffff18   array1(1,2)
#ffffff20   array1(2,2)
#ffffff28   array1(3,2)
```

```
real(dp) :: array2(3,2,2)
```

Speicher ↓

```
#ffffff00   array2(1,1,1)
#ffffff08   array2(2,1,1)
#ffffff10   array2(3,1,1)

#ffffff18   array2(1,2,1)
#ffffff20   array2(2,2,1)
#ffffff28   array2(3,2,1)

#ffffff30   array2(1,1,2)
#ffffff38   array2(2,1,2)
#ffffff40   array2(3,1,2)

#ffffff48   array2(1,2,2)
#ffffff50   array2(2,2,2)
#ffffff58   array2(3,2,2)
```

Reihenfolge der Feldbearbeitung

- Aus Effizienzgründen bei verschachtelten Schleifen Feldelemente möglichst **in derselben Reihenfolge** bearbeiten, **wie** sie **im Speicher** liegen. (Cache!)

```
real(dp) :: array1(3,2)
integer :: iCol, iRow

do iCol = 1, 2
  do iRow = 1, 3
    array1(iRow, iCol) = ...
  end do
end do
```

```
real(dp) :: array2(3,2,2)
integer :: i1, i2 i3

do i1 = 1, 2
  do i2 = 1, 2
    do i3 = 1, 3
      array2(i3, i2, i1) = ...
    end do
  end do
end do
```

- Felder solten entsprechend angelegt bzw. aufgefüllt werden:

```
real(dp) :: latVecs(2, 2), dot
integer :: ii

latVecs(1,1) = 1.0_dp                ! a1x
latVecs(2,1) = 0.0_dp                ! a1y
latVecs(1,2) = -0.5_dp               ! a2x
latVecs(2,2) = sqrt(3.0_dp)/2.0_dp !a2y

! Dot product. Just for demonstration.
! Use dot_product() in production!
dot = 0.0_dp
do ii = 1, 2
  dot = dot + latVecs(ii, 1) &
      & * latVecs(ii, 2)
end do
```

Statische Allokierung

- Wird die **Felddimensionen** bei der Deklaration **angegeben**:
  Speicher **beim Eintreten** in (Unter-)Programmblock automatisch **allokiert.**
  Speicher **beim Austreten** aus (Unter-)Programmblock automatisch **freigegeben.**

```
program test
  implicit none

  integer, parameter :: dp = &
      &selected_real_kind(15, 99)
  real(dp) :: latVecs(3, 3)

  latVecs(1, 1) = 1.0_dp

end program test
```

- Speicherbedarf: 9 x Größe von real(dp)
- Wird automatisch vor der Ausführung der ersten Anweisung allokiert.
- Speicher schon allokiert, Array kann bearbeitet werden.
- Speicher automatisch wieder freigegeben.

Vorteil:
- Allokierung/Deallokierung passiert automatisch

Nachteil:
- Statische Allokierung funktioniert nicht, wenn Feldgröße erst während der Laufzeit bekannt.

Dynamische Allokierung (#1)

Dynamische Allokierung ist notwending, wenn:

- **Araygröße** erst **zur Laufzeit ermittelt** bekannt

```
program DynamicAllocation
  implicit none

  integer, parameter :: dp = 8

  real(dp), allocatable :: array(:,:)
  integer :: arraySize1, arraySize2

  write (*,*) "Enter the sizes:"
  read (*,*) arraySize1, arraySize2
  allocate(array(arraySize1, arraySize2))
  array(1, 1) = 12.0_dp
  :
  deallocate(array)  ! You can omit this

end program DynamicAllocation
```

- Allokierbare Felder haben das **allocatable** Attribut.
- Anzahl der Arraydimensionen wird bei der Deklaration festgelegt
- Allokierbare Arrays müssen vor dem ersten Gebrauch mit dem **allocate** Befehl reserviert werden
- Speicher kann explizit mit dem **deallocate** Befehl freigegeben werden. Ansonsten wird es nach Verlassen des Blockes, wo es deklariert wurde, automatisch deallokiert.

Vor der Allokierung bzw. nach der Deallokierung darf auf das Feld nicht zugegriffen werden!

Dynamische Allokierung (#2)

- Allokierbare Arrays dürfen **nach der Deallokierung** wieder neu allokiert werden:

```
allocate(array(12, 12))
array(1, 1) = -12.0_dp
:
deallocate(array)
:
allocate(array(3, 5))
:
deallocate(array)
```

- **Nach** jeder **Allokierung** müssen die **Feldelemente** (neu) **aufgefüllt** werden. Wird ein Feldelement vor dessen Initialisierung verwendet, ist das Ergebnis unberechenbar!

```
allocate(array(12, 12))
write (*,*) array(1,1)! ???
```

- Allokierungsstatus eines **allokierbaren** Feldes wird mit **allocated** abgefragt:

```
if (.not. allocated(array)) then
  allocate(array(10, 20))
end if
```

- Noch nicht oder nicht mehr allokierte Arrays dürfen nicht deallokiert werden:

```
allocate(array(100,10))
:
deallocate(array)
:
! This will probably crash
deallocate(array)
```

Dynamische Allokierung (#3)

- Unnötige Allokierung/-deallokierung (insbesonders in häufig ausgeführten schnellen Schleifen!) sollte aus Effizienzgründen vermieden werden:

```
! Avoid this if possible!
do ii = 1, 100000
  allocate(array(100,100))
  array(1,1) = -12.0_dp
  :
  deallocate(array)
end do
```

- Indexbereich der Arrays kann bei der Allokierung frei gewählt werden

```
real(dp), allocatable :: array1(:,:,:), array2(:,:,:)

allocate(array1(10, 10, 10))
! equiv. definition: allocate(array1(1:10, 1:10, 1:10))
allocate(array2(-5:12, 20:30, -10:-8))
```

- Anzahl der Arraydimensionen kann bei der Allokierung nicht verändert werden:

```
integer, allocatable :: array(:,:)
allocate(array(3, 3, 3))                ! Compiler error
```

Dynamische Allokierung (#4)

- Wird der Status bei der Allokierung nicht abgefragt, bricht das Programm bei einer erfolglosen Allokierung ab.

```
real(dp), allocatable :: testArray(:,:)
integer :: arraySize

read (*,*) arraySize
! The program may crash here!
allocate(testArray(arraySize, arraySize))
```

```
real(dp), allocatable :: testArray(:,:)
integer :: arraySize, allocStat

read (*,*) arraySize
! Testing for allocation errors
allocate(testArray(arraySize, arraySize), stat=allocStat)
if (allocStat /= 0) then
  ! Allocation failed. Cleanup, notify user and STOP program
  :
end if
```

- **Statusabfrage:**
  `allocate(`*array*`,`**`stat=`*`status`**`)`
- **Allokierungsstatus (integer):**

`0`	Erfolgreich
alles anderes	Fehler!

Dynamische Allokierung (#5)

Ab Fortran 2003 werden dynamische Arrays auf der rechten Seite einer Zuweisung automatisch reallokiert:

```
integer, allocatable :: a1(:,:), a2(:,:)
allocate(a1(3, 2))
allocate(a2(2, 2))
a1(:,:) = -1
a2 = a1      ! deallocates a2, reallocates it to the shape
             ! shape of a1 and copies content of a1 into
             ! a2
print *, shape(a2)   ! 3 2
```

Wenn rechts ein Subarray-Ausdruck erscheint, muss **das Array schon vorher auf die richtige Größe allokiert** worden sein und es wird nicht reallokiert.

```
integer, allocatable :: a1(:,:), a2(:,:)
allocate(a1(2, 2))
allocate(a2(2, 2))
a1(:,:) = -1
a2(:,:) = a1     ! Does NOT change the allocation status of a2
print *, shape(a2)
```

Effizienter als a2 = a1, da keine Reallokierung

Initialisierung von Datenfelder (array constructors)

- **Statische** Arrays können bei Deklaration mit **Feldkonstruktoren** initialisiert werden:

```
real(dp), parameter:: eX(3) = (/ 1.0_dp, 0.0_dp, 0.0_dp /)   ! F95
real(dp), parameter :: eX(3) = [1.0_dp, 0.0_dp, 0.0_dp]      ! F03
```

- **Feldkonstruktoren** sind immer **eindimensionale** Felder, müssen für die Initialisierung von mehrdimensionalen Felder mit **`reshape`** umgeformt werden:

```
real(dp), parameter :: latVecs(2, 2) = &
    &reshape([1.0_dp, 0.0_dp, 0.0_dp, 1.0_dp], [2, 2])
```

Elemente als 1D-Array in der Reihenfolge, wie sie im Speicher abgelegt werden sollen.

Feldform als 1D-Array

- Feldkonstruktoren können auch im Anweisungsteil verwendet werden:

```
real(dp) :: latVecs(2,2)

latVecs(:,:) = reshape([1.0_dp, 0.0_dp, 0.0_dp, 1.0_dp], [2, 2])
```

Feldgrenzen, Grenzenprüfung

Es dürfen nur auf Feldelemente zugegriffen werden, die innerhalb des definierten Indexbereiches liegen!

Falsch!

```
real(dp) :: array(-3:10)
do ii = 1, 14
  ! Error if accessing array(11)
  array(ii) = ..
  :
end do
```

Richtig

```
real(dp) :: array(-3:10)
do ii = -3, 10
  ! OK, all elements exist
  array(ii) = ..
  :
end do
```

- Indizeriungsfehler werden beim Compilieren fast nie erkannt werden!
  Unberechenbares Verhalten, segmentation fault
- Die meisten Compiler haben eine Option zur Laufzeitprüfung (*bound check*)
  Macht das Programm langsam, deshalb nur fürs Debuggen verwenden!

```
> gfortran -fbounds-check -o badbound badbound.f90
> ./badbound
At line 10 of file badbound.f90
Fortran runtime error: Array reference out of bounds for array 'array'
upper bound of dimension 1 exceeded (11 > 10)
```

Zugriff auf Datenfelder als Ganzes – arithmetische Ausdrücke

- **Wertzuweisung mit einem Skalarwert:** jedes Feldelement dem Skalar gleich

```
! real(dp) :: array(10, 10)
array(:,:) = -3.0_dp
```

```
do i1 = 1, 10
  do i2 = 1, 10
    array(i2, i1) = -3.0_dp
  end do
end do
```

- **Anwendung eines Skalaroperators:** Operation für jedes Feldelement aus

```
! real(dp) :: array2(10, 10)
array2(:,:) = array1 * 0.5_dp
```

- **Anwendung einer Skalarfunktion:** Wendet die Funktion auf jedes Feldelement an

```
array2(:,:) = sin(array)
```

- **Anwendung eines binären Skalaroperators auf zwei Felder:** Elementenweise

```
! real(dp) :: array3(10, 10)
! NOT a matrix multiplication
array3(:,:) = array1 * array2
```

```
do i1 = 1, 10
  do i2 = 1, 10
    array3(i2, i1) = array1(i2, i1) &
         & array2(i2, i1)
  end do
end do
```

Operanden und Ergebnis müssen die selbe Form haben!

Zugriff auf Datenfelder als Ganzes – logische Audrücke

- Vergleich zweier Felder oder eines Feldes und eines Skalars liefert ein logisches Array (**Maske**) der selben Größe mit dem Ergebnis des elementenweisen Vergleiches.
- **any**, **all** und **count** können die entsprechende Maske auf einen Wert reduzieren:

all(*Maske*)	.true. wenn jedes Element in *Maske* .true., ansonsten .false.
any(*Maske*)	.true. wenn mindestens ein Element in *Maske* .true., sonst .false.
count(*Maske*)	Anzahl der .true. Elemente in *Maske.*

```
real(dp) :: array1(100)                    [.true., .true., .false., …]
:
if (all(array1 < 0.0_dp)) then
  write (*, *) "Every element below zero"
else if (any(array1 < 0.0_dp)) then
  write (*, *) count(array1 < 0.0_dp), "elements below zero"
else
  write (*, *) "No elements below zero."
end if
```

Teilbereiche von Felder (subarrays)

- Teilbereiche von Felder können mit dem `:` Operator ausgewählt werden. Die Teilbereiche können dann wie normale Felder verwendet werden.

```
integer :: array(10, 10), array2(10, 10)
integer :: vector(10)
:
! Set first five columns 1
array(:,1:5) = 1

! Multiply every even row by 2
! array(2:10:2,1:10) = array(2:10:2,1:10) * 2
array(2::2,:) = array(2::2,:) * 2

! Put 6th column into vector
vector(:) = array(:,6)
```

Teilbereich auswählen:

`[`*Start*`]:[`*Ende*`][:`*Schrittweite*`]`

- Teilbereiche können mit 1D-Index-Array ausgewählt werden:

```
! integer :: indices(3)
indices = [ 1, 7, 9 ]
vector(indices) = 1.0_dp                ! Set elements 1, 7, 9 to 0
write (*,*) vector([2, 9, 6, 2])        ! Print elements 2, 9, 6 and 2
```

Zeiger als Alias

- Wird ein Teilbereich häufig verwendet, kann ein Zeiger als Alias verwendet werden.
- Der Zeiger muss vom selben Typ sein, wie das Feld, auf das er zeigt
- Der Zeiger muss den selben Rang haben, als das Teilfeld, auf das er zeigt
- Der Zeiger muss mit dem **`pointer`** Attribut definiert werden.
- Das Feld, auf das gezeigt wird, muss mit dem **`target`** Attribut deklariert werden.
- Die Zuweisung wird mit dem **`=>`** Operator definiert
- Die Zuweisung solange gültig, bis der Zeiger nicht etwas anderem zugewiesen wird.
- Zeiger nur Alias, Operationen werden direkt an Feldkomponenten ausgeführt.

```
real(dp), target :: latVecs(3, 3)
real(dp), pointer :: vec1(:), vec2(:)
real(dp) :: dot

vec1 => latVecs(:, 1)
vec2 => latVecs(:, 2)
vec1(:) = 1.0_dp                          ! First lattice vector 1, 1, 1
vec2(:) = [0.0_dp, 2.0_dp, -1.0_dp]  ! 2nd lattice vector 0, -2, -1
vec2 => latVecs(:, 3)                  ! vec2 shows to 3rd lattice vec.
vec2(:) = vec1 * 0.5_dp                   ! 3rd lattice vec 0.5, 0.5, 0.5
dot = dot_product(vec1, vec2)      ! Dot of 1st and 3rd lattice vec.
```

Arraymanipulationsfunktionen

dot_product(*v1*, *v2*)	Skalarproduct zwei 1D Arrays
matmul(*array1*, *array2*)	Matrix-Matrix- bzw. Matrix-Vektor-Produkt
transpose(*array*)	Transponierte einer Matrix
maxval(*array*, [, *dim*])	Maximum einer Matrix (oder entlang einer Dimension)
minval(*array*, [, *dim*])	Minimum einer Matrix (oder entlang einer Dimension)
sum(*array*, [, *dim*])	Summe der Elemente
product(*array*, [, *dim*])	Produkt der Elemente
maxloc(*array,* [, *dim*])	Indizes des Maximalelementes (Array)
minloc(*array,* [, *dim*])	Indizes des Minimalelementes (Array)
cshift(*array*, *shift*)	Zyklische Verschiebung der Elemente
eoshift(*array*, *shift*)	Verschiebung der Elemente mit Abschneiden
spread(*array*, *dim*, *ncopies*)	Neues Array mit einer zusätzlichen Dimension aufgefüllt mit Kopien von *array*.

- Für Details siehe z.B. **Referenzhandbuch** http://www.lahey.com/docs/lang_revg.pdf

where Befehl

- Wenn nur diejenigen Elemente eines Feldes manipuliert werden sollen, die eine bestimmte Bedingung erfüllen:

```
where (logischer-Array-Ausdruck)
  Array-Zuweisungen
elsewhere
  Array-Zuweisungen
end where
```

optional

```
! real(dp) :: a(10, 10)
where (a > 0.0_dp)
  a = log(a)
end where
```

- where Befehl erstellt eine Maske, welche Elemente bearbeitet werden können
- Diese Maske kann auf jedes Feld angewendet werden, das die selbe Form hat

```
integer, parameter :: nPoint = 100
real(dp) :: deriv(nPoint)
logical :: calcDeriv2(nPoint)
:
where (abs(deriv) < 1e-10_dp)
  deriv = 0.0_dp
  calcDeriv2 = .true.
elsewhere
  calcDeriv2 = .false.
end where
```

Die kleinen Elemente von `deriv` werden ausgenullt. Die selben Elemente von `calcDeriv2` werden auf `.true.` gesetzt.

Die anderen Elemente von `calcDeriv2` werden auf `.false.` gesetzt.

forall Befehl

- Wenn die Reihenfolge der Zugriffe auf die Elemente eines Feldes in einer do Schleife nicht wichtig ist, kann eine parallele Ausführung effizient sein.
- Der forall Befehl teilt dem Compiler mit, dass parallele Ausführung erlaubt ist.
- Effizienz hängt vom Compiler und Prozessor ab.

```
forall (Index=Start:End[:Schritt] [, Index2=...] [, log. Ausdruck])
  Anweisungen
end forall
```

```
integer, parameter :: nn = 100
real(dp) :: aa(nn, nn), bb(nn, nn)
integer :: ii, jj

! Every internal element = sum of its neighbors
! Result also copied to bb
forall (ii = 2:nn-1, jj = 2:nn-1)
  aa(ii, jj) = aa(jj+1, ii) + aa(jj-1, ii) &
      &+ aa(jj, ii+1) + aa(jj, ii-1)
  bb(jj, ii) = a(jj, ii)
end forall
```

- Reihenfolge beliebig.
- Von aa wird eine temporäre Kopie erstellt.

Abschließende Bemerkungen

Theoretische Bemerkungen:

- Wenn möglich, `do` Schleifen für Felder vermeiden
- Möglichst Array- und Subarray-Ausdrücke verwenden
  (besser lesbar, theoretisch effizienter)
- Komplexere Aufgaben können mit `where` und `forall` gelöst werden
  (besser lesbar, theoretisch effizienter)
- Intrinsiche Funktionen statt do Schleifen verwenden
  (`transpose`, `dot_product`, `matmul`)

Praktische Bemerkungen:

- Subarray-Ausdrücke sehr effizient für Bereiche, die im Speicher kontinuierlich liegen (kein Kopieren der Bereiche notwendig). Arrays also versuchen, immer entsprechend anzulegen.
- Manche Compiler setzen `where` und besonders `forall` eventuell in langsameren Code um als für eine äquivalente do Schleife!

Aufgabe 1

- Ändern Sie ihr Fibonacciprogramm so, dass dieses „verallgemeinerte“ Fibonacci-Reihen beliebiger Ordnung erzeugen kann:
  - Dabei wird jeder Term als Summe beliebig vieler vorangehenden Terme (Ordnung) berechnet. Die Anfangsterme sollen alle 1 sein.
  - Bsp: Fibonacci Reihe der Ordnung 3 wäre:

    1, 1, 1, 3 (1+1+1), 5 (1+1+3), 9 (1+3+5), ...
- Die Ordnung der Reihe (`order`) soll von der Konsole eingelesen werden.
- Die Anzahl der zu berechnenden Terme (`nterm`) soll auch von der Konsole eingelesen werden.
  - Das Programm soll auch dann richtig funktionieren, wenn `nterm` kleiner als `order` ist!
- Es soll ein Feld der Größe `nterm` allokiert und mit den einzelnen Glieder der Folge aufgefüllt werden. Erst nach Auffüllen des Feldes sollen die Elemente auf den Bildschirm ausgegeben werden.
- Zur Summierung der einzelnen Terme sollten Sie die Funktion `sum()` verwenden.
- Testen Sie das Programm mit eingeschalteter Grenzenüberwachung (*bound check*).
- Implementieren Sie die Überlaufüberwachung (*overflow*) für den verallgemeinerten Fall.

Aufgabe 2

• Schreiben Sie Ihr verallgemeinertes Fibonacci-Program so um, dass unabhängig von der Anzahl der zu addierenden Terme (Ordnung) maximal zwei arithmetische Operationen (*, -, /, +) verwendet werden, um den nächsten Term zu berechnen.

Aufgabe 3 (optional)

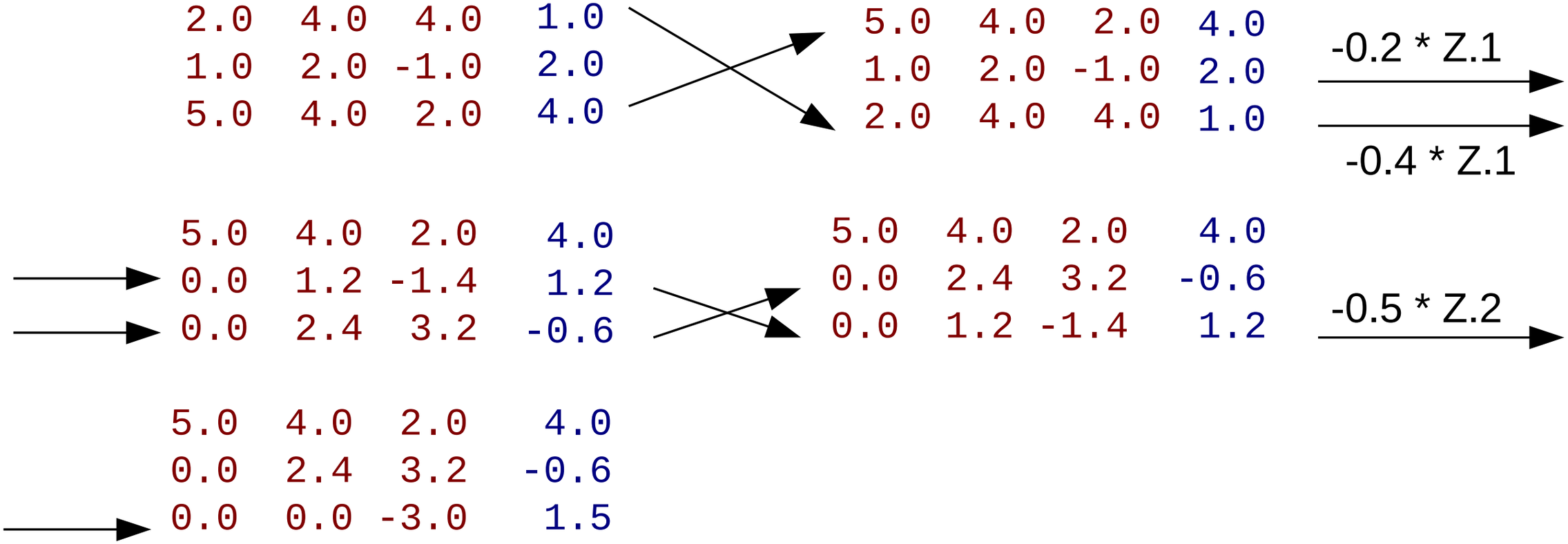
- Schreiben Sie ein Programm zur Lösung eines linearen Gleichungssystems (Ax=b) mittels Gauss-Elimination mit folgendem Programmablauf:
  1. Größe des Problems und Komponenten von A und b von der Konsole einlesen
  2. Gleichungssystem mittels Gauss-Elimination auf Dreieckform bringen. Dabei Zeilenpivot 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 10e-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 Ref.handbuch)

`abs(xx)`	Absolutwert von `xx` (`xx` kann Skalar oder Array sein)
`maxloc(array, dim=iDim)`	Index des Maximalelements eines Arrays entlang der angegebenen Dimension (1-Zeile oder 2-Spalte)

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...
```