Einführung in die Numerische Programmierung mit dem Schwerpunkt Fortran 90

Georg KRESSE, Robert LORENZ, Andreas EICHLER

Institut für Materialphysik and Center for Computational Material Science

Universität Wien, Sensegasse 8, A-1090 Wien, Austria

Literature

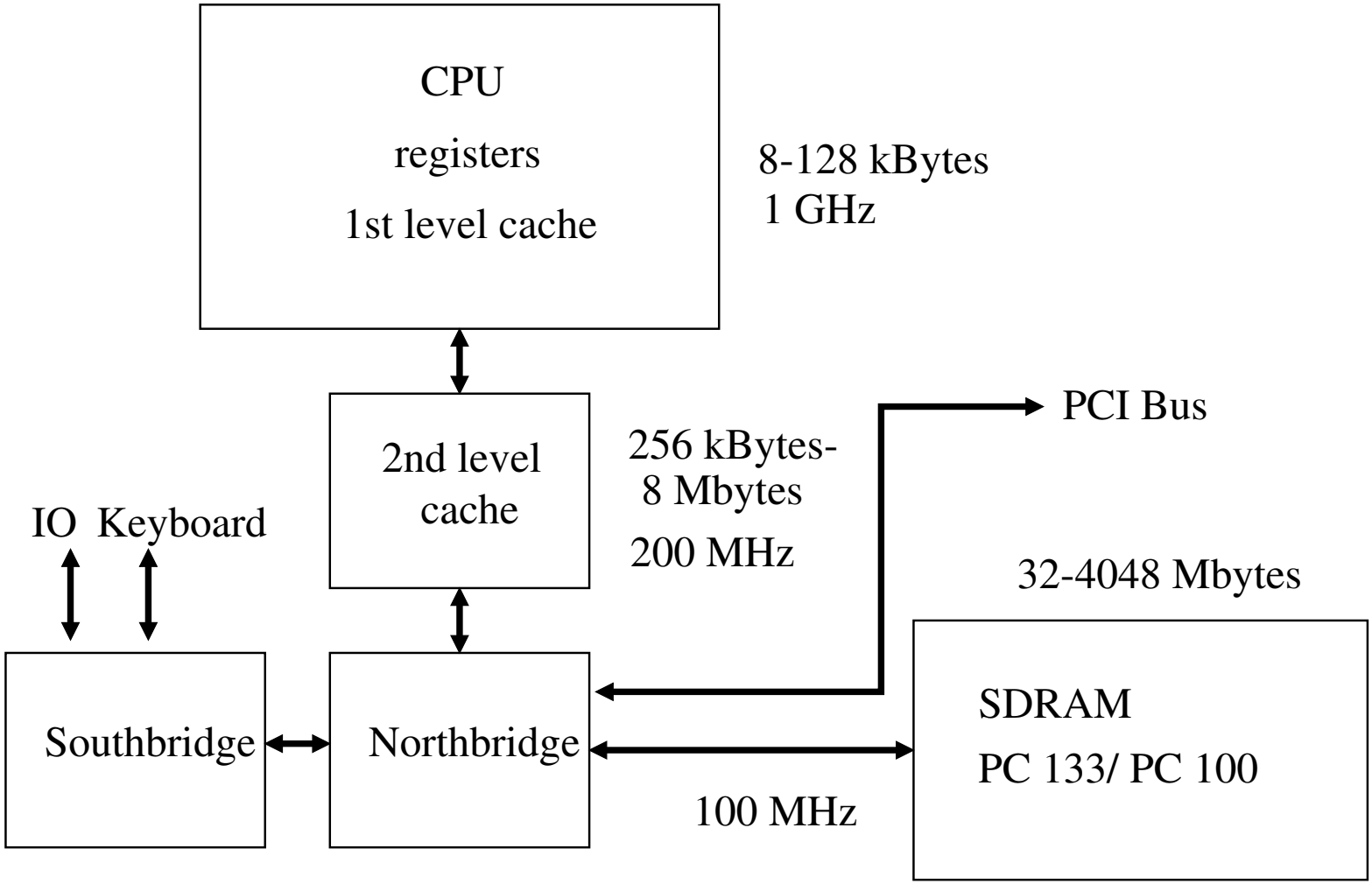
Fortran 90 explained
Michael Metcalf, John Reid, Oxford University Press

Fortran 90
Regionales Rechenzentrum für Niedersachsen RRZN

Fortran 90 programming
T.M.R. Ellis, Ivor R. Philips, Thomas M. Lahey, Addison-Wesley

An Introduction to Computer Simulation Methods
H. Gould, and J. Tobochnik, Addison-Wesley

Schematische Darstellung eines Computers



Repräsentation von Zahlen

Computer benutzt eine binäre Darstellung mit nur zwei Zustände 0 und 1

Bit	1 Speicherzelle	
Byte	8 Bits	0-255
Word	4 Bytes	real, Integer
long Word	8 Bytes	Double
1 KByte	1024 Bytes	
1 MByte	1024×1024	Bytes

kleinste adressierbare Einheit ist immer ein Byte (bei meisten modernen Computern sogar ein Wort)

Speicher ist eine Anordnung von Bytes (meist linear)

Darstellung von ganzen Zahlen

binäre Darstellung von ganzen Zahlen

323 =

					0	0	1	
					2^8			

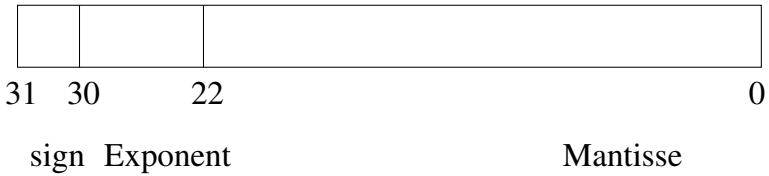
0	1	0	0	0	0	1	1	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	

Byte	0-255
2 Bytes	0-65535
Word	0-42949672961

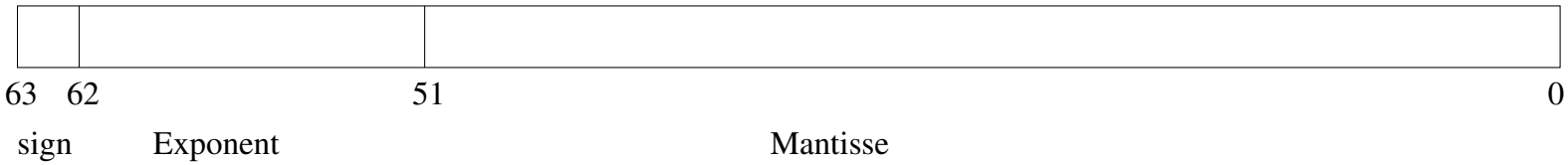
Darstellung von Fließkommazahlen

meisten Computer verwenden heute den IEEE Standard

float or REAL 1 word 4 bytes 32 bits



double 2 word 8 bytes 64 bits



Befehle

man unterscheidet grundsätzlich CISC und RISC Befehlssätze

CISC complex instruction set

RISC reduced instruction set

C=A*B

RISC: jeder Befehl 32 oder 64 Bit breit

CISC: 8,16 und 32 Bit breite Befehle

LOAD	A, register1
LOAD	B, register2
MUL	reg1, reg2, reg3
STORE	register3, C

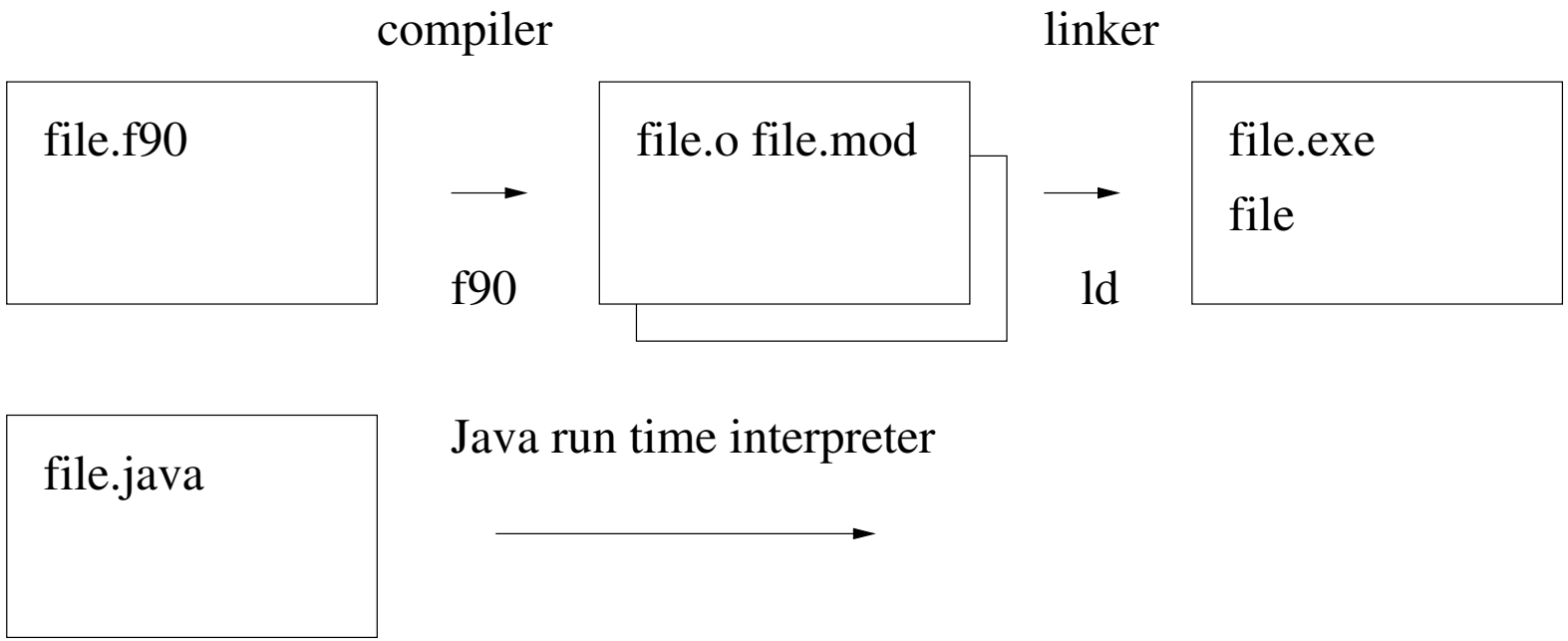
CISC: spart Speicherplatz, benötigt mehr “Silicon” auf der CPU

RISC: einfacher in “Silicon” zu gießen

Programmiersprachen

low level	high level
ASSEMBLER	C, Java, Fortran, Pascal, Basic
beste Performance	moderne Compiler

Compiler versus Interpreter



Sprachkonzepte

- Prozedurale Sprachen:

```
CALL calc_fact( a, fact)        SUBROUTINE calc_fact( a, fact)
```

  gefährlich, da das Unterprogramm jede Variable verändern darf (side effects)
  für große Projekte kaum verwendbar

- Funktionales Programmieren:

  Funktionen können niemals übergebene Werte verändern

  die einzige Methode um Ergebnisse zurückzuliefern ist das Ergebnis der Funktion selber

```
fact = calc_fact( a)        FUNCTION calc_fact( a)
```

  keine side effects $\Rightarrow$ völlig deterministisch

Modulares Programmieren

- Modulares Programmieren:

  auf dem Weg zur objektorientierten Programmierung
  faßt Prozeduren zusammen die auf irgendeine Weise zusammenhängen

  z.B. Prozeduren, die auf einen bestimmten Datensatz angewendet werden, befinden sich in einem Modul

  auf diesen Datensatz darf nur von diesen Prozeduren zugegriffen werden

- Objektorientiertes Programmieren:

  Datensatz ist das zentrale Objekt
  Zugriff auf das Objekt erfolgt nur mit Hilfe von Methoden (Prozeduren, die für dieses Objekt definiert sind)

  Vererbung

Wieso Fortran

History

1956	erste high level Sprache (enormer Erfolg)
1966	Fortran 66 (wichtige neue Konzepte)
1977	Fortran 77 (vereint viele Fortran Dialekte)
1990	Fortran 90 (moderne modulare Sprache, Sourcef.)

Fortran 90:

alle modernen Sprachelemente für prozedurale Sprachen sind vorhanden
vor allem Module und Strukturen

sehr mächtige Sprachelemente für Matrizen
für numerische Programmierung noch immer die Standardsprache

Let's get started

Einfaches Beispielprogramm in F90 und C

```
PROGRAM conversion1
  REAL :: a, result
  WRITE(*,*) "convert cal to Joules"
  READ(*,*) a   ! read in a number
  result = a * 4.186
  WRITE(*,"(F10.5)") result
END PROGRAM conversion1
```

```
main() {
  double a, result ;
  printf("convert cal to Joules\n") ;
  scanf("%10.5lf", &a) ;

  result = a * 4.186
  printf("%10.5lf", sum) ;
}
```

Sprachelemente von F90: Basic

ein F90 Programm darf nur Zeichen aus dem F90 Zeichensatz enthalten

- Buchstaben A ... Z und a ... z,
- Ziffern 0 ... 9,
- underscore “_”
- spezielle Zeichen

```
=  :  +  blank  -  *  /  (  )  ,  .  $  ’ (old)
!  "  %  &        ;  <  >  ?              (new)
```

Tokens (Worte) bestehen aus mehreren Zeichen

mehrere Tokens bilden eine Anweisung (Satz)

in F90 sind kleine und große Buchstaben außerhalb eines Strings ident

Sprachelemente von F90: Basics

F90 unterscheidet sechs Klassen von Tokens

```
Labels:      123              (1-5 numerals, not recommened)
Constants:   123.456789       "convert"   'hello world'
Keywords:    PROGRAM
Operators:   + - * / **   .AND.   .OR.    .NOT.
Names:       solve_equation (up to 31 characters,
                             including underscore '_')
Separators:  /   ( )   (/   /)   ,    =    =>    :    ::    ;    %
```

in F90 gibt es nur einen Namensbereich (name space)

Funktionen und Variablen müssen unterschiedliche Namen haben

Source Format und Statements

- 132 Zeichen pro Zeile
- Ausrufezeichen startet Kommentare “!”
  Kommentare gehen immer bis zum Zeilenende
- Semikolons “;” und Zeilenumbrüche trennen Statements
- nach einem Kaufmannsund “&” geht ein Statement in der folgenden Zeile weiter (die folgende Zeile kann mit “&” beginnen)

```
x= y   &   !  continue on next line
     * 12 ;  z =  y  * 24  ! line continues here
x= y   &   !  continue on next line
   &  * 12 ;  z =  y  * 24  ! line continues here
```

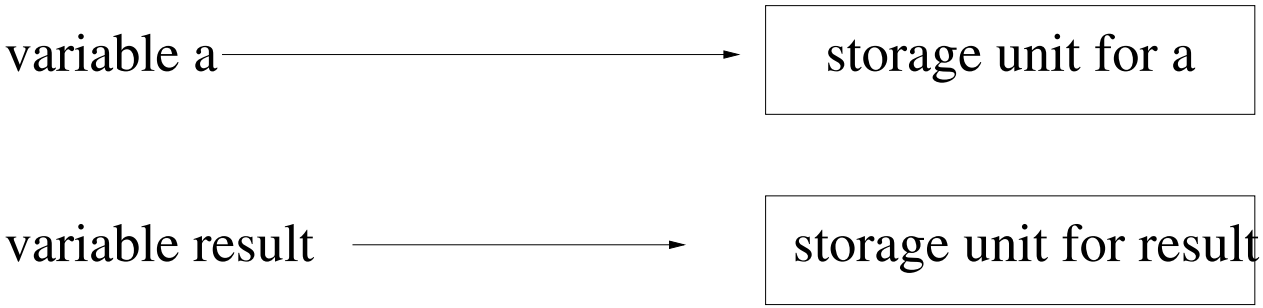
Spezifikations Anweisungen (specifications statements)

```
PROGRAM conversion
  REAL :: a, result
 ...
END PROGRAM conversion
```

die `PROGRAM` Anweisung zeigt den Beginn eines F90 Programms an

das `REAL` Statement deklariert die Variablen `a` und `result`

jede Variable zeigt auf eine Speicherstelle; eine Variable vom Typ `REAL` kann eine Fließkommazahl speichern



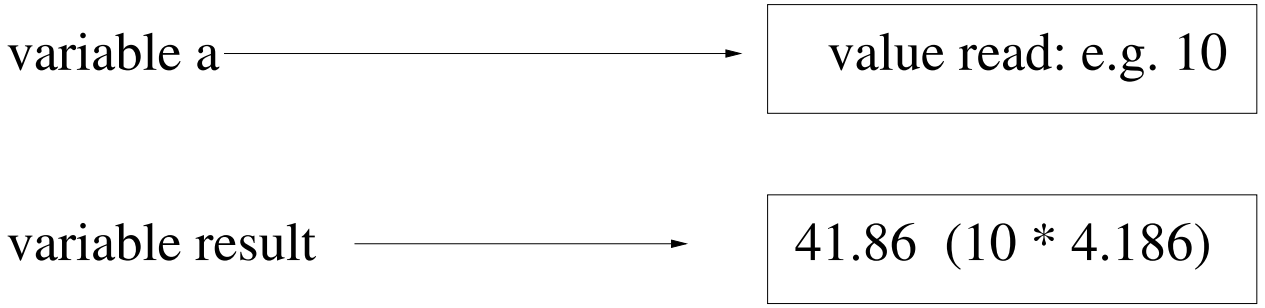
in F90 müssen alle Spezifikationsanweisungen vor ausführbaren Anweisungen stehen

Zuweisungen (Assignment)

in einer Zuweisung wird einer Variablen ein neuer Wert “zugewiesen”

im unserem Beispielprogramm wird `result` auf den Wert der Variablen `a` mal 4.186 gesetzt

```
result = a * 4.186
```



Symbolische Konstanten (named constant)

anstelle einer numerischen Konstante kann man auch symbolische Konstanten verwenden

```
PROGRAM conversion2
  REAL :: a, b, result
  REAL, PARAMETER :: cal_to_joule=  4.186
  WRITE(*,*) "convert cal to Joules"
  READ(*,*) a
  result= a * cal_to_joule
  WRITE(*,"(F10.5)") result
END PROGRAM conversion2
```

Programm wird übersichtlicher und besser lesbar

nachträgliche Änderungen sind einfacher durchzuführen

der Wert von symbolischen Konstanten kann nicht verändert werden

Ein- und Ausgabe

im Beispielprogramm, stoppt die Programmausführung wenn man zur `READ` Anweisung kommt

der Benutzer muß nun einen Wert eingeben und `<return>` oder `<enter>` drücken, danach setzt das Programm fort

das `WRITE` Statement schreibt den Inhalt der Variable `result` auf den Bildschirm

```
READ(*,*) a


WRITE(*,"(F10.5)") product
```

Oder einfach

```
WRITE(*,*) "input ",a," result ",result
```

Programmierstil

- alle F90 Keywörter sollten groß geschrieben werden
- man sollte einrücken
  - nach jedem `PROGRAM` Statement
  - nach `MODULE` Statements
  - nach `IF` Anweisungen
  - in `DO` Blöcken

Kompilieren

Source Files habe die Endung `.f90`, z.B.

```
conversion1.f90
```

benutzerlesbar und werden bei uns mit dem `Xemacs` Editor erstellt

müssen in eine computerlesbare Form gebracht werden
$\Rightarrow$ Executable oder ausführbares Programm

dazu dient der Compiler (selber auch ein ausführbares Programm)

```
> f90  conversion.f90 -o conversion
```

dieser erzeugt eine ausführbares Programm mit keine Endung (UNIX) oder der Endung `.exe` (WINDOF)

das Programm kann durch Eintippen des Programmnamens ausgeführt werden

```
>  conversion1
```

Programmerstellung

- Problemanalyse
- Programmdesign
- Programmierung
  - eintippen
  - Compilieren
  - Testen
  - abändern
- Microsoft
  - sell
  - re-design program
  - re-sell the same buggy stuff again

Beispielprogramm zwei

wir wollen nun nicht nur eine sondern fünf Zahlen von Kalorie nach Joule konvertieren

```
PROGRAM conversion3
  INTEGER, PARAMETER :: n = 5
  REAL :: a(n), r(n)
  REAL, PARAMETER :: c=  4.186
  WRITE(*,*) "convert cal to Joules"
  READ(*,*) a             ! read in five values and store in a
  r= a * c                ! multiply each value of a with c
  WRITE(*,"(F10.5)") r ! write result vector to screen
END PROGRAM conversion3
```

Konstanten und Typen

F90 ist eine “strongly typed language”
jede Variable und jede Konstante hat einen Typ

F90 unterstützt von sich aus 5 Typen mit Konstanten und Variablen für jeden Typ

```
INTEGER                 123      -15   +3
REAL                    1.5     100.76   1.5E+10    1.5
COMPLEX                 (1.5,1)     (1.5,0.3)
CHARACTER(LEN=40)       "1234"   'Hallo'   "Georg's"
LOGICAL                 .TRUE.   .FALSE.
```

Konstanten

Anmerkungen:

- Konstanten vom Typ REAL haben einen Gleitkommapunkt oder das Exponentierungssymbol `E`

  INTEGER Konstanten dagegen nicht

- CHARACTER Konstanten müssen in Anführungszeichen eingeschlossen sein (" oder ')

  CHARACTER können über mehrere Zeilen gehen

```
"Hallo this is &
&a test"
```

- es gibt nur zwei logische Konstanten: `.TRUE.` und `.FALSE.`

Der Typ von Variablen

natürlich hat auch jede Variable einen Typ

Variablen von einem bestimmten Typ werden mit den entsprechenden Anweisungen erzeugt:

```
INTEGER :: loop=0, i, j
REAL    :: x, y=0
COMPLEX :: c
CHARACTER     :: s1*(20), s2*(20) ! a string variable,
                                  ! which can hold  20 char
CHARACTER(LEN=20) :: s1, s2       ! same as above
```

symbolische Konstanten werden mit Hilfe des `PARAMETER` Attributs von normalen Variablen unterschieden

```
REAL, PARAMETER :: cal_to_joule=  4.186, stop_value=-1
```

sie können nicht überschrieben werden

Vektoren und Matrizen

werden von F90 unterstützt

eine Vektorvariable wird erzeugt, indem eine Klammer hinter den Variablennamen gesetzt wird

```
REAL :: a(5), r(5)  ! the var. x and y can store 5 values
INTEGER, PARAMETER :: n = 5
REAL :: a(n),r(n)   ! sames as above
```

in Ausdrücken kann auf jedes Element das Vektors zugegriffen werden

```
r(1) = c*a(1)
r(2) = c*a(2)
r(3) = c*a(3)
r(4) = c*a(4)
r(5) = c*a(5)
```

nicht besonders elegant, da das Programm ständig geändert werden muß

Ausdrücke mit Vektoren

ein Vektor kann genauso wie ein Skalar in F90 bearbeitet werden

wir können z.B. jedes Element mit einer Zahl (Skalar) multiplizieren

```
  r= c * a                    ! multiply each value of a with c
```

ähnlich wie in einer Formel $\vec{r} = c\vec{a}$

Fortran unterstützt auch Matrizen

```
REAL :: m(5,5)            !
INTEGER, PARAMETER :: n = 5
REAL :: m2(n,n)
```

Elemente können in ähnlicher Weise wie Vektoren adressiert werden

```
m=0             ! initialise all elements of the  matrix to 0
m(1,1) = 1      ! set one element
m2 = 2 * m      ! multiply each element of the matrix by 2
m2 = m + m      ! same as above
WRITE(*,*) m    ! print out the entire matrix
```

DO Schleifen

mit DO Schleifen kann man das obige Problem genauso lösen

```
  r(1) = c*a(1)
  r(2) = c*a(2) so nicht
 ...
```

```
PROGRAM conversion4
  INTEGER, PARAMETER :: n = 5
  INTEGER :: i
  REAL :: a(5), r(5)
  REAL, PARAMETER :: c=  4.186
  WRITE(*,*) "convert cal to Joules"
  READ(*,*) a            ! read in five values and store in a
  DO i=1,n
    r(i)=a(i)*c
  ENDDO
  WRITE(*,"(F10.5)") r   ! write result vector to screen
END PROGRAM conversion4
```

DO Schleifen im Detail

bei einer DO Schleife werden die Befehle in der Schleife (`r(i)=a(i)*c`) für i=1,2,...n ausgeführt

im Allgemeinen hat die DO Schleife folgende Form:

```
[name:] DO var=expr1,expr2[,expr3]
          f90-statement
          ...
          f90-statement
        ENDDO [name]    [] optional
```

`var` ist eine Integer Variable die deklariert werden muß

`expr1`, `expr2` und `expr3` sind drei Integerausdrücke

`expr1` Startwert

`expr2` Endwert

`expr3` Schrittweite

DO Schleifen im Detail

man kann der DO Schleife einen Namen geben (`named DO`)
dies sollte man in der Regel aber vermeiden

zwischen `DO` und `ENDDO` können beliebig viele Statements stehen
diese werden für

```
var = expr1, expr1+expr3, ... , expr1+ k*expr3
```

ausgeführt; `var` ist immer kleiner oder gleich `expr2`

wenn `expr2` kleiner als `expr1` ist, wird die Schleife nie ausgeführt
man kann zeigen, daß die Schleife exakt *ic* mal ausgeführt wird:

$$ic = \max((\texttt{expr2} - \texttt{expr1} + \texttt{expr3})/\texttt{expr3}, 0).$$

nach der Schleife hat die Variable `var` den Wert

$$\texttt{expr1} + ic * \texttt{expr3}$$

dies ist der erste Wert für den die Schleife nicht mehr ausgeführt wird

Unendliche Schleifen

```
[name:] DO
          f90-statement
          ...
          f90-statement
        ENDDO [name]
```

die Schleife kann man mit dem `EXIT` Statement verlassen

```
 EXIT [name]
```

Unendliche Schleifen: Beispielprogramm

```
PROGRAM conversion5
  REAL :: a, result
  LOGICAL :: is_zero
  DO
    WRITE(*,*) "please type in a number of convert, 0 to end"
    READ(*,*) a
    is_zero = a==0
    IF (is_zero) EXIT
    result = a * 4.186
    WRITE(*,"(F10.5)") result
  ENDDO
  ! exit jumps right here and the program stops
  WRITE(*,*) "stopping now"
END PROGRAM conversion5
```

neu ist hier vor allem die `IF` Anweisung und die logische Variable

Logische Variable

logische Variablen können nur die Werte `.TRUE.` und `.FALSE.`. besitzen

eine Zuweisung sieht sieht ähnlich wie in anderen Fällen aus:

```
logical_variable=logical_expression
```

ein logischer Ausdruck besteht meist aus einem Vergleich zweier numerischer Ausdrücke mit den Vergleichsoperatoren:

```
<     <=     ==     /=     >     >=
```

im Beispielprogramm ist die Variable `is_zero` nur dann `.TRUE.` wenn `a` Null ist

die Variable `is_zero` wird im `IF` Statement überprüft, und wenn sie `.TRUE.` ist, wird die `EXIT` Anweisung ausgeführt

meist würde man den Vergleich und die `IF` Abfrage kombinieren

```
IF (a==0) EXIT
```

Das `IF` *Konstrukt im Detail*

es gibt zwei Versionen des `IF` Konstrukts

```
IF (scalar-logical-expr) stmt
```

die Klammern sind notwendig, und das Statement `stmt` muß in der selben Zeile stehen
die Anweisung `stmt` wird nur ausgeführt, wenn `scalar-logical-expr` wahr ist

```
[name:]  IF (scalar-logical-expr) THEN
           f90-statement
           ...
           f90-statement
         ENDIF [name]
```

`name` ist optional
die Anweisungen im `IF ENDIF` Block werden nur ausgeführt wenn
`scalar-logical-expr` wahr ist

Das `IF` *Konstrukt im Detail*

```
[name:]  IF (scalar-logical-expr1) THEN
           block1
     [   ELSE IF (scalar-logical-expr2) THEN [name]
           block2
      [  ELSE IF (scalar-logical-expr3) THEN [name]
           block3
       [ ELSE [name]
           block_else of f90-statements
       ]
      ]
     ]
         ENDIF [name]
```

expr1 .TRUE.	$\Rightarrow$	block1 ausgeführt
expr1 .FALSE. und expr2 .TRUE.	$\Rightarrow$	block2 ausgeführt
kein Ausdruck wahr	$\Rightarrow$	else block ausgeführt

Beispielprogramm

```
PROGRAM calculator
  REAL :: res, a
  CHARACTER (LEN=1) :: operation
  res=0
  DO
    WRITE(*,*) 'result ',res
    READ(*,*) operation
    READ(*,*) a
    IF (operation == '*') THEN
       res=res*a
    ELSE IF (operation == '+') THEN
       res=res+a
    ELSE IF (operation == '-') THEN
       res=res-a
    ELSE IF (operation == '/') THEN
       res=res/a
```

```
    ELSE
       EXIT
    ENDIF
  ENDDO
END PROGRAM calculator
```

Ausdrücke und Zuweisungen im Detail

Wie schaut ein Ausdruck im Allgemeinen aus?
math. Formeln sind einfach nach F90 konvertierbar

$$f(x) = a + bx + cx^2$$

```
f = a+ b*x + c*x**2  ! multiplikation *
f = a+ b x + c x**2  ! compiler will report an error
```

zwei Klassen von Operatoren:

- dyadische (binary) Operatoren: `+ - * / ** .AND. .OR.`

  `expr1 + expr2`

- monadische (unary) Operatoren: `- .NOT.`

  `.NOT. expr1` `- expr2`

Ausdrücke `expr` sind wiederum gültige F90 Ausdrücke möglicherweise in Klammern

Skalare, numerische Ausdrücke

gültige Argumente sind vom Typ `INTEGER`, `REAL` oder `COMPLEX` (Konstanten, Variablen, Funktionsaufrufe oder F90 Ausdrücke in Klammern)

Gültige Operatoren:

```
**              exponentiation
*       /       multiplication and division
+       -       sum and diverence
+       -       unary operators (change of sign)
```

Ordnung der Ausführung ist die selbe wie in math. Formeln

```
f = a+ b*x + c*x**2*2   !  a + (b*x) + ((c*(x**2))*2)
```

Klammern können selbstverständlich verwendet werden

```
f = a+ b*x + (c*x)**2*2
```

Skalare, numerische Ausdrücke

man kann auch `INTEGER`, `REAL` oder `COMPLEX` Ausdrücke mischen

	I	R	C
I	I	R	C
R	R	R	C
C	C	C	C

eine skalare Zuweisung hat die Form:

```
numerical_variable=numerical_expression
```

Konversionen werden wieder automatisch durchgeführt (leider) Fließkommazahlen werden einfach abgeschnitten

Skalare Ausdrücke mit Strings

für Strings gibt es nur die Operation Anhängen

Argumente sind `CHARACTERS` (Konstanten, Variablen, Funktionsaufrufe)

```
//              concatenation
```

die Zuweisung ist wieder sehr einfach:

```
character_variable=character_expression
```

Beispiele

```
CHARACTER (LEN=4):: c
c      = "HE" // "LLO WORLD"    ! -> HELL
c      = "H"                    ! -> "H   "
c(1:2)= "HELLO WORLD"
c(3:4)= "HELLO WORD"(7:8)       ! -> "HEWO"
```

Vergleichsoperationen

eine Vergleichsoperation benötigt zwei Argumente vom Typ INTEGER, REAL, COMPLEX oder CHARACTER und liefert als Resultat einen logischen Wert

```
.LT.    <
.LE.    <=
.EQ.    ==
.NE.    /=
.GT.    >
.GE.    >=
```

Beispiele

```
LOGICAL :: f
f= "Hello"(1:2) == "He"      ! -> .TRUE.
f= 1.5 < 1                   ! -> .FALSE.
```

Logische Operationen

eine logische Operation benötigt ein oder zwei Argumente vom Typ `LOGICAL` und liefert als Ergebnis einen logischen Wert

```
.NOT.      unary negation
.AND.
.OR.
.EQV.      and .NEQV.
```

Beispiele

```
LOGICAL :: f
REAL    :: a = 1.5
f= 1 < a .AND. a < 2        ! a between 1 and 2
f = 1 < a  < 2              ! ** not allowed **
f= .NOT. (a < 1)            ! same as a >= 1
```

Priorität (Precedence)

```
 ----            monadic user defined
 **              numerical expression
 *       /
 +       -
 //              character concatenation
< > ==           relational
.NOT.            logical expression
.AND.
.OR.
.EQV.   .NEQV.
----             dyadic user defined
```

Klammern können verwendet werden um die Ordnung umzustellen

```
b*c+e          b*(c+e)
```

Arrays

jede Operation kann auch auf Arrays angewendet werden

Einschränkungen:

- Arrays müssen konform (conformable) sein
  gleiche Dimension und Größe haben
- die Operationen werden Element für Element durchgeführt

```
REAL :: a(5),b(5),c(5)
c= a*b
```

  ist gleichwertig zu

```
c(1)= a(1)*b(1)
c(2)= a(2)*b(2)
...
```

- ein Operand darf ein Skalar sein:

```
REAL :: a(5),c(5),b
c= a*b
```

ist gleichwertig zu

```
c(1)= a(1)*b
c(2)= a(2)*b
...
```

für Zuweisungen gilt ähnliches

- das zugewiesene Array muß konform sein
- Skalare können Feldvariablen zugewiesen werden

```
REAL :: a(5), b=1.5
a=b   ! each element of the vector a is set to 1.5
```

mehr Beispiele

```
REAL, DIMENSION(10):: x,y,z
z = x /y              !  z_i = x_i/ y_i   i=1,2,...,10
z = x + 1             !  z_i = x_i +1
z = x * 2             !  z_i = x_i *2
z = 0                 !  z_i =0
```

Summe von Eingabewerten

```
PROGRAM sum
  INTEGER, PARAMETER :: n=5
  INTEGER :: i
  REAL :: a(n), suma
  READ(*,*) a
  suma=0
  DO i=1,n
    suma=suma+a(i)
  ENDDO
  WRITE(*,*) 'sum is ',suma
END PROGRAM sum
```

```
suma=0
DO i=1,n
  suma=suma+a(i)
ENDDO
```

zuerst wird der Ausdruck rechts ausgewertet, und erst dann an die Variable zugewiesen

	vor Anweisung	nach Anweisung
1. Durchlauf	0	`a(1)`
2. Durchlauf	`a(1)`	`a(1)+a(2)`
3. Durchlauf	`a(1)+a(2)`	`a(1)+a(2)+a(3)`

Mittlere quadratische Abweichung

```
PROGRAM var
  INTEGER, PARAMETER :: n=5
  REAL :: a(n), suma, suma2
  READ(*,*) a
  suma=0
  suma2=0
  DO i=1,n
    suma=suma+a(i)
    suma2=suma2+a(i)**2
  ENDDO
  WRITE(*,*) sqrt(suma2/n-(suma/n)**2)
END PROGRAM var
```

Minimum

```
PROGRAM minimum
  INTEGER, PARAMETER :: n=5
  REAL  :: a(n), amin
  INTEGER :: i, ifound
  READ(*,*) a
  ifound=1
  amin=a(1)
  DO i=2,n
    IF ( a(i) < amin) THEN
      ifound=i
      amin=a(i)
    ENDIF
  ENDDO
  WRITE(*,*) amin,' found at ',ifound
END PROGRAM minimum
```

```
DO i=2,n
  IF ( a(i) < amin) THEN
    ifound=i
    amin=a(i)
  ENDIF
ENDDO
```

	vor Anweisung	nach Anweisung
1. Durchlauf	`a(1)`	min(`a(1)`,`a(2)`)
2. Durchlauf	amin	min(amin,`a(3)`)

Sortieren

```
PROGRAM sort
  INTEGER, PARAMETER :: n=5
  INTEGER :: a(n), amin, j, i, ifound
  READ(*,*) a
  DO j=1,n
    ifound=j ;  amin=a(j)
    DO i=j+1,n
      IF ( a(i) < amin) THEN
        ifound=i
        amin=a(i)
      ENDIF
    ENDDO
  a(ifound)=a(j) ;  a(j)=amin  ! swap data
  ENDDO
  WRITE(*,*) a
END PROGRAM sort
```

Inprodukt

```
PROGRAM dotprod
  INTEGER, PARAMETER :: n=5
  INTEGER :: j
  REAL :: a(n), b(n), cdot
  READ(*,*) a ; READ(*,*) b
  cdot=0
  DO j=1,n
    cdot=cdot+a(j)*a(j)
  ENDDO
  WRITE(*,*) cdot
END PROGRAM dotprod
```

Matrix mal Vektor

```
PROGRAM mat_vec
  INTEGER, PARAMETER :: n=2
  INTEGER :: i, j
  REAL :: a(n,n), b(n), c(n)
  READ(*,*) a ; READ(*,*) b
  c=0
  DO j=1,n
    DO i=1,n
      c(j)=c(j)+a(j,i)*b(i)
    ENDDO
  ENDDO
  WRITE(*,*) c
END PROGRAM mat_vec
```

FUNCTIONS und MODULES: Ein Mittelwertsprogramm

Mittelwertprogramm

```
PROGRAM calculate_mean1
  REAL :: m
  INTEGER, PARAMETER :: n=5
  REAL :: a(n)

  WRITE(*,*) "now please type in ",n," numbers"
  READ(*,*) a

  ! calculate mean value

  WRITE(*,"(F10.5)") m
END PROGRAM calculate_mean1
```

Berechnungsroutine fehlt; erste Möglichkeit (nicht elegant):

```
m=(a(1)+a(2)+a(3)+a(4)+a(5))/n
```

besser geht es natürlich mit `DO` loops

```
INTEGER :: i  ! must be placed in the declaration part


m =0
DO i=1,n
  m=m+a(i)
ENDDO
m=m/n          ! divide by number of elements
```

zuerst wird der Ausdruck rechts ausgewertet, und erst dann an die Variable zugewiesen

	vor Anweisung	nach Anweisung
1. Durchlauf	0	`a(1)`
2. Durchlauf	`a(1)`	`a(1)+a(2)`
3. Durchlauf	`a(1)+a(2)`	`a(1)+a(2)+a(3)`

FUNCTIONS und MODULES

definieren nun eine Funktion, die den Mittelwert ausrechnet

```
MODULE statistics
  CONTAINS
    FUNCTION mean(b)
      REAL ::  b(:)
      REAL :: mean
      INTEGER :: n, i

      n=size(b)
      mean =0
      DO i=1,n
        mean=mean+b(i)
      ENDDO
      mean=mean/n
    END FUNCTION
END MODULE statistics
```

```
PROGRAM calculate_mean1
  USE statistics
  REAL :: m
  INTEGER, PARAMETER :: n=5
  REAL :: a(n)

  READ(*,*) a

  ! write mean value

  WRITE(*,"(F10.5)") mean(a)
END PROGRAM calculate_mean1
```

FUNCTIONS und MODULES

- Module erlauben es eine Sammlung von Funktionen anzulegen, auf die man leicht zurückgreifen kann (`USE`)
- wenn eine Funktion aufgerufen wird, wird das dummy Argument `b` in der Funktion `mean` durch das aktuelle Argument `a` ersetzt

  beide Variablen müssen gleich definiert sein (REAL, INTEGER, Vektoren)
- der Dummyvektor `b` erhält automatisch die richtige Größe

  die Anzahl der Elemente im Vektor kann mit `SIZE()` festgestellt werden
- nach dem Aufruf wird jegliche Verbindung zwischen Dummyargument und Argument gelöst
- die lokalen Variablen `n` und `i` stehen nur in der Funktion zur Verfügung

Weitere Programmergänzungen

Berechnung der Varianz (nicht ganz richtig)

$$\sigma = \sqrt{\bar{a^2} - \bar{a}^2},$$

wobei $\bar{a}$ der Mittelwert ist; oder

$$\sigma = \mathrm{sqrt}\left\{\mathrm{mean}(a^2) - \mathrm{mean}(a)^2\right\},$$

in F90:

```
sigma = SQRT( mean(a**2) - mean(a)**2)
```

SUBROUTINES

Unterprogramme verhalten sich wie Funktionen, geben aber keinen Wert zurück; sie dürfen den Inhalt der Dummyvariablen verändern

```
SUBROUTINE mean_and_variance(b, m, variance)
   REAL :: b(:)
   REAL :: m
   REAL :: variance

   m=mean(b)
   variance= SQRT( mean(b**2) - m**2)
 END SUBROUTINE
```

Unterprogramme werden mit der `CALL` Anweisung aufgerufen

```
CALL mean_and_variance(a, m, sigma)
```

während der Ausführung des Unterprogramms werden die Dummyargument `b`, `m`, und `variance` mit den Variablen `a`, `m`, und `sigma` "verbunden"

Eingebaute Funktionen

eine Vielzahl von eingebauten Fuktionen steht zur Verfügung

Summe aller Elemente eines Vektors: `SUM`
damit kann man natürlich den Mittelwert wieder auf einfache Weise berechnen

```
m=SUM(a)/n
sigma = SQRT( (SUM(a**2)/n) - m**2),
```

drei eingebaute Funktionen haben wir schon kennen gelernt: `SUM`, `SQRT` und die `SIZE` Funktion

Dynamische Arrays

```
PROGRAM calculate_mean5
  USE statistics
  REAL :: m, sigma
  INTEGER :: n
  REAL, ALLOCATABLE :: a(:)
  WRITE(*,*) "how many numbers do you want to type in ?"
  READ(*,*) n
  ALLOCATE(a(n))

  WRITE(*,*) "now please type in the numbers"
  READ(*,*) a
  m    = mean(a)              ! calculate mean value
  sigma=SQRT( mean(a**2) - m**2)
  WRITE(*,"(F10.5)") m,sigma
  DALLOCATE(a)

END PROGRAM calculate_mean5
```

Dynamisch Arrays

die Deklaration

```
REAL, ALLOCATABLE :: a(:)
```

legt einen Vektor (mit noch unbekannter Größe) an

die Größe des Vektors wird zu einem späteren Zeitpunkt mit dem `ALLOCATE` Statement bestimmt

```
ALLOCATE(a(n))
```

allozierter Platz muß unbedingt mit `DEALLOCATE` freigegeben werden

PROGRAM units und MODULE im Detail

ein F90 Programm besteht aus

- einem Hauptprogramm (`PROGRAM` Statment)
- internen Prozeduren (nicht verwenden)
- externen Prozeduren (auch nicht zu empfehlen)
- Sammlungen von Modulen
- eingebauten Prozeduren (instrinsic procedures)

ein Modul ist eine Sammlung von Variablen und Prozeduren

alle Daten (und Prozeduren) sind lokal für ein Modul

- der Namen des Moduls ist global (case)
- alle Prozeduren eines Moduls können mit der `USE` Anweisung zugänglich gemacht werden

  Variablen oder Konstanten, die vor dem `CONTAINS` definiert sind, werden auch zugänglich

Das Main PROGRAM (Hautprogramm)

ein Hauptprogramm wird wie folgt definiert:

```
  PROGRAM program-name
    specification-stmts
    executable-stmts
[ CONTAINS
    internal-subprograms ]
  END [ PROGRAM [program-name]]
```

Ausdrücke in Klammern sind optional

das CONTAINS Statment leitet Prozduren ein, die nur lokal für dieses Programm definiert sind

es ist besser und sicherer `MODULE` zu verwenden

Das Main PROGRAM

Spezifikations Anweisungen (`specification-stmts`) sind `USE` Statements oder die Deklaration von Variablen

```
USE statistics
INTEGER, PARAMETER :: i=10
REAL :: a , b(i,i)
```

`USE` Statements müssen vor anderen Deklarationen stehen

ausführbare Anweisungen sind Zuweisungen, IO-Anweisungen, `CALL` Statments und `DO` und `IF` Anweisungen

Module

man sollte Prozeduren immer in `MODULE` “verpacken”

```
 MODULE modul-name
     specification-stmts
 [ CONTAINS
     module-subprograms ]   define !!
   END [ PROGRAM [modul-name]]
```

das einfachste Modul kann nur Konstanten definieren

```
MODULE constants
    REAL, PARAMTER :: cal_to_joule=  4.186
    REAL,PARAMETER :: pi =3.14159265358979323,tpi=2*pi
END MODULE
```

Kollektion von Unterprogrammen, die logisch zusammenhängen

mit dem `USE` Statment kann auf ein Modul zugegriffen werden

```
 USE  modul-name
```

Externe Prozeduren und Modul Prozeduren

```
[ RECURSIVE ]
  SUBROUTINE subroutine-name  [ ( [ dummy-arg-list] ) ]
  FUNCTION   function-name      ( [ dummy-arg-list] )
    specification-stmts
    executable-stmts
[ CONTAINS
    internal-subprograms ]
  END [ SUBROUTINE [subroutine-name]]
  END [ FUNCTION   [function-name]]
```

ein `RETURN`, `END SUBROUTINE` oder `CONTAINS` Statement führt zu einem Rücksprung zum aufrufenden Programm

der Compiler überprüft bei externen Prozeduren weder Typ noch Zahl der Argumente (F77)

interne Prozeduren werden in ähnlicher Weise definiert (keine weitere CONTAINS Anweisung, volle Überprüfung der Argumente)

Order of statements

die Ordnung der Anweisungen ist in F90 sehr strikt

```
PROGRAM, FUNCTION, SUBROUTINE, MODULE statements

  USE statements
  IMPLICIT statements
  derived type definition
  variable declarations

  executable statements

CONTAINS

internal or module procedures

END PROGRAM, FUNCTION, SUBROUTINE, MODULE statements
```

Dummy arguments - actual arguments - local variables

in F90 werden Argumente als Referenz übergeben

```
SUBROUTINE add(dum1,dum2,dum3)
  REAL :: dum1, dum2, dum3
  dum1=dum2+dum3
END add
```

Aufruf durch

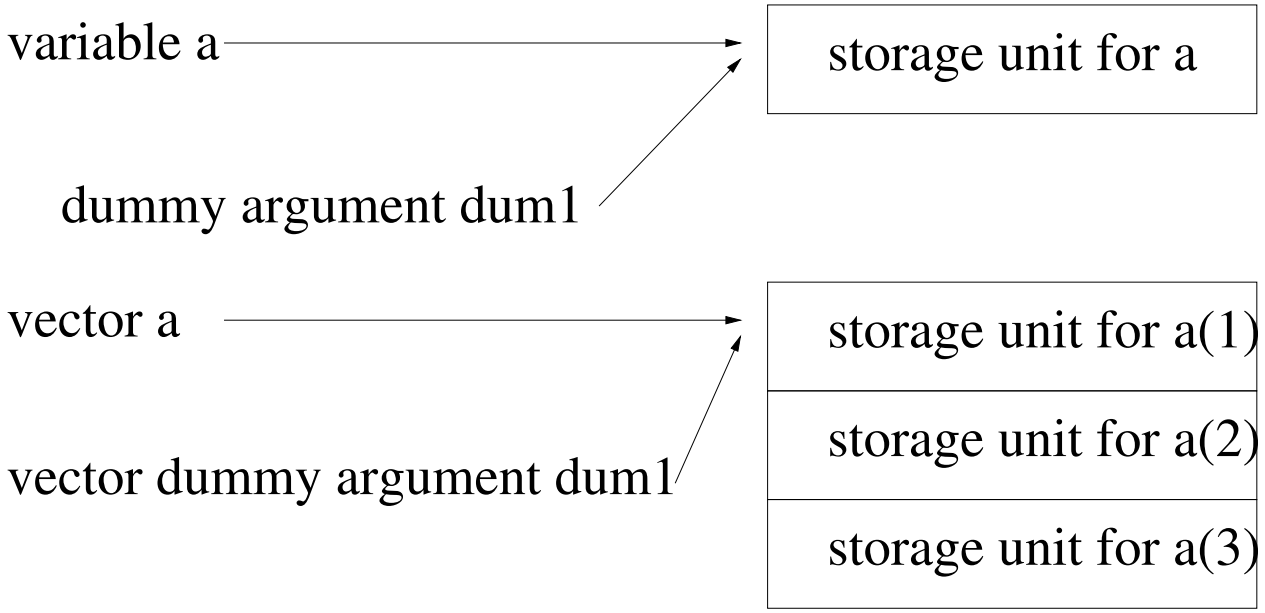
```
CALL add(a,b,c)
```

ist gleichwertig zu

```
a=b+c
```

`dum1`, `dum2` und `dum3` Dummyargument `a`, `b` and `c` aktuelle Argumente

cut und paste, wobei `dum1` überall durch `a` ersetzt wird



wenn `dum1` modifiziert wird, wird auch `a` modifiziert

“call by reference” versus “call by value” (C)

call by reference ist zwar flexibler aber unsicherer

Lokale Variablen

jede Prozedur kann ihren eigenen Satz von Variablen definieren

der Speicherplatz wird in dem Moment zugewiesen, in dem die Prozedur aufgerufen wird und wieder freigegeben wenn die Prozedur verlassen wird

mit dem `SAVE` Attribut kann man dieses Verhalten verändern

```
INTEGER, SAVE :: init
```

der Inhalt der Variable `init` bleibt auch zwischen Aufrufen erhalten

Funktionen

Funktionen verhalten sich ähnlich wie Prozeduren, sollten aber nie ihre Argumente verändern

```
[ RECURSIVE ]
 FUNCTION function-name([ dummy-arg-list]) RESULT(func_res)
   specification-stmts
   executable-stmts
 END  [ FUNCTION   [function-name]]
```

Funktionen werden in Ausdrücken über den Funktionsnamen aufgerufen

```
 function-name      ( [ actual-arg-list] )
```

`RESULT(func_res)` erzeugt eine Variable, der einen Wert zugewiesen werden sollte, bevor die Funktion verlassen wird

Default ist die Variable `function-name`

Examples: Only constants

```
MODULE constants
    REAL, PARAMTER :: cal_to_joule=  4.186
    REAL,PARAMETER :: pi =3.1415926535897932 3,tpi=2*pi
END MODULE
```

Examples: statistic routines

```
MODULE statistics
  USE constants
  CONTAINS
    FUNCTION mean(b)
      REAL ::  b(:)
      REAL :: mean
      INTEGER :: n, i

      n=size(b)
      mean =0
      DO i=1,n
        mean=mean+b(i)
      ENDDO
      mean=mean/n
    END FUNCTION mean
```

```
    SUBROUTINE mean_and_variance(b, m, variance)
       REAL :: b(:)
       REAL :: m
       REAL :: variance

       m=mean(b)
       variance= SQRT( mean(b**2) - m**2)
    END SUBROUTINE mean_and_variance
END MODULE statistics
```

```
MODULE simple_stack
  INTEGER, PARAMETER :: nmax=4
  REAL :: a(nmax)
  INTEGER :: nstored=0
  CONTAINS
    SUBROUTINE push(value)
      REAL :: value
      nstored=nstored+1
      IF (nstored>nmax) THEN
         WRITE(*,*) 'stack exhausted'
         STOP
      ENDIF
      a(nstored)=value
    END SUBROUTINE push
    FUNCTION pop()
      REAL :: pop
      IF (nstored<=0) THEN
         WRITE(*,*) 'stack empty'
         STOP
      ENDIF
      pop=a(nstored); nstored=nstored-1
    END FUNCTION pop
END MODULE simple_stack
```

```
PROGRAM rpn
  USE euro
  USE simple_stack
  CHARACTER :: l*(40)
  REAL :: b
  DO
    READ(*,'(A)') l
    IF (l(1:1)=='+') THEN
      b=pop()+pop()
    ELSEIF (l(1:1)=='-') THEN
      b=-pop()+pop()
    ELSEIF (l(1:1)=='s') THEN
      b=pop()*euro2sh
    ELSE
      READ(l,*) b
    ENDIF
    WRITE(*,*) b
    CALL PUSH(b)
  ENDDO
END PROGRAM rpn
```

Rekursive Funktionen: Die Faktorielle

klassische rekursive Funktion:

$$n! = \begin{cases} 1 & \text{for} \quad n <= 1 \\ n(n-1)! & \text{for} \quad n > 1 \end{cases}$$

```
RECURSIVE FUNCTION fac1(n) RESULT(f)
  INTEGER(8):: f,n
  IF (n <= 1) THEN
    f=1
  ELSE
    f=n*fac1(n-1)
  ENDIF
END FUNCTION fac1
```

die `RECURSIVE` Anweisung und die `RESULT` Anweisung sind hier notwendig

KIND Attribut

normale INTEGER Variablen belegen nur 4 Bytes im Speicher

Mit dem KIND Attribut kann man dies auf 8 erhöhen
dadurch können wir die Faktorielle für 20 berechnen

```
REAL     ( [KIND=] integer_constant ) :: list_of_variables
INTEGER  ( [KIND=] integer_constant ) :: list_of_variables
```

das `KIND=` Attribut ist optional A few examples:

```
REAL     ( KIND=8 ) :: a(3),b
INTEGER  ( 4 ) :: i
REAL (8) :: b(3)
```

Die Genauigkeit einer Konstanten kann mit _ erhöht werden

```
REAL(8) :: pi=3.14159265358979323_8
```

Faster Factorial

```
FUNCTION fac2(n)
  INTEGER(8):: fac2,n
  LOGICAL,SAVE       :: initialised=.FALSE.
  INTEGER, PARAMETER :: n_max=20
  INTEGER(8),SAVE    :: table(n_max)
  INTEGER :: i
  IF (.NOT. initialised ) THEN
    table(1)=1
    DO i=2,n_max
      table(i)=table(i-1)*i
    ENDDO
    initialised=.TRUE.
  ENDIF
  IF (n <= 1) THEN
    fac2=1
  ELSE IF (n > n_max) THEN
    WRITE(*,*) 'fac2 error: n >',n_max
    STOP
  ELSE
    fac2=table(n)
  ENDIF
END FUNCTION fac2
```

Scope; lexical Scoping

Scope gibt an wo eine Variable oder Prozedur definiert ist

- Labels sind lokal
- in F90 gibt es nur einem Namensbereich (Funktionen und Variablen müssen unterschiedliche Namen haben)

  alle Namen sind lokal, außer der Name der Programmeinheit selber (MODUL, PROGRAM, externe Prozeduren)

```
MODULE test
  REAL :: f=1.2
  CONTAINS
    FUNCTION f(x)    ! illegal, since f already def.
```

- *unterschiedliche* Programmeinheiten dürfen die selben Namen verwenden

```
MODULE test
  REAL :: f=1.2, pi = 3.1415926
END MODULE test


MODULE test2
  REAL :: pi = 3.1415 ! that's a different pi
  CONTAINS
    FUNCTION f(x)     ! this is legal
```

- "USE Assoziation": alle Symbole eines Moduls können mit `USE` importiert werden

```
USE modul
```

solche Symbole können nicht redefiniert werden

```
USE test2
REAL :: pi = 3.1415926  ! illegal, pi defined in test2
USE test
USE test2  ! illegal since pi is defined in both modules
```

- ”Host Assoziation”:
  interne Prozeduren, und Prozeduren in Modulen kennen alle Variablen des äußeren Blocks
  diese können redefiniert werden

```
MODULE test
  REAL :: x
  CONTAINS
    SUBROUTINE set_x_local(y)
      REAL :: x, y  ! x differs from first x
      x=y
    END SUBROUTINE set_x_local
    SUBROUTINE set_x(y)
      REAL :: y
      x=y   ! this will set x in the MODULE header
    END SUBROUTINE set_x
END MODULE
```

dies führt zu vielen unnötigen Fehlern

Funktionen als Argumente

```
MODULE functions
CONTAINS
  FUNCTION f1(x)  ! f1(x) = x^2
    REAL :: f1,x
    f1=x**2
  END FUNCTION f1
  FUNCTION f2(x)  ! f2(x) = e^x
    REAL :: f2,x
    f2=exp(x)
  END FUNCTION f2
END MODULE functions
```

```
MODULE integrate
  CONTAINS
  FUNCTION simple_int(f,xstart,xend,intersections)
    INTERFACE
      FUNCTION f(y); REAL :: f,y; END FUNCTION
    END INTERFACE
    REAL :: simple_int,xstart,xend,d
    INTEGER:: intersections,i

    simple_int=0
    d=(xend-xstart)/(intersections-1)
    DO i=0,intersections-1
      simple_int=simple_int + f(xstart+d*i)*d
    ENDDO
  END FUNCTION simple_int
END MODULE integrate

PROGRAM test
  USE functions
  USE integrate
  IMPLICIT NONE
  REAL a,b
  WRITE(*,*) simple_int(f1,0.,1.,100), &
             simple_int(f2,0.,1.,100)
END PROGRAM test
```

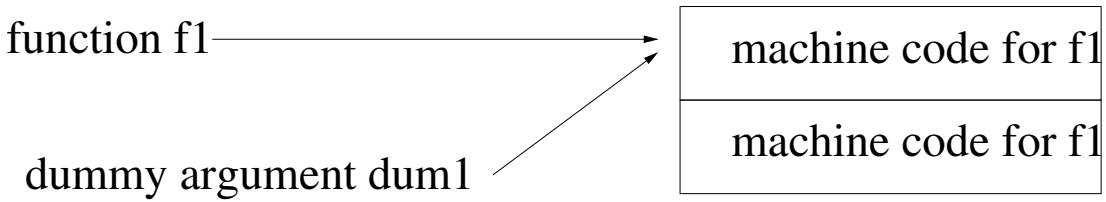
Funktionen als Argumente

einzig wirklich neue ist das `INTERFACE` Statement

```
INTERFACE
  FUNCTION f(y)
    REAL :: f,y
  END FUNCTION
END INTERFACE
```

gibt dem Compiler zwei Informationen: (i) `f` ist eine Funktion
(ii) welche Argumente hat `f`

zwischen dem `INTERFACE` und `END INTERFACE` Statement werden nur der Typ der Funktion und deren Argumente definiert



Langweilig aber notwendig: IO

Output: schreibt auf den Bildschirm oder Files

Input: liest von der Tastatur oder Files

IO wird natürlich während der Laufzeit durchgeführt

bei der Eingaben (READ) stoppt das Programm und wartet auf den Benutzer

in Fortran wird der Ort, von dem gelesen wird oder wohin geschrieben wird, durch Integer Ausdrücke angegeben (`UNIT`)

drei units sind “preconnected”

```
UNIT=5   ! stdin  usually the keyboard
UNIT=6   ! stdout usually the screen
UNIT=0   ! stderr (usually also the screen)
```

OPEN *und* CLOSE *Statments*

die OPEN und CLOSE Statments werde dazu verwendet ein File zu öffnen oder zu schließen

```
INTEGER :: u=10
CHARACTER :: file='my_file'
OPEN([UNIT=] u, FILE=file)
CLOSE([UNIT=] u)  ! file wieder schliessen
```

file ist ein String (oder ein gültiger Stringausdruck), und u ein ganzzahliger Ausdruck

der OPEN Befehl stellt eine Verbindung zwischen einem File und einer unit u her
danach kann dieses File über diese Unit in READ oder WRITE Befehlen angesprochen werden

```
REAL :: A=12
OPEN(10,FILE='data')  ! create the file data
WRITE(10,*) A*2+1     ! write the value of A*2+1 to file 'data'
CLOSE(10)
```

READ, WRITE *Statments*

Syntax:

```
READ ( [UNIT=] u, [FMT=] fmt) [iolist]
WRITE( [UNIT=] u, [FMT=] fmt) [iolist]
```

der INTEGER Ausdruck `u` gibt die unit an
der Ausdruck `fmt` spezifiziert das Format

Ausdrücke in Klammern sind optional

die Units 5 und 6 sind “preconnected”

- `READ(*,*)` ist gleichwertig zu `READ(5,*)`
  `READ(*,'(1F10.5)')` ist gleichwertig zu `READ(5,'(1F10.5)')`
- `WRITE(*,*)` ist gleichwertig zu `WRITE(6,*)`

`READ`, `WRITE` *Statments*

Format Spezifikation

- ein Stern “`*`” (list directed input/output)
- oder eine Zeichenkette

die `iolist` ist eine Liste von Komma-separierten Ausdrücken

in READ Befehlen muß jeder Ausdruck ein Ausdruck sein, der auch auf der rechten Seite einer Zuweisung stehen könnte (Variablen oder Vektoren)

List-directed input/output, `FMT= *`

einfachste Version: Stern-Format

es erlaubt eine beliebige Anzahl von Variablen einzulesen:

```
WRITE(*,FMT=*)"Number of data's and comment"
READ(*,FMT=*) n,comment
WRITE(*,*)'Please input ',n,'data'
READ(UNIT=*,*) (a(i),i=1,n)
```

Tastatureingabe:

```
3 "test" <enter>
1.5 2.3  <enter>
2.4      <enter>
```

list-directed input ist üblich, aber list-directed output sieht nicht sehr schön aus (und das Format ist außerdem Compiler-abhängig)

List-directed input, `FMT= *`

wenn das compilierte Fortran Programm auf ein READ Statment trifft, wird die Ausführung angehalten

der Input muß gewisse Regeln erfüllen:

- Items müssen durch Leerzeichen, Komma, den Slash "/" oder eine Return getrennt sein
- wenn ein "/" in der Eingabe gefunden wird, terminiert die Eingabe sofort (end of file condition)
- man kann eine Wiederholungsangabe vor jedes Item stellen: 3*3.5 entspricht drei mal der Zahl 3.5

```
REAL :: A(5)
READ(*,*) A

  3*0.0 2*1.0  <enter>
```

- logische Variablen: `T` und `F`
- Zeichenketten sollten in Hochkommas eingeschlossen sein

  wenn sie nicht in Hochkommas eingeschlossen sind, dürfen sie keine Leerzeichen, Komma oder Return enthalten

```
CHARACTER*(12) :: A
READ(*,*) A

"Hello world "    !  eine Zeichenkette  "Hello world "
Hello world       !  zwei Zeichenketten "Hello" und "world"
```

- Komplexe Zahlen werden in Klammern eingeschlossen:

```
( 1E4 separator 1.456 )
```

  wobei der Separator ein Leerzeichen, Komma oder Return ist

Formatierte Ein und Ausgabe

Formatierte Ausgabe macht ein Programm benutzerfreundlicher

jedem Eintrag im Format wird eine Eintrag in der Ein/Ausgabeliste zugeordnet

```
INTEGER :: i
REAL    :: flt
WRITE(*,'(I10,F10.3,A10)')     i,flt*2,'hello'

WRITE(*,FMT='(I10,F10.3,A10)') i*2+1,flt,'hello'

READ (*,'(I10,F10.3,A10)')     i,flt,'hello'
```

Ein und Ausgabe Deskriptoren

eine Format Spezifikation besteht aus einer Liste von Edit-Deskriptoren, die durch Komma getrennt sein und mit Klammern gruppiert sind

sie müssen immer in mindestens eine Klammer eingeschlossen sein

wichtigsten Edit-Deskriptoren

```
I  integer            rIw
F  fixed real         rFw.d
E  exponential real   rEw.d    rEw.dEe
L  logical            rLw
A  character          rAw
G  general            rGw.d    rGw.dEe

X  blank              rX
/  new-line          r/
```

Ein- und Ausgabe-Deskriptoren

`r`	Wiederholungszähler
`w`	Breite des Feldes
`d`	Anzahl der Nachkommastellen
`e`	Anzahl der Stellen im Exponent

in F90 sind Ausgaben generell rechtszentriert

Wiederholungszähler sind vor allem für Vektoren nützlich:

```
INTEGER :: i
REAL    :: f(10),a,b
WRITE(*,'(5F10.5)')  f  ! prints out the 10 numbers in f
WRITE(*,FMT='(5F10.5)') a,b  ! prints two numbers a and b
```

wenn die Edit-Deskriptoren erschöpft sind, werden Sie vom Start wiederholt (eine Zeilenumbruch wird zusätzlich erzeugt)

es ist zulässig mehr Edit-Deskriptoren im Format zu haben, als es Einträge in der Ein- bzw. Ausgabeliste gibt

Gruppieren von Ein- und Ausgabe-Deskriptoren

Edit-Deskriptoren können mit Klammern gruppiert werden:

```
 '(3(2F10.4,5X))'

   dd.dddd   dd.dddd~~~~~   dd.dddd   dd.dddd~~~~~
12345678901234567890123456789012345678901234567890123
0         1         2         3         4         5
```

wenn die Edit-Deskriptoren erschöpft sind und ein Gruppierung mit Klammern existiert, wird die letzte Gruppe in Klammern wiederholt

```
 WRITE(*,'(I10,F10.3)') i,a,j,b
```

Ausgabe:

```
      iiii    aa.aaa <newline>
       jjj  bbbb.bbb
12345678901234567890123456789012345678901234567890123
0         1         2         3         4         5
```

Integer Format: rIw

Ausgabe: Zahlen werden rechtszentriert ausgegeben:

```
  WRITE(8,'(2I10)') 12,156
        12       156
12345678901234567890123456789012345678901234567890
0         1         2         3         4         5
```

Eingabe: `w` Zeichen werden eingelesen, Leerzeichen gelöscht und die Eingabe in eine ganze Zahl umgewandelt:

```
 READ(8,'(2I5)') i,j
```

Beispiel:

```
   12  156        -> i=12 j=156
 1  2156          -> i=12 j=156
1234512345
```

Float Format (F-Format): rFw.d

Ausgabe: Zahlen werden rechtszentriert mit `d` Dezimalstellen ausgegeben (Rundung erfolgt automatisch)

```
  WRITE(*,'(F8.3)') 13.568924
  13.569
12345678901234567890123456789012345678901234567890
0         1         2         3         4         5

  WRITE(*,'(A10,4F10.3)') "Hello  ",13.568924,1.,0.
   Hello       13.569     1.000     0.000
12345678901234567890123456789012345678901234567890
0         1         2         3         4         5
```

Eingabe: `w` Zeichen werden eingelesen und Leerzeichen gelöscht
wenn die eingelesen Zahl kein Komma besitzt, wird an der Position `d` von rechts ein Komma eingefügt:

```
  READ(*,'(F10.3)') a

  13.569     ->   13.569
      1233   ->    1.233 (decimal point ins. at 3. pos)
  1 3 4      ->    .134  (blanks removed, dec. point at 3. pos)
     1.E10   ->    1E10
   1E10      ->    unpredictable, a dec. point is inserted
                   but where ?? not to be used !!!!
1234567890123456789012345678901234567890123456789
0         1         2         3         4         5
```

für Tastatureingabe ziemlich sinnlos !

Exponential Format (E-Format): rEw.d or rEw.dEe

Eingabe: ident zu F Format

Ausgabestring: `__s.ddddEveeee`

- `_` Leerzeichen
- `s` Vorzeichen
- `d` Mantisse mit `d` Digits
- `v` Vorzeichen des Exponents
- `e` Exponent mit `e` Digits (Default 2 or 3)

die Feldbreite sollte mindestens `w = z+e+5` sein

wenn die `Ee` Spezifikation fehlt, sollte die Feldbreite mindestens `w = z+8` sein

Logical Format (L-Format): rLw

Ausgabe: T oder F rechtsbündig

Eingabe: erster Buchstabe der sich vom Leerzeichen und Punkt "." unterscheidet bestimmt ob die Variable auf .TRUE. oder .FALSE. gesetzt wird

```
LOGICAL :: L
READ(*,'(L3)') L
T              -> L= .TRUE.
.Txx           -> L= .TRUE.
  F            -> L= .FALSE.
```

Character Format (A-Format): rAw

Ausgabe: Zeichenkette rechtszentriert

wenn `w` zu klein ist werden die rechten Zeichen des Strings abgeschnitten

```
WRITE(*,'(A10,A10)')  "Hello world",'right'
Hello worl     right
12345678901234567890123456789012345678901234567890
0         1         2         3         4         5
```

Eingabe: `w` Zeichen werden eingelesen und in der Variable abgelegt

General Format (G-Format): rGw.dEe

allgemeines Ein und Ausgabeformat

Blank Format (X-Format): rX

Ausgabe: gibt `r` Leerzeichen aus

Eingabe: überliest `r` Zeichen

New Record (/-Format): r/

Ausgabe: Zeilenumbruch

Eingabe: lese bis zum Zeilenumbruch

More about arrays

Arrays are natively supported in F90, and far more flexible than discussed in the main lectures.

- Arrays can have any dimension
- the array bounds need not to start at 1.

Dimensions can be specified either with the `DIMENSION` attribute or in parentheses after the name of the variable.

General syntax for array specification

```
DIMENSION(extent-list)   or  variable_name(extent-list)
extent:   [lower :] upper
```

Here an `extent-list` is a list of `extents`, which are comma separated. An `extent`, consists of an optional lower bound and an upper bound for the array index. The default for the lower bound is 1.

Here are few more examples for the specification of arrays:

```
REAL, DIMENSION(3) :: x,y                  ! simple vector
INTEGER, PARAMETER :: n=50, m=100
REAL :: rho(0:n,0:m), vel(3,0:n,0:m) ! rho is a matrix
                                           ! vel is a tensor of rank 3
REAL, DIMENSION(3,3) :: a,b                ! two matrices
REAL :: c(-10:3)                           ! a vector
```

Storage layout

The storage layout of arrays has not been changed since F77 and is a row first ordering. This means that a matrix with 3×3 elements

```
a(1,1)      a(1,2)      a(1,3)
a(2,1)      a(2,2)      a(2,3)
a(3,1)      a(3,2)      a(3,3)
```

is stored in the memory in the following manner:

```
a(1,1) a(2,1) a(3,1)   a(1,2) a(2,2) a(3,2)   a(1,3) a(2,3) a(3,3)
```

This is important for the optimisation of program. If arrays are used in nested `DO` loops, one should try to let the first index of all arrays correspond to the innermost variable of the `DO` loop.

Literal constants

Literal constants for arrays are constructed by enclosing a list of literal real or integer constants in `(/` and `/)`:

```
(/ 2, 4, 3 /)
```

Array constructors

```
array = (expression, var=expr1, expr2, expr3)
```

The variable `var` takes all values it would take in a DO loop of the form:

```
DO var=expr1, expr2, expr3
```

This is called implicit `DO` loop. For any of the possible values of `var` one element is added to the final vector. Such implicit `DO` loops can be nested as shown in the examples below:

```
(/ (i,i=1,7,2) /)          % same as (/1,3,5,7/)
(/ (j*i,i=1,3),j=1,3) /) % matrix
```

Rank, shape and size

The rank of an array—that is the number of dimensions— can be changed only during compile time and is fixed in the source code. The shape and size of an array can be determined during run-time using the following commands (with examples assuming the previous declarations):

shape:	sequence of extends,	`SHAPE(rho)` $\longrightarrow$ (/ 51, 101/)
size:	total number of elements,	`SIZE(rho)` $\longrightarrow$ 5151

The extend of an array is the upper bound minus the lower bound plus 1.

Array-Section

It is possible to address only parts of an array, using array-sections. This is an extremely flexible and powerful feature, which often allows to avoid complicated `DO` loops. Some simple examples are indicated below:

```
x(1:2)                ! same as vector (/ x(1), x(2) /)
rho(:,1)              ! vector, which equals the first column of rho
rho(3,:)              ! vector, which equals the third row of rho
```

One can even select sub-parts of a matrix:

```
rho(2:4,3:6)
```

This selects only the area between the 2nd and 4th row, and the 3rd and 6th column.

Zero sized arrays

If the lower bound of an array exceeds its upper bound, the array is called zero size array. All intrinsic subroutines and all F90 expressions can handle such arrays in the way one would expect.

Assumed shape array

In subroutines, dummy arguments can assume the shape of actual arguments.
Unfortunately the rank must agree with the calling routine, which sometimes limits the usefulness of this feature:

```
REAL :: a(3:12),m(10,10)
CALL t(a,m)


SUBROUTINE t(b,n)
REAL :: b(:)
REAL :: n (-1:,-1:)
```

lower bound in the subroutine is by default always 1 (bounds are not passed !!!!).
Only the extend (upper–lower bound) and the size are handed down to the subroutine.

The actual bounds can be found by calling the `SHAPE` and `SIZE` functions:

```
SIZE (b)   ! sufficient for one dimensional arrays:  10
SHAPE(b)   ! in this case it would be a vector with on
           ! element: (/10/)
```

Automatic arrays

In subroutines it is often required to temporarily create an array that has the same shape as an array passed to the subroutine. This can be done by means of automatic arrays:

```
SUBROUTINE swap(a,b)
 REAL:: a(:), b(:), work(SIZE(a))
 work=a ; a=b; b=work
END
```

In this case arrays are allocated from the stack or an internal heap and freed after the RETURN.

Elemental intrinsic functions

All elemental intrinsic functions can be applied to arrays. The resulting array has the same shape as the initial array:

```
REAL, DIMENSION (10): a,b,c,d
a=b+c
d=SQRT(a)
```

Array valued function

The result of a function can be an array (for instance automatic objects!):

```
FUNCTION gurk(a)
  REAL:: a(:),gurk(size(a))
  gurk=a**2
END FUNCTION
```

Allocatable arrays

Arrays can be allocated at runtime with the `ALLOCATE` command. This feature was already discussed in the main section, but here is another example:

```
REAL, ALLOCATABLE :: a(:,:), b(:)
READ(*,*) n
ALLOCATE(a(n,n),b(n))
...
DEALLOCATE(a,b)
```

There are a few things to be careful with:

- Presently, F90 does not offer an automatic garbage collection or deallocation. This means that one *must* deallocate any array that has been allocated. Otherwise, only at the termination of the program the allocated space is freed.
- It is particularly important to deallocate arrays which are allocated in subroutines. If this is not done, memory becomes unaccessible (memory leakage).

General syntax for allocatable arrays

```
ALLOCATE (allocation-list, STAT=stat)
DEALLOCATE (allocate-object-list, STAT=stat)
NULLIFY (pointer-object-list, STAT=stat)

allocation-list:
  allocate-object [(array-bounds-list)]
array-bound
  [lower-bound:] upper-bound
```

- The default for the lower bound is 1.
- The variable `stat` must be an `INTEGER` variable
- if stat is not 0 after the `ALLOCATE` command, the allocation failed.
- If STAT=`stat` is missing, the `ALLOCATE` commands terminates the program, if it fails to allocate sufficient space.

It is possible to test, whether arrays were already allocated somewhere else in the program by means of the `ALLOCATED` intrinsic function:

```
ALLOCATED(pointer)
```

This function returns `.TRUE.` or `.FALSE.`

WHERE stmt

Many Do-loops with arrays may be replaced by the `WHERE` construct:

```
REAL :: A(10)
  WHERE (A>0)
    A=1.0 /A
  ELSE WHERE
    A=0
  END WHERE
```

General syntax of the WHERE stmt

```
  WHERE (logical-array-expr)
   array-assignments
[ ELSE WHERE
   array-assignments ]
  END WHERE
```

or

```
  WHERE (logical-array-expr) array-assignments
```

The logical-array-expr must have the same shape as each array-assignment.

Derived Types

- In F90, it is possible to define derived data types. For anything but the most trivial program this is a feature that one should rely on heavily.
- A derived type is a collection of variables into one single combined structure.

A derived type is declared for instance with the following statement:

```
TYPE person
  CHARACTER(10) ::  name
  REAL          ::  age
  INTEGER       ::  id
END TYPE person
```

Nesting of types

Types can be nested, for instance:

```
TYPE couple
  TYPE(person) :: male,female
  TYPE(date)   :: date_of_marriage
END TYPE couple
```

Structures

As for other types, one can define variables of a derived type. Such variables are called structures in Fortran90:

```
TYPE(person) :: you, me  ! declare two variables you and me
```

Literal constants of a derived type are constructed with the following syntax:

```
you=person('Georg',27,210767)
```

where an values must be supplied for each of the elements of the derived type. In expressions it is possible to select specific components of a derived type, using the `%` operator followed by the name of the component:

```
you%age  -> 27   you%name   -> Georg   couple%male%age
```

Operator overloading

- Initially no operations are defined for derived types.
- F90 allows to define the meaning of the build in operators for derived types using operator overloading (it is even possible to define new operators).
- This extends the flexibility and the power of F90 significantly.

Algorithm

New Term: Programm ist eine Sammlung von geschriebenen Instruktionen oder ein ausführbares Stück Software

Programmierstil:

- Unstrukturiertes Programmieren (sollte man vermeiden)
- Prozedurales Programmieren (FORTRAN90)
- Modulares Programmieren (FORTRAN90)
- Objektorientiertes Programmieren (C++, Java, Smalltalk)

Der IEEE Standard

- IEEE 754, Computer verwenden das Binärsystem

$$x := \pm 0.m_1 m_2 m_3 \ldots m_p \cdot \beta^e = \beta^e \sum_{i=1}^{p} m_i \beta^{-n_i}$$

  $0.m_1 m_2 m_3 \ldots m_p$ ist die Mantisse, β Basis, e der Exponent

- $\beta = 2$, $p = 24$ für einfache Genauigkeit
- $\beta = 2$, $p = 53$ für doppelte Genauigkeit

Achtung viele Kommazahlen können im Binärsystem nicht exakt dargestellt werden (β=10)

$$0.1 \sim 1.10011001100110011001101 \cdot 2^{-4}$$

Rundungsfehler

- Fehler durch die binäre Darstellung einer Zahl
- Rundungsfehler: nach der Anwendung mathematischer Funktionen (Plus, Minus) müssen die Ergebnisse wieder mit einer endlichen Zahl von Bits dargestellt werden
- es gibt ein ε mit

  $$1.0 + \varepsilon = 1.0$$

  großes Problem wenn man kleine und große Zahlen addiert
- numerische Differentiation

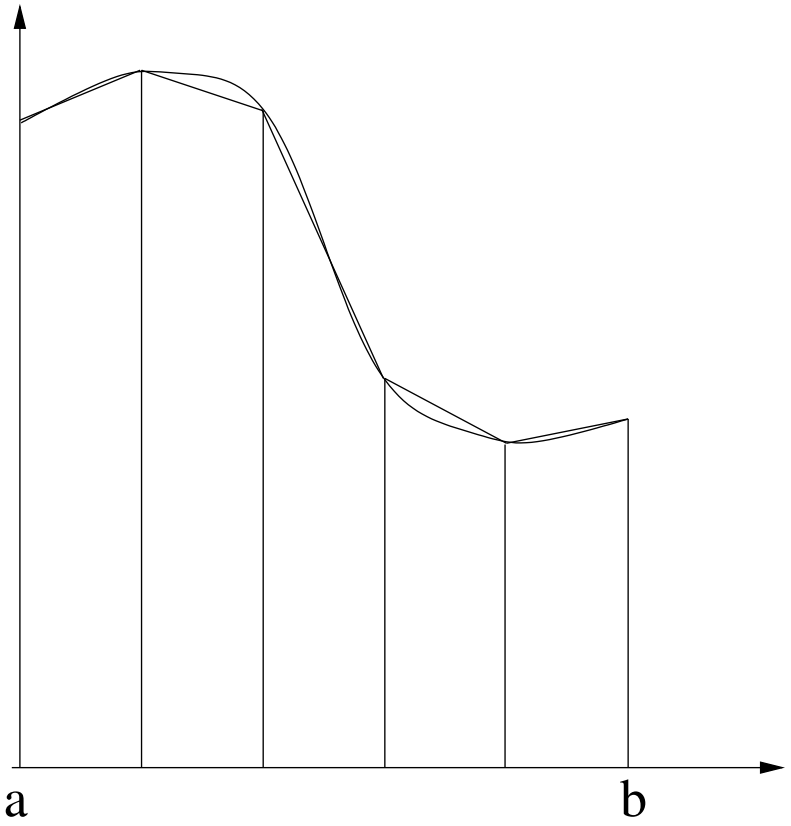
Numerische Integration, Newton-Cotes

- eindimensionales Integral

$$I(f) = \int_a^b f(x)dx$$

- Trapezregel:

  Funktion wird stückweise linear interpoliert, und die Fläche unter den Trapezen wird berechnet



$$I \approx \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]$$

Simpson's Integrationsformel

- interpoliert man die Funktion quadratisch (3 Stützpunkte) erhält man für das Integral:

$$I \approx \frac{1}{3}\Big[f(x_0)+4f(x_1)+f(x_2)\Big]$$

- Führt man eine solche Interpolation stückweise durch so ist das Integral für ein endliches Intervall durch

$$\begin{aligned} I \quad \approx \quad & \frac{h}{3}\Big[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)\ldots \\ & \ldots 2f(x_{n-2})+4f(x_{n-1})+f(x_n)\Big]. \end{aligned}$$

  gegen

- Der Fehler in der Integration ist proportional zu h^n:

Trapezregel: h^2

Simpsonregel: h^4

Approximation von bestimmten Integralen

Eine Näherung des bestimmten Integrals

$$I(f) = \int_a^b f(x)\mathrm{d}x \quad , \tag{1}$$

durch die Form

$$Q_n(f) = \sum_{i=0}^{n} \sigma_i f(x_i) \tag{2}$$

nennt man Quadratur. Die Punkte x_i sind die Abscissen

Newton–Cotes Formeln

- Integriere $f(x)$ zwischen a und b
- Ester Schritt: interpoliere $f(x)$ mit einem Interpolationpolynom, wobei $f_i := f(x_i^{(n)})$

$$f(x) = \sum_{i=0}^{n} f_i L_i^{(n)}(x) + r_n(x)$$

der Rest $r_n(x)$ gibt die Differenz zum exakten Wert an

- Wahl der Polynome: Lagrange Darstellung

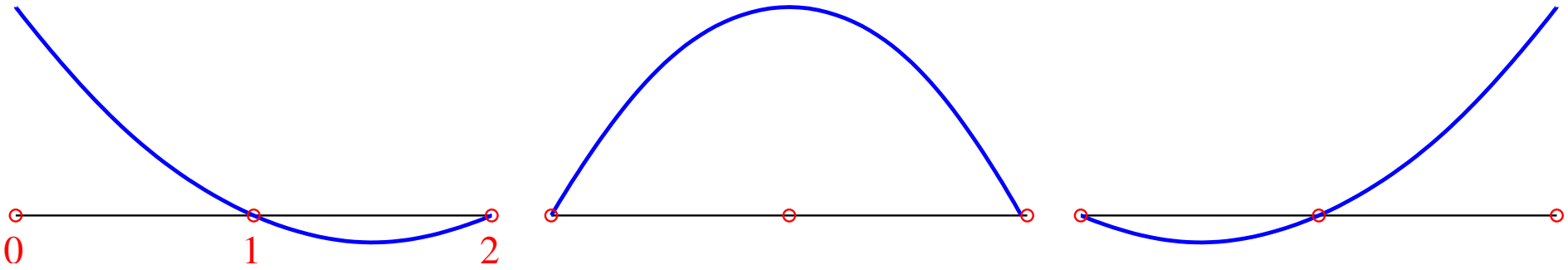
$$L_i^{(n)}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad \text{für} \quad i = 0, 1, \ldots, n \tag{3}$$

$$L_i^{(n)}(x_k) = \delta_{ik} \tag{4}$$

Simpson Integration

$n = 2$: $\quad x_0 = 0\,,\, x_1 = 1\,,\, x_2 = 2$

$$L_0(x) = \frac{(x-1)\cdot(x-2)}{2} \qquad L_1(x) = -\frac{x\cdot(x-2)}{1} \qquad L_2(x) = \frac{x\cdot(x-1)}{2}$$



$$\int_0^2 L_0(x)dx = \frac{1}{3} \qquad \int_0^2 L_1(x)dx = \frac{4}{3} \qquad \int_0^2 L_2(x)dx = \frac{1}{3}$$

$\Rightarrow$ Simpson Regel

$$\frac{1}{3}(f_0 + 4f_1 + f_2)$$

Geschlossene Newton–Cotes Integrationsgleichungen

- äquidistante Punkte $x_i^{(n)}$
- Integral:

$$I(f) = \sum_{i=0}^{n} \left(\int_a^b L_i^{(n)}(x)\mathrm{d}x \right) f(x_i^{(n)}) + \int_a^b r_n(x) \tag{5}$$

$$= (b-a) \sum_{i=0}^{n} \sigma_i^{(n)} f(x_i^{(n)}) + R_n(f) = (b-a) \sum_{i=0}^{n} \sigma_i^{(n)} f_i^{(n)} + R_n(f) \tag{6}$$

- Gewichte

$$\frac{1}{b-a} \int_a^b L_i^{(n)}(x)\mathrm{d}x = \sigma_i^{(n)}$$

- mit (Computer adaptiert)

$$x_i^{(n)} := a + i\frac{b-a}{n} = a + ih \tag{7}$$

Beispiele für geschlossene Newton–Cotes Formeln

n	$s\sigma_i^{(n)}$					s	$R_n(f)$	Name
1	1	1				2	$-\frac{h^3}{12} f''(\xi)$	Trapezoidal
2	1	4	1			6	$-\frac{h^5}{90} f^{(4)}(\xi)$	Simpson
3	1	3	3	1		8	$-\frac{3h^5}{80} f^{(4)}(\xi)$	$\frac{3}{8}$ rule
4	7	32	12	32	7	90	$-\frac{8h^7}{945} f^{(6)}(\xi)$	Milne rule

Problem: Uneigentliche Integrale z.B.

$$\int_0^1 \frac{1}{x^{1/2}} dx = 2x^{1/2}\Big|_0^1$$

Offene Newton–Cotes Formeln

- Wir vermeiden einfach die Endpunkte des gegebenen Integrals (a und b)
- Integral:

$$I(f) = \sum_{i=1}^{n-1} f_i \int_a^b \bar{L}_i^{(n)}(x)\mathrm{d}x + \bar{R}_n(f) \tag{8}$$

$$= (b-a)\sum_{i=1}^{n-1} \bar{\sigma}_i^{(n)} f_i + \bar{R}_n(f) \tag{9}$$

mit

$$\bar{L}_i^{(n)}(x) = \prod_{\substack{j=1 \\ j\neq i}}^{n-1} \frac{x-x_j}{x_i-x_j} \quad , \tag{10}$$

$$\bar{\sigma}_i^{(n)} = \frac{1}{n}\int_0^n \prod_{\substack{j=1 \\ j\neq i}}^{n-1} \frac{s-j}{i-j}\mathrm{d}s \quad . \tag{11}$$

Beispiele für offene Newton–Cotes Formeln

n	$s\sigma_i^{(n)}$					s	$R_n(f)$	name
2	1					1	$-\frac{h^3}{3} f''(\xi)$	Rectangular
3	1	1				2	$-\frac{h^3}{4} f''(\xi)$	
4	2	-1	2			3	$-\frac{14h^5}{45} f^{(4)}(\xi)$	
5	11	1	1	11		24	$-\frac{95h^5}{144} f^{(4)}(\xi)$	
6	11	-14	26	-14	11	20	$-\frac{41h^7}{140} f^{(6)}(\xi)$	

Zusammengesetzte Newton–Cotes Formeln

- geschlossene Newton–Cotes Formeln $\rightarrow$ kein konvergentes Quadraturverfahren
- besser: Formeln niedrigen Grades $\rightarrow$ kleine Teilintervalle
- Intervall $[a,b] \rightarrow N$ äquidistante Teilintervalle
- Zerlegung des bestimmten Integrals $I(f)$ in eine Summe von Integralen

$$I(f) = \int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{N-1} \int_{x_i}^{x_i+1} f(x)\mathrm{d}x \tag{12}$$

äquidistante Teilintervalle

- Das Integrationsintervall $[a,b]$ wird in N Intervalle $[x_i, x_{i+1}]$ zerlegt
- Berechnung der einzelnen Stützstellen:

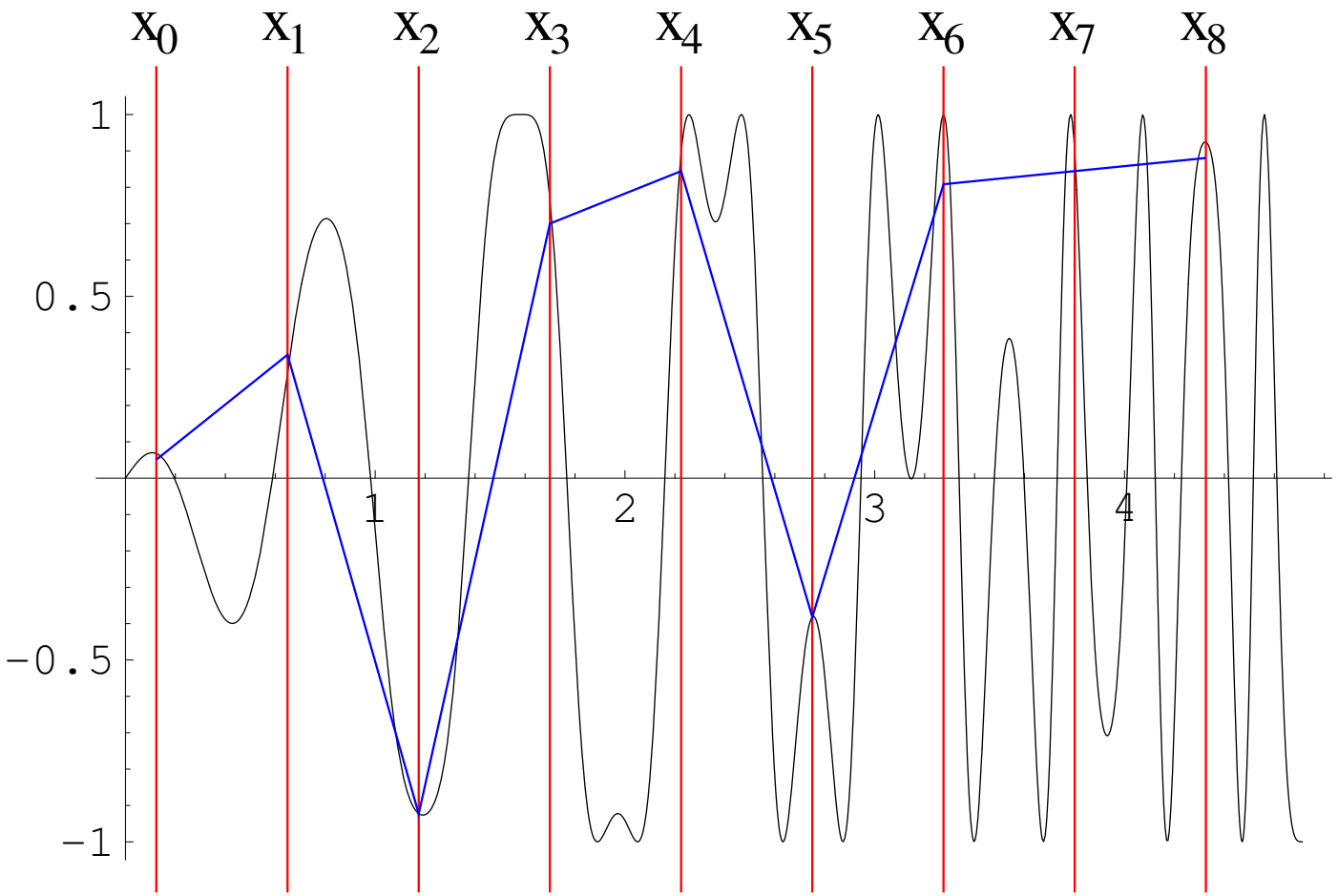
$$x_i = a + ih \quad \text{for} \quad i = 0,1,\ldots,N \quad \text{und} \quad h := \frac{b-a}{N} \tag{13}$$

  Berechnung der Stützstellen
  - Fehlerfortpflanzung beachten
  - schneller Algorithmus (geringe CPU Belastung)

Algorithmus:

- Zerlegung des Integrationsbereiches in Teilintervalle
- Für die Integrale $[x_i, x_{i+1}]$ $\longrightarrow$ Newton–Cotes Formeln
- Aufsummieren der Teilbeiträge auf den $[x_i, x_{i+1}]$
- Konvergenz beachten
  - Analyse der zu integrierenden Funktion, Daten - oder -
  - mehrere Durchläufe mit verschiedener Anzahl an Stützstellen

schlecht gewählte Stützpunkte:



$$f(x) = \sin(x\cos(8 * x)), \qquad [0, 1.5\pi] \tag{14}$$

Anwendung: stückweise Integration mit der Trapezformel

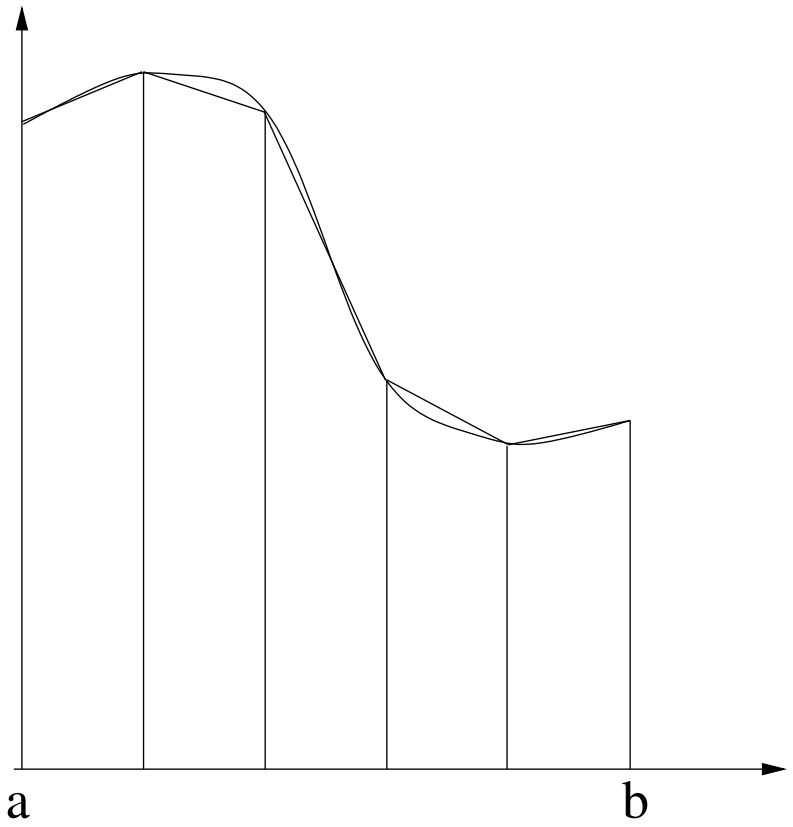
- zur Wiederholung - geschlossene Newton–Cotes:

$$I(f) \approx (b-a) \sum_{i=0}^{n} \sigma_i^{(n)} f_i^{(n)} \tag{15}$$

mit $s = 2$, $n = 1$ und $s\sigma_i^{(n)} = 1,1$

- Trapezformel:

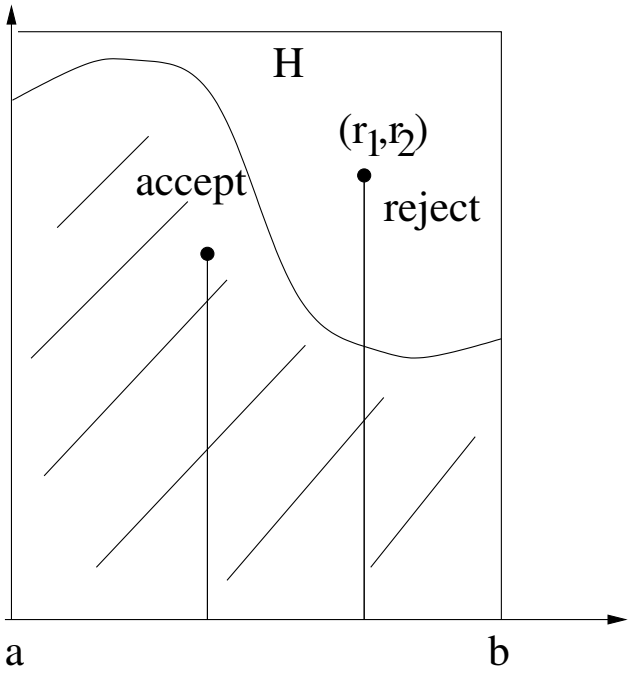
$$\begin{aligned} I(f) &\approx \sum_{i=0}^{N-1} \left\{ (x_{i+1} - x_i) \left[\frac{f(x_i)}{2} + \frac{f(x_{i+1})}{2} \right] \right\} \\ &= \frac{h}{2} \left[f(a) + 2 \sum_{i=1}^{N-1} f(x_i) + f(b) \right] - \frac{h^2(b-a)}{12} \frac{1}{12} \sum_{i=0}^{N-1} f''(\xi_i) \quad , \end{aligned} \tag{16}$$



$$I \approx \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]$$

Monte Carlo Integration

Rejection Methode



- das Maximum der Funktion im Intervall $[a,b]$ soll H sein
- erzeuge zwei Zufallszahlen: r_1 im Intervall $[a,b]$ und r_2 im Intervall $[0,H]$
- überprüfe ob der Punkt (r_1,r_2) unterhalb der Kurve liegt $(r_2 < f(r_1))$

- das Integral ist gegeben durch

$$F_n \approx (b-a)H\frac{n_s}{n},$$

  wobei n_s die Anzahl der Punkte unterhalb der Kurve ist, und n die Zahl der erzeugten Punkte ist

- Dieses Verfahren wird hauptsächlich bei der Berechnung von hochdimensionalen Integralen verwendet.

$$I(f) = \int\int\int\ldots\int f(x)\mathrm{d}x^n$$

Monte-Carlo

```
MODULE functions
CONTAINS
  FUNCTION f1(x)
    REAL :: f1,x
    f1=x**2
  END FUNCTION f1
  FUNCTION f2(x)
    REAL :: f2,x
    f2=exp(x)
  END FUNCTION f2
END MODULE functions

PROGRAM test
  USE functions
  USE integrate
  IMPLICIT NONE
  REAL a,b
  WRITE(*,*) mc_int(f1,0.,1.,100),mc_int(f2,0.,1.,100)
  WRITE(*,*) simp_int(f1,0.,1.,100), &
             simp_int(f2,0.,1.,100)
END PROGRAM test
```

```
MODULE integrate
  CONTAINS
  FUNCTION mc_int(f,xstart,xend,n)
    INTERFACE
      FUNCTION f(x); REAL f,x; END FUNCTION
    END INTERFACE
    REAL :: mc_int,xstart,xend,d,x
    INTEGER:: n,i

    simple_int=0
    d=(xend-xstart)
    DO i=1,n
      CALL RANDOM_NUMBER( x)
      mc_int=mc_int + f(xstart+d*x)
    ENDDO
    mc_int=mc_int/n
  END FUNCTION mc_int
```

```
  FUNCTION simp_int(f,xstart,xend,intersections)
    INTERFACE
      FUNCTION f(x); REAL f,x; END FUNCTION
    END INTERFACE
    REAL :: simp_int,xstart,xend,d
    INTEGER:: in,intersections,i
    in=(intersections/2)*2  ! round towards 2

    simp_int=0
    d=(xend-xstart)/in
    DO i=1,in-1,2
      simp_int=simp_int + f(xstart+d*(i-1))*d/3.
      simp_int=simp_int + f(xstart+d*i)  *d*4/3.
      simp_int=simp_int + f(xstart+d*(i+1))*d/3.
    ENDDO
  END FUNCTION simp_int
END MODULE integrate
```

Monte-Carlo: Konvergenz

Monte-Carlo Methoden konvergieren sehr langsam:

N	MC x^2	Simpson x^2	MC $\exp(x)$	Simspon $\exp(x)$
5	0.2994335	0.3333333	2.204745	1.718319
10	0.4476958	0.3333333	1.785828	1.718283
100	0.3611916	0.3333333	1.793215	1.718282
1000	0.3512844	0.3333333	1.719919	1.718282
100000	0.3344397	0.3333333	1.716187	1.718282

der Fehler fällt mit $\sqrt{1/N}$ ab

nehmen wir aber an, daß die Funktion von k Variablen abhängt sieht die Sache anders aus:

Monte-Carlo: $\sqrt{1/N} = (1/N)^{1/2}$
Simpson: $(1/N^{1/k})^4 = (1/N)^{4/k}$

Numerische Differentiation

Interpolationsformeln

- reelle differenzierbare Funktion f
- numerische Näherung der Ableitung von f an $\bar{x}$
- ähnlich *numerischer Quadratur*
  ersetze f durch ein Polynom ϕ,

$$f(x) = \phi(x) + R(x) \tag{17}$$

ϕ ... Polynom mit bestimmten Grad, $R(x)$ ... Fehler der Näherung

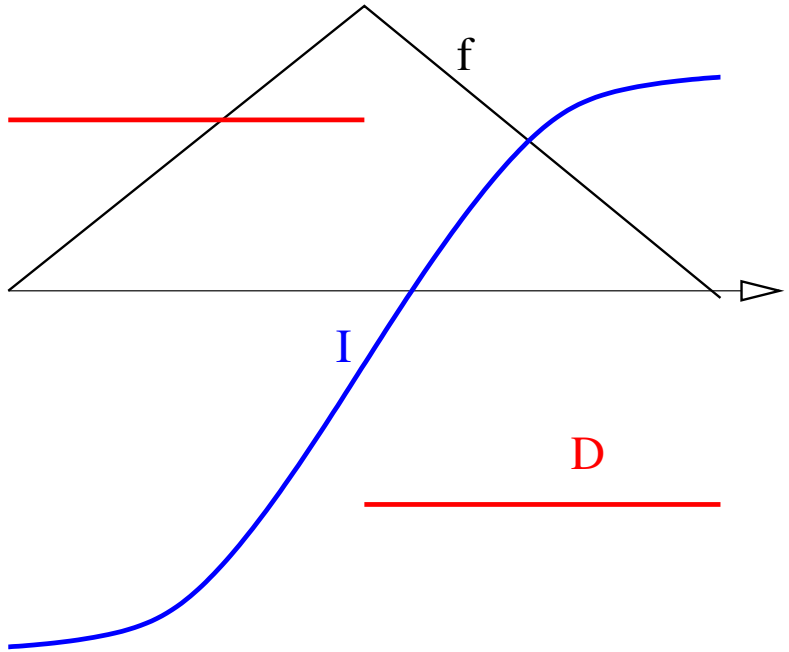
Numerische Differentiation

Differentiation "rauht" auf

- kleine Änderungen in x können das Ergebnis stark verändern
- Rest $R(x)$ wird durch Differentiation in der Ordnung *erhöht*

$$R(x) \approx o(x^2) \quad dR(x)/dx = o(x)$$

- Rundungsfehler stellen das zweite Problem dar



Linear Approximation

Lineare Interpolation: $x_0 = \bar{x}$, $x_1 = \bar{x} + h$

Lagrange Darstellung:

$$\begin{aligned} \phi(x) &= f(\bar{x}) \frac{x - (\bar{x}+h)}{\bar{x} - (\bar{x}+h)} + f(\bar{x}+h) \frac{x - \bar{x}}{(\bar{x}+h) - \bar{x}} \\ &= \frac{f(\bar{x}+h) - f(\bar{x})}{h} x + \frac{(\bar{x}+h) f(\bar{x}) - \bar{x} f(\bar{x}+h)}{h}. \end{aligned} \tag{18}$$

Ableitung

$$\frac{f(\bar{x}+h) - f(\bar{x})}{h}$$

Fehler ist von der Ordnung h

Wegen Rundungsproblem kann man h aber nicht beliebig klein machen

x	$(e^{1+x} - e^x)/x$	REAL(KIND=8)
10^{-2}	2.858844	2.85884195487388
10^{-4}	2.719879	2.71964142253323
10^{-6}	2.741814	2.71829541995672
10^{-8}	4.768372	2.71828196840573
10^{-10}	0.000000	2.71828204390090
10^{-12}	0.000000	2.71831446241322
exact	2.718281	2.71828182845904

Quadratische Näherung

Quadratische Interpolation: $x_0 = \bar{x} - h$, $x_1 = \bar{x}$, $x_2 = \bar{x} + h$.

Lagrange Form:

$$\begin{aligned}\phi(x) \quad = \quad & \frac{f(\bar{x}-h) - 2f(\bar{x}) + f(\bar{x}+h)}{2h^2} x^2 - \\ & - \frac{(2\bar{x}+h)f(\bar{x}-h) - 4\bar{x}f(\bar{x}) + (2\bar{x}-h)f(\bar{x}+h)}{2h^2} x + \\ & + \frac{(\bar{x}^2+h\bar{x})f(\bar{x}-h) - 2(\bar{x}^2-h^2)f(\bar{x}) + (\bar{x}^2-h\bar{x})f(\bar{x}+h)}{2h^2}.\end{aligned} \tag{19}$$

Differentiation an $\bar{x}$ gibt den Quotienten:

$$\begin{aligned} \phi'(\bar{x}) &= \frac{-hf(\bar{x}-h)+hf(\bar{x}+h)}{2h^2} \\ &= \frac{f(\bar{x}+h)-f(\bar{x}-h)}{2h} \end{aligned} \tag{20}$$

Fehler ist nun von der Ordnung h^2

Um die Genauigkeit zu erhöhen $\rightarrow$ Erhöhung der Zahl der Gridpunkte

$$\begin{aligned} f(x) &= \sum_{i=0} n f_i L_i^{(n)}(x) + R_n(x) \\ \frac{\mathrm{d}}{\mathrm{d}x} f(x) &= \sum_{i=0} n f_i \frac{\mathrm{d}}{\mathrm{d}x} L_i^{(n)}(x) + R_n(x). \end{aligned} \tag{21}$$

äquidistantes Gitter um $\bar{x}$

$$f'(x) = \frac{1}{h} \sum_{i=-k} k\beta_i^{(k)} f_i + r_{2k}(f;\bar{x}), \tag{22}$$

mit

$$x_i = \bar{x} + ih \qquad \text{und} \qquad f_i = f(x_i) \qquad \text{für} \qquad i = -k, \ldots, k \tag{23}$$

$r_{2k}(f;\bar{x})$ Differenz zwischen Näherung und exakten Wert

Symmetrische Formeln für erste Ableitungen:

$2k$	$s\beta_i^{(k)}$							s	$r_{2k}(f;\bar{x})$
2	-1	0	1					2	$-\frac{h^2}{6} f'''(\xi)$
4	1	-8	0	8	-1			12	$-\frac{h^4}{30} f^{(5)}(\xi)$
6	-1	9	-45	0	45	-9	1	60	$-\frac{h^6}{140} f^{(7)}(\xi)$

Algebra

In den modernen numerischen Rechnungen sind Matrix–Matrix und Matrix–Vektor Multiplikationen und Additionen sehr häufig. In vielen sehr großen Programmpaketen wird der größte Teil der gesamten Rechenzeit auf Matrix–Operationen oder Eigenwert und Eigenvektor Problemen verwendet. $\rightarrow$ OPTIMIERUNG

Matrix–Multiplikation

- zwei $n \times n$ Matrizen A und B, $\rightarrow AB = C$
- sehr zeitaufwendig, Multiplikationen und Additionen
- C:

$$c_{i,k} = \sum_{j=1}^{n} a_{i,j} b_{j,k},$$

- jeder der n^2 Elemente braucht $O(n)$ Rechenzeit

manuelle Matrix–Matrix Multiplikation

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{pmatrix} \tag{24}$$

- 8 Multiplikationen und 4 Addition (normalerweise dauern Multiplikationen viel länger als Additionen
- Load und Store Operationen: hängt stark von der Architektur ab (Anzahl der Register, Cache etc.)

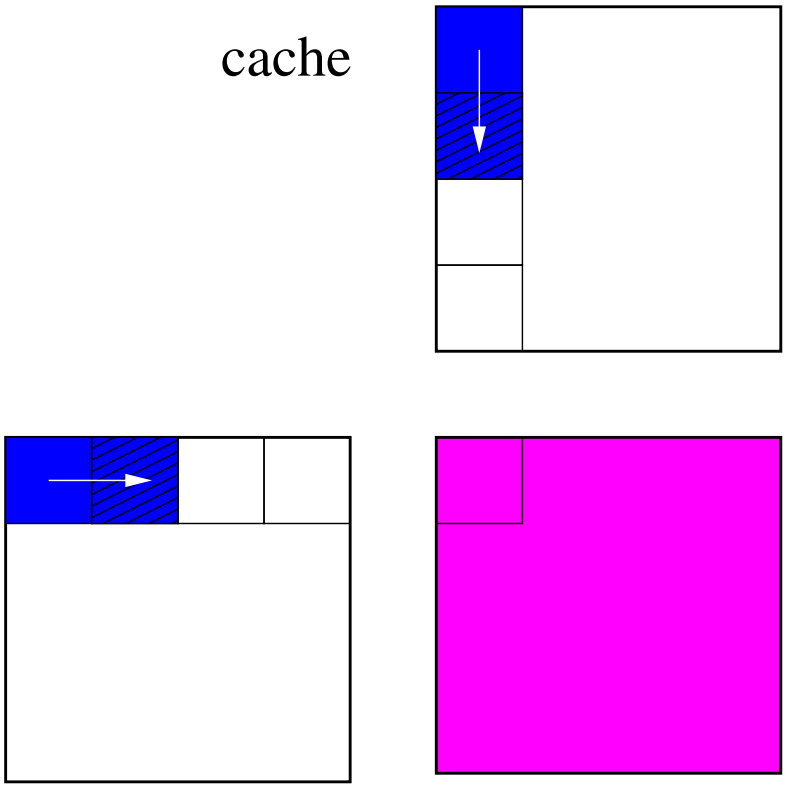
Block Optimierung: Basic Linear Algebra (BLAS)

zerlegt die Matrix in kleine (32×32) große Blöcke und multipliziert diese miteinander
Blöcke liegen vollständig im Cache auf der CPU $\Rightarrow$ Peak performance

- Speicherzugriff von einen Block auf den nächsten: $2 \times 32 \times 32$
- Rechenoperationen aber: $32 \times 32 \times 32$ zur Berechnung eines Subblocks
- 90% der Peakperformance (5 GFlops)
- BLAS:

  `http://www.cs.utexas.edu/users/kgoto/`

  `http://math-atlas.sourceforge.net/`



Optimierung

- Strassen 1969 (siehe Referenzen im Skript)
- Idee: $n \times n$ Matrix in 4 Teile mit $n/2 \times n/2$
- Untermatrizen mit der Dimension $n/2 \times n/2 \rightarrow$ einzelne Elemente
- Produkt von zwei 2×2 Matrizen
- rekursive Zerlegung
- normale Matrix Multiplikation: 8 Multiplikationen und 4 Additionen
- Strassen: 7 Multiplikationen und 18 Additionen

Beispiel: Multiplikation zweier komplexer Zahlen

wir wollen Multiplikationen vermeiden weil sie teuer sind

- Multiplikation zweier komplexer Zahlen $x_1 + ix_2$ und $y_1 + iy_2$ ist gegeben durch

$$(y_1 + ix_2)(y_1 + iy_2) = z_1 + iz_2$$

mit

$$\begin{aligned} z_1 &= x_1y_1 - x_2y_2, \\ z_2 &= x_1y_2 + x_2y_1. \end{aligned}$$

- aber es genügen auch drei Multiplikationen:

$$\begin{aligned} p_0 &= x_1y_1, \\ p_1 &= x_2y_2, \\ p_2 &= (x_1 + x_2)(y_1 + y_2), \end{aligned}$$

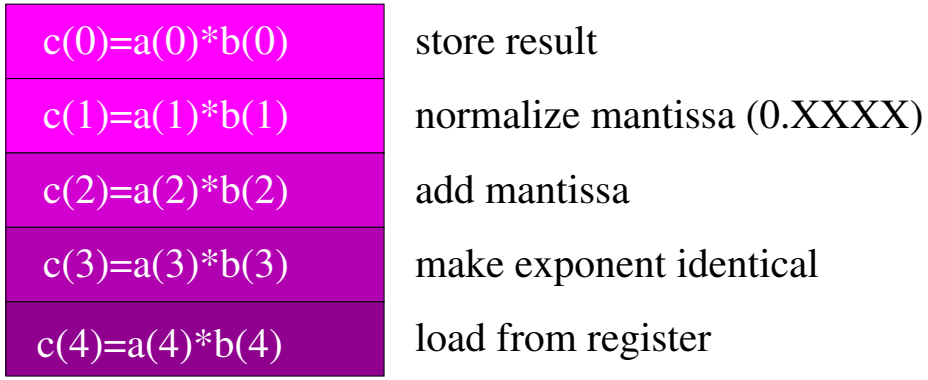
dann ist

$$z_1 = p_0 - p_1 \qquad z_2 = p_2 - p_0 - p_1$$

Computer–Architekturen

- Vektorcomputer: kann Vektoren bestimmter Länge (128, 256 Elemte) sehr schnell verarbeiten, “Supercomputer”
- Skalarprozessor: verarbeitet ein Datum nach dem anderen
- Super–Skalar (x86, RISC) kann mehrere skalare Anweisungen in Pipelines verarbeiten

  mehrere Instruktionen werden gleichzeitig verarbeitet während eine initialisiert wird, “retired” die andere



- Multiply–Add, manche Rechnerarchitekturen können in einem Schritt eine Addition und eine Multiplikation ausführen (IBM Power RISC)
- symmetrische Parallel–Computer: mehrere Super–Skalar Prozessoren → Kommunikations–Zeiten

Rechenzeit

Die meisten Optimierungen versuchen die Skalierbarkeit des Problems zu verbessern: Matrix–Matrix Multiplikation: $O(n^3)$ Problem $\rightarrow$ doppelt so viele Elemente ergibt die 8–fache Rechenzeit.

Besser: Verfahren niedrigerer Ordnung

- $T(n)$ Zeit für die Multiplikation von 2 $n \times n$ Matrizen.
- Strassen: $T(n) = 7T(n/2) + 18n^2$ ergibt die Ordnung $T(n) \in O(n^{\log_2 7}) \in O(n^{2.81})$
- Bestes Verfahren: Coppersmith und Winograd, 1987 (siehe Referenzen im Skript) $O(n^{2.376})$
- Änderung der Methoden in der Physik $\rightarrow O(n)$ oder $O(n\log(n))$ Methoden

Rechner für NIC

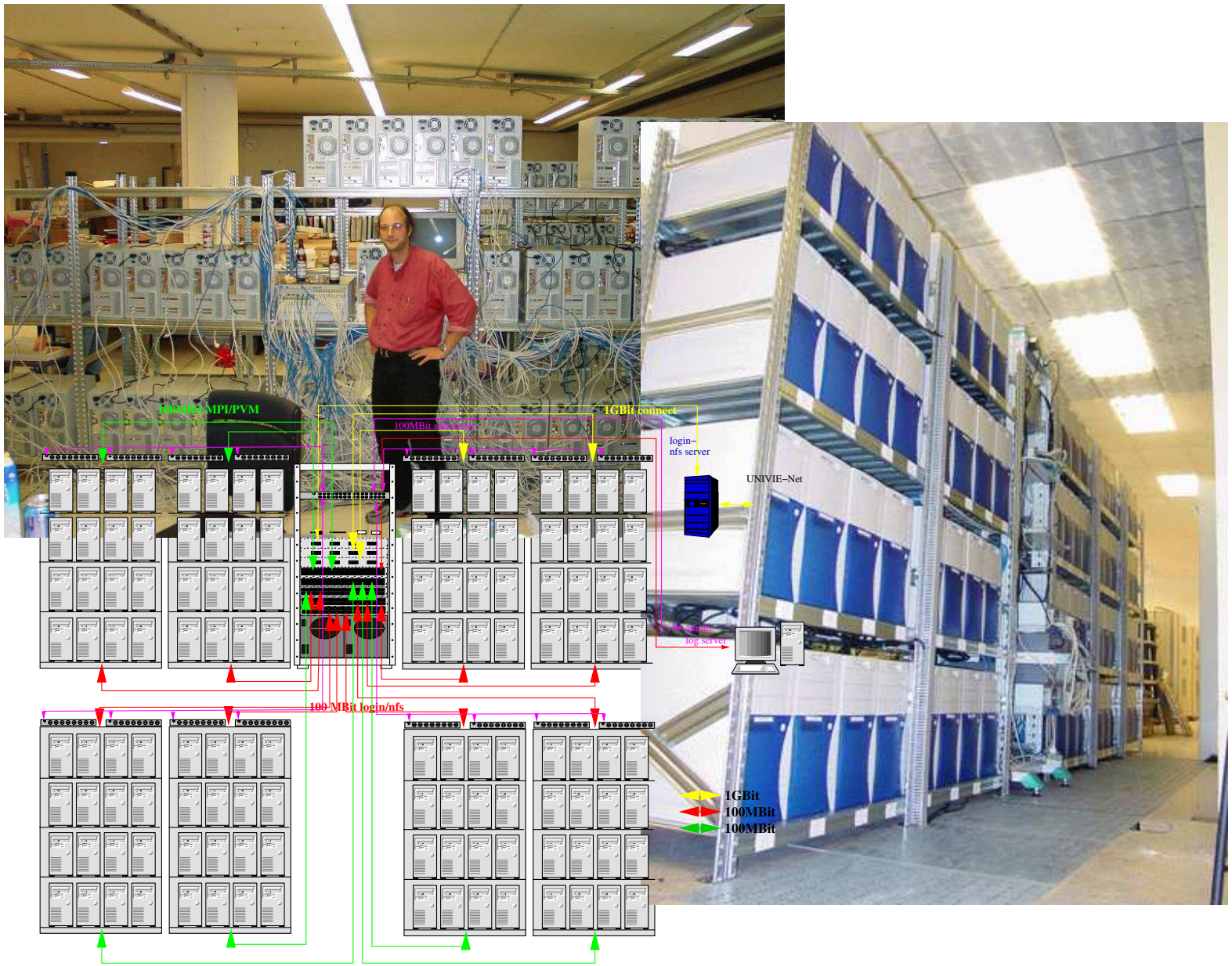
Schrödinger II, massiv-paralleler Cluster an der Uni Wien

- 196 P4 2533 MHz Prozessoren, 2 GB RAM
- File-, Login–Server, 250GB HD, 100MBit fileshare Netz
- 196x 1 Gbit Kommunikations-Netz (MPI)
- Leistung $>$ 504.30 GFlops
- Betriebssystem: Linux

IBM SP/3 an der TU-Wien

- 48 IBM Power 3 Prozessoren (375MHz)
- 3x 16 Proz. Power3 SMP High Node
- 1 GByte/CPU, 360GB HD für das ganze System
- Betriebssystem: IBM AIX
- Leistung: $>$ 45 GFlops

Schrödinger I



Die Schnellsten in Österreich (der Welt)

IBM SP3

NEC SX–4 Vektorcomputer

SGI Origin 2000

Cray SX–6

IBM SP3

www.top500.org

ASCI White (IBM)	7226 GFlops
Es45 (Compaq)	4059 GFlops
ASCI Red (Intel)	2379 GFlops

Die Liste der schnellsten Computer der Welt: `www.top500.org`

1	IBM/DOE	United States/2004	BlueGene/L beta-System BlueGene/L DD2 beta-System (0.7 GHz PowerPC 440) / 32768 IBM
2	NASA/Ames Research Center/NAS	United States/2004	Columbia SGI Altix 1.5 GHz, Voltaire Infiniband / 10160 SGI
3	The Earth Simulator Center	Japan/2002	Earth-Simulator / 5120 NEC
4	Barcelona Supercomputer Center	Spain/2004	MareNostrum eServer BladeCenter JS20 (PowerPC970 2.2 GHz), Myrinet / 3564 IBM
5	Lawrence Livermore National Laboratory	United States/2004	Thunder Intel Itanium2 Tiger4 1.4GHz - Quadrics / 4096 California Digital Corporation
6	Los Alamos National Laboratory	United States/2002	ASCI Q ASCI Q - AlphaServer SC45, 1.25 GHz / 8192 HP
7	Virginia Tech	United States/2004	System X 1100 Dual 2.3 GHz Apple XServe/Mellanox Infiniband 4X/Cisco GigE / 2200
8	IBM - Rochester	United States/2004	BlueGene/L DD1 Prototype (0.5GHz PowerPC 440 w/Custom) / 8192 IBM/ LLNL
9	Naval Oceanographic Office	United States/2004	eServer pSeries 655 (1.7 GHz Power4+) / 2944 IBM
10	NCSA	United States/2003	Tungsten PowerEdge 1750, P4 Xeon 3.06 GHz, Myrinet / 2500 Dell
100	Credit Suisse	Switzerland/2004	BladeCenter HS20 Xeon 2.8 GHz, Gig-Ethernet / 1500 / IBM
295	CSC (Center for Scientific Computing)	Finland/2002	pSeries 690 1.1GHz / 512 IBM
> 500	Univ. Wien	Austria/2003	P4 2533 MHz Prozessoren / 196 /init.at

1	70720	91750
2	51870	60960
3	35860	40960
4	20530	31363
5	19940	22938
6	13880	20480
7	12250	20240
8	11680	16384
9	10310	20019.2
10	9819	15300
100	2026	8400
295	1170	2253
>500	504	971

Zufalls–Zahlen

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

John Von Neumann (1951)

Pseudo Zufallszahlen: Die "Zufallszahlen" werden mittels eines mathematischen Algorithmus berechnet $\rightarrow$ die "zufl̈ligkeit" kann vorausberechnet werden.

- Pseudo Zufallszahlen: unabhängige, gleichverteil auf $U([0,1[)$
- andere Verteilungen mittels Transformation möglich
- Verschiedene Verfahren für verschiedene Probleme $\rightarrow$ "eingebaute" Funktionen nicht immer brauchbar

Anwendungen

- Kryptographie: Bei der Verschlüsselungen sind session keys die für die eigentliche Übertragung verwendet werden, sehr wichtig. Gute Gleichverteilung. Bei besonderen Inhalten: Hardwarelösungen (Banken, Militär)
- Monte Carlo Methoden: Hier wird der Phasenraum mit der Verteilungsfunktion beschrieben und entsprechende Algorithmen verwendet
  - schnelle Generierung der Zufallszahlen
  - lange Zykluslängen (bis sich die Pseudo Zufallszahlen wiederholen), bei großen Systemen sehr wichtig

einfache Algorithmen

multiply-with-carry oder recursion-with-carry Generator von George Marsaglia:

$$\begin{aligned} S &= 2111111111\Delta X_{n-4} + 1492\Delta X_{n-3} + 1776\Delta X_{n-2} + 5115\Delta X_{n-1} + C \\ X_n &= S \quad \text{modulo} \quad 232 \\ C &= \text{floor} \quad (S/232) \end{aligned} \tag{25}$$

- X und C 32–bit unsigned integers
- S 64–bit unsigned integer
- initialisierung von X und C (Uhrzeit, Netzwerk–Traffic etc.) **Spezifikationen:**
  - 32-bit integer output
  - cycle length is 3×1047
  - very good randomness

professionelle Algorithmen

The R250 Pseudo-random number generator

- Kirkpatrick, S., and E. Stoll, 1981; A Very Fast Shift-Register Sequence Random Number Generator, Journal of Computational Physics, V. 40. p. 517
- Maier, W.L., 1991; A Fast Pseudo Random Number Generator, Dr. Dobb's Journal, May, pp. 152 - 157
- it has a very long period 2^{249}
- very fast generator, but with 250 numbers latency
- for each bit:

$$I_k = c_1 I_{k-1} + c_2 I_{k-2} + c_3 I_{k-3} + \ldots + c_{p-1} I_{k-p+1} + I_{k-p} \mathrm{mod} 2 \tag{26}$$

- useful: $q = 103$, $p = 250 \rightarrow I_k = I_{k-103} + I_{k-250}$

Linear congruential Generator: LCG

Der linear congruential generator (LCG) wurde von Lehmer 1948 vorgeschlagen. Pseudo Zufallszahlen Generator mit Rekursion:

- Algorithmus

$$y_{n+1} = a y_n + b (\text{modulo} \quad m) \tag{27}$$

- Startwert y_0
- Funktion:

$$LCG(m, a, b, y_0) \tag{28}$$

Generating Gaussian Random Numbers

- Algorithmus: This transformation takes random variables from one distribution as inputs and outputs random variables in a new distribution function. Probably the most important of these transformation functions is known as the Box-Muller (1958) transformation. It allows us to transform uniformly distributed random variables, to a new set of random variables with a Gaussian (or Normal) distribution.

$$\begin{aligned} y_1 &= \sqrt{(}-2\ln(x1))\cos(2\pi x_2) \\ y_2 &= \sqrt{(}-2\ln(x1))\sin(2\pi x_2) \end{aligned} \tag{29}$$

- two problems:
  1. It is slow because of many calls to the math library.
  2. It can have numerical stability problems when x1 is very close to zero.

The polar form of the Box-Muller transformation is both faster and more robust numerically. The algorithmic description of it is:

```
FLOAT x1, x2, w, y1, y2;
DO {
  x1 = 2.0 * ranf() - 1.0;
  x2 = 2.0 * ranf() - 1.0;
  w = x1 * x1 + x2 * x2;
} WHILE ( w >= 1.0 );

w = SQRT( (-2.0 * LN( w ) ) / w );
y1 = x1 * w;
y2 = x2 * w;
```

where ranf() is the routine to obtain a random number uniformly distributed in [0,1].

Normale Differential Gleichungen

Einführung

- Beispiele für Differentialgleichungen

$$\frac{du}{dt} = -cu(t) \quad , \qquad \text{Wachstumsgleichung;} \tag{30}$$

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = F(t) \quad , \qquad \text{Pendel;} \tag{31}$$

$$L\frac{d^2Q}{dt^2} + R\frac{dQ}{dt} + \frac{Q}{C} = U(t) \quad , \qquad \text{Gedämpfter Oszillator;} \tag{32}$$

t ist die unabhängige Variable

Normale Differential Gleichungen

- Generell kann man alle obigen Differentialgleichungen als Lösung der Gleichung:

$$f(t, y, \dot{y}, \ldots, y^{(n)}) = 0, \tag{33}$$

  schreiben, wobei $y(t)$ eine reale Funktion ist

- $y^{(n)}$ steht für $\frac{d^n y}{dt^n}$.

- z.B.

$$L\frac{d^2Q}{dt^2} + R\frac{dQ}{dt} + \frac{Q}{C} - U(t) = 0$$

$$f = -U(t) + \frac{1}{C}y + Ry^{(1)} + Ly^{(2)}$$

- IVP initial value problem, Anfangswertproblem

- BVP boundary value problem, Randwertproblem

Beispiele (IVP)

Feder und träge Masse:

$$\ddot{x} + \frac{a}{m}\dot{x} + \frac{k}{m}x = g + \frac{1}{m}F(t), \tag{34}$$

mit den Anfangsbedingungen:

- $x(0) = x_0, \dot{x}(0) = v_0$; für $m = 10, k = 140, a = 90,$
- mit einer externen Kraft: $F(t) = 5\sin(t)$

Lösung für $x_0 = 0, v_0 = -1$:

$$x(t) = \frac{1}{500}\left(-90e^{-2t} + 99e^{-7t} + 13\sin(t) - 9\cos(t)\right). \tag{35}$$

Beispiele (BVP)

Differentialgleichung

$$\ddot{y} + \pi^2 y = 0, \tag{36}$$

mit den Randbedingungen:

- $y(0) = 2, y(1/2) = -2$
- Lösung ist gegeben durch

$$y(t) = 2\cos(\pi t) - 2\sin(\pi t)$$

System von Differentialgleichungen

System von normalen gekoppelten Differentialgleichungen erster Ordnung:

$$\dot{\mathbf{Y}} = \mathbf{F}(t, \mathbf{Y}), \quad \mathbf{Y}(a) = \mathbf{A}. \tag{37}$$

Hier sind $\mathbf{Y}$ und $\mathbf{A}$ n-Vektoren und $\mathbf{F}$ ist eine nicht lineare vektorwertige Funktion in $R \times R^n$.

$$\mathbf{Y} = \begin{pmatrix} Y_1 \\ Y_2 \\ \dots \\ Y_n \end{pmatrix}, \mathbf{A} = \begin{pmatrix} A_1 \\ A_2 \\ \dots \\ A_n \end{pmatrix}, \mathbf{F} = \begin{pmatrix} F_1(t, Y_1, Y_2, \dots, Y_n) \\ F_2(t, Y_1, Y_2, \dots, Y_n) \\ \dots \\ F_n(t, Y_1, Y_2, \dots, Y_n) \end{pmatrix}. \tag{38}$$

Differentialgleichung zweiter Ordnung

läßt sich als zwei gekoppelte Differentialgleichungen erster Ordnung schreiben

$$0 = -U(t) + \frac{1}{C}y + R\dot{y} + L\ddot{y}$$

mit $Y_1 = y$ und $Y_2 = \dot{y} = \dot{Y}_1 \Rightarrow$ wird zu

$$0 = -U(t) + \frac{1}{C}Y_1 + RY_2 + L\dot{Y}_2$$

zwei gekoppelte Gleichungen

$$\begin{aligned} \dot{Y}_1 &= Y_2 \\ \dot{Y}_2 &= -\frac{1}{L}\left(-U(t) + \frac{1}{C}Y_1 + RY_2\right) \end{aligned}$$

Numerische Lösung

Skalarer Fall - Diskretisierung

$$\dot{y} = f(t,y), \quad y(a) = \mathbf{A} \tag{39}$$

- Sequenz von Näherungswerten für $y(t) \rightarrow y_1, y_2, y_3, \ldots$
- an den Punkten t_1, t_2, t_3
- konstanter Abstand h
- $y(t_i)$ Lösung von (39) für $t = t_i$
- y_i Näherung von $y(t_i)$.

Fehler

- der wahre oder globale Fehler:

$$\varepsilon_{i+1} = y(t_{i+1}) - y_{i+1}. \tag{40}$$

für $t = t_{i+1}$
sehr schwierig zu berechnen!

- lokale Fehler: für

$$\dot{u} = f(t,u), \quad u(t_i) = y_i, \tag{41}$$

der lokale Fehler an $t = t_{i+1}$ ist gegeben durch

$$d_{i+1} = u(t_{i+1}) - y_{i+1}. \tag{42}$$

- die meisten Methoden schätzen den lokalen Fehler in jeden Schritt ab und bestimmen h

Ein-Schritt Methoden

- Näherung für Lösung der Gleichung (39) im Intervall $[a,b]$
- t Punkte haben gleichmäßigen Abstand, mit n und $h=(b-a)/n$, $t_i = a+ih$, $i = 0,1,\ldots,n$
  wobei $a < b$
- in der Ein-Schritt Methode sieht ein Schritt wie folgt aus:

$$y_{i+1} = y_i + h\delta(t_i, y_i), \quad y_0 = y(t_0), \tag{43}$$

- δ ist eine Funktion die unsere Methode charakterisiert

Taylor Methoden

Expansion der Lösung $y(t)$:

$$y(t_{i+1}) = y(t_i) + h\left(\dot{y}(t_i) + \ldots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}\right) + y^{(p+1)}(\xi_i)\frac{h^{p+1}}{(p+1)!}, \tag{44}$$

wobei $t_i \leq \xi_i \leq t_{i+1}$. Die Kontinuität von $y^{(p+1)}(t)$ impliziert, daß der Fehler begrenzt ist in $[a,b]$ und daher gilt

$$y^{(p+1)}(\xi_i)\frac{h^{p+1}}{(p+1)!} = O(h^{p+1}) = hO(h^p). \tag{45}$$

mit $\dot{y} = f(t,y) \rightarrow$

$$y(t_{i+1}) = y(t_i) + h\left(f(t_i,y(t_i)) + \ldots + f^{(p-1)}(t_i,y(t_i))\frac{h^{p-1}}{p!}\right) + hO(h^p), \tag{46}$$

mit

$$\begin{aligned} f^{(1)}(t,y) &= \frac{df}{dt} = \frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}\frac{dy}{dt} = f_t(t,y) + f_y(t,y)f(t,y), \\ f^{(k)}(t,y) &= f_t^{(k-1)}(t,y)f(t,y), \qquad k = 2,3,\ldots. \end{aligned} \tag{47}$$

Implementation

Näherungslösung von der Ordnung p des IVP (17) auf $[a,b]$, mit $h=(b-a)/n$:

$$y_{i+1} = y_i + h\left(f(t_i,y_i)+\ldots+f^{(p-1)}(t_i,y_i)\frac{h^{p-1}}{p!}\right), \tag{48}$$

$$t_{i+1} = t_i + h, \qquad i=0,1,\ldots,n-1, \tag{49}$$

mit $t_0=a$, und $y_0=\mathbf{A}$.

Taylorreihenentwicklung von t um t^i bis zur ersten Ordnung führt zu Eulers Methode (einfache Vorwärtsdifferenzen)

$$\begin{aligned} y_{i+1} &= y_i + h\dot{y}_i + o(h^2) \Rightarrow \\ y_{i+1} &= y_i + hf(t_i,y_i) \end{aligned}$$

Runge-Kutta Methode

die Standardmethode für gewöhnliche Differentialgleichungen

- gewöhnliche Differentialgleichung erster Ordnung

$$\frac{dy}{dt} = f(y,t)$$

- Runge-Kutta Methode zweiter Ordnung

$$\begin{aligned} k_1 &= f(t_i, y_i)h \\ k_2 &= f(t_i + h/2, y_i + k_1/2)h \\ y_{i+1} &= y_i + k_2 \end{aligned}$$

  der Anstieg wird in der Mitte des Intervalls und nicht am Anfangspunkt ausgewertet

  korrekt bis zur zweiten Ordnung im Zeitschritt h

- läßt sich auf beliebig hohe Ordnung erweitern, benötigt aber leider bei Implementierung in Ordnung p, mindestens p Funktionsauswertungen

Runge-Kutta Methode nochmals

$$\begin{aligned} y_{i+1} &= y_i + h\left((1-\gamma)f(t_i,y_i) + \gamma f(t_i + \frac{h}{2\gamma}, y_i + \frac{h}{2\gamma}f(t_i,y_i))\right), \\ t_{i+1} &= t_i + h, \qquad i = 0,1,\ldots,n-1, \end{aligned} \tag{50}$$

- $\gamma = 0$: Eulers Methode
- $\gamma = 1/2$: verbesserte Euler Methode

$$y_{i+1} = y_i + h\,(1/2 f(t_i,y_i) + 1/2 f(t_i + h, y_i + h f(t_i,y_i)))$$

- $\gamma = 1$: Euler-Cauchy Methode, Runge-Kutta zweiter Ordnung

Beispielprogram

- Gekoppelter Harmonischer Oszillator

$$\begin{aligned} \dot{Y}_1 &= Y_2 \\ \dot{Y}_2 &= -\frac{1}{L}\left(-U(t)+\frac{1}{C}Y_1+RY_2\right) \end{aligned}$$

- Runge-Kutta Methode zweiter Ordnung

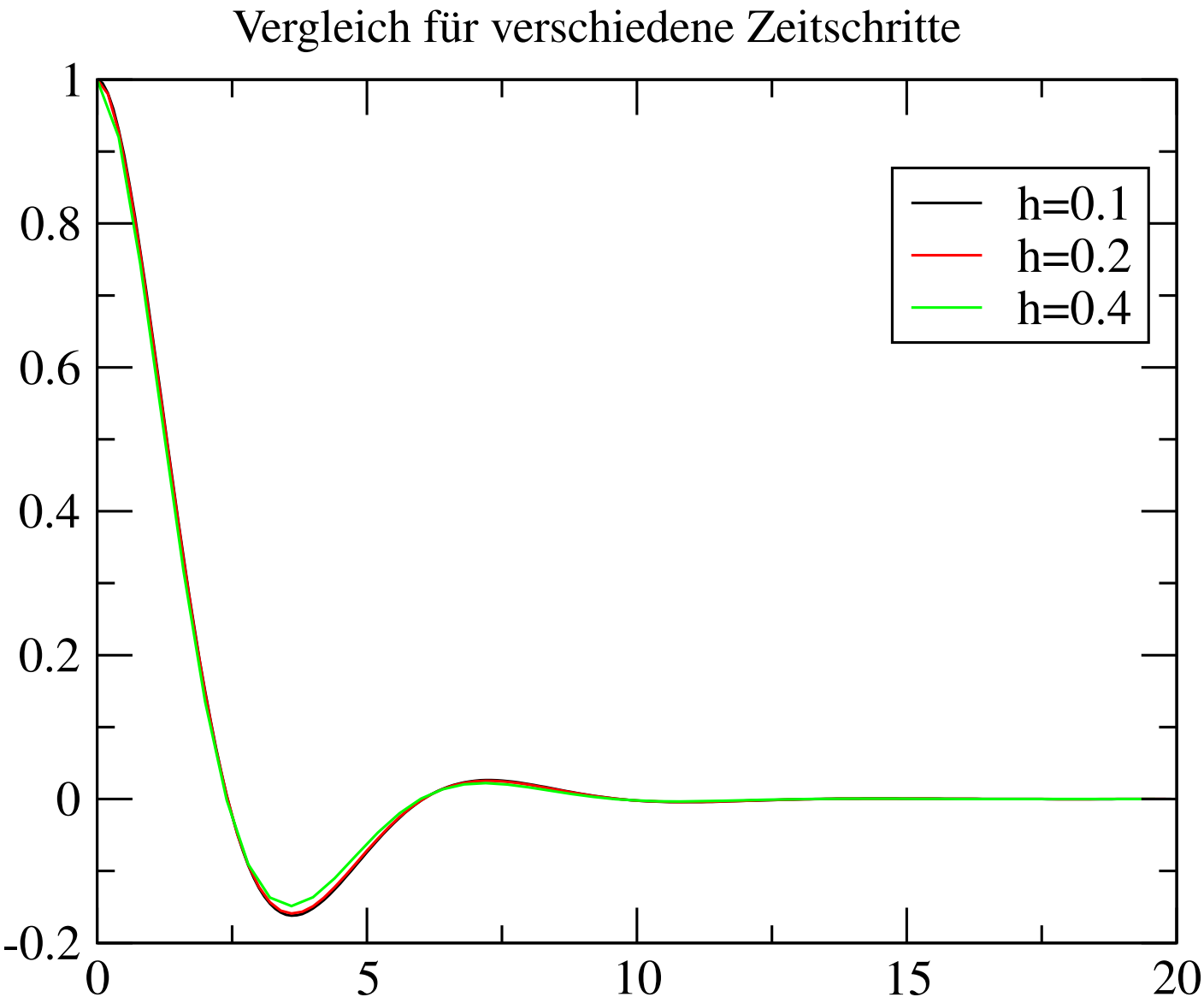
$$\begin{aligned} k_1 &= f(t_i,y_i)h \\ k_2 &= f(t_i+h/2,y_i+k_1/2)h \\ y_{i+1} &= y_i+k_2 \end{aligned}$$

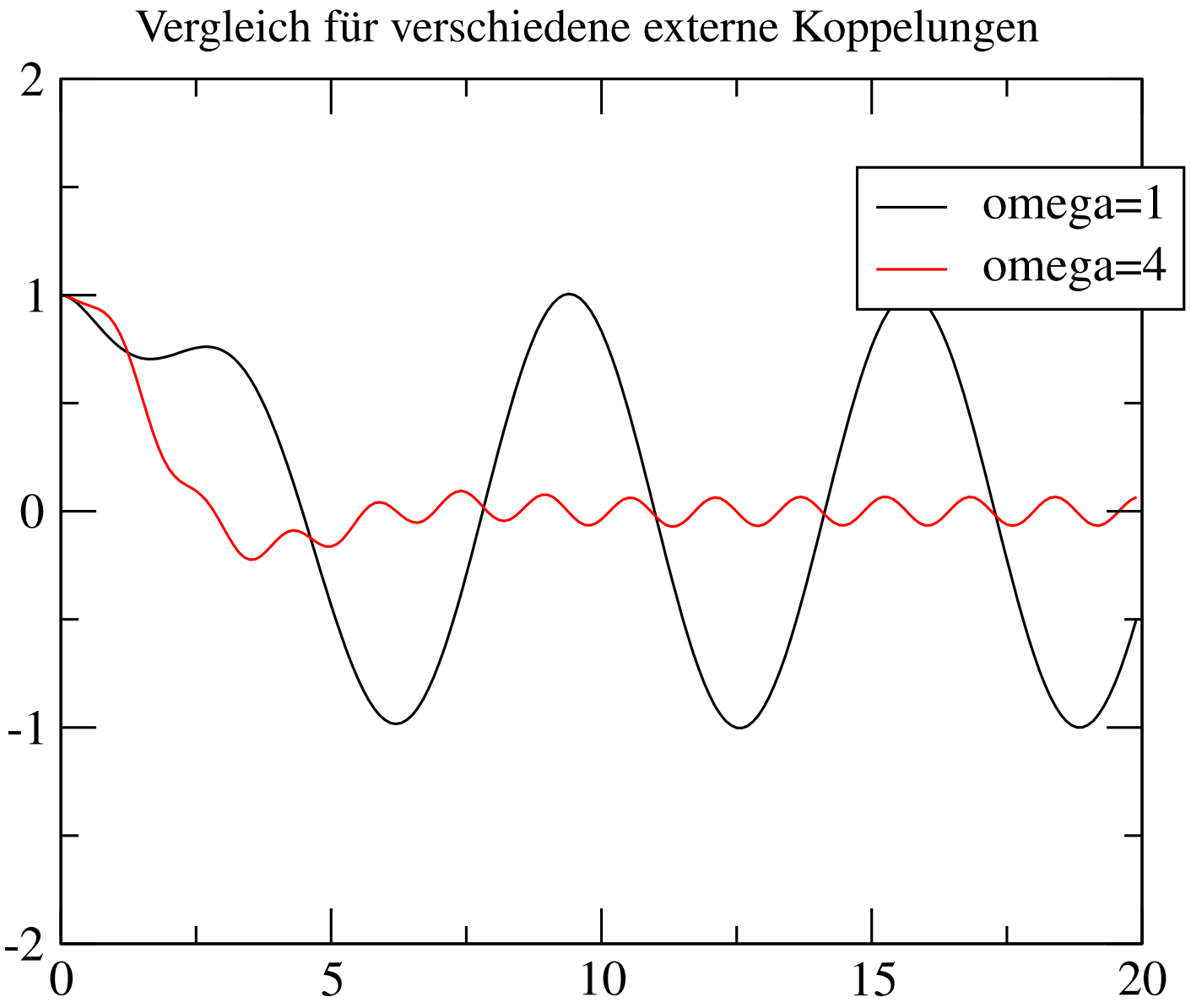
```
MODULE myfunction
  REAL, PARAMETER :: OMEGA = 4 , C=1 , R=1 , L=1 , U=0
  CONTAINS
  FUNCTION der_damped_osz( t, y)
    REAL :: der_damped_osz(2), t, y(2)
    der_damped_osz(1) = y(2)
    der_damped_osz(2) = -1/L * &
        ( - U* sin(OMEGA * t) +  1/C * y(1) + R * y(2))
  END FUNCTION der_damped_osz
END MODULE myfunction

MODULE solve_diff
  CONTAINS
    SUBROUTINE Euler_Cauchy( f, y, t, h)
      INTERFACE
         FUNCTION f( t, y)
           REAL :: f(2), y(2)
         END FUNCTION f
      END INTERFACE
      REAL :: y(2), t, h
    ! local
      REAL :: k1(2), k2(2)

      k1= f(t, y)*h
      k2= f(t+h/2, y+k1/2)*h
      y=y+k2 ; t=t+h
    END SUBROUTINE Euler_Cauchy
END MODULE solve_diff
```

```
PROGRAM MAIN
  USE myfunction
  USE solve_diff
  REAL :: h
  REAL :: y(2), t=0
  INTEGER i, nstep
  READ(*,*) h,nstep
  t=0
  y(1)=1
  y(2)=0
  DO i=1,nstep
     WRITE(*,'(2F14.7)') t,y(1)
     CALL Euler_Cauchy(der_damped_osz, y, t, h)
  ENDDO
END PROGRAM MAIN
```


Vergleich für verschiedene Zeitschritte
1
0.8
0.6
0.4
0.2
0
-0.2
0
5
10
15
20
h=0.1
h=0.2
h=0.4


Vergleich für verschiedene externe Koppelungen
2
1
0
-1
-2
0
5
10
15
20
omega=1
omega=4