Fortran 77 Gyakorlati Jegyzet

Eötvös Lóránd Tudományegyetem

Meteorológiai Tanszék

1. Elõszó

A jegyzet célja, hogy gyors áttekintést nyerjünk a Fortran 77 programozási nyelv használatáról. Ezen jegyzet a Fortran 77 és 90 programokhoz tartozó kézikönyv (T.F. Coleman - C. Van Loan : Fortran 77 for Engineers and Scientist with an Introduction to Fortran 90 c. könyv) alapján készült. Fontos megjegyezni, hogy ez az írásmû a fent említett kézikönyv kivonata, tehát a részletes ismertetés helyett inkább csak az alapvetõ tudnivalókat emeljük majd ki (elsõsorban a lineáris algebrára fókuszálva).

A jegyzetet eredetileg Erik Boman írta, de késõbb Sarah T. Whitlock és Paul H. Hargrove több pontban is kiegészítette illetve módosította.

E jegyzet oktatási, nemkereskedelmi célokra szabadon felhasználható, a szerzõk nevének feltüntetésével.

Paul H. Hargrove, Stanford, 1997. december
Sarah T. Whitlock, Stanford, 1997. január
Erik Boman, Stanford, 1995. december

2. Mi is a Fortran?

A Fortran egy programozási nyelv, melyet elsõsorban matematikai számítások (pl. mérnöki alkalmazások) megkönnyítésére fejlesztettek ki. Maga a Fortran szó két angol szó speciális rövidítése: FORmula TRANslation (=formula fordítás). Eredetileg csupa nagybetûvel irták (FORTRAN-nak), csak késõbb tértek a ma is elfogadott nagy kezdõbetûs (Fortran) írásmódra. A Fortran volt ez elsõ ún. magasszintû programozási nyelv. Ezt a programozási nyelvet az IBM cég fejlesztette ki az ötvenes években, azóta több újabb verziója is megjelent. A különféle Fortran verziókat a megjelenési évük feltüntetésével különböztetik meg, a Fortran szó mögötti két számjegy erre utal.

A jelenleg is használatban lévõ verziók

- Fortran 66
- Fortran 77
- Fortran 90 (95)

Manapság a leginkább elterjedt a 77-es verzió, de egyre népszerûbb a 90-es is. A Fortran 95 a 90-es verzió módosítása, mely 1996-ban jelent meg. Természetesen ezeken kivül vannak speciális verziók is, jó példa erre a HPF (High Perforamance Fortran = Magas Szintû Fortran).

Fontos, hogy a legtöbb Fortran 77-es szerkesztõ speciális kiegészítéseket is támogat, pl nem alapértelmezett kiterjesztéseket is elfogad. E jegyzet azonban az alapértelmezett Fortran 77-et ismerteti.

Miért érdemes a Fortrant megismerni?

Mind a mai napig a Fortran a legelterjedtebb programozási nyelv a mérnöki (és egyéb magas szintû matematikai) számítások elvégzéséhez.Ezért alapvetõ, hogy a jövõ mérnökei (és természettudósai) is tisztában legyenek e program használatával. Idõrõl idõre elhangzik: hamarosan a Fortran úgyis kiszorul, kikopik a használatból, de ez a jóslat eddig még sosem vált be. Jelenleg a Fortran a legrégebb óta használatban lévõ programozási nyelv. Többi oka is van annak, hogy a Fortran ilyen régóta alkalmazott és elõreláthtólag az is marad, ilyen ok például a *"szoftver-inercia"*. Tehát ha egy cég, ami hosszú évek során több millió dollárt költött egy megbízható szoftverre, nehezen fog átállni egy újabb nyelvre, fõleg annak az ismeretében, hogy egy pontos és hiteles szoftver-fordítás és átállás eléggé

nehezen kivitelezhetõ.

Mobilitás

A másik fontos elõnye a Fortrannak, hogy két követelményrendszernek (ANSI-nak és az ISO-nak - lsd lábjegyzet) is eleget tesz. Tehát, ha például a programunkat ANSI Fortran 77-ben írtuk és csak a standard alapverziót vettük igénybe, programunkat bármilyen Fortran 77 szerkesztõvel rendelkezõ gépen is le tudjuk majd futtatni, azaz a Fortran programok szabadon mozgathatók a különféle gépekés platformok között. (Fortran Standard Dokumentumokat itt találhatunk)

Lábjegyzet :

ANSI = American National Standards Institute (Amerikai Nemzeti Hitelesítési Intézet)
ISO = International Standards Organization (Nemzetközi Hitelesítési Szervezet)

3. Fortran 77 alapok

Egy Fortran program egymásutáni sorokból álló szöveg. Azonban a szövegnek bizonyos *struktúrát* kell követnie, hogy abból egy lefutattható program álljon elõ. Elsõnek nézzünk egy egyszerû példát:

```
      program circle
      real r, area

c Ez a program valos (real)  szamokat (r) olvas r és nyomtat
c az r sugaru kor terulete.

      write (*,*) 'Give radius r:'
      read  (*,*) r
      area = 3.14159*r*r
      write (*,*) 'Area = ', area

      stop
      end
```

A sor elején olvasható "c" betû az ún. *comment* (=megjegyzés) sort definiálja, ezen sorokban szereplõ szövegeket a program nem hajtja végre, sõt meg se jeleníti; igazi célja, hogy segítségével érthetõbb legyen a program szövege annak, aki olvassa. Eredetileg a Fortran programokat csupa nagybetûvel írták. Manapság a legtöbb ember már kicsi betûkkel írja a programszöveget, és mi is így fogjuk begépelni a parancsokat. A Fortran nem érzékeny a betûfajtára, tehát az "X" és "x" -et a program nem különbözteti meg.

Egy Fortran program felépítése

A Fortran programok egy fõprogramból és ha szükséges, több kisebb ún. szubprogramból (subroutinból) állnak. Egyelõre csak a fõprogramot fogjuk használni, a szubprogramokkal majd késõbb ismerkedünk meg. A fõprogram felépítése az alábbi :

```
program neve
    deklarációk
    utasítások/parancsok
stop
```

```
      end
```

Ebben a jegyzetben a *dõlt betûvel* szedett programrészekbe helyettesíthetjük be a konkrét programszövegeket, parancsokat.

A `stop` utasítás talán feleslegesnek tûnik, hiszen a program futtatása befejezõdik, ha eléri a végét, de erõsen ajánlott beleírni, hogy a program valóban befejezze az utasítások végrehajtását.

Fontos megjegyezni, a változó neve nem lehet ugyanaz, mint a program neve!

Formai követelmények, szabályok

Egy Fortran 77 program formája *nem tetszõleges.* Szigorú szabályokat kell figyelembe vennünk, ezek közül a legfontosabbak:

```
Col. 1    : Üres, vagy a "c" vagy r "*" a commenteknek
Col. 1-5  : Utasítás cimke (szabadon választható)
Col. 6    : Folytatósor (akkor, ha nem fértünk ki egy sorba, ez is szabadon
választható)
Col. 7-72 : Utasítások, parancsok
Col. 73-80: Szekvencia szám (szabadon választható, manapság ritkán használatos)
```

A legtöbb sor tehát 6 üressel kezdõdik és befejezõdik még a 72. oszlop elõtt , fõleg az utasítások mezõ a használatos.

Commentek

Az a sor, amelyik "c"-vel kezdõdik, az az ún comment sor. Commentet akárhová elhelyezhetünk a programunkba. Jól megírt commentek segítik a megértést és átláthatóbbá teszik a kódolásunkat. Egy átlagos Fortran programszöveg 50%-a ilyen megjegyzésekbõl áll. Találkozhatunk olyan Fortran verziókkal is, ahol egy felkiáltó jellel (!) különböztetik meg ezeket a commenteket. Ezt az eredeti Fortran 77 még nem támogatja, de néhány módosított verziója illetve a Fortran 90 már igen.

Folytatósorok

Elõfordulhat, hogy utasításunk nem fér ki egy sorbana rendelkezésre álló 66 oszlopban. Ilyenkor sortörést alkalmazunk, és így az utasítást több sorba írjuk. Ilyenkor a folytatásra utaló jelzést beleírjuk a 6. oszlopba. Például:

```
c23456789 (Ez most oszlop pozícióját demonstrálja!)

c A kovetkezo parancs ket sorban jelenik meg
      area = 3.14159265358979
     +       * r * r
```

Akármilyen karakter használható a sortörésre a + helyett. Azonban célszerû vmilyen logika szerint alkalmazni. (pl. az elsõ +, a továbbiakat meg egyesével számozni.., tehát + , 2 , 3 ..stb)

Üres helyek

A Fortran77 program az üres helyeket a futtatásnál figyelmen kivül hagyja. Tehát ha nem hagyunk ki üres sorokat, a program lefut, de maga a programszöveg átláthatlan lesz, annak aki olvassa.

4. Változók, típusaik és a deklaráció

Változók nevei

A változók nevei a Fortran programban max 6 karakter hosszúak lehetnek, melyek betûkbõl (a-z , angol abc, tehát nem használhatunk ékezetes betûket, (sehol sem!)) és/vagy számjegyekbõl (0-9) állhatnak. A név elsõ karakterének mindenképp betûnek kell lennie. A Fortran 77 általában nem különbözteti meg a kicsi és nagybetûket. Habár a futtatásnál minden betût nagybetûnek kezel, a fordítóprogramok többsége elfogadja a kis betûk használatát is. Tehát találkozhatunk olyan Fortan 77 verzióval is, ami csak a nagybetûk használatát támogatja, ezt a programunk megírásánál figyelembe kell vennünk.

Vannak ún. *speciális (védett) szavak*, amelyeket nem használhatunk változó névként: az utasítások kódjai tartoznak ide. Eddigi példáinkban is láthattunk ilyeneket: "program", "real", "stop" és "end".

Változók típusai és deklarációjuk

Minden változót *deklarálni* (=definiálni, meghatározni) *kell*! Több változó *típus* is használatos, a leggyakoribbak:

```
integer (=egész)  változók listája
real    (=valós)  változók listája
double precision ("pontosabb", több értékes valós) változók listája
    complex (=komplex) változók listája
    logical (=logikai) változók listája
character (=karakter) változók listája
```

A *változók listája* a változók neveinek felsorolása (lehet egy, de több is), az egyes neveket vesszõvel választjuk el. Mindent változót elég egyszer (de akkor pontosan!) deklarálni. Ha ezt nem tesszük meg a Fortran a saját, *beépített automata deklarációját* alkalmazza. Ezt azt jelenti, hogy azon változókat amelyeknek kezdõbetûi i és n betûk között vannak, egésznek (integernek) kezeli, az összes többet valósnak (realnek). A legtöbb régebbi Fortran 77 program e logika szerint használja a változókat, de *mi ne így járjunk el.* Az esetleges programhibák elõfordulási esélyei drasztikusan megemelkedhetnek, ha változóinkat nem következetesen deklaráljuk.

Az egész és nemegész változók

A Fortran 77 csak egyfajta egész változót ismer, melyet általában 32 bites (4 bytes) adatként tárol. Ezért az egész változók értékei egy intervallumban [-m és m] mozoghatnak, ahol az m értéke kb. 2*10^9.

A Fortran 77 kétfajta nemegész változót kezel, ezek a `real` (= valós) és `double precision`. (="pontosabb" valós). A `real`-t használjuk rendszerint, de a nagyon precíz számításoknál a `double precision` használata az ajánlott. Általában a real 4 byte-os, a double precision 8 byte-os változó,de ez számítógépfüggõ. Nehány nem standard Fortran verzió a `real*8` elnevezést használja, a 8 byte-os törtváltozókra.

A `parameter` utasítás

Néhány konstans akár többször is felbukkanhat a programban. Ilyenkor tanácsos egyszer, a program elején definiálni ezeket. Erre alkalmas a `parameter` utasítás, mellyel a programunkat átláthatóbbá is tehetjük Például a kör területének kiszámítására írt programot célszerû így megírni:

```
      program kor
      real r, ter, pi
      parameter (pi = 3.14159)

c Ez a program valós r szamot olvas be és irat ki.
c Az r sugaru kor terulete.
      write (*,*) 'Add meg a sugarat: r:'
      read  (*,*) r
      ter = pi*r*r
      write (*,*) 'Terulet = ', ter
      stop
      end
```

A parameter utasítás formája:

```
      parameter (név = konstans, ... , név = konstans)
```

A `parameter` utasítás használatának szabályai:

- Az a *név,* amit a `parameter` utasítással definiáltunk nem variálható késõbb. (Tehát a programunk írása során nem változtathatjuk meg az értékét)
- A `parameter` utasítás(oka)t az elsõ végrehajtó utasításunk elé kell megírnunk.

Néhány elõnye a `parameter` utasításnak:

- Csökkenti a programszöveg terjedelmét
- Könnyebb megváltoztatni a konstans értékét (elég csak ott, egy helyen), ha sokszor elõfordul a programban
- Olvashatóbbá válik programunk

5. Kifejezések és elnevezéseik

Konstansok

A legegyszerûbb kifejezési típus a *konstans*. A 6 adattípusnak megfelelõen 6 féle konstans típus van.

Néhány egész konstans:

```
      1
      0
      -100
      32767
      +15
```

Illetve néhány valós:

```
      1.0
      -0.25
      2.0E6
      3.333E-1
```

Az E-jelölés a tizes szám vmilyen hatványra utal, ami az "E" mögött van, azt a tizes hatványszámaként kell értelmezni. (tehát ez a normált alak jelölése). Példánkban, a 2.0E6 = 2*10^6 , azaz kétmillió, vagy a 3.333E-1=3,333*10^-1, ami 0,3333.

Azoknál a konstansoknál, amelyeknek az értéke túllépi a megengedettet, vagy nagyon nagy pontosságot igényelnek, az ún double precision forma használatos. A jelölésben ilyenkor a "E" helyett "D" betût használunk, de használata megegyezik a valóséval. Példák:

```
2.0D-1
1D99
```

Itt a 2.0D-1 egy double precision formájú 1/5, míg az 1D99, 1-es számjegy mögötti 99 nullát jelöl.

A következõ típus a komplex. Ezt egy konstans párral jelöljük (ami lehet integer vagy real), vesszõvel elválasztva a számokat és zárójelbe téve. Szemléltetve:

```
(2, -3)
(1., 9.9E-1)
```

Az elsõ szám a valós, míg a második a képzetes részt jelöli.

Az ötödik a logikai típus. Ez csak kétféle lehet:

```
.TRUE.  (=igaz)
.FALSE. (=hamis)
```

Fontos, hogy a két pont közé tegyük ezen szavakat.

Az utolsó típus a karakteres (szöveges) konstans. Legtöbbször *karakterek halmaza, tömbje,* szokták *stringnek* is nevezni. Ezeket aposztróf jelek közé tesszük.

```
'ABC'
'Barmi johet ide!'
'Ez egy szep nap.'
```

Ezek a jelölések nagyon érzékenyek. Angol szöveg esetén például az aposztrof a szövegben is felbukkanhat. Ebben az esetben a szövegen belül további aposztrófokat kell tennünk.

```
'It''s a nice day'
```

Kifejezések

A legegyszerûbb nem konstans-kifejezés az alábbi formájú:

```
operandus operátor operandus
```

egy példa rá:

```
x + y
```

A kifejezés eredménye is operandus, tehát többet is tehetünk egy kifejezésbe, például:

```
x + 2 * y
```

Ez mindjárt felveti a matematikai elsõbbrendûség kérdését is: a legutóbbi példánkban látott kifejezést a program hogy értelmezi `x + (2*y)` vagy `(x+2)*y` ?

A Fortran 77 az alábbi mûveleteket kezeli alapverzióban, kiértékelési sorrend (mûveletek közötti elsõbbség) szerint csoportosítva:

```
**   {exponciális}
```

```
*,/  {szorzás, osztás}
+,-  {összeadás, kivonás}
```

Mindegyik operátort balról jobbfelé értelmezi a program, kivéve az exponenciálist, melyet jobbról balfelé kezeli. Ha más mûveleti sorrendet kívánunk, zárójeleket kell alkalmaznunk. Ezek alapján a fenti kérdésre a válasz: a program az elsõ alakban értelmezi, hiszen a szorzás magasabb rendû mûvelet, mint az összeadás.

A fenti operátorok mind binárisak: tehát létezik az inverz opeáror is, a negatív jel - például elsõbbséget élvez a többihez képest. Tehát a -x+y kifejezest a program úgy értelmezi, ahogy várnánk.

Fokozott figyelmet kell fordítani az osztás operátorra, melynek kicsit más a jelentése egész illetve valós számoknál. Ha a operandusok egészek, akkor egész osztásként értelmezi, egyébként valósnak. Például: 3/2 egyenlõ 1, de 3./2. már 1.5-tel egyenlõ (fontos a tizedespont használata)

Elnevezés

Az elnevezésnek a formája:

```
változó neve = kifejezés
```

Az interpretáció az alábbi: a jobb oldalon álló kifejezésbõl számoljuk ki a bal oldal változóját. A jobboldali kifejezésben többfajta változó is lehet, de ezeknek az értéke fix. Például,

```
ter = pi * r**2
```

csak a `ter` értékét változtatja meg, a `pi` és `r` értékét nem

Típusok átalakítása

Mikor többfajta adattípus is megjelenik ugyanazon kifejezésben, *típus átalakítás*t kell alkalmazni, akár impliciten, akár expliciten. A Fortran is alkalmaz néhány beépített átalakítást, például

```
real x
x = x + 1
```

aminél az integer egyet real formájú egyre alakítja át, és így adja hozzá az x értékhez. Azonban , fõleg bonyolultabb formuláknál, érdemes explicite kiírnunk minden változónk elé a típus átalakító parancsokat, amik számok esetében az alábbiak lehetnek:

```
int
real
dble
ichar
char
```

Az elsõ három értelme egyértelmû, az `ichar` utasítás egy karaktert egész számmá alakít, míg a `char` pontosan az ellenkezõjét teszi.

Példa: Hogyan szorozzunk össze két valós számot (x és y), double precision alakot használva és az eredményt (w) szintén double precision formában tárolva?

```
w = dble(x)*dble(y)
```

Fontos, hogy ez mást jelent mint a

```
w = dble(x*y) kifejezés.
```

6. Logikai kifejezések

A logikai kifejezésnek csak két értéke lehet: `.TRUE.(=igaz)` or `.FALSE.`(=hamis). Egy logikai kifejezésben vagy állításban az alábbi *matematikai relációkat* használhatjuk:

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

Tehát összehasonlításra *nem használhatjuk* a "kacsacsõrös" < vagy = szimbólumokat, csak a fent olvasható rövidítéseket két pont között írva!

Speciálisi *logikai operátorokat* is felhasználhatunk logikai állításokhoz: `.AND.(=és) .OR. (=vagy) .NOT.` .(=nem)

Logikai változók és az elnevezés

Igaz értéket tárolhatjuk logikai *változó formájában* is. Az elnevezés teljesen analóg az aritmetikaival. Például:

```
logical a, b
a = .TRUE.
b = a .AND. 3 .LT. 5/2
```

A mûveleti sorrend ebben az esetben nagyon fontos. A szabály az alábbi: számításoknál az aritmetikai (számításos) mûveleteket hajtja végre legelõször a program, majd a relációkat, végül a logikai operátorokat. Példánkban a b értéke `.FALSE.`. A logikai operátork között (feltéve, ha nincs zárójel) a `.NOT.` élvez elsõbbséget, majd az `.AND.` legvégül az `.OR.` .

Logikai változókat elég ritkán használnak a Fortranban, de a logikai kifejezéseket annál gyakrabban, például az `if` (feltételes) utasításoknál.

7. A feltételes utasítások

Egy programozási nyelv fontos részei az ún. *feltételes utasítások.* A Fortranban az `if` utasítás a legelterjedtebb, melynek több formája is lehet :

```
if (logikai kifejezés) végrehajtási utasítás
```

Ezt egy sorba kell írni. Nézzünk egy példát, ahol x elõjelét kapjuk meg:

```
if (x .LT. 0) x = -x
```

Ha több utasítást is végre kívánunk hajtani a feltélel teljesülése esetén, akkor az alábbi formalizmust kell követnünk:

```
if (logikai kifejezés) then
   utasítások
endif
```

A legáltalánosabb forma az `if` utasításhoz, az alábbi:

```
if (logikai kifejezés) then
   utasítások
elseif (logikai kifejezés) then
   utasítások      :
 :
else
   utasítások
endif
```

A végrehajtás felülrõl lefelé halad. A program végighalad a feltételeken, és ahol igazat talál, ott végrehajta a hozzá kapcsolodó utasítást. Miután a program megtette ezt, az `endif`. utasítással záródik le a futtatás.

Egymásba ágyazott `if` állítások

`Az if` utasítás állhat több egymásba ágyazott, többszintû feltételes parancsokból is. Az olvashatóság érdekében célszerû bekezdéseket használni. Nézzünk erre egy példát:

```
if (x .GT. 0) then
   if (x .GE. y) then
      write(*,*) 'x  pozitiv es x >= y'
   else
      write(*,*) 'x  pozitiv de x < y'
   endif
elseif (x .LT. 0) then
   write(*,*) 'x negativ'
else
   write(*,*) 'x nulla'
endif
```

Azonban nem érdemes túl sok elágazást alkalmazni, mert a sok feltételt nehéz követni.

8. Ciklusok

Ismétlõdõ utasítások végrehajtására ún. *ciklusokat* alkalmazunk. Ha az Olvasó jártasabb a programozási nyelvek terén, talán hallhatott a *for, while* vagy *until* ciklusokról. A Fortran 77 azonban csak egyfajta ciklust használ, a neve a *do* ciklus. A *do ciklus* a többi programozási nyelvben elõforduló *for* ciklushoz hasonló fogalom, a többi ciklustípust az `if` és `goto` utasításokkal kell megoldanunk.

A do ciklusok

A `do` ciklust egyszerû számlálásra használjuk. A következõ egyszerû példa összeadja a számokat 1-tõl n-ig (az összeg (sum) és a n-et persze a ciklus elõtt deklarálni kell):

```
integer i, n, sum

sum = 0
```

```
      do 10 i = 1, n
         sum = sum + i
         write(*,*) 'i =', i
         write(*,*) 'sum =', sum
  10  continue
```

A 10-es szám az utasítás *cimke.* Jellemzõen egy programban több ciklus és/vagy utasítás is lehet. A programozón múlik milyen cimkéket használ programjában. Emlékeztetõ: az 1-5 poziciójú oszlopokba kell írni ezen cimkéket. Azon kivül, hogy egész számokat kell használnunk, nincs semmiféle megkötés, tehát a sorrend sem fontos. Elterjedt a 10-es szám ill. egész többszöreseinek használata (10, 20, 30 ...).

Alapértelmezettként a ciklusban a parancsok egymásután, egyesével hajtódnak végre, tehát egyes lépésközben. Ha ettõl eltérõ lépésközt szeretnénk, akkor a *step* (= lépés) kiegészítést is fel kell tüntetnünk, melynek értéke bármilyen 0-tól különbözõ szám lehet. Ez a program-szegmens (részlet), az 1 és 10 közötti páros számokat írja ki, csökkenõ számsorrendben:

```
      integer i

      do 20 i = 10, 1, -2
         write(*,*) 'i =', i
  20  continue
```

Az általános forma tehát az alábbi:

```
      do cimke  var =  expr1, expr2, expr3
         utasítások
cimke continue
```

var a ciklus-változó (gyakran *ciklus index* a neve) melynek egésznek kell lennie. *expr1* a *var* kezdõértékét, *expr2* a *var* végértékét, és az *expr3* a lépésközt (step) jelenti.

Megjegyzés: A `do` ciklus-változóját sose változtassuk meg utasítással a cikluson belül, mert ez komoly zavart okoz.

Sok Fortran 77 programfordító megengedi a ciklusok `enddo` paranccsal való lezárását. Ennek elõnye, hogy a ciklus cimkéjét elhagyhatjuk, mivel az `enddo` lezárja az õt megelõzõ ciklust. A standard Fortran 77-nek azonban ez nem része.

Meg kell jegyezni, hogy a Fortran a ciklus kezdetének, végének és lépésközének egyszeri feltûntetését támogatja, még az utasítások végrehajtása elõtt. Ennek következtében a következõ példában szereplõ ciklus le fog záródni és nem lesz egy végtelen ciklus (ami egy másik programnyelvben elõfordulhat).

```
      integer i,j

      read (*,*) j
      do 20 i = 1, j
         j = j + 1
  20  continue
      write (*,*) j
```

while ciklusok

Kézenfekvõ, hogy a `while` ciklus formája az alábbi legyen::

```
      while (logikai kif) do
         utasítások
```

```
      enddo
```

vagy esetleg,

```
      do while (logikai kif)
         utasítások
      enddo
```

A program ellenõrzi a feltételt és végrehajtja a ciklusban szereplõ utasításokat, ameddig a `while` ciklusban az állítás igaz. Habár ezt a formulizmust sok fordító támogatja, a gyári, eredeti Fortran 77 nem. A helyes alak, az `if` és `goto` parancsokat felhasználva ez:

```
cimke if (logikai kif) then
         utasítások
         goto cimke
      endif
```

Az alábbi példa kiszámolja és megjeleníti 2-nek minden 100-nál kisebb, vagy azzal egyenlõ hatványát.

```
     integer n

     n = 1
  10 if (n .le. 100) then
        write (*,*) n
        n = 2*n
        goto 10
     endif
```

until ciklusok

Ha a feltételünket a ciklus végére akarjuk írni, akkor until ciklust használunk. Ekkor programunk addig hajtja végig az utasításokat, amig a záró feltétel igaz. Kézenfekvõ, hogy formája ilyen legyen:

```
      do
         utasítások
      until (logikai kif)
```

Azonban (a fent leírtakat figyelembe véve) az eredeti Fortran 77-ben az alábbi alak a helyes, az `if` és `goto` használatával:

```
cimke continue
         utasítások
      if (logikai kif) goto cimke
```

9. A tömbök

A tudományos számításoknál elõfordulhatnak vektorok és mátrixok. Ezen adattípusokat a Fortran *tömbökként* kezeli. Az egydimenziós tömb a vektorral ekvivalens, míg a kétdimenziós tömb a mátrixot jelöli. A tömbök fortranos kezeléséhez azonban nemcsak a szintaxisát használát, hanem ezen objektumok memóriában való tárolását is meg kell ismernünk.

Egydimenziós tömbök

A legegyszerübb tömbtípus, az egydimenziós, elemek egymásutáni sorozatából áll, melyek folyamatosan tárolódnak a memóriában. Például, a

```
real a(20)
```

deklarációval, megneveztünk egy `a` 20 hosszúságú valós egydimenziós tömböt, ami 20 valós számból áll, melyek folyamatosan tárolódnak a gép memóriájában. A konvenció alapján a Fortranban az elemek indexelését 1-tõl kezdjük felfelé. Tehát az elsõ elemét a tömbünknek az `a(1)` jelöli, míg az utolsót az `a(20).` Ettõl függetlenül mi is definiálhatunk egyéni indexezést saját tömbünkre, az alábbi módon:

```
real b(0:19), weird(-162:237)
```

Itt a `b` teljesen megegyezik az elõzõ példabeli `a` -val, kivéve hogy itt az index 0-tól fut 19-ig. A `weird` a tömb hossza, ebben a példában 237-(-162)+1 = 400.

A tömb elemének típusa bármilyen alaptípus lehet. Például:

```
integer i(10)
logical aa(0:1)
double precision x(100)
```

Minden egyes elem felfogható akár egy-egy különálló változóként is. Az i-edik elemre az `a(i)`-vel hivatkozhatunk. A következõ kis programszegmens az elsõ 10 négyzetszámot tárolja egy `sq` tömbben:

```
      integer i, sq(10)

      do 100 i = 1, 10
         sq(i) = i**2
  100 continue
```

Gyakori hiba, hogy a program olyan tömbelemre is hivatkozik, ami nem definiált vagy kivülesik a definiált tömbön. Ezért fontos a programozó figyelmessége, mert ezen hibatípust a Fortran-fordítók nem ismerik fel!

Kétdimenziós tömbök

A lineáris algebrában alapvetõ a mátrixok használata. A mátrixokat általában kétdimenziós tömbök formájában jelenítjük meg. Például a

```
real A(3,5)
```

deklaráció, definiál nekünk egy 2 dimenziós tömböt, mely 3*5=15 valós számból áll. Hasznos tudni, hogy az elsõ index a sorindex, a második pedig az oszlop index. Így az alábbi grafikus képet kapjuk:

```
(1,1)  (1,2)  (1,3)  (1,4)  (1,5)
(2,1)  (2,2)  (2,3)  (2,4)  (2,5)
(3,1)  (3,2)  (3,3)  (3,4)  (3,5)
```

Kétdimenziós tömböknél is definiálhatunk egyéni indexezést. Az általános formalizmus az alábbi:

név (low_index1 : hi_index1, low_index2 : hi_index2)

A tömb teljes mérete tehát :

```
méret = (hi_index1-low_index1+1)*(hi_index2-low_index2+1)
```

(low=kezdõindex, hi=legmagasabb index)

A Fortranban gyakran szoktak nagyobb tömböt deklarálni, mint amekkora mátrixra a számításhoz és tároláshoz szükség lenne. (Oka: a Fortran nem rendelkezik rugalmas és dinamikus tárolási funkcióval) Ez teljesen elfogadott eljárás. Például:

```
      real A(3,5)
      integer i,j
c
c     Mi csak a felso 3 * 3 -ast részét hasznoljuk a tombnek
c
      do 20 j = 1, 3
         do 10 i = 1, 3
            a(i,j) = real(i)/real(j)
   10    continue
   20 continue
```

A részmátrix elemei, A(1:3,4:5), pontosan definiáltak

A kétdimenziós tömbök tárolási formája

A Fortran a magasabb dimenziojú tömböket is elemek folyamatos sorozataként tárolja. Fontos tudni, hogy a kétdimenziós tömböket *oszlop szerint* tárolja. Tehát a fenti példát nézve, a tömb (2,1)-vel jelzett elemét (3,1) elem követi. Majd ezután végighalad a kettes, majd a többi oszlopon is.

Térjünk kicsit vissza ahhoz a példához, mikor csak a felsõ 3*3 részmátrixot használtuk, a 3*5-ös `A(3,5)` tömbbõl. A kilenc számunkra fontos elem a memória elsõ 9 helyén fog tárolódni, míg a következõ hat helyet nem használjuk fel. Ez kicsit talán fura, hiszen a *fõdimenziója* mindkét mátrixnak (az eredeteinek és a tároltnak) ugyanaz. Azonban gyakran a fõdimenziója a használt tömbnek nagyobb lesz, mint a definiáltnak. Tehát a mátrix nem folyamatosan fog tárolódni, habár maga a tömb folytonos. Például a fenti példánkban, ahol `A(5,3)` volt, lesz két "kihasználatlan" cellánk (az elsõ oszlop alja és a második eleje között (ismét a 3*3 mátrix példáját nézve) .

Ez most talán bonyolultnak tûnik, de a használat során majd megtapasztaljuk, hogy nem az. Probléma esetén, nem árt megnézni, az egyes elemekre való *memória-hivatkozást (azaz memória címét).* Minden egyes tömbhöz hozzárendelõdik egy, a tömb elsõ eleme (1,1) alapján kiosztott memória cím. Az (i,j)-edik elem memóriacíme:

```
cím[A(i,j)] = cím[A(1,1)] + (j-1)*lda + (i-1)
```

ahol `lda` a fõ (többnyire sor) dimenziója `A`-nak. Fontos, hogy az `lda` nem azonos az aktuális mátrix dimenziójával. Ezek keveréséből sok programhiba származhat!

Három- vagy még több dimenziós tömbök

A Fortran 77-ban maximum hétdimenziós tömböt használhatunk. A kezelés szintaxisa és formalizmusa teljesen analóg a kétdimenzióséval, ezért erre nem vesztegetjük az idõt.

A `dimension` utasítás

Más úton is definiálhatunk egy tömböt a Fortran 77-ben:

```
real A, x
dimension x(50)
dimension A(10,20)
```

ami ekvivalens ezzel:

```
real A(10,20), x(50)
```

A `dimension` utasítás használata manapság már egyre kevésbé jellemzõ.

10. Segédprogramok

Amikor egy program már több száz sor hosszú, nehéz már követni. Sõt, a valódi mérnöki (vagy akármilyen más) számításoknál sokszor több ezer sorból álló programokat kell alkalmazni. Az egyetlen módja, hogy e hatalmas "kód-zuhatagot" kezelni tudjuk a *modulátorok* és a programba beágyazott *szub(vagy segéd vagy al)programok* használata*,* amikkel programunk szövegét több kisebb részre bonthatjuk.

A szubprogram egy kisebb kódrészlet, ami egy konkrét segédfeladatot old meg. Egy nagyobb programban elõfordulhat, hogy ugyanazt azt alproblémát kell megoldani többször, de különbözõ adattokkal. Ilyenkor is alkalmazhatunk szubprogramot, a parancsok ismétlése helyett, azaz az egyszer megírt segédprogramot kell elõhívni, miután a bemenõ adatokat megfelelõen módosítottuk.

A Fortranban kétféle szubprogram van: a *függvények* és a *subroutine*-ok.

Függvények

Egy Fortran-függvény hasonló jelentésû, mint egy matematikai: mindkettõhõz meg kell adni a bemenõ argumentumot (paramétereket) és a egy függvényalakot, hogy értéket nyerjünk belõle. A Fortran esetében beszélhetünk a *felhasználó által definiált szubprogramokról* (ezekrõl volt fent is szó), de léteznek *beépített függvények* is.

Egy egyszerû példa a függvények használatára:

```
x = cos(pi/3.0)
```

Itt a `cos` a cosinus függvény, tehát `x` értékére 0.5 adódik majd. (feltéve ha `pi` értékét korrekten definiáltuk, mivel a Fortran 77-nek nincsenek beépített konstans értékei). Nézzünk példákat a Fortran 77 saját, beépített függvényeire:

```
abs     abszolútérték
min     minimum érték
max     maximum érték
sqrt    gyökvonás
sin     sinus
    cos     cosinus
    tan     tangens
atan    cotanges
exp     exponenciális (természetes, e)
log     logaritmus (természetes alapú, ln)
```

Általában a függvényeknek *típusuk* van. A legtöbb beépített függvény, amiket elõbb is is láttunk, *általános tipusú.* Azaz, például a fenti esetet tekintve a `pi` és `x` értéke lehet `real` vagy `double`

`precision` is. A programolvasó ellenõrzi az adatok típusát és a helyes `cos` verziót fogja alkalmazni (real vagy double precision-t). Sajnos a Fortran nem elég rugalmas nyelv e téren, ezért nagyon fontos a változók típusainak és a függvények korrekt megválasztása!

Most térjünk vissza a felhasználó által írt függvényekre. Tekintsük az alábbi problémát: Egy meteorológus miután tanulmányozta a Balaton-vidék havi csapadék viszonyait, segítségével egy modellt alkot:: *r(m,t)-et,* ahol *r* a havi csapadékösszeg, az *m* a hónap száma, és *t* a földrajzi helyre utaló paraméter, Ha r-re egy formulát és t-nek egy értéket adunk az éves csapadékösszeget kiszámolhatjuk.

A legegyszerübb megoldás egy ciklus megírása, ami havonként összegzi a csapadékmennyiségeket. Itt most r-et nem mérjük, hanem számítjuk. Az r kiszámítása egy külön elszigetelt probléma, ezért célszerû egy szubprogram megírása értékének meghatározásához. A fõprogram alakja az alábbi (más megoldás is lehet):

```
      program eso
      real r, t, sum
      integer m

      read (*,*) t
      sum = 0.0
      do 10 m = 1, 12
         sum = sum + r(m, t)
 10   continue
      write (*,*) 'Eves csapadekosszeg ', sum, 'mm'

      stop
      end
```

Megjegyzés: r-et egy valós változónak definiáltuk (hiszen egy konkrét értéket ad) A szubprogramban majd r-et Fortran függvénynek kell definiálni. A meteorológus az alábbi formulát találta:

```
      r(m,t) = t/10 * (m**2 + 14*m + 46) (ha ez pozítiv)
      r(m,t) = 0                         (egyébként)
```

Ez Fortranban lekódolva:

```
      real function r(m,t)
      integer m
      real t

      r = 0.1*t * (m**2 + 14*m + 46)
      if (r .LT. 0) r = 0.0

      return
      end
```

Láttuk, hogy egy függvény formailag szinte teljesen megegyezik a fõprograméval. Azonban vannak különbségek, melyek közül a fontosabbak:

- A függvényeknek típusuk van, amit a hívó programban is definiálni kell
- A függvény által adott értéket ugyanazon néven kell a hívó programban megnevezni, mint magát a függvényt a segédprogramban.
- A függvényeket a *return* paranccsal kell lezárni a *stop* helyett.

Összefoglalásképp egy Fortan 77 függvény általános alakja:

```
      típus function név (kül. változó típusok szerint)
```

```
      deklarációk
      utasítások
      return
      end
```

A függvény típusát pontosan kell megadni a hívó programban. Ha olyan függvényt alkalmazunk, amit elõtte nem deklaráltunk, a Fortran pótlásképp saját maga fogja deklarálni, nem kizárt, hogy hibásan. Egy függvényt nevének és paramétereinek leírásával (a paramétereket zárójelben a név mögött soroljuk föl) hívhatjuk meg.

Meg kell jegyezni, hogy a gyári, alapértelmezett Fortran 77 nem engedi meg a rekurzív (önmagát generáló számításokat), de némelyik fordító már elfogadja használatukat.

Subroutin-ok

Egy Fortran függvény csak egy értéket ad vissza. Azonban gyakran elõfordul, hogy kettõ vagy több értéket (sõt néha semmilyet sem) akarunk megkapni segédprogram segítségével. A `subroutine` utasítást erre használjuk. A szintaxis az alábbi:

```
      subroutine név (argumentumok listája)
      deklarációk
      utasítások
      return
      end
```

Fontos, hogy a subroutin-oknak nincsen típusuk és nem is lehet deklarálni a fõprogramban. Ezért meghívásuk is más, mint a függvényeknél: a *call* (=hívás) szót kell leírnunk nevük és paramétereik elé.

Nézzünk egy egyszerû subroutine-t. A feladat két egész szám megcserélése:

```
      subroutine csere (a, b)
      integer a, b
c Lokalis valtozok
      integer tmp

      tmp = a
      a = b
      b = tmp

      return
      end
```

Itt kétféle deklarációt is alkalmazunk. Elõször a bemenõ/kimenõ paramétereket kell, amik közösek a hívó és a hívott programban. De ezután deklarálni kell a *lokális (helyi) változókat*, amiket csak ebben a szubprogramban használhatunk. Ezért nyugodtan használhatunk ugyanolyan változó-neveket a különbözõ segédprogramokban, a programfordító tudni fogja, hogy ezek különbözõ változók, habár nevük formailag ugyanaz.

A Call-by-hivatkozás

A Fortran 77 használ ún.*call-by-reference (=a hivatkozás alapján hívás)* formát is. Ez azt jelenti, hogy nem csak a függvény/subroutine argumentumjait *(call-by-value=érték alapján hívás)* , hanem vele együtt az argomentumok memória-címeit is meghívja. A következõ rövid példa bemutatja a kettõ közötti különbséget:

```
      program callex
      integer m, n
c
      m = 1
      n = 2

      call csere(m, n)
      write(*,*) m, n

      stop
      end
```

Ennek a programnak az eredménye "2 1", ahogy el is várjuk. De, ha a Fortran 77 call-by-value típusú hívást alkalmazott volna az eredmény "1 2" lenne, azaz m és n felcseréletlen maradt volna! Ennek az az oka, hogy a második esetben csak az értéket másolta volna a csere subroutine-ba, ahol habár a és b értékét megcserélte volna, az új (felcserélt) értékeket nem adta volna vissza a fõprogramba.

A fenti példa is rávilágosított, hogy jobb a call-by-reference hívás De bánjunk óvatosan ezzel is, mert könnyen elronthatjuk vele programunkat. Például néha csábító lehet a szubprogramba bemenõ változót egyben lokális változónak is használni, és így változtatni meg az értékét. Azonban a szubprogramból kimenõ érték már ez a megváltozott lesz, ami többnyire nem kívánatos. Bizonyos esetektõl eltekintve (amikor direkt akarjuk megváltoztatni a fõprogrambeli változónk értékét, lásd elõzõ példa) *kerüljük* ezt a megoldást.

Késõbb még folytatjuk e téma taglalását, de elõtte tisztáznunk kell a tömbök (mint argumentumok) szerepét a segédprogramokban.

11. Tömbök a segédprogramokban

A Fortran szubprogramok meghívásának elvi alapja a *call by reference*. Ez azt jelenti, hogy a hívó paraméterek nem másolódnak a meghivanandó segédprogramba, hanem e helyett a paraméretek (váltózók) memóriacimjei kerülnek továbbításra. Tömbökkel programozva , ezzel sok memóriahelyet megspórolhatunk. Nincs szükség plussz tárolási helyre, mert a subroutine is, meg a hívó program is ugyanazon memória felületen dolgozik. Egy programozónak errõl is tudnia kell, és eszerint is kell mérlegelnie.

Lehetséges lokális tömbök definiálása a Fortran szubprogramokban, de nem ez a jellemzõ. Ehelyett inkább az összes tömböt és dimenziójukatm már a fõprogramban deklaráljuk, és ezekre alkalmazzuk a segédprogramokat.

Változó hosszúságú tömbök

Az egyik alapvetõ vektoroperátor a *saxpy* . Ez az alábbi kifejezést jelenti:

```
      y := alpha*x + y
```

ahol alpha skalár, de x és y vektor. Nézzünk egy egyszerûbb subroutine-t, felhasználva a fentit::

```
      subroutine saxpy (n, alpha, x, y)
      integer n
      real alpha, x(*), y(*)
c
c Saxpy: Szamold ki y := alpha*x + y,
```

```
c ahol x és y n hosszusagu vektorok
c
c lokalis valtozok
      integer i
c
      do 10 i = 1, n
         y(i) = alpha*x(i) + y(i)
   10 continue
c
      return
      end
```

Az újdonságot a deklarációban megjelenõ csillagok jelentik: *x(*)* és *y(*)*. Ezzel a jelöléssel az x és y vektorunk hossza tetszõleges lehet. Ennek elõnye, hogy ugyanazon segédprogramokat használhatjuk a különféle hosszúságú vektorokra. Utalva arra, hogy a Fortran call-by-reference típusú hivatkozáson alkalmazza, nem szükséges további memóriahelyeket lefoglalni, a segédprogram is a hívó programban definiált tömb elemein dolgozik.A programozó felelõssége az x és y vektorok hosszának helyes megválasztása, azaz hogy a program semelyik részén se forduljon elõ a vektor nem létezö elemére (megengedett elemszámnál nagyobb értékû helyre) való hivatkozás. Ennek elmulasztása gyakran visszatérõ hibának tûnik.

Így definiálhatunk a tömböket :

```
      real x(n), y(n)
```

A legtöbb programozó inkább a csillagos x(*) jelölést kedveli, hiszen nem mindig tudni elõre, hogy milyen hosszúságú vektorok kellenek majd. Néhány régebbi Fortran 77 programban ilyen deklarációt is láthatunk:

```
      real x(1), y(1)
```

Megtévesztõ lehet, de ez a jelölés akkor is helyes, ha a tömb mérete nagyobb, mint 1. Lehetõleg kerüljük e jelölés használatát.

Tömbök részleteinek átvitele

Segédprogramot kívánunk írni egy mátrix és egy vektor összeszorzására. Két módszer is lehetséges, az egyik beépített programokat alkalmaz, a másik a fenti saxphy kódot használja. Nézzük most a második esetet, melynek a formája:

```
      subroutine matvec (m, n, A, lda, x, y)
      integer m, n, lda
      real x(*), y(*), A(lda,*)
c
c szamold ki: y = A*x, ahol A egy (m x n) matrix
c Az A fodimenzioja lda
c
c Lokalis valtozok
      integer i, j

c Induljon y
      do 10 i = 1, m
         y(i) = 0.0
   10 continue
```

```
c Matrix-vektor szorzas eredmenye a saxpy segitsegevel A oszlopaira
c Fontos, az egyes oszlopk hossza m es nem n!
      do 20 j = 1, n
         call saxpy (m, x(j), A(1,j), y)
   20 continue

      return
      end
```

Néhány megjegyzést kell tennünk. A mátrix méretét (itt n és m) általánosan is megválaszthatjuk a csillagos kóddal, de a fõdimenziót továbbra is pontosan kell definiálni. Miért? Mert a (máshol változtatható méretre vonatkozó) * jelölés ebben az esetben a tömb utolsó dimenziójára alkalmazható. Erre magyarázatot a Fortran 77 sokdimenziós tömbtárolási mechanizmusa adhat (lsd a tömbökrõl szóló fejezetet)

Amikor kiszámítjuk a saxpy operátorral az y = A*x by értéket, szükségünk van A oszlopainak értékeire is. Az A mátrix j-edik oszlopa A(1:m,j). Azonban a Fortran 77 nem tudja kezelni ezt az összetett indexezést (a Fortran 90 már igen!). Ezért szükségünk van egy közelítésre, a *mutatóvektorra*, amivel az oszlop elsõ elemére A(1,j) hivatkozunk. (nem pontos a fogalom, de a megértést segítheti). Tudjuk, hogy a következõ memóriahelyeken, ezen oszlop elemei lesznek. A saxpy segédprogram az A(1,j)-t egy vektor elsõ elemének fogja kezelni, mit sem tudva arról, hogy ebben az esetben ez a vektor egy mátrix oszlopát jelenti.

Végül azt is meg kell jegyeznünk, hogy a konvenció alapján a mátrixok m sorból és n oszlopból állnak. Az i indexet sorindexként (1-tõl m-ig), a j indexet oszlopindexként (1-tõl n-ig) használjuk. A legtöbb lineáris algebrás számításra írt Fortran-program ezeket a jelöléseket használja, ami megkönnyíti a programszöveg olvasását.

Különbözõ dimenziók

Sokszor megéri az egydimenziós tömböt kétdimenziósként kezelni vagy fordítva. Az dimenziók közti "átváltás" elég könnyû, sõt sokak szerint túl könnyû.

Nézzünk egy könnyen érthetõ példát. A másik alapvetõ vektromûvelet a *nyújtás (=skale)*, amikor a vektor minden elemét egy konstans számmal megszorozzuk. Ez programnyelven így szól:

```
      subroutine scale(n, alpha, x)
      integer n
      real alpha, x(*)
c
c Lokalis valtozok
      integer i

      do 10 i = 1, n
         x(i) = alpha * x(i)
   10 continue

      return
      end
```

Miután megkaptuk az m x n -es mátrixunkat, meg szerenénk "nyújtani". Ahelyett, hogy erre is írnánk egy másik segédprogramot, elég ezt a mátrixok vektorként kezelve megnyújtani, alkalmazva a `scale` subroutine-t. Kézenfekvõ az alábbi metódus:

```
      integer m, n
```

```
      parameter (m=10, n=20)
      real alpha, A(m,n)

c Nehany utasitassal defininalni az A-t...

c Most megnyujtjuk A-t
      call scale(m*n, alpha, A)
```

Fontos megjegyezni, hogy a fenti példa mûködik, mert az eredetileg deklarált dimenziója A-nak megegyezik, azzal a mátrixéval, amiben éppen (számítás közben) tároljuk az A-t. Ha ez nem így van, akkor a fenti metódus nem jó. Rendszerint e kétfajta mátrix dimenzió nem egyezik meg, ilyenkor körültekintõen kell eljárnunk.Ebben az esetben a programunk így lesz korrekt:

```
      subroutine mscale(m, n, alpha, A, lda)
      integer m, n, lda
      real alpha, A(lda,*)
c
c Lokalis valtozok
      integer j

      do 10 j = 1, n
         call scale(m, alpha, A(1,j) )
   10 continue

      return
      end
```

Így már minden esetben mûködõképes lesz a programunk: a számításhoz szükségtelen sorok és oszlopok (amivel nagyobb az eredeti mátrix, mint az aktuális) elemeihez nem nyúl.

12. Közös tömbök

A Fortran 77 nem rendelkezik *globális* változókkal, azaz olyan változókkal, amelyeken több programrész is felhasznál, osztozik. Eddig csak egyféle módszert láttunk az egyes subroutinok közötti információcserére, ez volt a paraméter lista. Azonban ennek a használata nem mindig célravezetõ, például akkor, mikor sok segédprogram van, amik nagy mennyiségû paraméteren osztoznak. Ilyenkor célszerû a *közös tömbök* használata. De azért e módszer alkalmazását nem szabad túlzásba vinnünk.

Példa

A közös tömbök használatára lássuk az alábbi példát: Tegyük fel van két paraméterünk: alfa és beta, amelyeket több segédprogram is felhasznál.

```
      program fo
      deklarációk
      real alpha, beta
      common /coeff/ alfa, beta

      utasítások
      stop
      end

      subroutine sub1 (néhány argumentum)
```

```
az argumentumok deklarációi
real alfa, beta
common /coeff/ alfa, beta

utasítások
return
end

subroutine sub2 (néhány argumentum)
az argumentumok deklarációi
real alfa, beta
common /coeff/ alfa, beta

utasítások
return
end
```

Itt defin——áltunk egy `coeff` nevû közös tömböt, mely két elemből áll: `alfa` és `beta` változókból. Természetesen tetszõleges számú elemet is tehetünk egy ilyen közös tömbbe. Sõt, még az elemek típusainak se kell egyformának lenniük, mivel minden subroutin, amely ezt a tömböt alkalmazza, az egész tömböt deklarálja.

Megjegyzés: Ebben a példákban is könnyen elkerülhettük volna a közös tömb használatát, ha alfa-t és beta-t eleve paraméterként(argumentumként) kezeljük. Amikor csak lehet, kerüljük a közös tömbök használatát. Természetesen azonban vannak olyan problémák is, mikor csak a közös tömbök használata jelenti a megoldást.

Közös tömbök szintaktikája

```
common / név / a változók listája
```

Fontos tudnivalók:

- A `common` utasításnak és a változók deklarációjának együtt kell megjelennük, még a végrehajtó utasítások elõtt
- Különbözõ közös tömböknek különbözõ nevet kell adnunk (hasonlóan a változókhoz)
- Egy változó csak *egyetlen* tömbhöz tartozhat
- A közös tömbben lévõ változók neveinek nem kell megegyeziük minden egyes megjelenésüknél, ugyanarra az elemre más-más néven is hivatkozhatunk, (de célszerû mindig ugyanazt a nevet használni), viszont fontos, hogy a hivatkozásnál ugyanabban a sorrendben kell felsorolni õket, és a típusnak és méretnek is meg kell egyezniük.

Ezen szabályok figyelembevételét is szemlélteti a következõ példánk:

```
subroutine sub3 (néhány argumentum)
az argumentumok deklarációi
real a, b
common /coeff/ a, b

utasítások
return
end
```

Ez a deklaráció ekvivalens a korábban látott, alfa-ra és beta-ra vonatkozó deklarációval. Célszerû ugyanazon neveket használni ugyanazon elemekre, mert könnyen összezavarodhatunk. Nézzük erre

egy elrettentõ példát:

```
subroutine sub4 (néhány argumentum)
az argumentumok deklarációi
real alfa, beta
common /coeff/ beta, alfa

utasítások
return
end
```

Itt most az alfa egyben beta is. Ha ilyesmit látunk, akkor nagy valószinûséggel hibát fedeztünk fel. De az ilyen típusú hibákat igen nehéz megtalálni.

Vektorok a közös tömbökben

Közös tömbökben definiálhatunk vektorokat is. De mégegyszer hangsúlyozzuk, ezt nem ajánlatos alkalmazni. A legfõbb oka ennek a flexibilitás. A következõ példában is láthatjuk, hogy mennyire rossz ez az ötlet. Tegyük fel az alábbi deklarációkat írjuk meg a fõprogramban:

```
program main
integer nmax
parameter (nmax=20)
integer n
real A(nmax, nmax)
common /matrix/ A, n
```

Ebben a közös tömbben az A mátrix elemei vannak, illetve egy egész szám, az *n*. Szeretnénk ezt az A mátrixot néhány subroutinban alkalmazni. Ekkor az alábbi deklarációt kell megejteni minden egyes segédprogramban:

```
subroutine sub1 (...)
integer nmax
parameter (nmax=20)
integer n
real A(nmax, nmax)
common /matrix/ A, n
```

Különbözõ dimenziójú vektorok nem lehetnek egy közös tömbben, ezért az *nmax* értékének pontosan meg kell egyeznie a fõprogrambeli értékével. Emlékezzünk vissza: a mátrix méretét pontosan kell ismernie a fordítóprogramnak, ezért nmaxot definiálnunk kell egy paraméter utasítással.

Ez a kis példa is rámutatott arra, hogy a nem érdemes vektorokat közös tömbben kezelnünk. Sokkal célszerûbb a vektorokat argumentumként alkalmazni a subroutinokban (persze a fõdimenzió figyelembevételével).

13. Az adatbeolvasás másik módja: a data és data block utasítások

A data utasítás

Az olyan változók, amelyek már a program megírásánál is ismertek, a data (=adat) utasítással is bevihetõk. Formalizmusban nagyon hasonló az elnevezéses utasításokhoz. A metódus a következõ:

```
data változók listája/ értékek listája/, ...
```

ahol a három pont azt jelenti, hogy ezen beírás akárhányszor megismétlehetõ, például:

```
data m/10/, n/20/, x/2.5/, y/2.5/
```

De össze is vonhatók:

```
data m,n/10,20/, x,y/2*2.5/
```

A "hagyományos" elnevezéses utasítással a fenti m,n,x,y változókat így írtuk volna:

```
m = 10
n = 20
x = 2.5
y = 2.5
```

Ez a data utasítás tömörebb, mint a hagyományos, ezért széleskörben elterjedt a használata. Azonban fõleg az összevont esetnél, ügyelnünk kell arra, hogy ugyanazon értékeket nem elég egyszer leírni.

A data utasítást elég egyszer, közvetlenül a végrehajtó utasítások *elõtt*, a program elején alkalmaznunk. Ezért ezt a data utasítást általában a fõprogramban használják, és nem a subroutinokban.

Természetesen a data utasítással tömböket (vektorokat, mátrixokat) is definiálhatunk. A következõ programrészlet biztosítja, hogy a mátrix elemei mind zérusok, mikor a programfutattás megindul.

```
real A(10,20)
data A/ 200 * 0.0/
```

Néhány fordító automatikusan alkalmazza ezt a tömbökre, de nem mind, ezért ha ragaszkodunk, hogy az elemek mind nullák legyenek, akkor a fenti szintaktikát kell beírnunk. Természetesen zérustul különbözõ tömb elemeket is megnevezhetünk, akár így:

```
data A(1,1)/ 12.5/, A(2,1)/ -33.3/, A(2,2)/ 1.0/
```

de akár így is, kilistázva egy kis tömbben:

```
integer v(5)
real B(2,2)
data v/10,20,30,40,50/, B/1.0,-3.7,4.3,0.0/
```

Többdimenziós esetben oszlopok szerint rendezi és kezeli a tömb elemeit.

A block data utasítás

A data utasítást nem alkalmazhatjuk egy közös tömb/blokk elemeire. Erre egy speciális "subroutine" áll rendelkezésünkre: a `block data` utasítás. Igazából ez nem egy segédprogram, de hasonló fogalom,

mert egy önálló programrészként jelenik meg. Nézzünk rá egy példát is:

```
block data
integer nmax
parameter (nmax=20)
real v(nmax), alfa, beta
common /vector/v,alfa,beta
data v/20*100.0/, alfa/3.14/, beta/2.71/
end
```

A Fortran 77 fordító, a data utasításhoz hasonlóan, a block data-t is program elején hajtja végre, mielõtt még a fõprogram futtatása elindulna. A block data elhelyezése a forráskód szempontjából közömbös, persze csak addig amíg nem beágyazott utasításként szerepel vagy a fõprogramban vagy egy segédprogramban. Ezért célszerû mindig ezek eléjére írnunk.

14. A bemenet és kimenet I. (fájlokra)

Eddig is látthattuk a Fortranban be- és kimenet (azaz I/O = input/output) használata teljesen megegyezik a hétköznapival. Sõt, akár fájlokból is beolvashatunk vagy fájlokba is kiirathatunk, amelyeket valamilyen fizikai egységen tároljuk: merevlemezen, floppyn, cd-n, dvd-n. A Fortranban minden egyes fájlhoz egy *egyedi szám* tartozik, melynek értéke egy 1 és 99 közötti egész szám lehet. Néhány egyedi szám azonban már foglalt: az 5 a standard bemenet, a 6 a standard kimenetre utal.

Fáljok megnyitása és bezárása

Használat elõtt a fájlt meg kell nyitnunk az *open* paranccsal.

```
open (különféle specifikációk listája)
```

ahol a leggyakoribb specifikációk az alábbiak:

```
[UNIT=]  u
IOSTAT=  ios
ERR=     err
FILE=    fname
STATUS=  sta
ACCESS=  acc
FORM=    frm
RECL=    rl
```

Az egyedi szám az *u*, ami egy 1-99 közötti egész szám. A program ezzel azonosítja fájlt. Fontos, hogy egy az egyes fájlok egyedi számai különfélék legyenek, épp ezért egyedi.

ios egy a be és kimenet (I/O) állapotát jelzi, ami egy egész szám. Ha az alkalmazás és visszatérés után az *ios* nulla, akkor sikeres, ha nem nulla, akkor sikertelen volt utasítás-végrehajtás.

err egy cimke, ahova akkor ugrik a program ha hibát talál..

fname a fáljnévre utal

sta a kezelendõ fálj korábbi állapotát jelzi, ami lehet NEW (=új, azaz a futtatáskor hozza létre), OLD (=régi, azaz már meglévõ fájl), és SCRATCH(= ideiglenes fálj, ami megnyitáskor vagy a program indításakor) keletkezik, de a bezárásnál (vagy a program lefutása után) azonnal törlõdik.

acc a fálj elérését jelenti, ami kétféle lehet vagy SEQUENTIAL (=szekvenciális, közvetett) vagy DIRECT (=direkt, közvetlen) Az alapértelmezett a SEQUENTIAL.

frm a fálj formátumát jelöli, ami lehet FORMATTED (=formázott) vagy UNFORMATTED (=formázatlan). Az alapértelmezett az UNFORMATTED.

rl jelöli egy direkt elérésû fálj rekordjainak a hosszúságát (a fálj méretére utal)

Miután megnyitottunk egy fájlt, utasításokat (kiiratást stb.) eszközölhetünk rajta. Miután ezekkel végeztünk a fáljt be kell zárnunk, ezt az alábbi módon kell megtennünk:

```
close ([UNIT=]u[,IOSTAT=ios,ERR=err,STATUS=sta])
```

ahol a zárójelben lévõ paraméterek értelme és használata szinte teljesen megegyezik a feljebb tárgyaltakéval. Egy kis eltérés azért van, ez a *sta*: ami kétféle lehet ebben az esetben: vagy KEEP (=megtartani a fájlt, ez az alapértelmezett) vagy DELETE (=letörölni a fájlt)

A read és write utasítás

Az egyetlen eltérés a korábban definált read/write utasításokhoz képest, az egyedi szám egzakt megnevezése és specifikációja.

```
read ([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s)
write([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s)
```

ahol a legtöbb specifikációt már feljebb értelmeztük. az END=s egy utasítás cimkére utal, ahova a program ugrik, miután elérte a kezelt fájl végét.

Egy példa

Adva van egy *points.dat* nevû adatfájl, amiben néhány pont xyz koordinátája van felsorolva. A pontok számát e fálj elsõ sorában találjuk.Az egyes koordináták formátuma F10.4 (errõl és a FORMAT utasításról majd késõbb lesz szó) A következõ kis program beolvassa az adatokat egy x,y,z (3 dimenziós) tömbbe.

```
      program inpdat
c
c  Ez a program beolvassa n pont adatait egy 3 dimenzios x,y,z tombbe
      integer nmax, u
      parameter (nmax=1000, u=20)
      real x(nmax), y(nmax), z(nmax)

c  Megnyitjuk az adatfile-t
      open (u, FILE='points.dat', STATUS='OLD')

c  Beolvassuk a pontok szamat
      read(u,*) n
      if (n.GT.nmax) then
         write(*,*) 'Error: n = ', n, 'ha nagyobb mint nmax =', nmax
         goto 9999
      endif

c  Beolvasasi ciklus a ponthalmazra
      do 10 i= 1, n
         read(u,100) x(i), y(i), z(i)
   10 enddo
```

```
  100 format (3(F10.4))

c  Close the file
      close (u)

c  Most kovetkezik a beolvasott adathalmaz alkalmazasa
c  (hianyzo resz)
9999 stop
      end
```

15. A bemenet és kimenet II. (más esetben)

Bármilyen programnyelvnél fontos a bemenet és kimenet (I/O)kezelése. Eddig példáinkban a két legelterjedtebb konstrukciót láttuk a `read` és `write` utasításokat. Fortranban az I/O használata elég gyakran elbonyolódik, a jegyzetünkben ezért csak az egyszerübb esteket vizsgáljuk meg.

A read és a write

`A Read` parancsot a bemenetre, míg a `write` utasítást a kimenetre használjuk. A formalizmus az alábbi:

```
read (egyedi szám, formázási szám) változók listája
write(egyedi szám, formázási szám) változók listája
```

Az egyedi szám utalhat a standard, alapértelmezett I/O-ra vagy vmilyen fájlra. A formázási számról késõbb lesz szó (a format parancs cimkéjére utaló szám).

Lehetséges ezen utasítások leirását egyszerûsítéshetjük a csillag (*) jelöléssel is, mint ahogy tettük is az eddigi példáinkban is. Ezt néha *általános* beolvasásnak és kiiratásnak is szokták nevezni.

```
read (*,*) változók listája
write(*,*) változók listája
```

Az elsõ parancs az alapértelmezett bemeneten a változók összes értékeit olvassa be, míg a második az alapértelmezett kimeneten keresztül írja ki az értékeket.

Példák

Nézzük az alábbi Fortran programszegmenst (részletet):

```
integer m, n
real x, y, z(10)

read(*,*) m, n
read(*,*) x, y
read(*,*) z
```

Itt a bemenet a standard (például egy adatfájlon keresztül az alapértelmezett bemenetre vonatkoztatva). Ez az adatfájl rekordokból (sorokból) áll, ahogy ezt a Fortran terminológiában megszoktuk. Példánkban minden rekord tartalmaz egy számot (lehet egész vagy valós). Az egyes rekordokat üres helyekkel vagy vesszõkkel választjuk el. Konkrét esetben a fenti bemeneti file így néz ki:

```
   -1  100
  -1.0 1e+2
  1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
```

Vagy vesszõkkel ugyanez:

```
   -1, 100
 -1.0, 1e+2
  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
```

Fontos tudni, hogy a Fortran 77 bemenete érzékeny a sorok helyére, ezért fontos, hogy ne tegyünk bele extra oszlopokat (mezõket) és sorokat (rekordokat). Azaz, ha így módosítjuk a felállást, azaz kiveszünk egy sort:

```
   -1, 100, -1.0, 1e+2
  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
```

akkor m és n értékét -1-nek és 100-nak fogja venni, és a sor utolsó két elemét (-1.0, 1e+2) elhagyja, x és y értékének az 1.0 és 2.0 -t veszi, a második sor többi tagját figyelmen kivül hagyja. A z elemeit pedig definiálatlannak veszi.

Ha pedig egy adott sorban túl kevés a bemeneti adat, akkor a következõ sorbeliekkel egészíti ki, tehát a

```
   -1
  100
 -1.0
 1e+2
  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
```

kombináció eredménye ugyanaz lesz, mint az elsõ két (jó) eset volt.

További változatok

További általános I/O is rendelkezésünkre áll:

```
      read  *, változók listája
      print *, változók listája
```

aminek jelentése pontosan ugyanaz, mint a feljebb leírt "csillagos", azaz általános read és write utasításoknak. Ez a verzió mindig az standard bemenet és kimenetet használja, ezért a * itt a formára utal.

16. Formázási lehetõségek a beolvasásnál és kiiratásnál (A Format parancs)

Eddig mi többnyire az ún.*kötetlen formázást* alkalmaztuk. Ez alapértelmezett szabályokat alkalmazott, mind a bemenet, mind a kimenet különféle típusaira. Azonban gyakran kivánunk egyéni, különleges I/O formátumot alkalmazni, például hány tizedesjegyet tekintsünk egy valós számnál. Erre van a *format* utasítás, melyet mind a bemenetre, mind a kimenetre is alkalmazhatunk

A format utasítás szintaktikája

```
      write(*, label) változók listái
 cimke format formázási kód
```

A format alkalmazását szemlélteti az alábbi kis példa. Tegyük fel, hogy van egy egész változónk, amit 4 karakteres alakban illetve egy valós számunk, amit fixpontos kiírással és 3 tizedesjeggyel kívánunk kinyomtatni.

```
      write(*, 900) i, x
  900 format (I4,F8.3)
```

A formázási cimke (900) megválasztása tetszõleges. Az elterjedt gyakorlat az, hogy a formázási cimke számot mindig megasabbnak választják, mint a többi utasításokét. A `format` utasítás mögötti zárójelben lévõ kódok utalnak a formázás alakjára. Ebben az esetben az I4 jelenti a 4 karakter hosszúságú egész számot, míg az F8.3 jelöli a fixpontos 8 karakter hosszúságú valós számot, melybõl 3 karakter a tizedesjegyeké.

A format utasítás a program bármelyik részében elõfordulhat. Két programozási stílus is van: az egyik a format utasítást azonnal a read/write parancsok mögé írja, a másik a format parancsot egy helyre összegyüjtve, a (szub)program végén jeleníti meg.

Alapvetõ formázási kódok

A leggyakoribb format kódok az alábbiak:

```
A - szöveg
D - pontosabb valós szám, exponenciális alak (normált alak)
E - valós szám, exponenciálos alak (normált alak)
F - valós szám, fixpontos formátum
I - egész
X - horizontális ugrás (space=szóköz)
/ - vertikális ugrás (új sor)
```

Az F (és a D és E) kódok általános alakja F*w.d* ahol *w* a szám hosszúságát (karaktereinek számát), *d* pedig egy egész szám, mely a tizedeshelyek számát adja meg.

Egész számoknál csak a szám hosszúságát kell megadni, tehát a szintakszis I*w*. Ehhez hasonló a szöveges formátumnál is megadható a hosszúság (A*w*), de ezt ritkán tüntetik fel.

Ha egy szám vagy szöveg nem tölti ki a teljes (elõre megadott) mezõt, akkor üres helyekkel egészíti ki a program. Általában hiányos a szöveget jobbról egészíti ki, de vannak olyan fordítók is, amik máshogy járnak el.

Szóközökre az *n*X kódot használják. Ez *n* szóközt jelent. Ha *n* -et elhagyjuk, akkor azt *n=1* -nek értelmezi (azaz egy szóköz).. Vertikális ugrásokra (új sor kezdésre) a / (törtvonal) jelölést alkalmazzuk. Minden ilyen törtvonal egy-egy új sort jelent. Fontos megjegyeznünk, hogy alapértelmezettként minden read vagy write utasítás egy ilyen új sorkezdéssel zárul. (Ebben a Fortran és a C között eltérés van.)

Néhány példa

Tekintsük az alábbi Fortran program részletet:

```
      x = 0.025
```

```
      write(*,100) 'x=', x
  100 format (A,F)
      write(*,110) 'x=', x
  110 format (A,F5.3)
      write(*,120) 'x=', x
  120 format (A,E)
      write(*,130) 'x=', x
  130 format (A,E8.1)
```

amit, ha lefutattjuk, akkor a kimenet az alábbi lesz:

```
x=      0.0250000
x=0.025
x=  0.2500000E-01
x= 0.3E-01
```

Érdemes megnézni, hogy a Fortran hogyan helyezi el az egyes eredményeket, illetve azt, hogy alapértelmezettként 14 karakter széles helyet hagy egy való számnak. A Fortran kerekítési szabályai: 0-4 és közötti számot lefelé, az 5-9 közötti értéket fölfelé kerekíti.

Ebben a példában minden egyes write utasításhoz, különféle formázási utasítást használtunk. Természetesen lehet, egyfajta formátumot több I/O utasításhoz is rendelni. Pont ebben rejlik a format parancs elõnye. Például egy táblázat elemeinek nyomtatásánál a sorokat egyfajta formátumban kívánjuk megjeleníteni, ilyenkor is praktikus a format utasítás.

A format parancs összevonása a read/write utasítással

Ahelyett, hogy egy különálló format utasítással adjuk meg az alakot, beépíthetjük közvetlenül az I/O parancsba is, az alábbi módon:

```
      write (*,'(A, F8.3)') 'A valasz x = ', x
```

ami ugyazt jelenti, mint a következõ: :

```
      write (*,990) 'A valasz x = ', x
  990 format (A, F8.3)
```

Néha a szöveges formátumot is megadjuk a format utasítással, ekkor ez a megoldás is ugyanazt adja, mint a fenti kettõ:

```
      write (*,999) x
  999 format ('A valasz x = ', F8.3)
```

Ciklusok és ismétlõdõ számítások

Most tekintsünk egy kicsit bonyolultabb esetet it. Tegyük fel, hogy van egy kétdimenziós, egész elemekbõl álló tömbünk, melynek csak balfelsõ 5*10-es részmátrixot kívánjuk kinyomtatni, 5 sorban tizesével.

```
      do 10 i = 1, 5
         write(*,1000) (a(i,j), j=1,10)
   10 continue
 1000 format (I6)
```

Két ciklusunk van, egy explicit, ami a sorokon szalad végig, és egy implicit, mely az oszlopokon (j index).

Gyakran egy format utasítás ismétlõdésekbõl állhat, például:

```
  950 format (2X, I3, 2X, I3, 2X, I3, 2X, I3)
```

De erre is van egy összevont írásmód:

```
  950 format (4(2X, I3))
```

Lehetséges az is, hogy megengedjük az ismétlést, anélkül, hogy feltüntetnénk hányszor is kéne ismételnünk. Tegyük föl, hogy van egy vektorunk, aminek az elsõ 50 elemét kívánjuk kinyomtatni, tizesével megjelenítve. Egy lehetséges megoldás:

```
      write(*,1010) (x(i), i=1,50)
 1010 format (10I6)
```

Ez a format parancs 10 szám kinyomtatását jelenti. De mi a write utasítással 50 szám kiírását akarjuk elérni, tehát miután kinyomtatta az elsõ 10 darab számot, "visszaugrik" és kinyomtatja a következõ tizet, majd még ezt háromszor megismétli.

Implicit do ciklusokat alkalmazhatunk sokdimenziós (3 fölött) esetekre is, de mi inkább kerüljük a használatát, mivel túlzottan elbonyolítja a programunkat és megnehezíti a megértését. A fent tekintett példa egy egyszerû esete volt ennek, a sokdimenziós problémák sokkal összetettebbek.

17.1 BLAS

BLAS egy angol mozaikszó: Basic Linear Algebra Subroutines. (=Alapvetõ Lineáris Algebrai Segédprogramok) Ahogy már a névbõl is sejthetjük, ez különféle segédprogramokat tartalmaz alapvetõ vektor és mátrixmûveletekhez. A BLAS-ból építették föl a bonyolultabb kódrendszereket is, mint például a LAPACK-ot. A BLAS és annak forráskódja a Netliben keresztül hozzáférhetõ. Sok számítógépkereskedõnek már egy a saját gépükre optimizált, feltunningolt verziója van. Ez az egyik fõ elõnye a BLAS-nak: minden verziója kompatibilis a másikkal, tehát a BLAS bármilyen olyan gépen müködik, ahova egyszer már telepítették. A BLAS (és továbbfejlesztése: a LAPACK) egy egyszerüen kezelhetõ, kombatibilis, de egyben magas szinvonalú program, mely fõleg (de nemcsak) a lineáris algebrai számításokat könnyíti meg.

Szintek és elnevezési szokások

A BLAS segédprogramjai 3 szint (level) szerint osztályozhatóak:

- **Level 1:** Vektor-vektor operátorok. *O(n)* adat és *O(n)* számítás
- **Level 2:** Mátrix-vektor operátorok. *O(n^2)* adat és *O(n^2)* számítás.
- **Level 3:** Mátrix-mátrix operátorok. *O(n^2)* adat és *O(n^3)* számítás

Minden egyes BLAS és LAPACK subroutinenak vannak további verziói, például megkülönböztetjük ezeket a számolt adat típusa szerint is. A segédprogram nevének elsõ betûje erre utal:

```
S      Valós
D      Pontosabb valós
C      Komplex
Z      Pontosabb komplex
```

A double precision-os (pontosabb) komplex nincs egyértelmûen definálva a Fortran 77-ben, de a legtöbb fordító elfogadja az alábbi deklarációt:

```
double complex változók listája
complex*16     változók listája
```

BLAS 1

Néhány eleme a BLAS 1 segédprogram családnak:

- xCOPY - egy vektort a másikba másol
- xSWAP -felcserél két vektort
- xSCAL - konstans értékkel szorozza a vektort (nyújtja)
- xAXPY - két vektor összeadása
- xASUM - normált formája a vektornak
- xNRM2 - más típusú normált forma
- IxAMAX - megkeresi egy vektor legnagyobb elemét

Az elsõ betû (x) lehet akár S,D,C,Z is, ez a pontosságtól függ. Egy gyors (angol nyelvû) leírás található az alábbi címen: http://www.netlib.org/blas/blasqr.ps

BLAS 2

Néhány eleme a BLAS 2 segédprogram családnak:

- xGEMV - mátrix-vektor szorzás
- xGER - mátrix rangja
- xSYR2 - szimmetrikus mátrix rangja
- xTRSV - 3 ismeretlenes 3 egyenletbõl álló feladat megoldása

További részleteket a BLAS 2-rõl ezen a helyen találunk: http://www.netlib.org/blas/blas2-paper.ps. (ez is angolul van)

BLAS 3

Néhány eleme a BLAS 3 segédprogram családnak:

- xGEMM - mátrixok összeszorzásageneral matrix-matrix multiplication
- xSYMM - mátrixok szimmetrikus összeszorzása
- xSYRK - szimmetrikus mátrix rangja
- xSYR2K - szimmetrikus mátrix rangja - továbbfejlesztett verzió

A magasabb szintû mátrix operátorokat, mint például magasabb rendû lineáris egyenletrendszerek megoldása, csak a LAPACK-ban találunk. További információkért ide érdemes kattintanunk: http://www.netlib.org/blas/blas3-paper.ps. (angol nyelvû)

Példák

Elõször nézzünk egy egyszerû BLAS segédprogramot, az SSCAL-t

A subroutin meghivása az alábbi:

```
call SSCAL ( n, a, x, incx )
```

ahol *x* egy vector, *n* a vektor hosszúsága (*x* azon elemeinek száma, amit használni kivánunk). Az *incx* jelöli a növekményt. Általában az *incx=1,* ekkor visszakapjuk az eredeti egydimenziós x vektorunkat. Ha *incx>1*, akkor a kifejezés azt jelenti, hogy mennyit kell ugranunk az x vektor egyes elemei között.Például ha *incx=2* , akkor ezt azt jelenti hogy csak minden második elemét kell a vektornak nyújtani (tehát a vektor dimenziójának minimum 2n-1 - nek kell lennie). Ezek ismeretében nézzük az alábbi példát, ahol *x* -et valós `real x(100)` alakban deklaráltuk.

```
call SSCAL(100, a, x, 1)
call SSCAL( 50, a, x(50), 1)
call SSCAL( 50, a, x(2), 2)
```

Az elsõ sor mind a 100 elemet szorozza *a*-val. A második sorban már csak az utolsó 50 elemét nyújtja meg *a*-val. A harmadik sorban lévõ pedig a páros indexû elemeket szorozza.

Fontos tudni, hogy ezzel az utasítással x-et felülírjuk az új értékekkel. Ha szükségünk van az eredeti x-re, másolatot kell róla készítenünk, a SCOPY használatával.

Most tekintsünk egy bonyolultabb esetet is. Tegyük fel, hogy van 2 db kétdimenziós tömbünk (mátrixunk) : A és B. A feladat az A*B szorzás eredményének megadása. A megoldás az, hogy kiszámoljuk A i sorának elemeinek és B j oszlopának elemeinek szorzatát, majd összeadjuk. Erre használhatjuk a BLAS1 segédprogramját az SDOT-ot. A nehézség a helyes indexezésben és a növekményben rejlik. Az SDOT formulizmusa az alábbi:

```
SDOT ( n, x, incx, y, incy )
```

Tegyük föl, hogy a tömböket az alábbi módon definiáltuk:

```
real A(lda,lda)
real B(ldb,ldb)
```

de a programban az A aktuális mérete m*p, míg B-é p*n. A ideik sora az *A(i,1)* elemmel kezdõdik. De tudjuk, hogy a Fortran oszlop szerint számol, tehát a következõ *A(i,2)* elemet *lda*-ban tárolja átmenetileg (mivel *lda* az oszlop hosszúság). Ezért *incx = lda* beállítást alkalmazzuk. B oszlopainál nem áll fent ez a probléma, tehát itt *incy = 1*. A belsõ produktum hossza *p.* Tehát a korrekt válasz ez:

```
SDOT ( p, A(i,1), lda, B(1,j), 1 )
```

Hol szerezzük be a BLAS-t?

Elõszõr nézzük meg, hátha van a számítógépünkön BLAS. Ha nincs, akkor itt megtalálhatjuk: http://www.netlib.org/blas.

Dokumentációk

A BLAS subroutine-ok szinte majdnem teljesen maguktól értetõdõek.. Ha tudjuk, hogy melyik routinra van szükségünk, megkeressük és elolvassuk a hozzá tartalmazó leírást, ahol a bemenet és kimenet paraméterek használatát részletesen ismertetik. Majd mi is nézünk erre egy példát a következõ fejezetben, ahol is a LAPACK segédpogramokról lesz szó.

17.2 LAPACK

A LAPACK magasabb számításokhoz (pl. összetett egyenletrendszerek megoldása, szingularitási problémák stb.). szükséges Fortran segédprogramok gyûjteménye. A LAPACK mára már kiváltotta a régebbi LINPACK illetve EISPACK programokat. A LAPACK megírásánál törekedtek a BLAS minél hatékonyabb felhasználására.

Routine-ok

Talán a legelterjedtebb LAPACK routine-ok az egyenlet megoldó segédprogramok:

- xGESV - AX=B típusú egyenlet megoldása, ha A tetszõleges mátrix
- xPOSV - AX=B típusú egyenlet megoldása, ha A szimmetrikus pozítiv definít mátrix

Természetesen még sokféle segédprogram van, a különféle igényeknek megfelelõen.

A forráskód és a végrehajtási parancssor (korlátozottan) ezen a linken érhetõ el: http://www.netlib.org/lapack. A LAPACK Felhasználói Kézikönyv teljes verziója is megtekinthetõ a Neten (angol nyelven).

Dokumentáció

A BLAShoz hasonlóan a LAPACK segédprogramjai is majdnem teljesen maguktól értetõdõek. A bemeneti és kimeneti paraméterek részletes leírásait az egyes fájlokhoz csatoltan megtaláljuk (ún. header (fej) rész). Például nézzük meg SGESV segédprogram angol nyelvû, eredeti leírását. (A szakirodalom angol nyelvû, ezért szinte bizonyos, hogy csak angol nyelven juthatunk hozzá.)

```
      SUBROUTINE SGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
*
*  -- LAPACK driver routine (version 2.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     March 31, 1993
*
*     .. Scalar Arguments ..
      INTEGER            INFO, LDA, LDB, N, NRHS
*     ..
*     .. Array Arguments ..
      INTEGER            IPIV( * )
      REAL               A( LDA, * ), B( LDB, * )
*     ..
*
*  Purpose
*  =======
*
*  SGESV computes the solution to a real system of linear equations
*     A * X = B,
*  where A is an N-by-N matrix and X and B are N-by-NRHS matrices.
*
*  The LU decomposition with partial pivoting and row interchanges is
*  used to factor A as
*     A = P * L * U,
*  where P is a permutation matrix, L is unit lower triangular, and U is
*  upper triangular.  The factored form of A is then used to solve the
*  system of equations A * X = B.
*
```

```
*  Arguments
*  =========
*
*  N       (input) INTEGER
*          The number of linear equations, i.e., the order of the
*          matrix A.  N >= 0.
*
*  NRHS    (input) INTEGER
*          The number of right hand sides, i.e., the number of columns
*          of the matrix B.  NRHS >= 0.
*
*  A       (input/output) REAL array, dimension (LDA,N)
*          On entry, the N-by-N coefficient matrix A.
*          On exit, the factors L and U from the factorization
*          A = P*L*U; the unit diagonal elements of L are not stored.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  IPIV    (output) INTEGER array, dimension (N)
*          The pivot indices that define the permutation matrix P;
*          row i of the matrix was interchanged with row IPIV(i).
*
*  B       (input/output) REAL array, dimension (LDB,NRHS)
*          On entry, the N-by-NRHS matrix of right hand side matrix B.
*          On exit, if INFO = 0, the N-by-NRHS solution matrix X.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B.  LDB >= max(1,N).
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
*          > 0:  if INFO = i, U(i,i) is exactly zero.  The factorization
*                has been completed, but the factor U is exactly
*                singular, so the solution could not be computed.
*
*  =====================================================================
```

17.3 Könyvtárfájlok használata UNIX alatt

Egy Fortran segédprogram-csomag akár több száz fájlból is állhat. Ha csak egyet is használni kívánunk közülük, idõpocsékoló és értelmetlen újra és újra lefordíttattni az összeset. UNIX operációs rendszerû számítógépen ezt a problémát elkerülhetjük az ún. *könyvtárfájl (library file)* használatával. Mivel a könyvtárfájl egy teljesen "normális" fájl, elég csak a fõprogramunkat lefordítani, majd ezt linkelni a könyvtárfájl(ok)hoz. A linkelés sokkal gyorsabb, mint a fordítás.

A könyvtárfájlok neve `lib` -bel kezdõdik és `.a`.-val végzõdik. Nem kizárt, hogy néhány könyvtárfájlt már a rendszergazda telepített a gépre, általában ezekbe a könyvtárakba: `/user/lib` és/vagy `/user/local/lib`. Tehát például lehetséges, hogy a BLAS könyvtárfájl ebben a fájlban található./ `user/local/lib/libblas.a`. A `-l` használatával linkelhetjük össze a könyvtárfájlt programunkkal, az alábbi módon:

```
f77 fo.f -lblas
```

Természetesen több könyvtárfájlt is linkelhetünk:

```
f77 fo.f sub.f -llapack -lblas
```

Fontos a felsorolás sorrendje, a fenti példánkban -llapack hamarabb listázódik ki, mint -lblas, mivel a LAPACK hívja a BLAS routine-okat.

Ha egy saját könyvtárfájlt kívánunk létrehozni, akkor miután a forrás kódokat lefordítottuk célkódokká egy közös fájlba kell gyûjtenünk ezeket. Ebben a példában egy `my_lib` nevû könyvtárfájlt hoznuk létre:

```
f77 -c *.f
ar rcv libmy_lib.a *.o
ranlib libmy_lib.a
rm *.o
```

Az és utasításokról több infót akármelyik UNIX kézikönyvben is találhatunk. Miután megvan a könyvtárfájlunk, már linkelhetjük is:

```
f77 main.f -L. -lmy_lib
```

Ebben rejlik a könyvtárfájlok nagy elõnye, elég egyszer lefordítani, de ezután akárhányszor használhatjuk.

17.4 Hol találhatjuk meg azt a numerikus szoftvert, amire épp szükségünk van?

Egy átlagos matematikai probléma megoldásához szükségünk van különféle kódrészletekre. Gyanítjuk, hogy ezt már valaki elõttünk megírta, de nem tudjuk, hogy hol találjuk meg. Néhány jótanács:

1. Kérdezzük meg munka- vagy hallgatótársunkat! Célszerû mindig ezzel kezdeni, mert ha nem, akkor rengeteg idõt vesztegethetünk el a kereséssel, és lehet, hogy a mellettünk dolgozó, tanuló személynél is megtalálnánk azt amire szükségünk van.
2. Keressük fel a neten a Netlib-et. Talán a keresés funkciójuk segíthet.
3. Nézzük meg itt is : GAMS, (=Guide to Available Mathematical Software=Kalauz az Elérhetõ Matematikai Szoftverekhez, a NIST terméke)

18. Fortran programozási stílusok

Nagyon sokféle programozási stílus van, de azért érdemes a következõ elvi megoldásokat, ötleteket alkalmaznuk.

Mobilitás

Célszerû a standard, "gyári" Fortran 77-et használnunk. Az egyetlen eltérés, amit mi ettõl a standard verziótól megengedtünk, az alacsony betük használata volt.

A programunk felépítése

Programunk legyen jól tagolt, moduláris. Minden egyes subroutine/segédprogramunk egy-egy konkrét (rész)feladatot oldjon meg. Sokan az egyes segédprogramokat külön-külön fájlokba írják.

A commentek

Nagyon fontos, hogy a commentek segítségével írjuk le a forráskódban, hogy mi is történik éppen! Még ennél is fontosabb, hogy az egyes segédprogramokhoz érthetõ leírást adjunk (ún. header-t) a be- és kimenetrõl és hogy mit csinál konkrétan.

Bekezdések

Mindig használjunk bekezdéseket a ciklusokra és a a feltételes utasításokra, ahogy ebben a leírásban is láthattuk.

Változók

Mindig deklaráljuk az összes változót. Próbáljuk ne impliciten, összevissza, hanem egy helyen megtenni ezt .Lehetõség szerint a változók neve ne legyen 6 karakternél hosszabb, és fontos, hogy egyedi legyen.

Szubprogramok, segédprogramok

Sose engedjük meg az ún."side effects" -et, azaz sose változtassuk a bemenõ paraméterek értékét. Nagyon fontos a segédprogramok megfelelõ használata is.

A deklarációkban válasszuk szét a paramétereket, a közös tömböket és a helyi változókat.

Ha lehetséges, kerüljük a közös tömbök használatát.

Goto

Minél kevesebbszer használjuk a *goto* parancsot. Persze bizonyos ciklusoknál kénytelenek vagyunk használni, mivel a *while* nem alapértelmezett a Fortranban.

Tömbök, vektorok

Az esetek döntõ többségében a legjobb a tömböket már a fõprogramban deklarálni és argumentumként továbbítani a különbözõ segédprogramokba. Fontos, hogy a fõdimenziót is továbbítsuk. Kerüljük a mátrixok számításközbeni méretváltoztatását!

Hatékonyság

Mikor egy dupla ciklussal kezeljük egy kétdimenziós tömb elemeit, célszerû mindig az elsõ (sor) indext tenni a belsõbb ciklusba. Ennek magyarázata a Fortran adattárolási szisztémájában rejlik.

Amikor egy feltételes elágazásunk van (if, else if), célszerû a legáltalánosabb feltétellel kezdeni, majd szûkíteni a kört.

19. Hibakeresés

Egy becslés szerint egy kereskedelmi szoftver kifejlesztési idejének 90%-a csak a hibakeresés és a tesztelés. Pont ezért is elsõdlegesen fontos a nemcsak helyes, hanem az érhetõ kód megírása.

Ennek ellenére a használat során mindig találni hibákat (bugokat), Néhány tanács, ezen hibák detektálására és kezelésére:

Hasznos fordító funkciók

A legtöbb Fortran fordító rendelkezik olyan funkciókkal, amelyeket a felhasználó szabadon alkalmazhat. A következõkben a SUN Fortran 77 fordítóprogramhoz olvashatunk hasznos dolgokat, de a legtöbb más fordítóhoz is hasonló megállapításokat tehetnénk. (maximum ott a betük jelentése más).

-ansi
: Figyelmeztett a nem standard Fortran 77 kiegészítésekre a mi verziónkban.

-u
: Minden beépített típus deklaráció szabályt mellõz, azaz kikapcsolja az automatikus deklarációt. Ha ezt a funkciót a mi fordítónk nem támogatja, egy szinte teljesen hasonló dolgot kaphatunk, ha az alábbival egészítjük ki a deklarációnkat

```
implicit none (a-z)
```

(segéd)programjaink elején.

-C
: Ellenõrzi a tömbök kiterjedését, méretét, és megpróbálja detektálni a tömbön kivüli elemeket. Sajnos nem tudja az összes hibát így megtalálni.

Néhány gyakori hibaforrás

Néhány gyakori hibaforrás, ezekre (is) kiemelten figyeljünk:

- Ellenõrizzük, hogy minden sorunk legkésõbb a 72 oszlopban befejezõdik, mert az ezen túliakat figyelmen kivül hagyja!
- Összeillenek-e a hívó és hívott programok paraméter listái?
- Összeillenek a közös tömbök?
- Nem egész számú osztást alkalmaztunk a valós helyett?
- Nme gépeltünk *o*-t a 0 helyett vagy *l*-t az 1 helyett?
- stb. stb. stb.

További hibakeresési tanácsok

Ha hibát észlel a fordító, próbáljuk megtalálni. Szintaktikai (elírási) hibákat könnyü megtalálni. Az igazi problémát a logikai vagy a futattási hibák jelentik. A régi módszer erre, hogy sok write utasítással nyomon követjük a változók értékeit. Ez egy kicsit fárasztó és "macerás" mód, mivel újra és újra fordítani kell a programot, mivel mindig kicsit módosítjuk a programunkat. Újabban speciális hibakeresõket is alkalmazhatunk. Soronként haladva vagy egyéni felosztás szerint végignézzük a programot, megjelenítjük a részeredményeket és még sok minden... A legtöbb UNIX rendszerû számítógépen a *dbx* és *a gdb* nevû hibakeresõket találjuk. Sajnos ezek többsége régimódi szöveges megjelenésû, de már létezik grafikus verzió is, mint például az *xdbx* vagy *dbxtool*.

20. Hasznos weboldalak a Fortranról

- Fortran FAQ (gyakran ismételt kérdések)
- Fortran Piac
- Fortran Standard Dokumentumok
- Jegyzet: A Fortran 90-rõl Fortran 77 programozóknak

Fontos: ezen linkek mindegyike angol nyelvû!