PROGRAMACIÓN CIENTÍFICA
CURSO 2025-2026

EVA CERNADAS GARCÍA

Coordinadora da materia

Centro Singular de Investigación
en Tecnoloxías Intelixentes da USC (CiTIUS)
Despacho 207

Grupo de clases expositivas **CLE1**
Grupos de clases interactivas **CLI1, CLI2** e **CLI3**

PROGRAMACIÓN CIENTÍFICA
CURSO 2025-2026

MANUEL FERNÁNDEZ DELGADO

Centro Singular de Investigación
en Tecnoloxías Intelixentes da USC (CiTIUS)
Despacho 207

Grupo de clases expositivas **CLE2**
Grupos de clases interactivas **CLI4, CLI5, CLI6**

Ubicación: Centro Singular de Investigación en Tecnoloxías Intelixentes da USC (CiTIUS)

Facultade de Matemáticas

CiTIUS, 2ª planta Despacho 207

Obxectivos da asignatura

1) Dominar a programación en linguaxes estructuradas.
2) Analizar, deseñar, programar e implementar algoritmos de resolución de problemas matemáticos sinxelos.

- **FORTRAN**: linguaxe de programación compilada.
- **OCTAVE/MATLAB**: linguaxe interpretada (Octave é a versión libre de Matlab).

Material da asignatura

- Todo o material da asignatura atópase na páxina web (presentacións, exercicios resoltos, solucións de exames):

  http://persoal.citius.usc.es/manuel.fernandez.delgado/programacion/

  Utilizaremos os foros do campus virtual da USC: **https://cv.usc.es**

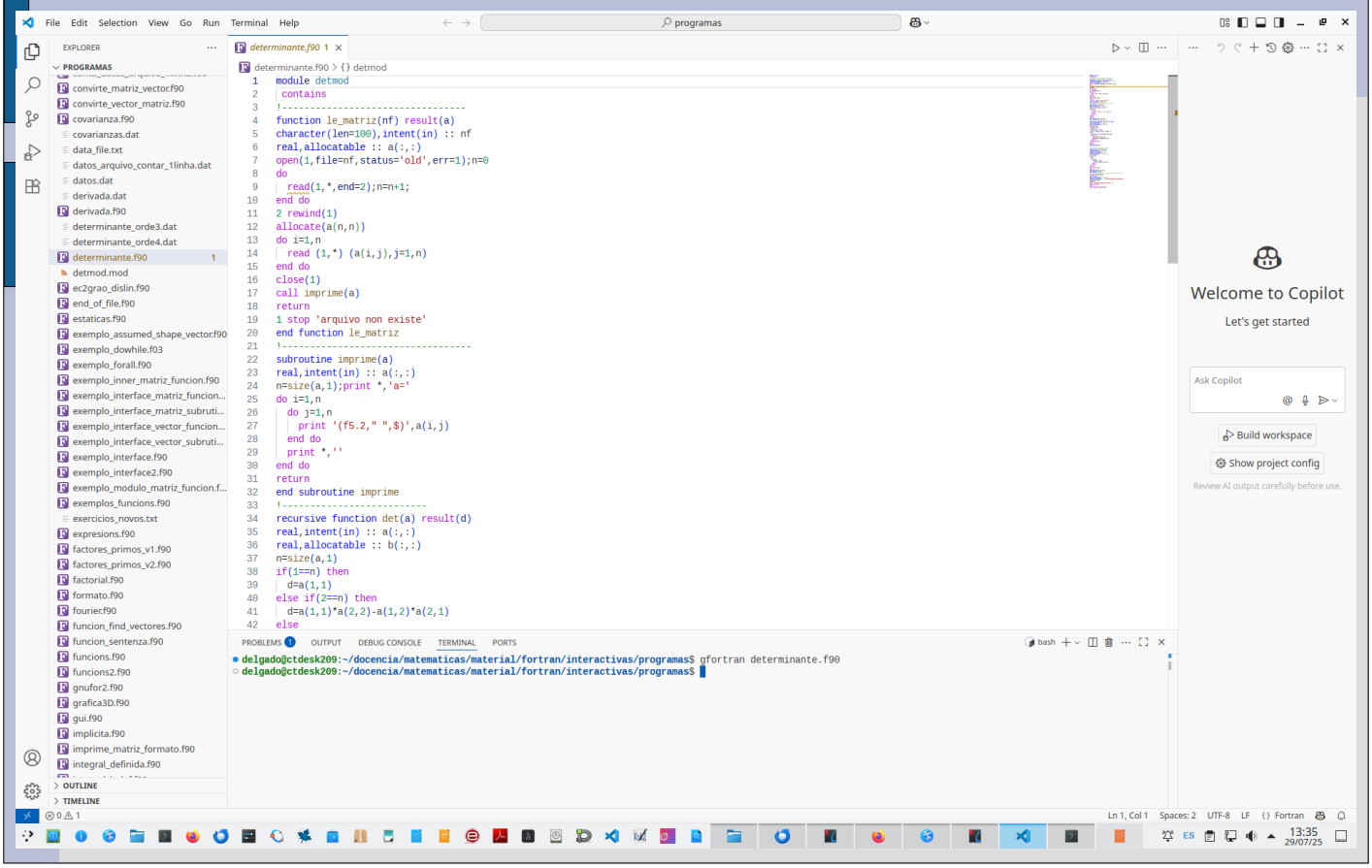
- Enlace para encargar apuntes:

  https://servizosdixitais.fundacionusc.gal/lista-productos-csd/

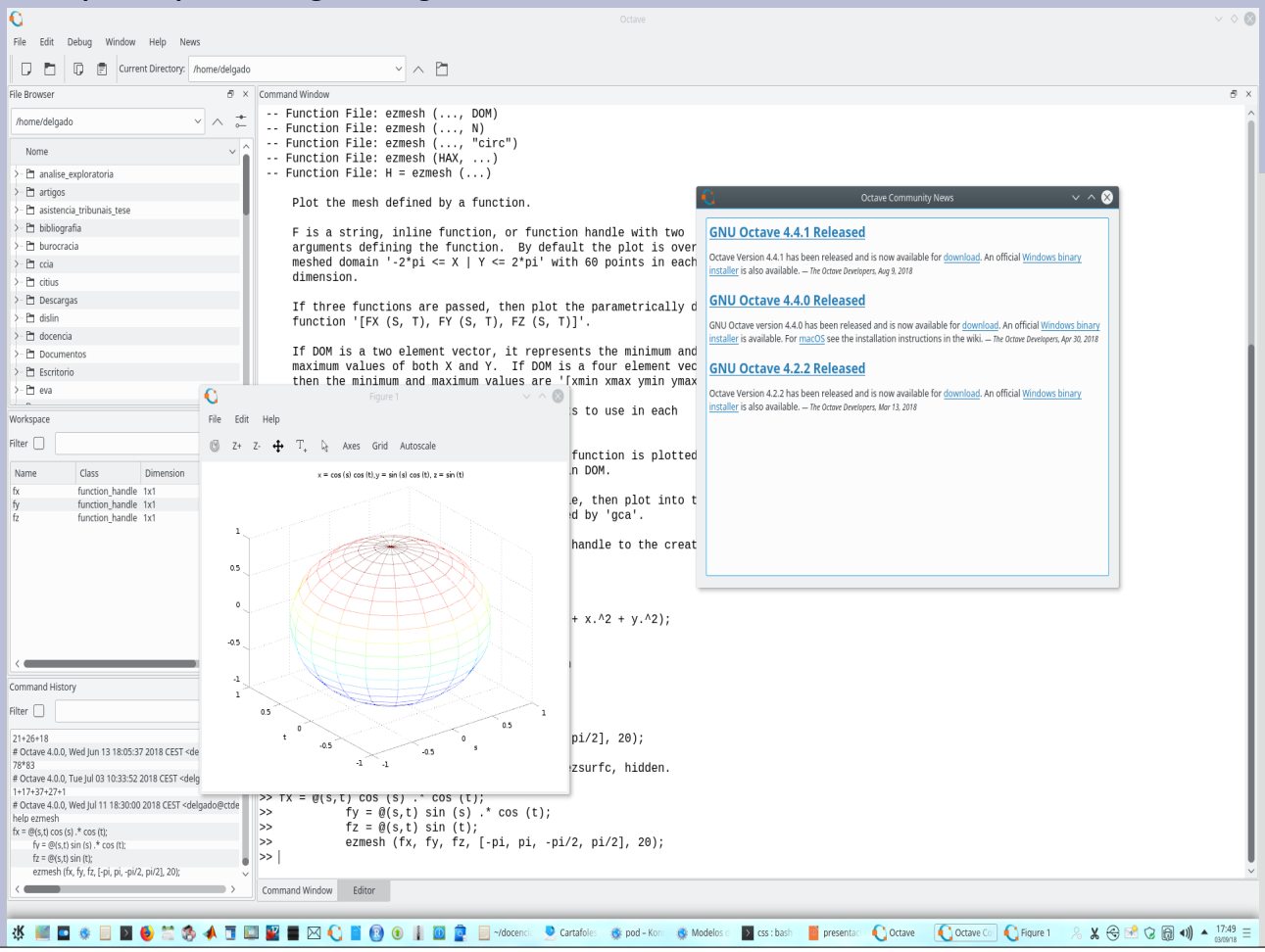
(filtra por Profesores y selecciona Manuel Fernández/Eva Cernadas)

FUNDACIÓN USC

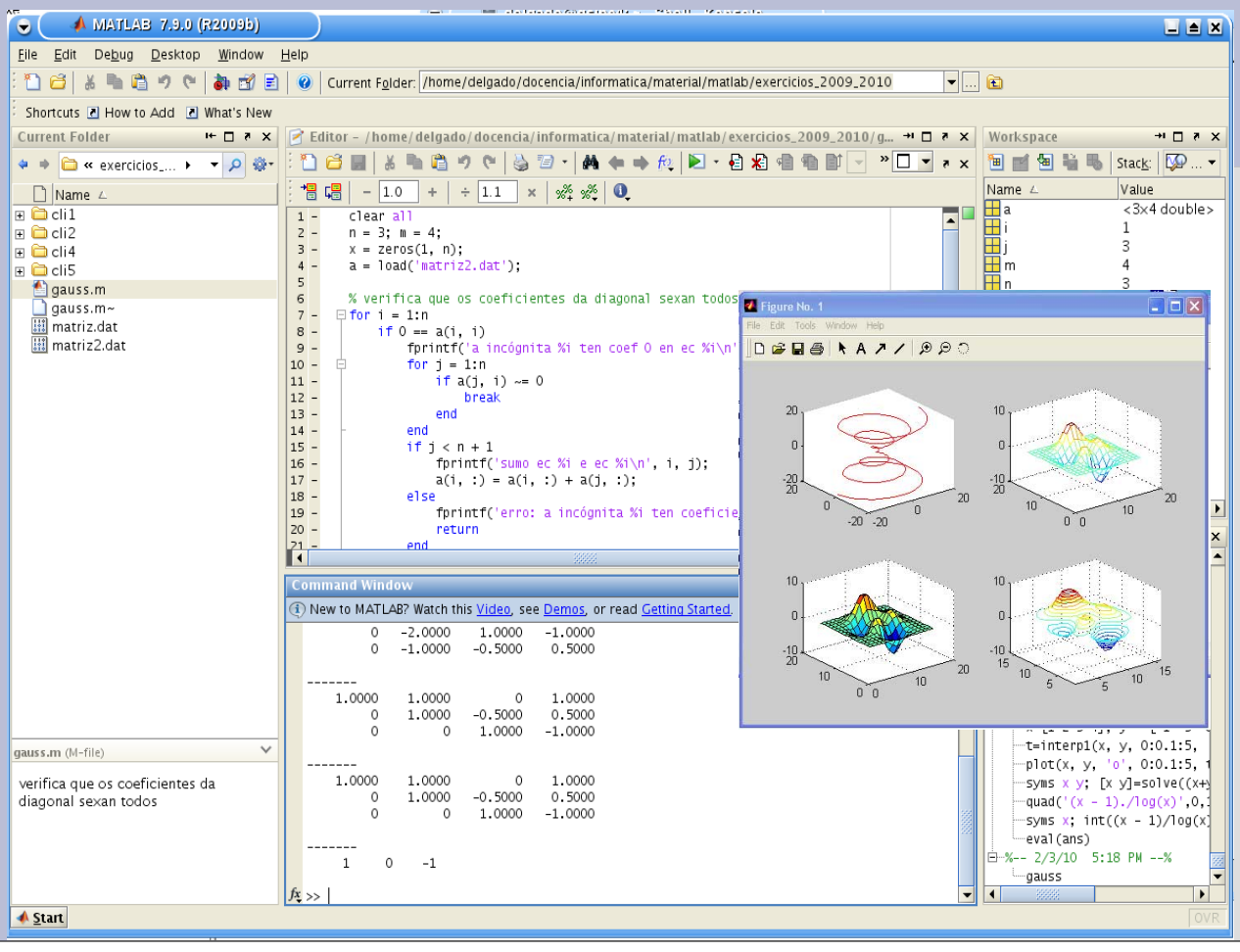
Compilador **Gfortran:**https://www.equation.com/ftpdir/gcc/gcc-13.2.0-32.exe
Entorno **VSCode**: https://code.visualstudio.com/download



Octave:
https://ftpmirror.gnu.org/octave/windows/octave-10.2.0-w64-installer.exe



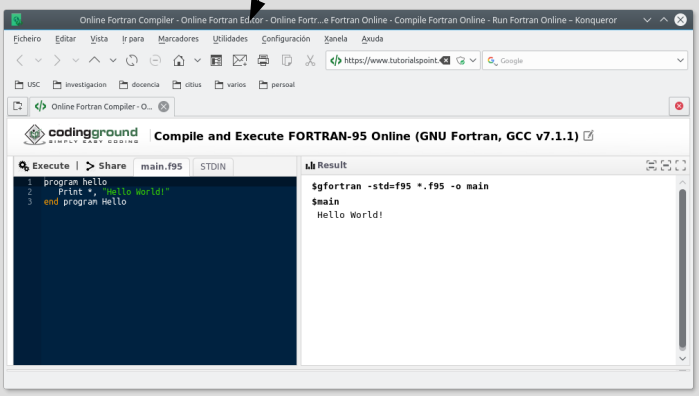
Matlab: https://es.mathworks.com/academia/tah-portal/universidad-de-santiago-de-compostela-31485094.html



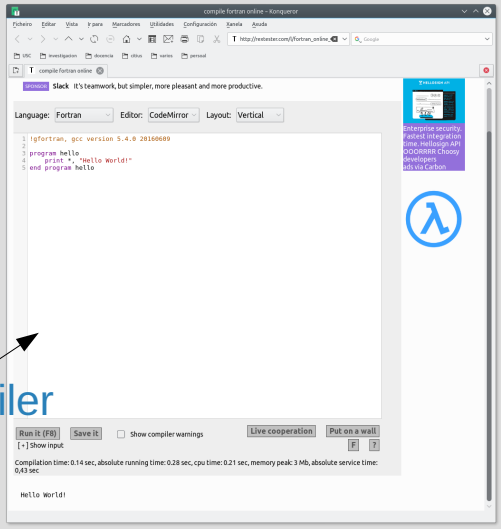
Execución on-line de Fortran

Fortran pódese executar online en dous sitios:

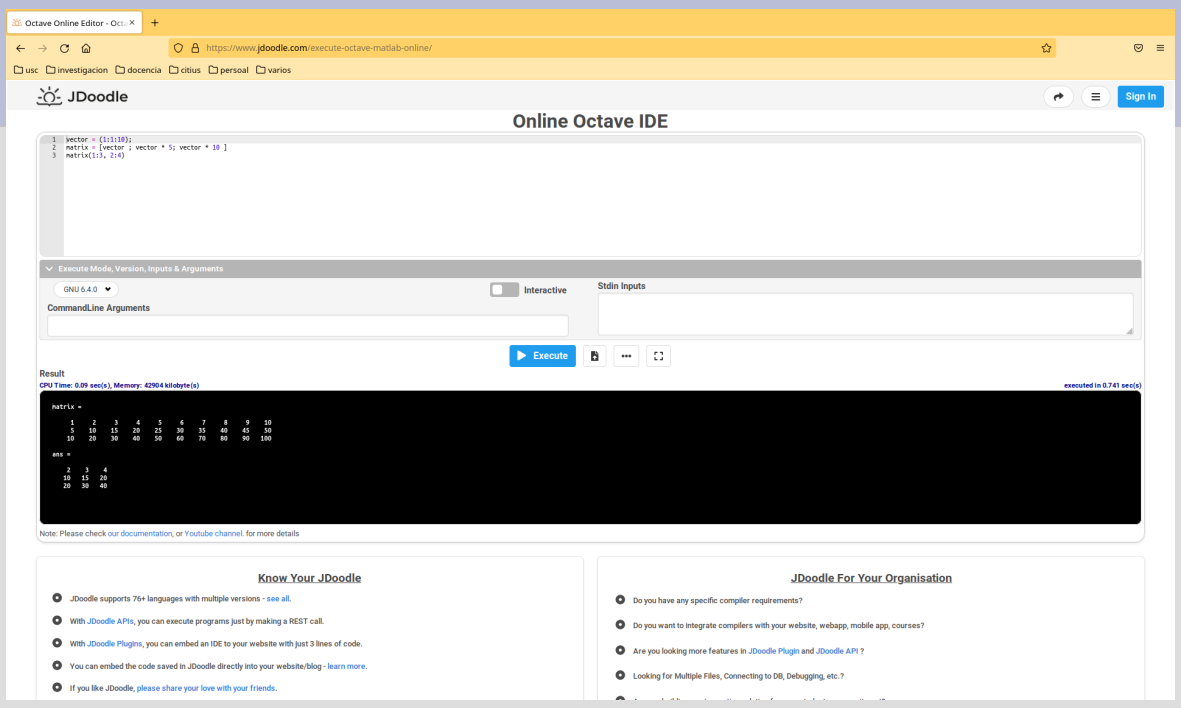
https://www.tutorialspoint.com/compile_fortran_online.php



http://rextester.com/l/fortran_online_compiler



Execución on-line de Octave



https://octave-online.net

https://www.jdoodle.com/execute-octave-matlab-online/

Metodoloxía docente

- **Nas clases expositivas**:
  - Expoñemos os conceptos básicos da programación e vemos exemplos de programas importantes.
- **Nas clases interactivas**:
  - En **Fortran** e **Octave**: escribes, depuras e executas **programas**, resolves paso a paso problemas de programación de forma planificada e razoada, adoptas decisións de deseño para optimizar a eficiencia (tempo, memoria RAM).
  - En **Octave** tamén executas comandos que realizan operacións matemáticas e representacións gráficas.

Avaliación

- **Avaliación continua (2 puntos)**: realización de exercicios durante as clases interactivas, diante do ordenador, que se entregan para a súa avaliación.
- **Exame final (10 puntos)**: exame diante do ordenador co material do curso en papel ou memoria USB. Contén 2 partes: Fortran e Octave, tes que obter como mínimo 1 punto en cada parte.
- **Avaliación final = exame final + avaliación continua**
- Tódolos exames de anos anteriores están resoltos neste enlace.

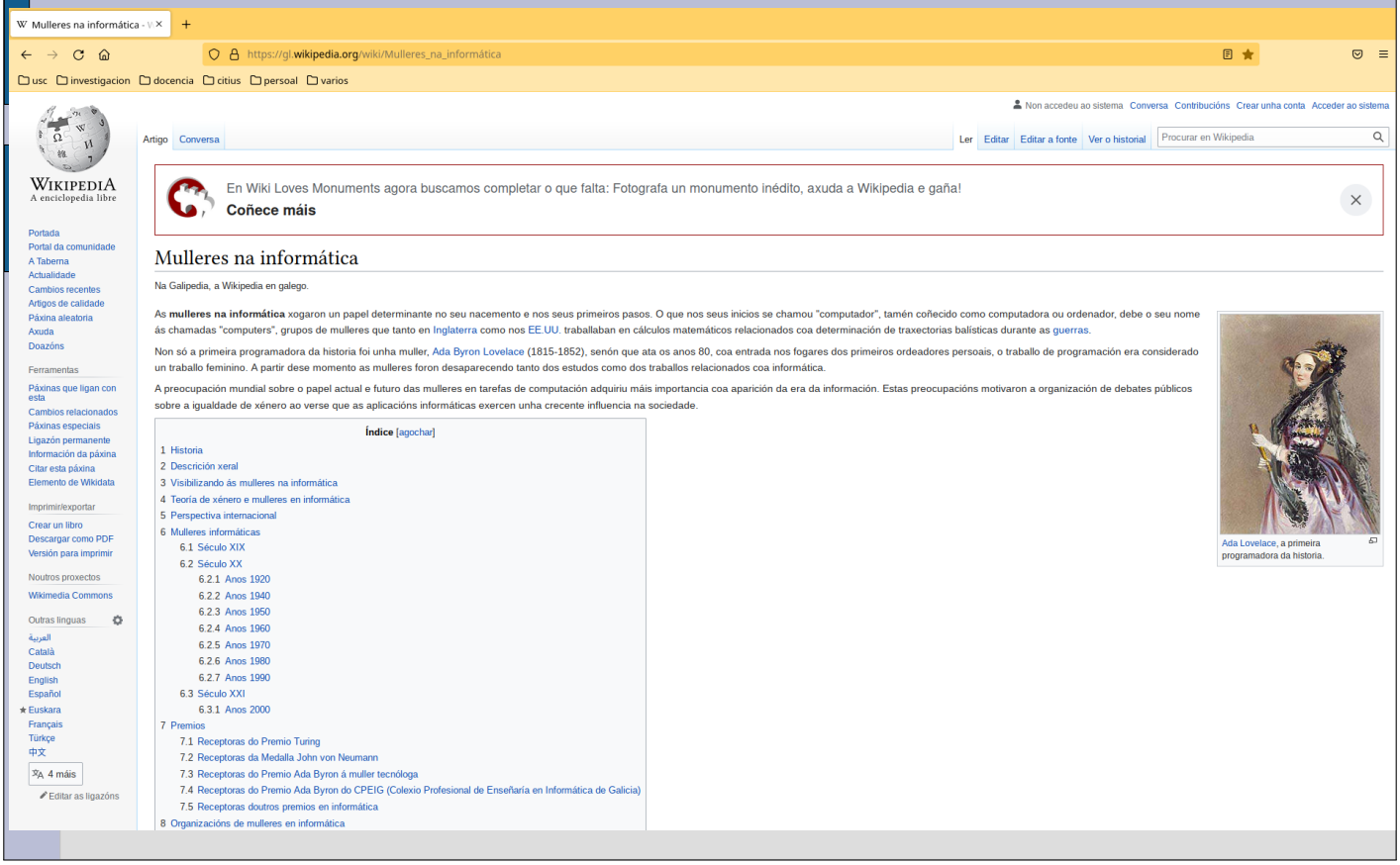
Recomendacións

- **Asistencia a clases expositivas e interactivas.**
- **Realización no ordenador os exercicios propostos por semana e revisar os exames resoltos.**

- **Contidos fundamentais en Fortran e Octave:**

1) Vectores e matrices.
2) Sentenzas de selección e iteración.
3) Subprogramas con paso de vectores e matrices.

Mulleres na informática



Bibliografía

Páxina web da asignatura

- Fortran: **Programación estructurada con Fortran 90/95.** J. Martínez Baena, I. Requena Ramos, N. Marín Ruiz, Editorial Universidad de Granada, 2006

- Matlab: **Matlab©: Una introducción con ejemplos prácticos**. A. Gilat, Editorial Reverté

PROGRAMACIÓN ESTRUCTURADA EN FORTRAN

As mulleres na programación e na informática

Proceso de programación en Fortran

- Escribe o programa fonte *programa.f90* no **entorno** *Visual Code*
- Compilación do programa dende a **terminal** do VSCode:

gfortran programa.f90

- Corrección dos erros de compilación
- Creación de programa executábel: *a.exe*
- Execución de programa na terminal: .*a.exe*
- Corrección de erros de execución e lóxicos (edición, compilación, execución)
- Se executas *gfortran programa.f90 -o programa.exe*, o executable chamarase *programa.exe*
- **Coidado**: non poñas *programa.f90* no canto de *programa.exe*: se o fas, sobrescribes o *programa.f90* co executable

Programa básico en Fortran

- Resumo do proceso: *programa.f90* Compilador *gfortran* *a.exe*

```
program proba
print *,'ola!'
end program proba
```

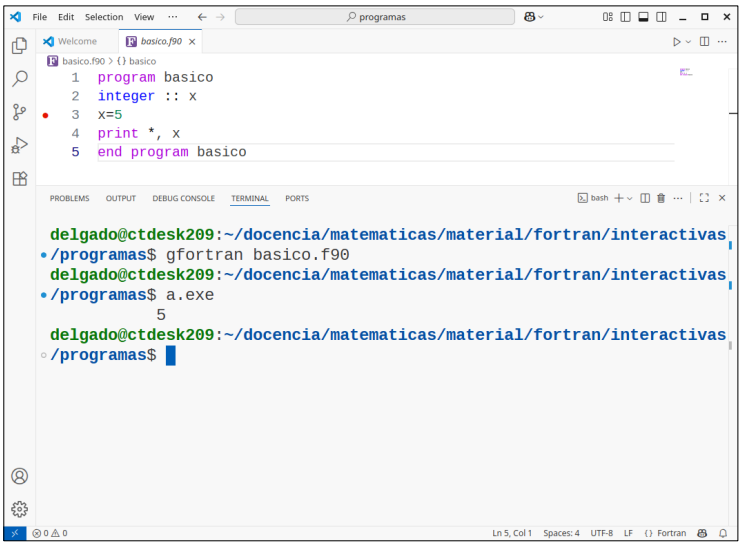
- Comeza con sentenza: "*program nome*"
- Remata con "*end program nome*"
- "*nome*" é o nome do programa: non pode haber variábeis nin subprogramas con ese nome
- Logo de *program*:
  - declaración de variábeis (ao principio)
  - sentenzas executábeis: operacións, entrada / saída, ...

Programa básico en Fortran

- Sentenza “*stop*”: remata a execución:
- *stop ‘mensaxe’*: remata e imprime a mensaxe
- Exemplo de programa básico:

```
program basico
integer :: x
x=5
print *, x
end program basico
```

Imprime o nº 5 na terminal de comandos



- Liñas de comentarios: con ! ao comezo da liña

Entrada e saída estándar

- Entrada de datos estándar: le datos dende o teclado e almacénaos en variábeis
  - Sentenza *read*: *read *, x*
  - Pódese producir un erro se a variábel é numérica e introducimos un carácter (p.ex.)
- Saída de datos estándar: visualiza na terminal (pantalla)
  - Sentenza *print*: *print *, ‘resultado=‘, 2*x*
  - Con formato: *print ‘(“n=”,i0)’,n*
    print ‘(“x=”,f10.6)’,x

Ancho do nº enteiro

Ancho e nº de decimais

Algoritmo e diagrama de fluxo

Algoritmo: método para a resolución dun problema, detallado completamente en tódolos seus pasos.

- **Entrada** de datos: dende teclado ou arquivos
- **Procesamento**: operacións cos datos
- **Saída** de resultados: por pantalla ou a arquivos

Lectura de datos por teclado ou escritura de datos na terminal:

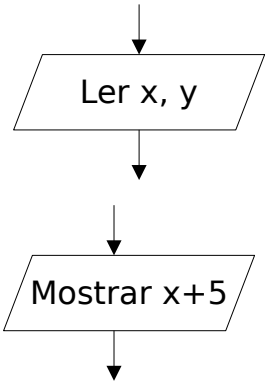
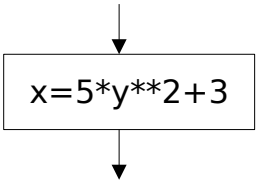


Diagrama de fluxo:

- Forma de representar gráficamente un algoritmo como unha secuencia de operacións
- Diagrama de caixas (operacións) e flechas que as conectan (orden de execución).

Operación aritmética ou asignación:



Estrutura de selección básica

- IF/ELSE: avalía unha serie de condicións e executa unhas sentenzas ou outras dependendo de que condicións se cumpren

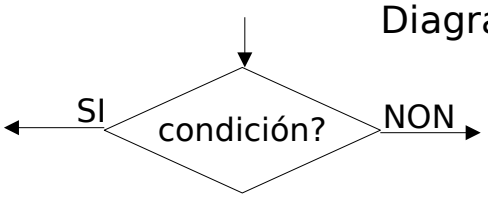
Sintaxe:

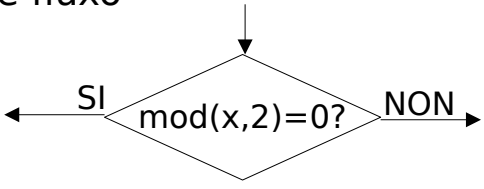
```
if(condicion) then
   sentenzas1
else if(condición2) then
   sentenzas2
else
   sentenzas3
endif
```

Exemplo:

```
if (x > 3) then
    print *, 'alto'
else
    print *, 'baixo'
end if
```

Diagrama de fluxo





Estrutura iterativa *do* definida

- Permite repetir unha ou varias sentenzas un certo número de veces

Sintaxe:

```
do var=ini, fin, paso
   sentenzas
end do
```

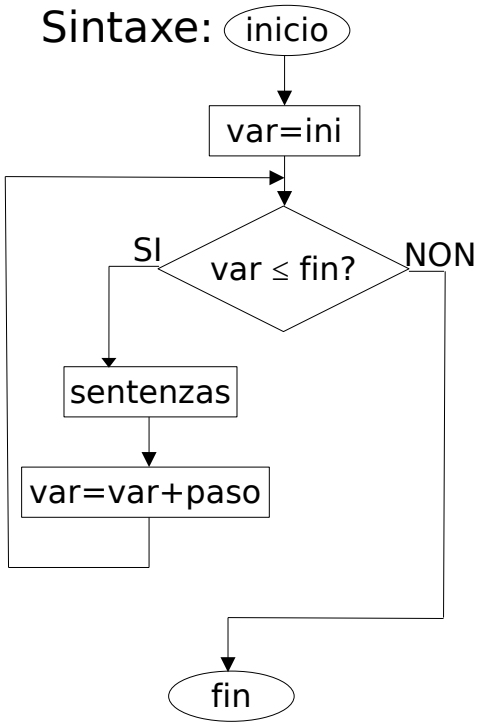
Exemplo:

```
do i=1, 10
   print *,i+2
end do
```

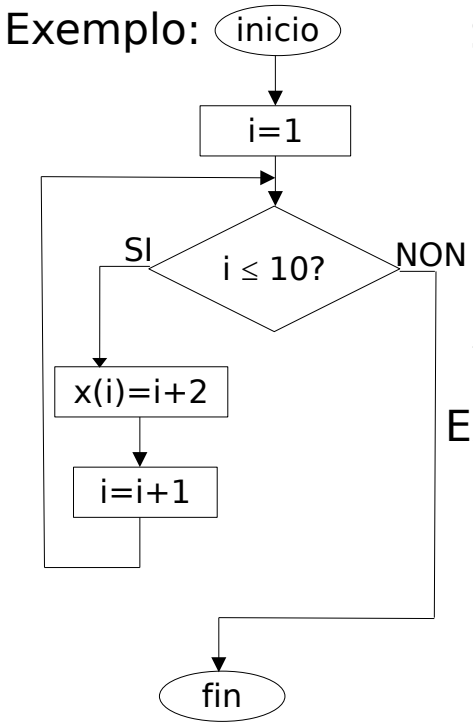
- Repite as sentenzas dende que a variábel *var* adopta o valor *ini*, ata que adopta o valor *fin*.
- En cada repetición, *var* increméntase en *paso*
- O incremento *paso* vale 1 por defecto

Diagrama de fluxo *do* definido

Sintaxe:

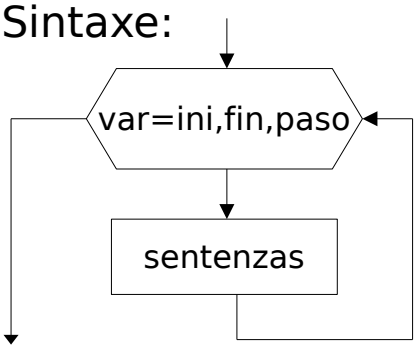


Exemplo:

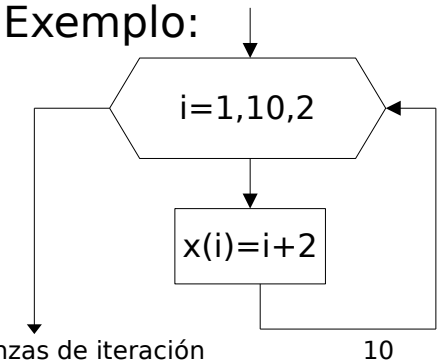


Versión simplificada:

Sintaxe:



Exemplo:



Acumulador

- Variable que aparece na esquerda e na dereita dunha asignación dentro dun bucle *do*: Exemplo: *x=x+i*; *p=p*j*
- Emprégase en operacións cun número variábel de operandos, que require un bucle *do*.
- O acumulador debe inicializarse co elemento neutro da operación.
- Acumulador de suma e de producto dun número variábel (*n*) de valores:

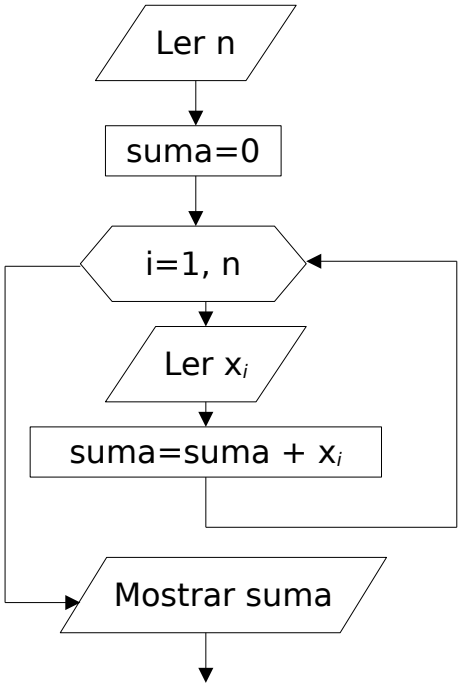
$\sum_{i=1}^{n} x_i$

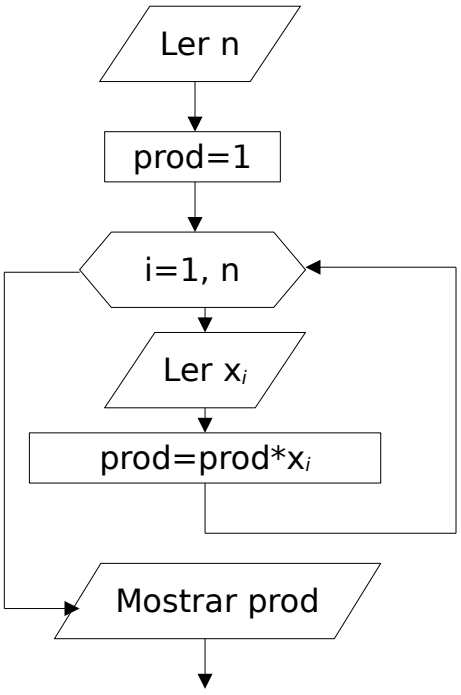
```
suma=0
do i=1,n
    print *,'x? '
    read *,x
    suma=suma+x
end do
```

$\prod_{i=1}^{n} x_i$

```
prod=1
do i=1,n
    print *,'x? '
    read *,x
    prod=prod*x
end do
```


Diagrama de fluxo de suma e producto





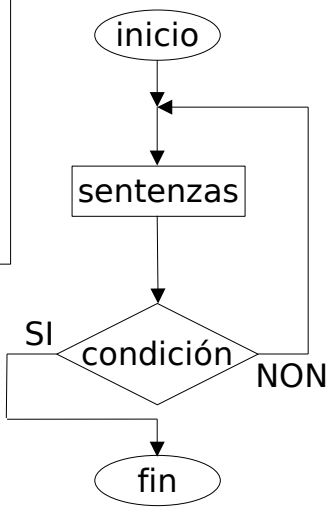
Estruturas iterativas *do* indefinidas

- Bucle *do/exit*: repite as sentenzas e remata cando se cumpre unha condición.

```
do
   sentenzas
   if(cond) exit
   sentenzas
end do
```

```
x=0;dx=0.1
do
   x=x+dx
   print *,x,x*x
   if(x>1) exit
end do
```

```
x=0;dx=0.1
do
   if(x>1) exit
   x=x+dx
   print *,x,sin(x)
end do
```



- Bucle *do while*: repite mentres se cumpre a condición.

```
do while(condición)
   sentenzas
end do
```

```
x=0;dx=0.1
do while(x<1)
   print *,x,x*x
   x=x+dx
end do
```


Grace Murray Hopper (1906-1992)

- Inventora de compiladores das linguaxes de programación A0 e B0, para o cálculo de nóminas
- Contra-almirante da US Navy
- Muller do ano en Informática (1969), primeira muller na British Computer Society (1973)

Linguaxes de programación

- Linguaxe de programación: conxunto de sentenzas, regras sintácticas e palabras chave que permiten especificarlle ao ordenador unha tarefa a realizar
- Programa: arquivo de texto que contén sentenzas (comandos) na linguaxe de programación: programa fonte
- Estas sentenzas convértense en sentenzas executábeis (programa executábel) polo microprocesador usando un **compilador**, ou son executadas por un **intérprete**

Linguaxes compiladas e interpretadas

- Linguaxes de programación:
  - **Compiladas**: un programa traduce o programa fonte a programa executábel. Hai arquivos fonte e executábel (pódese executar só): Fortran, C/C++
  - **Interpretadas**: un programa interpreta o arquivo fonte, traduce e executa os seus comandos liña a liña. Non hai arquivo executábel. Só se pode executa-lo programa fonte co programa intérprete: Octave, Matlab, R, Python, Java

Linguaxes compiladas e interpretadas que usaremos

- Linguaxe **compilada**: Fortran. O compilador (*gfortran*) traduce o programa fonte (*programa.f90*) a programa executábel (*programa.exe*). Executas: *programa.exe*.
- Xera programas máis rápidos que os interpretados. É máis difícil de programar nel por ser de baixo nivel, é decir, proporciona menos utilidades.
- Linguaxe **interpretada**: Octave e Matlab. Non hai arquivo executábel. Para executa-lo programa fonte, necesítase o intérprete: *octave programa.m* ou *matlab -batch programa*
- Xera programas máis lentos que os compilados. Máis fácil de programar por ser de máis alto nivel, xa que ten máis utilidades.

Etapas do proceso de programación (I)

1) Análise do problema
2) Deseño do algoritmo
3) Codificación do programa fonte
4) Depuración do código
5) Proba do programa
6) Mantemento

Análise do problema: especificar resultados a obter, datos de partida, posíbeis erros ou situacións límite, comportamento nestos casos, medidas de rendemento

Deseño do algoritmo: técnica do deseño descendente: división do problema en subproblemas máis simples; resolución individual de cada subproblema se a súa complexidade o permite; caso contrario, nova división en subproblemas

Etapas do proceso de programación (II)

Codificación do programa fonte:

- Escritura do arquivo de texto nun entorno de desenvolvemento (VSCode)
- Escritura de sentenzas que resolven cada subproblema por separado
- O código debe ser entendible: nomes de variábeis, coherencia, documentación, ...
- É inevitábel cometer erros, que hai que correxir ...

Depuración do programa: corrección dos erros cometidos na codificación

- Erros de **compilación** (sintaxe)
- Erros de **execución**: prodúcense durante a execución do programa, aínda que non se viola a sintaxe da linguaxe, por exemplo división por cero, multiplicación de matrices non permitida, lectura de datos inválidos ... Producen o remate prematuro do programa
- Erros **lóxicos**: o programa remata ben pero non da resultados correctos

Etapas do proceso de programación (III)

Proba:

- Execución do programa con múltiples datos para comprobar que funciona ben en tódalas posíbeis situacións

Mantemento:

- Corrección de erros que aparezan durante a explotación do programa
- Realización dos cambios necesarios para adaptarse a cambios no entorno (formato das entradas e saídas, librarías usadas, ...).

Joan Clarke (1917-1996)

"A veces, la persona a la que nadie imagina capaz de nada, es la que hace cosas que nadie imagina"

Joan Clarke (1917)
Descrifó los mensajes alemanes de Enigma.

- Informática do exército británico especializada en criptografía
- Conseguiu descifrar o código Enigma dos alemáns durante a 2ª guerra mundial
- Nomeada cabaleira da Orde do imperio británico en 1947

Constantes e variábeis

- Os datos poden ser:
  - constantes: non se poden modificar durante a execución; dánselle valores no momento en que se comezan a usar: poden ter nome ou non
  - variábeis: pódense modificar, sempre teñen nome
- O nome dun dato pode ter letras, números, "_":
  - Non pode comezar por números
  - Non pode ter símbolos especiais: +?-=)/*$#

Nomes e tipos de datos

- Fortran NON distingue entre maiúsculas e minúsculas. Sempre usaremos minúsculas
- Os nomes deben ter significado: radio, temperatura, altura, ... Tamén deben ser curtos
- Os datos almacénanse en memoria RAM
- Datos de distintos tipos: enteiros, reais, reais de dobre precisión, complexos, lóxicos e carácter. Ocupan un nº de bytes distinto, e teñen un rango de valores distinto
- Fortran proporciona funcións intrínsecas para realizar operacións estándar (p. ex: funcións matemáticas estándar)

Declaración de variábeis e constantes

- En xeral, os datos deben ser declarados. Na declaración indícase o seu nome e tipo:

integer x *integer :: x*

- Toda declaración debe estar antes de calquer outra sentenza que non sexa unha declaración. Se non, erro de compilación.
- Pódense inicializar na declaración, neste caso hai que poñer o símbolo :: por exemplo *integer :: x = -5*
- As constantes con nome decláranse co atributo *parameter* e sempre hai que inicializalas. Ex:

integer, parameter :: x = -10

- Tamén se poden usar constantes sen nome dos distintos tipos en expresións aritméticas: *print *, x + 3.5*

Funcións relacionadas cos tipos dos datos

- *huge(...)*: indica o valor máximo que pode acadar un dato do tipo indicado

  integer x

  *print *, huge(x)* => 2147483647

- *precision(...)*: indica o nº de decimais

  real y

  real(kind = 8) z

  *print *, precision(y), precision(z)* => 6 15

- *kind(...)*: indica o nº de bytes que ocupa

  *print *, kind(x), kind(z)* => 4 8

Datos enteiros e reais

- Datos **enteiros**: *integer x*
  - ocupan 4 bytes, signo +/-, sen punto decimal
  - rango valores: $-2^{31}...2^{31}-1$
- Datos **reais**: *real x*
  - ocupan 4 bytes: signo +/-, partes enteira e decimal, expoñente (máx. 2 díxitos)
  - sen expoñente: *x=-1.2*
  - con expoñente: *x=-0.45e-18*
  - rango: $-3.4\cdot10^{38}...-3.4\cdot10^{-38}$, $3.4\cdot10^{-38}...3.4\cdot10^{38}$
  - precisión: 6 cifras decimais

Overflow e underflow; reais dobres

- Overflow / underflow: erro que se produce cando superamos o rango de valores dunha variábel real
  - Overflow: supérase o límite máximo: $\pm 3.4 \cdot 10^{38}$
  - Underflow: supérase o límite mínimo: $\pm 3.4 \cdot 10^{-38}$

```
real :: y = -1.2e+80
          1
Error: Real constant overflows its kind at (1)
```

```
real :: y = -1.2e-80
          1
Warning: Real constant underflows its kind at (1)
```

- Reais de **dobre precisión**:
  - 8 bytes
  - Precisión: 15 cifras decimais

Exemplo: cálculo da media de 100 millóns de datos:
• Se sumamos e dividimos por N => overflow
• **Solución**: calcular 100 medias de 1 millón e a media destas 100

Reais dobres e complexos

- Reais de dobre precisión (continuación):
  - Declaración:

    real(8) x

    real(kind=8) x

    double precision :: x
  - Rango: $-1.79 \cdot 10^{308}$, ..., $-1.79 \cdot 10^{-308}$, $1.79 \cdot 10^{-308}$, ..., $1.79 \cdot 10^{308}$
- **Complexos**: parte real e imaxinaria

complex c

c=(-1.3, 1.5e-18)

complex :: c=(-1,2)

Mostrar por pantalla: *print *, c* => (-1.3, 1.5e-18)

Lóxicos e carácter

- **Lóxicos**: só poden toma-los valores *.true.* e *.false.* (constantes)
  - Declaración: *logical x*
  - Inicialización: *x=.true.*
  - Mostrar por pantalla: *print *, x* => T
- **Cadeas de caracteres**:
  - Hai que indica-lo seu tamano máximo: non se pode superar
  - Se non se indica o tamano máximo, vale 1: *character* :: *s='a'*
  - Declaración: *character(100) s*
  - Inicialización: s='ola que tal'
  - Lonxitude da cadea (nº de caracteres que realmente ten): *len_trim*(*s*)
  - Mostrar por pantalla: *print *, s* => hola que tal
  - Teste de igualdade: *s==t*
  - Relación de orde alfabética: *s<t*

Declaración implícita (I)

- Na práctica, os datos básicos (reais e enteiros) non é necesario declaralos:
- Os datos enteiros non é necesario declaralos cando os seus nomes comezan polas letras *i j k l m n*
- Os datos reais non é necesario declaralos cando os seus nomes comezan polo resto de letras
- En resumo: se non declaramos un dato:
  - Se o seu nome comeza por {i j k l m n}, é enteiro
  - En caso contrario, é real

Declaración implícita (II)

- Se queremos datos doutro tipo (real dobre, complexo, lóxico, carácter, vector ou matriz), hai que declaralos
- A sentenza *implicit none* anula a declaración implícita no subprograma actual (*f95* da erro de compilación se hai variábeis sen declarar)
- A sentenza:

implicit tipo(rango), ..., tipo(rango)

permite asignar rangos de letras a tipos. Ex:

implicit integer(a-b), real(c-d), complex(e-z)

Vectores e matrices (I)

- Vector: **colección de datos** do mesmo tipo almacenados xuntos na memoria RAM, un índice. Declaración: *integer :: x(10)*
- Matriz: dous índices (fila e columna). Declaración: *integer :: a(3,3)*
- Acceso a elementos e grupos de elementos: *x(i), x(i:j), x(i:), a(i,:), a(:,i), a(i:j,k:l)*. Vectores e matrices son **arrais**. Deben declararse.
- Inicialización de vector: *x=[1,2,3], x=[(i,i=1,10,2)], x=(/1,2,3/)*
- Inicialización de matriz *a*: a=*reshape([1,2,3,4,5,6,7,8,9], shape(a))*: os valores van por columnas
- **Vector/matriz estático:** mesma lonxitude en tódalas execucións do programa.

```
real v(3)
integer :: a(3,3)
v(1)=2
v=[1,2,3]
a(2,3)=5
```

```
real v(10),a(2,2)
v=[(2*i+1,i=1,10)]
a=reshape(([((i*j,j=1,2),i=1,2)],shape(a)))
```

Vectores e matrices (II)

- **Vector/matriz dinámico**: o vector pode ter distintas lonxitudes en distintas execucións.
- Resérvase a memoria so cando se coñece o nº de elementos
- Se se accede a un elemento do matriz/vector **dinámico** antes do *allocate* ou despois do *deallocate* da erro de segmentación

```
real,allocatable :: v(:)
integer,allocatable :: a(:,:)
read *,n,nf,nc
allocate(v(n),a(nf,nc))
v(3)=2
a(2,3)=5
deallocate(v,a)
```

No *deallocate* non se indica o nº de elementos

- Pódese engadir elementos a un vector dinámico, inicialmente baleiro ou non:

```
real,allocatable :: v(:)
v=[1]
do i=1,3
    v=[v,i]
end do
```

```
real,allocatable :: v(:)
allocate(v(0))
do i=1,3
    v=[v,i]
end do
deallocate(v)
```


Vectores e matrices (III)

- Outra forma de declaración: *tipo :: nome(N_1:N_2)*. É para que o 1º elemento do vector non sexa o 1 e o último non sexa o *N*
- Exemplo: *integer :: x(-5:5)*. Ten elementos x(-5)...x(5)
- Con esta declaración, o vector ten $N_2 - N_1 + 1$ elementos: as funcións *lbound(x)* e *ubound(x)* dan N_1 e N_2. Cun vector dinámico sería: *real,allocatable :: x(:)*, e logo *allocate(x(-15:15))*
- **Ler** vector por teclado: *read *,x*
- **Ler** matriz:

```
do i = 1,n
   read *, (a(i, j),j=1,m)
end
```

```
read *,a
a=transpose(a)
```

Le a matriz por columnas: hai que traspoñer

Vectores e matrices (IV)

- **Escribir** vector: *print *,x* (tódolos elementos),

  *print *,x(1:n)* } só *n* primeiros
  *print *,(x(i),i=1,n)* } elementos

- **Escribir** matriz:

```
do i = 1,2
   print *, (a(i, j),j=1,2)
end
```

- **Inicialización** con bucle *do* explícito:

```
do i=1,n
    v(i)=i**2
end do
```

```
do i=1,n
    do j=1,m
        a(i,j)=i*j+i
    end do
end do
```

- **Inicialización** con bucle *forall*:

```
forall(i=1:10) v(i)=i*i+i-1
```

```
forall(i=1:5,j=1:6) a(i,j)=i*j-i+j
```


Funcións e vectorización

- Escribir unha expresión con vectores/matrices como se fose con números
- Sexan *x,y* dous vectores (de igual lonxitude) e *a,b* dúas matrices (de iguais dimensións).
- Dimensións: *size(x), size(a),size(a,1), size(a,2), shape(x), v=shape(a)*
- Asignación de valores a un arrai ou asignación dun arrai a outro: *x(:)=5; a(:,:)=7; x=y; b=a*
- Operacións aritméticas compoñente a compoñente: *x+2*y, x*y, x/y, 1/x, x**y, 1/a, a*b, 2**a, a**2, a**b*
- Suma e producto de elementos: *sum(x), sum(x(2:)), sum(a), sum(a,1), sum(a,2), product(x), product(a)*. Media dun vector: *sum(x)/size(x)*
- Valores e índices dos elementos máximo e mínimo dun vector: *maxval(x), minval(x), maxloc(x), minloc(x)*. O mesmo para matrices.
- Producto escalar: *dot_product(x,y)*
- Transposta dunha matriz: *b=transpose(a)*
- Producto de dúas matrices: *p=matmul(a,b)*

```
integer :: a(2,2)=reshape((/1,2,
   3,4/), shape(a)),x(2)
x=sum(a,2)
print *,sum(a,1)
```

Atopar o índice do elemento dun vector cun certo valor

- Índice do 1º elemento dun vector cun valor:

  i=***findloc***(vector, valor, 1)

- Retorna o índice *i* do primero elemento do vector con ese valor; 1 é a dimensión do vector
- Exemplo:

  integer :: x(4)=[3,1,4,4]

  *i=**findloc**(x,1,1)*

  *print *,'i=',i* (mostra a posición 2 por pantalla)

Ada Lovelace (1815-1852)

- Inglaterra, século XIX (1815-1852)
- Inventora do primeiro programa de ordenador
- Precursora en máis de 100 anos da informática
- Linguaxe de programación de sistemas de tempo real chamado Ada na súa honra

Expresións aritméticas

- Operadores aritméticos: + - * / ** (exponenciación): x + 3, y – z, 4.3E-2*x, x/y, x**2
- Son binarios (2 operandos). O operador – pode ser unario: *print *, -x*
- Precedencia (prioridade):

  1) **

  2) * /

  3) - unario

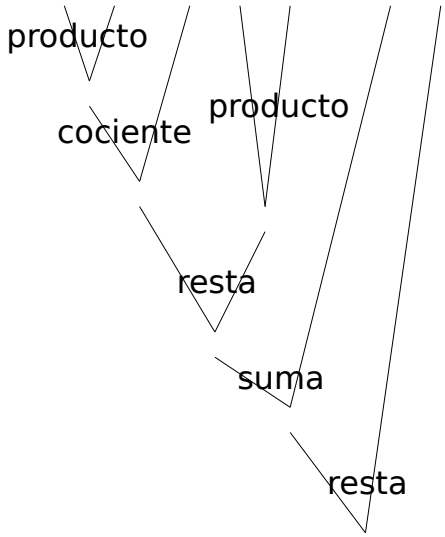
  4) + -
- Operadores de igual prioridade: execútanse de esquerda a dereita, agás a exponenciación (**), que se executa de dereita a esquerda

Prioridades na exponenciación

- Exponenciación: $x^{y^{z^t}}$ -> x**y**z**t
- Execútase de dereita a esquerda:
  - 1) z**t
  - 2) y elevado ao resultado de (1)
  - 3) x elevado ao resultado de (2)
  - Poderiamos escribir: x**(y**(z**t)), pero non é necesario poñer parénteses porque as prioridades xa fan que se execute nesa orde

Prioridades cos restantes operadores

a*b/c-4*y+3-c



As operacións combinan constantes e variábeis

Uso de parénteses

- As prioridades pódense modificar usando parénteses en función das nosas necesidades
- Exemplo: $\frac{\frac{x+y}{z-t}+1}{x+\frac{1}{y+z}}$ =>((x+y)/(z-t)+1)/(x+1/(y+z))
- Se o puxéramos sen parénteses: Distinto de

  x + y/z – t + 1/x + 1/y + z=> $x+\frac{y}{z}-t+\frac{1}{x}+\frac{1}{y}+z$
- O nº de parénteses de apertura-peche debe coincidir. Se non coinciden, erro de compilación

Tipo (do resultado) dunha expresión

- Operación aritmética: dous operandos que poden ser de tipos distintos (p.ex., real e enteiro)
- Cando un operador actúa sobre datos de distintos tipos, o resultado é do tipo máis alto de ambos (*double* é máis alto que *real*, e *real* é máis alto que *integer*)
- Isto non impide a posíbel perda de información, dependendo dos seus valores.
- Exemplo: na división de enteiros, xa que o resultado será enteiro: perda de parte decimal, se existe: sexan *n=3,m=2* enteiros: *n/m* debería ser 1.5, pero como é enteiro, será 1
- Solución: declarar *n* e/ou *m* como reais; ou ben usa-la función *real(n)* para convertir *n* (ou *m*) en *real*

Tipos das expresións

- Sexan *x=1.2,y=1.0* reais, *n=3,m=2* enteiros. Na expresión:

$$x*y + n/m$$

- Erro común: pensar que na división *n/m* o resultado é real porque *x* e y son reais.
- Pero como *n* e *m* son enteiros, *n/m* é enteiro. Debería ser 1.5, pero vai ser 1 (enteiro): perda de información
- Hai que ver o tipo do resultado de cada operador individual (de dous operandos)

Sentenza de asignación

- A expresión da dereita pode ser unha constante, variábel, expresión aritmética ou función intrínseca
- Asigna o resultado da **expresión dereita** á **variábel esquerda**
- No lado esquerdo só pode haber unha variábel: non pode haber constante, expresión nin chamada a subprograma: erro de compilación
- Ambos lados poden ser de tipos distintos

Sentenza de asignación

- O valor da dereita convírtese ao tipo da esquerda: pode haber perda de información. Exemplo:

  n = x + 3.4 (*n* enteiro, *x* real)

A variábel da esquerda da asignación debe ter un tipo non **inferior** ao da expresión da dereita

integer < real < double

- Erro común: dúas asignacións consecutivas á mesma variábel. Ex:

  x = 3.4 – y

  x = z – y !pérdese o valor anterior de x

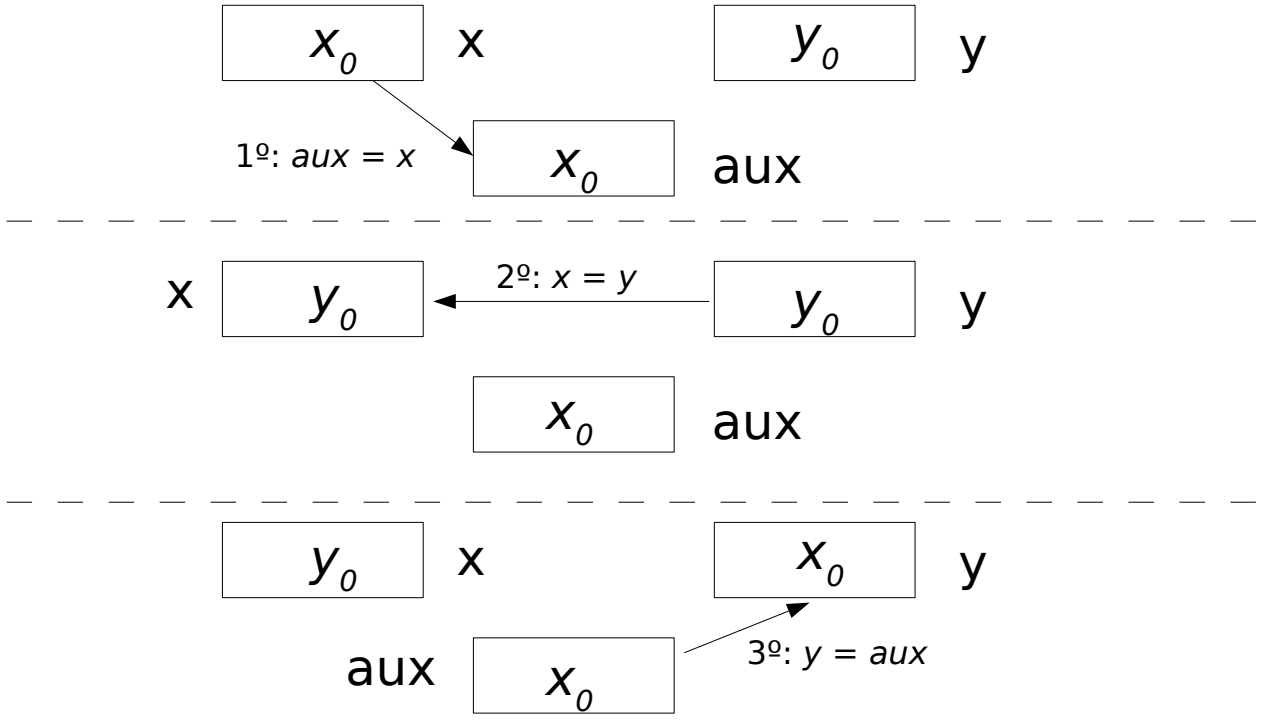
- Pódense encadear varias asignacións na mesma liña con “;”. Ex: *x = y + z; t = 2*x*

Exemplo: intercambio dos valores de dúas variábeis

- Temos dúas variábeis *x* e *y*: *x* ten o valor x_0, e *y* ten o valor y_0: queremos intercambiar os seus valores
- Distinguir entre o valor que se almacena nunha variábel e a variábel (que é o contedor do valor)
- Necesitamos unha variábel para almacenar temporalmente un dos dous valores: *aux*
- 1º paso: *aux* = *x* (almacena x_0 en *aux*)
- 2º paso: *x* = *y* (copia y_0 de *y* á variábel *x*)
- 3º paso: *y* = *aux* (copia x_0, que está almacenado na variábel *aux*, á variábel *y*)

Ollo: escribe na mesma variábel que salvou antes en *aux*

Intercambio de 2 variábeis



Intercambio de 2 variábeis

- Resumindo:

$aux = x$

$x = y$ ← Sempre escribe nunha variábel que se salvou na sentenza anterior a outra variábel

$y = aux$

- No intercambio, a variábel na dereita dunha sentenza de asignación debe aparecer na esquerda da seguinte sentenza de asignación
- Non pode estar a mesma variábel na esquerda de dúas sentenzas consecutivas de asignación

Algunhas funcións intrínsecas útiles de Fortran

- Valor absoluto: *abs(x)*. Argumento real/dobre.
- Raíz cadrada: *sqrt(x), csqrt(z)* para números complexos
- Resto da división enteira: *mod*(*m*,*n*) resto de *m*/*n*
- Trigonométricas: *sin*,*cos*,*tan*,*asin*,*acos*,*atan*
- Hiperbólicas: *sinh*,*cosh*,*tanh*,*asinh*,*acosh*,*atanh*
- Exponencial/logarítmica: *exp(x)*,*log(x)*
- Números aleatorios (0<*x*<1): *call random_number(x)*,*x=rand()*
- Arrai aleatorio: *real a*(3,3);*call random_number*(*a*)
- Truncamento a enteiro: *int*, *floor, ceiling, nint*
- Executar comando de Windows: *call system('comando')*. Ex: *call system('dir')*: mostra o contido do directorio actual

Funcións de sentenza

- Se queres definir unha función matemática, p.ex. *f(x,y)=x*y* que se poda calcular nunha única sentenza, podemos definir e chamar a unha **función de sentenza** (ou **de liña**) como a calquera función intrínseca
- Así non tes que poñer a expresión varias veces
- Hai que definila antes de ser chamada
- Os valores que se pasan como argumentos deben ter o mesmo tipo que o definido implícitamente polos argumentos da función de liña.

```
program exemplo
f(x,y)=x*y
print *,f(3.,2.5)
stop
end program exemplo
```

- Permite definir unha función como se fose intrínseca
- Só se pode chamar dende o programa principal no que se define.
- Verémolas de novo no tema de subprogramas

Programadoras da NASA no programa espacial de viaxe á lúa (1969)

Katherine Johnson: **matemática** e programadora da NASA
Primeira civil en recibir a **Medalla de Ouro** do Congreso USA

Mary Jackson: enseñeira da NASA

Película
“Figuras ocultas”

Dorothy Vaughan: programadora en **Fortran** do primeiro ordenador da NASA

Operadores relacionais (I)

- Definen as relacións de equivalencia (igualdade) e de orde numérica (con caracteres, a orde alfabética)
- Operadores: < <= > >= == /=
- Exemplos: x < 3, y >= z
- Operandos numéricos, resultado lóxico
- x < y: .*true*. se x < y, .*false*. en caso contrario
- x > y: .*true*. se x > y, .*false*. en caso contrario
- x == y: .*true*. se x e y son iguais, .*false*. en caso contrario
- *x /= y*: .*true*. se x e y son distintos, .*false*. en caso contrario

Operadores relacionais (II)

- Permiten definir condicións de igualdade / desigualdade ou maior/menor sobre números ou caracteres (orde alfabética)
- Úsanse en sentenzas de selección para crear distintas "rutas" de execución do programa, dependendo do cumprimento ou non dunha condición
- Sobre vectores e matrices, aplícanse por compoñentes:
  integer :: x(3)=[1,2,3]
  *print *,x>2 => F F T*

```
integer :: a(2,2)
forall(i=1:2,j=1:2) a(i,j)=i*i+j
print *,mod(a,2)==0
```

- Función *count*: nº de elementos dun vector/matriz que cumpren unha relación: *count(x>5)*, *count(x/=0)*, *count(mod(x,2)==1)*
- Caracteres: *character(10) :: s='ola'*

  s=='ola': T s>'pepe': F

```
integer :: a(2,2)
forall(i=1:2,j=1:2) a(i,j)=i*i+j
print *,count(mod(a,2)==1)
```

Operadores lóxicos (I)

- Combinan unha ou dúas condicións (definidas cos operadores relacionais) mediante relacións de:
  - **Conxunción**: *condición1* e tamén *condición2*
  - **Disxunción**: *condición1* ou *condición2*
  - **Negación**: non é certo *condición1*
  - **Equivalencia lóxica**: *condición1* e *condición2* son iguais (ou ben ambas se cumpren ou ben ningunha se cumpre)
  - **Non equivalencia lóxica**: *condición1* e *condición2* son distintas (se se cumpre unha non se cumpre a outra e ao revés)

Operadores lóxicos (II)

- Operadores:
  - Conxunción: .*and*.
  - Disxunción: .*or*.
  - Negación : .*not*.
  - Equivalencia lóxica (condicións iguais): .*eqv*.
  - Non equivalencia lóxica (condicións distintas): .*neqv*.

a	*.true.*	*.false.*	*.true.*	*.false.*
b	*.true.*	*.true.*	*.false.*	*.false.*
a.and.b	*.true.*	*.false.*	*.false.*	*.false.*
a.or.b	*.true.*	*.true.*	*.true.*	*.false.*
.not.a	*.false.*	*.true.*		
a.eqv.b	*.true.*	*.false.*	*.false.*	*.true.*
a.neqv.b	*.false.*	*.true.*	*.true.*	*.false.*

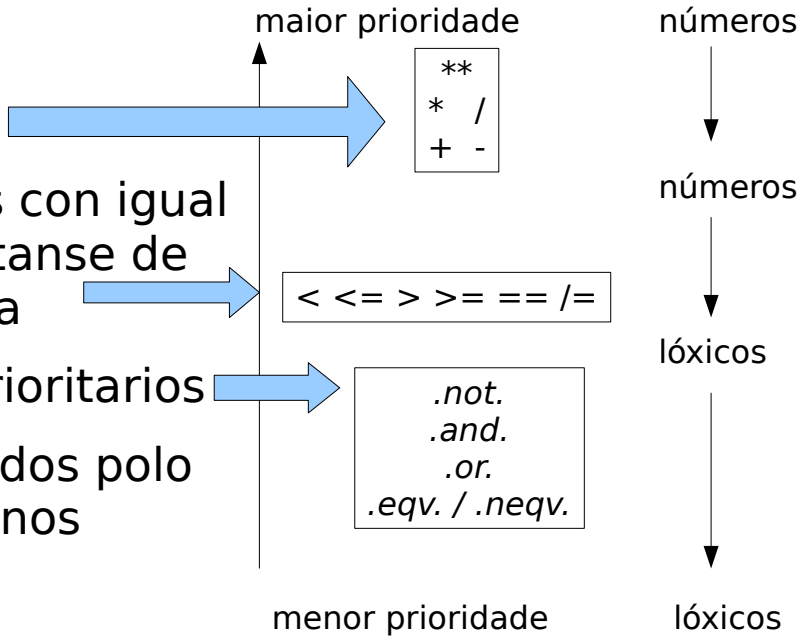
- Operandos lóxicos (resultado de operadores relacionais ou lóxicos). Resultado lóxico

Exemplos de operadores lóxicos

- AND: *x>5.and.x<=7*: x debe estar en (5,7] para que se cumpra a condición
- OR: *x==2.or.x==3*: x debe ser 2 ou 3 para que a condición sexa .true.
- NOT: *.not.(x>3)* é certa so se x<=3
- EQV: *mod(x,2)==0.eqv.mod(x,3)==0*: é certa se x é múltiplo de 2 e de 3, e tamén se x non é múltiplo de 2 nin de 3
- NEQV: *mod(x,2)==0.neqv.mod(x,3)==0*: é certa se x é múltiplo de 2 pero non de 3, e tamén se x non é múltiplo de 2 pero si de 3

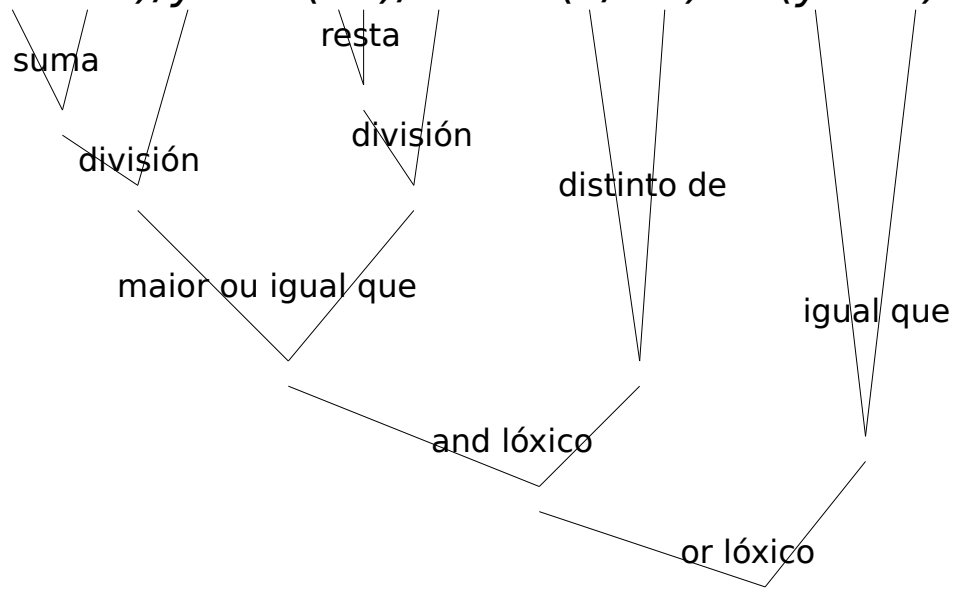
Precedencia de operadores

- Aritméticos: máis prioritarios
- Relacionais: todos con igual prioridade, execútanse de esquerda a dereita
- Lóxicos: menos prioritarios
- Operadores definidos polo usuario: os de menos prioridade



Exemplo de prioridades de execución

*(z+3.4)/y >= (z-t)/x.*and.*(x/=0).or.(y==z)*



Funcións lóxicas *all* e *any* para vectores/matrices

- *all(v/=0)*: *true.* se tódolos elementos do vector *v* son non nulos.
- *all(a>0)*: *true* se tódolos elementos da matriz *a* son positivos
- *all(a,1)*: executa o *all* por columnas (*a* debe ser *logical*)
- *all(a,2)*: executa o *all* por filas
- *any(a>5)*: *true* se *a* ten algún elemento maior que 5
- *any(mod(a,2)==0,1)*: *true* para columnas con elemento par.
- *any(a==3,2)*: *true* para as filas cun 3

```
integer :: a(2,3)=reshape([1,-1,3,1,5,-2],shape(a))
print *,all(a>0)
print *,all(a>0,1)
print *,any(a==3,2)
```

```
False
False  True  False
True  False
```

```
1  3  5
-1 1 -2
```

Edith Clarke (1883-1959)

- Graduada en matemáticas e astronomía (1908)
- Calculadora (1912) en American Telephone and Telegraph (ATT)
- Creadora dunha calculadora gráfica patentada en 1925
- Enxeñeira eléctrica estadounidense e profesora de matemáticas, física e enxeñaría eléctrica

Sentenzas de selección

- Programa: execución secuencial de instrucciónss (*print*, *read*, asignacións, ...)
- Sentenza de selección: executa bifurcacións na execución secuencial do programa
- Avalía unha expresión lóxica: se se cumpre, executa unhas sentenzas; se non, executa outras ou non fai nada
- Permite obter programas complicados, con múltiples rutas de execución dependendo de datos de entrada
- Moi importantes na programación: permiten controla-lo fluxo de execución do programa

if lóxico / bloque *if*

- ***if* lóxico**: avalía unha condición (expresión lóxica) e dependendo do seu valor (.*true*./.*false*.) executa ou non unha única sentenza

```
if(condición) sentenza
```

  - A condición vai entre parénteses. Só válido cunha única sentenza. Se non se cumpre, non fai nada.
- **Bloque *if***: igual, pero permite varias sentenzas

```
if(condición) then
      sentenzas
end if
```

```
if(x>=10.or.y/=-5) then
    x = x+y; print *, x
end if
```

  - Atención ao *then*: se falta, erro de compilación

Bloque *if/else*

- Avalía a condición: se esta se cumpre, executa un bloque de sentenzas; se non se cumpre a condición, executa outro bloque distinto

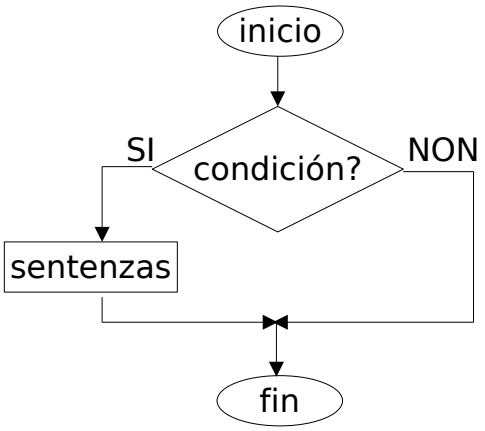
```
if(condición) then
    sentenzas1
else
    sentenzas2
end if
```

```
if(x==0.or.y<-1) then
    x=x+y;print *, x
else
    y = y – x; print *, y
end if
```

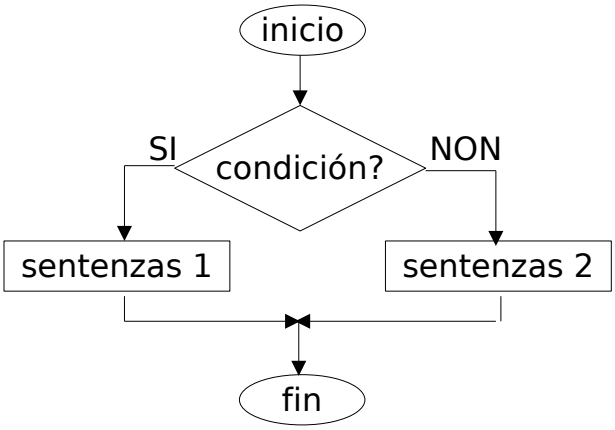
- As sentenzas de selección poden conter outras sentenzas de selección, formando unha estructura que pode ser moi sofisticada.

Diagramas de fluxo de *if* lóxico, bloque *if* e *if/else*

if lóxico e bloque *if*



if/else

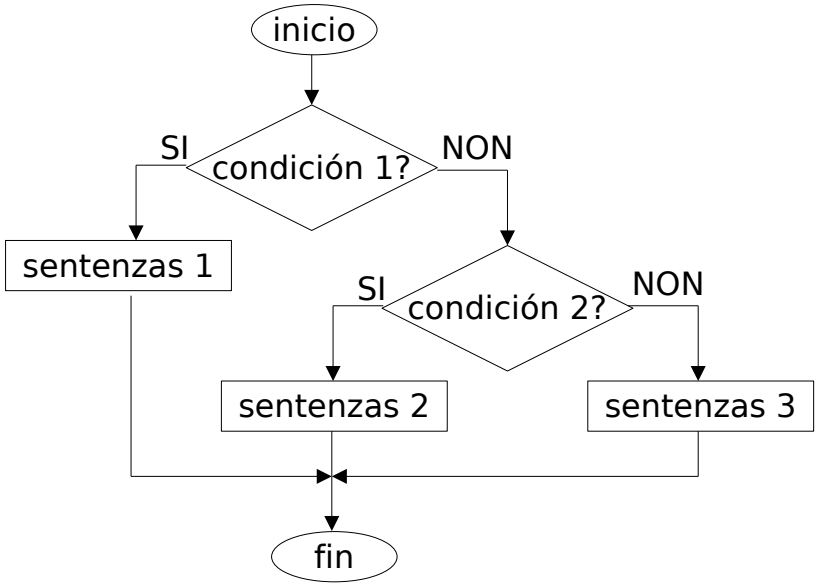


Bloque *if/else* múltiple

- Avalía *N* condicións: cada una ten un bloque de sentenzas, que se executa se se cumpre esa condición (entón xa non se avalía ningunha outra condición)
- A *i*-ésima condición so se avalía se non se cumpre a condicion (*i-1*)-ésima (nin as *i* condicións anteriores)
- Opcionalmente, pode haber un bloque *N+1* que se executa se ningunha condición se cumpre

```
if(condición1) then
    sentenzas1
else if(condición2) then
    sentenzas2
...
else if(condiciónN) then
    sentenzasN
else
    sentenzasN+1
endif
```

Diagrama de fluxo do bloque *if/else if*



Sentenza *select*

- Compara secuencialmente unha *expresión* **enteira** con *N* selectores: valor único *(0)*, conxunto de valores *(1,2,3)* ou rango de valores *(:1)*
- Se a expresión é igual a algún selector, execútase ese bloque e remátase
- So se executa un bloque
- Pode haber opcionalmente un bloque *default* que se executará se a expresión non se corresponde con ningún selector

```
select case (expresión)
   case (selector1)
      sentenzas1
   case (selector2)
      sentenzas2
   ...
   case (selectorN)
      sentenzasN
   case default
      sentenzasN+1

end select
```

Exemplos de *if/else* e *select*

- Exemplo de *if/else* múltiple: función a cachos:

$$f(x)=\begin{Bmatrix} -1 & x<0 \\ 0 & x=0 \\ 1 & x>0 \end{Bmatrix}$$

```
if(x < 0) then
    f = -1
else if(x == 0) then
    f = 0
else
    f = 1
end if
```

- Exemplo de *select*: código de operación: (*n* enteiro)

```
select case (n)
case (-1,0,1)
    print *, 'erro'
case (2:5)
    print *, 'correcto'
case default
    print *, 'apagado'
end select
```


Ángela Ruiz Robles (1895-1975)

- Mestra e creadora da primeira enciclopedia mecánica
- Libro electrónico fabricado en 1949 no Parque de Artillería do Ferrol
- Usaba bobinas automáticas e un sistema de botóns a presión conectados a unha serie de abecedarios que construían textos de temas
- Recibiu a Cruz da Orde de Afonso X en 1947 en recoñecemento á súa traxectoria profesional

Depuración con *gdb* (I)

- O *gdb* é un depurador: un programa que permite executar o programa paso a paso para así atopar erros de execución e lóxicos
- Para usar o gdb, compila o programa coa opción -g: *gfortran -g programa.f90* (isto engade información de depuración no executábel *a.exe*)
- Entra no *gdb* cargando o executábel: *gdb a.exe*
- Establece un *punto de ruptura* no programa, é decir, unha liña na que se rompe a execución do programa
- O programa execútase normalmente ata esa liña na que hai o punto de ruptura
- Para establecer un punto de ruptura na liña 45: *break 45* ou ou tecleando *b* 45 (*b*=abreviatura de *break*)

Depuración con *gdb* (II)

- Executa o programa no *gdb* co comando *run ou r.*
- O *gdb* executa o programa normalmente ata esa liña na que hai o punto de ruptura
- Cando a execución se para na liña 45, executas liña a liña co comando *next* ou *n*
- En todo momento podes ver o valor de calquera variábel co comando *print variable*
- Por exemplo, para ver a variábel *i*, executa *print i* ou *p i*
- Se a seguinte senteza é unha chamada a un subprograma (que veremos máis tarde) e queres entrar nel, executa *step* ou *s*. Se non, executa *next*
- Sae do *gdb: quit* ou *q*

Corrección de erros de execución

- Compila coa opción -g: *gfortran -g programa.f90*
- Carga o executábel no *gdb*: *gdb a.exe*
- Executa o programa no *gdb*: *run*
- Mira en qué liña do programa se produce o erro de execución co comando *where* ou *w*
- Selecciona o nivel situado en *programa.f90*: *frame 3* (se o nivel #3 é o de *programa.f90*)
- Inspecciona os valores das variábeis: *print i* ou *p i*. Isto permite ver en que punto toman valores incorrectos, adiviñar por que, e correxir o fallo.
- Executa paso a paso: *step*
- Sair do gdb: *quit*

Depuración visual con VSCode (I)

- Debes ter instaladas as extensións *Modern Fortran* e *cppdebug* (botón *Extensions*, último en barra vertical esquerda)
- Crea un directorio chamado *.vscode* no directorio fortran
- Nel, tes que poñer 3 arquivos:

1)extensions.json →

```
{
  "recommendations": [
    "fortran-lang.linter-gfortran"
  ]
}
```

Depuración en VSCode (II)

2) lauch.json:

3) tasks.json:

launch.json:

```
{
 "version": "0.2.0",
 "configurations": [
  {
   "name":          "debug",
   "type":          "cppdbg",
   "request":       "launch",
   "program":       "${workspaceFolder}/a.exe",
   "cwd":           "${workspaceFolder}",
   "MIMode":        "gdb",
   "preLaunchTask":  "compile",
  }
 ]
}
```

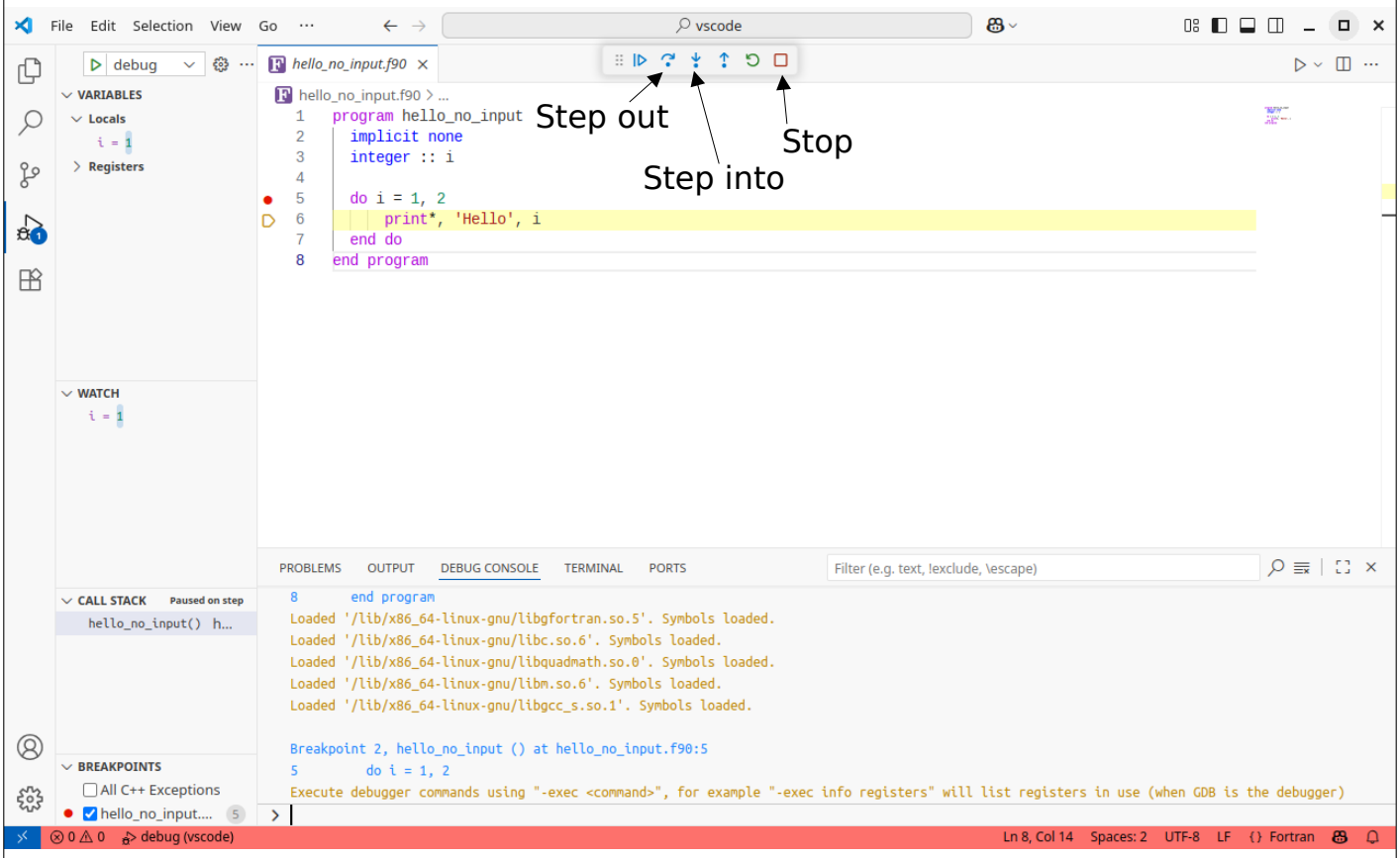
```
{
 "version": "2.0.0",
 "tasks": [
  {
   "label":   "compile",
   "type":    "shell",
   "command": "gfortran",
   "args":   ["-Wall", "-g", "programa.f90"],
   "group":   "build",
  },
 ]
}
```

Desafortunadamente, hai que cambiar o nome do programa no arquivo *tasks.json* para cada programa que fagas

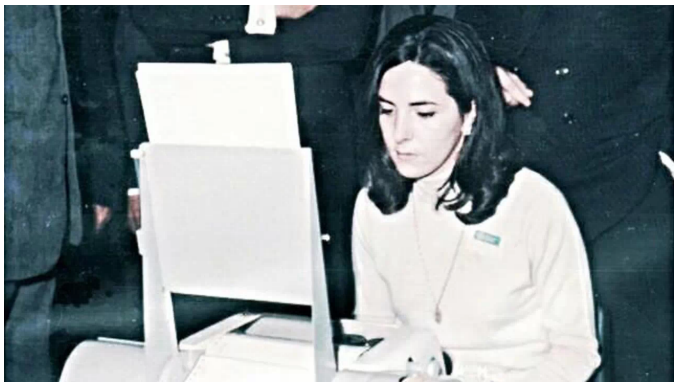
Depuración en VSCode (III)

- No VSCode, abre a carpeta *fortran* e o *programa.f90*
- Vai ao meu *Run*, *Start debugging*, ou pulsa *F5*
- Así executa o programa e podes ver tódalas variábeis poñendo o rato sobre o seu nome (ver volcado de pantalla en páxina seguinte)
- Tamén podes engadir no apartado *Watch* (panel esquerdo medio) as variábeis que queres que te mostre
- Móstrase na parte superior unha barra de botóns. Pulsando no botón *Step Over* (*F10*) podes executar paso a paso
- Se queres entrar nun subprograma, pulsa no botón *Step Into* (*F11*)
- Para rematar a execución, botón *Stop* (pulsa *shift+F5*)

Depuración en VSCode (IV)



Ana María Prieto López (1942-2018)



- Primeira programadora en España, de Santiago
- Dende os 21 anos traballou en IBM e foi a primeira programadora en España da empresa informática Bull
- Programou en COBOL e código máquina
- Traballou dende 1969 na Caixa de Aforros de Santiago

Sentenzas de iteración

- Repiten a execución dun bloque de sentenzas ...
  - Un certo nº de veces predeterminado (*iteración definida*)
  - Mentres se cumpra unha condición (ou ata que deixa de cumprirse): non se sabe de antemán o nº de repeticións (*iteración indefinida*)
- Grande importancia en programación
- Sentenza *do*: ambos tipos de iteración

Sentenza *do* definida

```
do var=ini,fin,paso
    sentenzas
end do
```

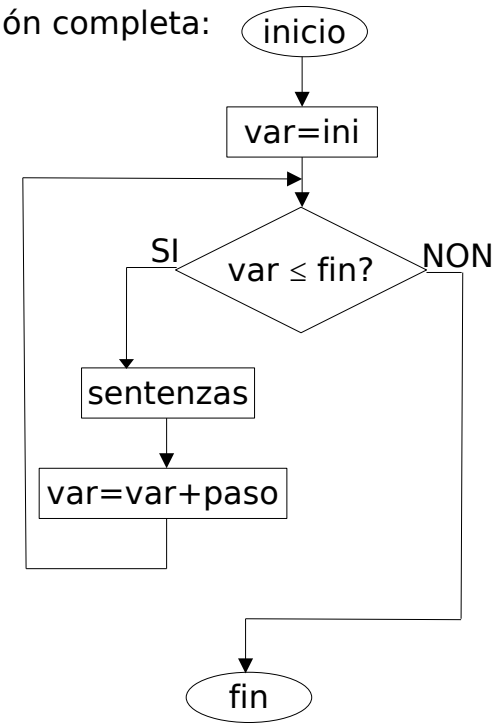
```
do i=1,10
    print *, v(i)
end do
```

imprime os elementos dun vector

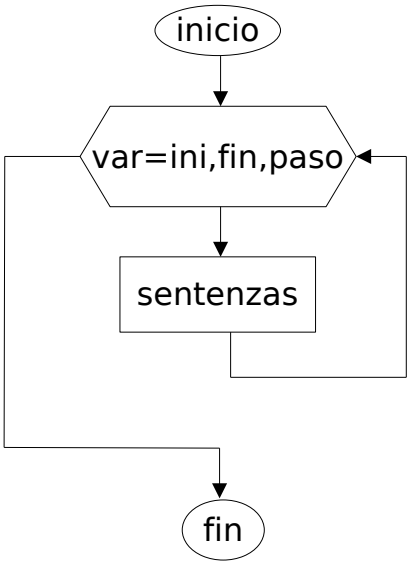
- Antes de executar nada, asigna *ini* a *var*
- Repite as sentenzas mentres *var<=fin*
- Ao rematar unha iteración (repetición completa das sentenzas), executa *var=var+paso*
- O valor *paso*: opcional, por defecto vale 1
- Se modificamos *var* dentro das sentenzas: erro de compilación
- O parámetro *ini* debe ser enteiro, non real

Diagrama de fluxo *do* definido

Versión completa:



Versión simplificada



Sentenza *do* definida

- Nº de iteracións do bucle *do*:

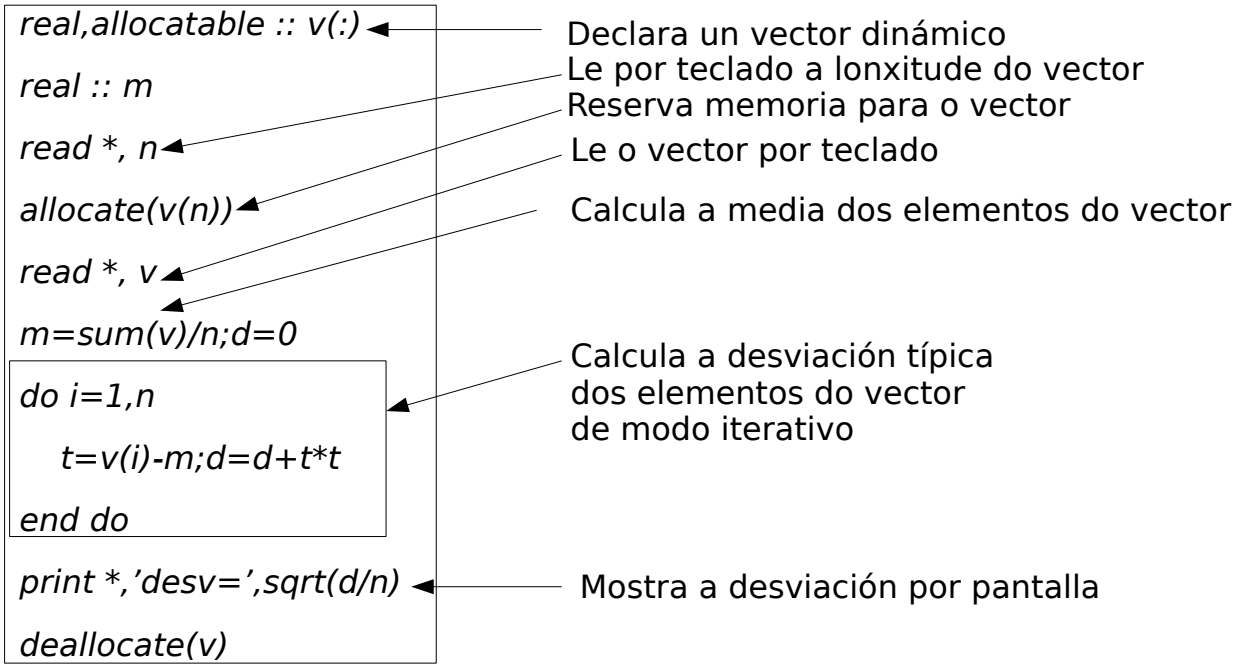
$$N = max\left\{0, \left\lfloor \frac{fin - ini + paso}{paso} \right\rfloor\right\}$$

- Non pode ser *paso = 0*: erro de compilación
- Se *paso < 0*, entón debe ser *fin < ini*, e a condición de continuidade do bucle é que *var >= fin*. Ex:

```
do i = 10, 1, -1
      print *, v(i)
end do
```

- Un bucles *do* pode estar dentro doutros bloques *do* ou *if/else*, ou conter bloques *if/else*.

Exemplo 1: cálculo de desviación típica dun vector



Exemplo 2: búsqueda de elementos comúns a dús vectores

- So con bucles *do*:

```
integer,parameter :: n=10
integer :: x(n),y(n),z(n)
k=0
do i=1,n
    u=x(i)
    do j=1,n
        if(u==y(j)) then
            k=k+1;z(k)=u;exit
        end if
    end do
end do
print *,z(1:k)
```

- Vectorizado:

```
integer,parameter :: n=10
integer :: x(n),y(n),z(n)
k=0
do i=1,n
    u=x(i)
    if(any(u==y)) then
        k=k+1;z(k)=u
    end if
end do
print *,z(1:k)
```

Exemplo 3: acumulador, vectorización e incremental

- Queremos calcular:

$$y_i = \sum_{j=1}^{i} x_j, i = 1, \ldots, n$$

- **Versión 1**: acumulador

```
do i=1,n
    s=0
    do j=1,i
        s=s+x(j)
    end do
    y(i)=s
end do
```

Acumulador e dous bucles, fácil de entender, moitas iteracións e operacións

- **Versión 2**: vectorizada

```
s=0
do i=1,n
    y(i)=sum(x(1:i))
end do
```

Máis curto, fai sumas innecesarias

- **Versión 3**: incremental

```
s=0
do i=1,n
    s=s+x(i)
    y(i)=s
end do
```

Aforra moitas operacións (sumas)

Erro de segmentación

- Usualmente a xestión de vectores e matrices realízase mediante bucles *do* definidos ou indefinidos. Exemplo:

```
integer,dimension(3) :: v
do i = 1, 3
    v(i) = 2*i + 4
end do
```

- É posíbel que nos saiamos dos límites do vector, p.ex. se *i* chega a valer *10*
- En tal caso: *erro de segmentación*: é un exemplo de erro de execución (ver metodoloxía da programación), provoca o remate anticipado do programa.

Detección de erros de segmentación

- Compila coa opción *-fcheck=all*
- Cando se produce o erro de segmentación durante a execución, indica en que liña se produce o erro
- Exemplo:
- Compilación:
  f95 -fcheck=all proba.f90 -o proba

```
program erro_segmentacion
real :: x(10)
do i=0,20
    x(i)=i
    print *,x(i)
end do
stop
end program erro_segmentacion
```

Erro de segmentación por chegar a ser i=0 ou i>10

```
At line 5 of file proba.f90
Fortran runtime error: Index '0' of dimension 1 of array
'x' below lower bound of 1
```


Depuración con *gdb*

- Se a causa do erro é moi evidente, pódese correxir o programa directamente
- Se non é así, hai que usar o depurador *gdb* para atopar a causa dun erro de segmentación e correxir o erro, tal como se explicou no apartado *Depuración*:

1) Compila con *-g*: *gfortran -g programa.f90*
2) Entra no gdb: *gdb a.exe*
3) Executa o programa: *run*
4) Mira en que liña se produce o erro: *where*
5) Imprime as variábeis: *print i*
6) Busca o punto onde o programa se sae do rango de índices dalgún vector ou matriz
7) Sae do *gdb*: *quit*

Sentenza *exit*

- Provoca o remate inmediato das iteracións
- Sempre vai dentro dunha sentenza de selección (asociada a unha condición)
- Se ésta se cumpre, execútase o *exit* e remata o bucle (se hai varios anidados, o bucle máis interno)
- A condición debe ter variábeis, e algunha delas debe cambiar o seu valor dentro do bucle para que éste remate

```
do i = 1, 10
   print *, “introduce un número (-1 para rematar):”
   read *, n
   if(-1 == n) exit  !remata a iteración neste intre
   suma = suma + n
end do
```


Bucle indefinido *do/exit*

```
do
   sentenzas
   if(condición) exit
end do
```

Executa as sentenzas antes de avaliar a condición (execútanse sempre polo menos unha vez)

```
x2=0.9
do
   x1=x2;x2=x1**2
   if(abs(x1-x2)<0.01) exit
   print *,x1,x2
end do
```

```
do
   if(condición) exit
   sentenzas
end do
```

Executa as sentenzas logo de avaliar a condición (poden non executarse nunca)

```
x=1
do
   if(x>10) exit
   x=x+0.1
   print *,x,x+1
end do
```

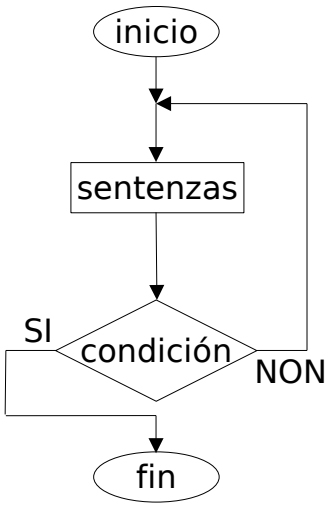
```
do
   sentenzas1
   if(condición) exit
   sentenzas2
end do
```

Remata cando se cumpre a condición: hai sentenzas antes e despois do *exit*: é máis xeral

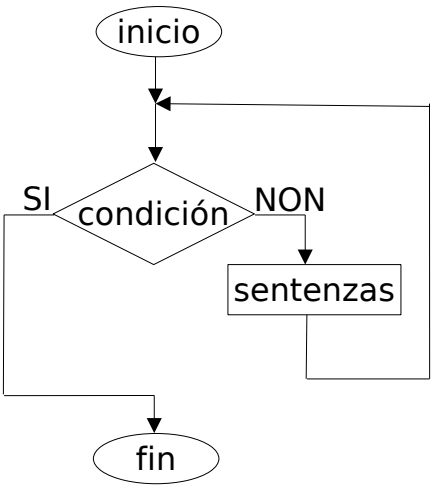
O bucle *do/exit* executa as sentenzas ata que se cumpre a condición, que polo tanto é **condición de remate**.

Diagramas de fluxo *do/exit*

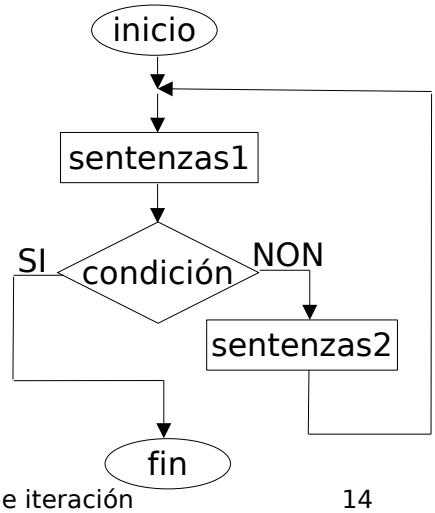
```
do
   sentenzas
   if(condición) exit
end do
```




```
do
   if(condición) exit
   sentenzas
end do
```




```
do
   sentenzas1
   if(condición) exit
   sentenzas2
end do
```



Bucle indefinido *do/while*

- Executa as sentenzas mentres se cumpre a condición.
- A **condición é de continuidade**, non de remate.
- A condición avalíase antes de cada iteración.
- É menos flexíbel que o *do/exit*, porque a condición non se pode avaliar en calquera parte do bucle.
- Outro inconvinte: debes inicializar as variábeis da condición antes do *while*
- Tamén pode levar un *if/exit* dentro do bucle.

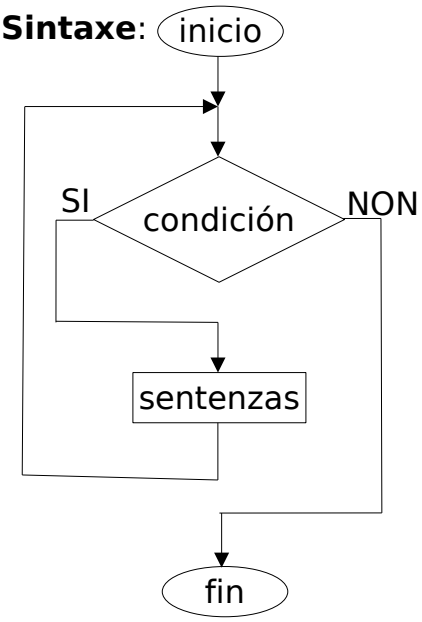
```
do while(condición)
   sentenzas
end do
```

```
s=0
do while(s<10)
   read *,x
   s=s+x
end do
```

```
s=0
do while(s<10)
   read *,x
   if(x==-1) exit
   s=s+x
end do
```

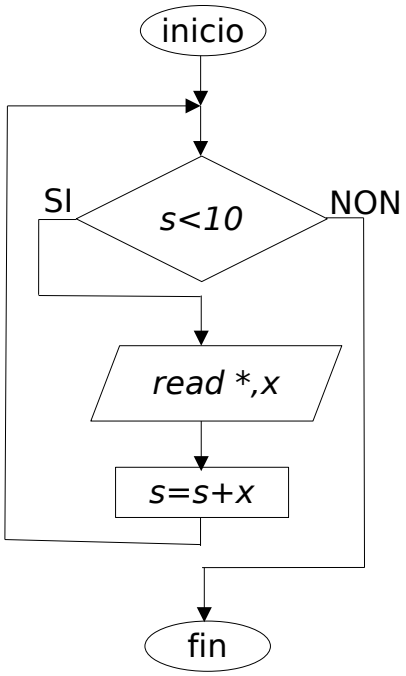
Diagrama de fluxo *do/while*

Sintaxe:



Exemplo:

```
do while(s<10)
    read *,x
    s=s+x
end do
```



Bucle *do* indefinido

- O nº de iteracións depende do nº de veces que se non se cumpre a condición de remate

```
do
    print *, "introduce un nº (-1 para rematar)"
    read *, n
    if(n == -1) exit
end do
```

- Non se deben usar == nas condicións os erros en punto flotante poden evitar a igualdade: execución infinita. P. ex: ~~*if(x==y) exit*~~; *if(abs(x-y)<1E-5) exit*
- Bucle *do* infinito:

```
do
    sentenzas
end do
```

```
do while(.true.)
    sentenzas
end do
```

O programa debería rematarse dende fóra, p.ex. con CTRL+C

Bucle *do* indefinido

- **Exemplo**: aproximación dunha suma (serie) de infinitos sumandos: sumar só os sumandos > 1e-5

$$\sum_{n=1}^{\infty}\frac{1}{n^2}\simeq\sum_{n\in A}\frac{1}{n^2},A=\left\{n\in\mathbb{N}:\frac{1}{n^2}>10^{-5}\right\}$$

```
suma=0;n=1
do
   sumando=1./n**2
   if(sumando<1e-5) exit
   suma=suma+sumando
   n=n+1
end do
```

Debe ser real ou dará resultado 0

Versión con *do/exit*

Versión con *do-while*

```
suma=0;sumando=1;n=1
do while(sumando>1e-5)
   sumando=1./n**2
   suma=suma+sumando
   n=n+1
end do
```


Relación entre bucles definidos e indefinidos

- Un **bucle definido** pódese construir cun **bucle indefinido**, no cal a inicialización, actualización e teste de continuidade ou remate sobre a variábel fanse manualmente:

```
var = ini
do
     sentenzas
     var = var + paso
     if(var > fin) exit
end do
```

teste de remate

```
do var = ini, fin, paso
     sentenzas
end do
```

teste de continuidade

```
var = ini
do while(var<=fin)
     sentenzas
     var = var + paso
end do
```

- As versións indefinidas son útiles se queremos que a variábel a actualizar sexa real, xa que o bucle *do* definido so permite variábeis enteiras.

Exemplo de bucle indefinido: percorrido cíclico dun vector

- Cíclico: cando chegas ao final do vector, volves ao principio
- O *do* definido non vale: usar *do-exit* ou *do-while*
- Cando chegas ao final do vector, voltas ao principio
- Tamén pode ser decrecendo o índice

```
integer :: x(5)=[1,2,3,4,5],s=0
i=1
do while(s<10)
   print *,'i=',i,' x=',x(i),' s=',s
   s=s+x(i)
   i=i+1
   if(i>10) i=1
end do
print *,'fin: s=',s
```


Bucle *do* híbrido

- É unha mestura de bucle definido e indefinido:

```
do var = ini, fin, paso
   sentenzas
   if(condición) exit
   sentenzas
end do
```

```
suma = 0
do n=1,100
   sumando = 1./n**2
   if(sumando < 1e-5) exit
   suma = suma + sumando
end do
```

```
sumando=1;suma = 0;n=1
do while(sumando>1e-5.and.n<100)
   sumando = 1./n**2
   suma = suma + sumando
   n=n+1
end do
```

- Executa as sentenzas un nº máximo de veces, pero pode rematar antes no *exit* se se cumpre a condición
- Evita unha execución infinita se a condición non está ben deseñada.
- Evita saírse dun vector ou matriz:

```
integer :: x(10)
s=0;i=1
do while(s<1.and.i<=10)
   s=s+x(i);i=i+1
end do
```

Sentenza *cycle*

- Provoca que se salte as sentenzas que restan por executar na iteración actual. Sempre vai asociado a unha condición mediante unha sentenza de selección
- A execución salta ao comezo da seguinte iteración (con bucles *do*'s aniñados, salta á seguinte iteración do bucle *do* máis interno)

```
do i = 1, 10
      print *, “introduce un número natural:”
      read *, n
      if(n <= 0) then
         print *, “dixen que fora natural!!”
         cycle    !salta ao print na iteración i+1
      end if
      suma = suma + n
   end do
```


Nomeamento de bucles *do*

- Como pode haber bucles aniñados, pode resultar útil nomear os bucles:

```
nome1: do i = 1, 10,2
   nome2: do j = 1, 50, 4
      sentenzas
      if(condición) exit nome1
   end do nome2
end do nome1
```

- Permite, p. ex., rematar con *exit* o bucle *do* máis externo (se se desexa)
- Con *cycle*, permite saltar á seguinte iteración dun bucle *do* que non é o máis interno: *cycle nome1*

Sentenza *where/elsewhere*

- Executa sentenzas de asignación para tódolos elementos dun vector/matriz nos que se cumpre (ou non) unha condición.

```
where condición-arrai
    asignacións_arrai_1
elsewhere
    asignacións_arrai_2
end where
```

```
real,dimension(10) :: v,w
where(v>w)
    v=v+w
elsewhere
    v=v-w
end where
```

```
real :: a(3,3),b(3,3)
where(a==b)
        a=b+3
end where
```

Permite vectorizar expresións, evitando bucles *do* e executando operacións para moitos elementos á vez.

Sentenza *forall*

- **Sentenza *forall***: executa iterativamente sentenzas de asignación sobre vector ou matriz, similar a sentenza *where*

```
forall (var=ini:paso:fin,condición) asignacións
forall (var1=ini1:paso1:fin1,var2=ini2:fin2,condición)
    asignacións
end forall
```

```
forall (i=1:n,j=1:m,y(i,j)/=0.and.i/=j)
   x(i,j)=1./y(i,j)
end forall
```

- **Bucle *do* implícito**: *print *, (v(i), w(i), i = 1, 10)*
  - Imprime un vector nunha única liña
  - Imprime unha matriz, cada fila nunha liña da terminal

```
do i = 1, 3
   print *, (m(i, j), j = 1, 3)
end do
```

Sentenza *pack* con vectores

- Retorna un vector *k* cos índices, ou os valores, dos elementos do vector que cumpren unha condición. Se ningún a cumpre, vector baleiro (*size(k)*=0): similar á función *find* de Octave
- *pack(x,condición):* valores que cumpren a condición
- *pack([(i,i=1,n)],condición)*: índices dos elementos

  n=nº elementos do vector ou matriz que se testea

  condición: expresión lóxica que ten un vector *x*: p.ex. *x>=2*

Retorna valores que cumpren a condición

```
integer :: x(4)=[2,3,5,2]
print *,'x>2: ',pack(x,x>2)
```

```
integer :: x(4)=[2,3,5,2]
integer,allocatable :: z(:)
z=pack(x,mod(x,2)==0)
print *,'x par: ',z
```

```
integer :: x(4)=[2,3,5,2]
integer,allocatable :: k(:)
k=pack([(j,j=1,4)],x==2)
print *,k  ! k(1)=1, k(2)=4
```

Retorna os índices dos elementos que cumpren a condición

Se queres almacenar os valores ou índices nun vector, hai que declaralo como *allocatable* sen reservar memoria

Sentenza *pack* con matrices

Retorna os valores da matriz que cumpren a condición

```
integer :: a(2,3)=reshape((/7,2,1,8,10,3/),shape(a))
integer,allocatable :: z(:)
print *,'a>2: ',pack(a,a>2)
z=pack(a,mod(x,2)==0)   ! valores pares
```

Retorna os índices dos elementos que cumpren a condición

```
integer :: a(2,3)=reshape((/7,2,1,8,10,3/),shape(a))
integer :: i(2,3)=reshape((/1,2,3,4,5,6/),shape(i))
integer,allocatable :: k(:)
k=pack(i,mod(a,2)==0)  ! k=(2,4,5)
k=pack(i,a>2)  ! k=(1,4,5,6)
k=pack(i,a==7)  ! k=1
```

Uso de *pack* para vectorizar expresións

- Eliminar elementos dun vector:

```
integer,allocatable :: x(:),y(:),j(:),k(:)
x=[(i**2,i=1,10)]
y=pack(x,x>=3.and.x<=5)  ! baseándose no valor de x(i)
j=[(i,i=1,10)]
k=pack(j,j/=2)
x=x(k) ! baseándose no índice: borra x(2)
```

- Eliminar filas e columnas dunha matriz

```
integer :: a(3,3)=reshape((/1,2,3,4,5,6,7,8,9/),shape(a))
integer,allocatable :: b(:,:),k(:),m(:)
k=[(i,i=1,3)]
m=pack(k,k/=2)  ! borra o elemento 2 de vector k
b=a(m,m)    ! colle filas 1,3 e columnas 1,3 de a
```


Conversión entre vectores e matrices

Función *reshape(x,forma)*: o argumento *forma* pode ser *(/n/)* (para convertir a vector de lonxitude *n*), ou (/*nf*,*nc*/), para convertir a unha matriz *nf*x*nc*.

- Conversión de vector a matriz:
  a=reshape(v,(/nf,nc/))
- Conversión de matriz a vector:
  v=reshape(a,(/n/))

```
real :: v(4)=(/1,2,3,4/)
real,dimension(2,2) :: a
a=reshape(v,(/2,2/))
```

Convirte de vector a matriz por columnas

```
real :: v(4)
real,dimension(2,2) :: a=reshape((/1,2,3,4/),shape(a))
v=reshape(a,(/4/))
print *,'v=',v
```

Función *shape(v), shape(a)*: retorna un vector coas dimensións do vector/matriz.

Programa de conversión de vector a matriz

```
program vector_matriz
real,allocatable :: x(:)
real,allocatable :: a(:,:)
real,parameter :: rand_max=2147483647
n=5;m=floor(sqrt(real(n)));k=1
allocate(x(n),a(m,m))
do i=1,n
      x(i)=floor(10.*irand()/rand_max)       !   nº enteiro aleatorio entre 0 e 10
end do
do i=1,m  ! conversión por filas
      do j=1,m
            a(i,j)=x(k);k=k+1            ! para convertir por columnas: a(j,i)=x(k)
      end do
end do
print *,'x=',x                ! imprime vector
print *,'a='                  ! imprime matriz
do i=1,m
      print *,(a(i,j),j=1,m)
end do
deallocate(x,a)
stop
end program vector_matriz
```


Programa de conversión de matriz a vector

```
program matriz_vector
real,allocatable :: x(:)
real,allocatable :: a(:,:)
n=3;m=n**2;k=1
allocate(x(m),a(n,n))
do i=1,n
      do j=1,n
            a(i,j)=i**2*j+i     !   inicializa a(i,j)
      end do
end do
do i=1,n  ! conversión por filas
      do j=1,n
            x(k)=a(i,j);k=k+1            ! para convertir por columnas: x(k)=a(j,i)
      end do
end do
print *,'a='                  ! imprime matriz
do i=1,n
      print *,(a(i,j),j=1,n)
end do
print *,'x=',x                ! imprime vector
deallocate(x,a)
stop
end program matriz_vector
```

Programadoras do ENIAC

- Betty Snyder Holberton (1917-2001)
- Betty Jean Jennings Bartik (1924-2011)
- Ruth Lichterman Teitelbaum (1924-1986)
- Kathleen McNulty (1921-2006)
- Frances Bilas Spence (1922-2012)
- Marlyn Wescoff Meltzer (1922-2008).

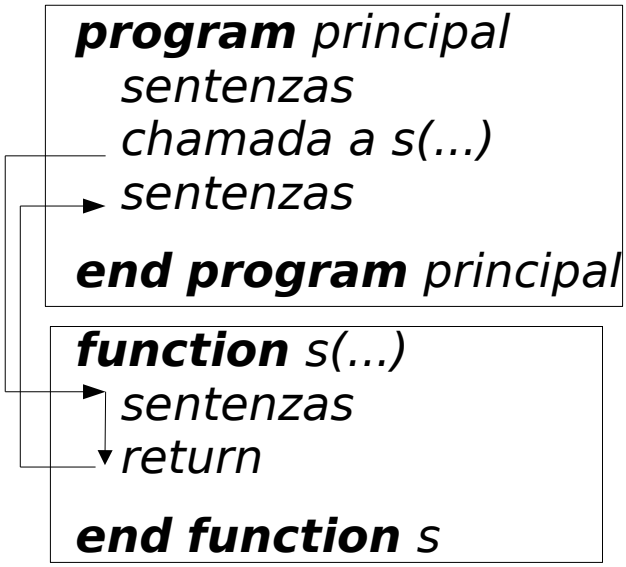
- ENIAC: Electronic Numerical Integrator And Computer
- Primeiro ordenador de propósito xeral (1946-1955)
- Elas desenvolveron as bases da programación de ordenadores

Subprogramas (I)

- Subprograma: conxunto de sentenzas que realizan unha tarefa clara, cunhas entradas (datos) e saídas (resultados) ben definidas. Exemplos:
  - subprograma que recibe un vector e calcula a súa media aritmética
  - subprograma que recibe unha matriz e calcula o seu determinante
- Hai dúas cousas:
  - Chamada ao subprograma dende o programa principal (*program*)
  - Corpo do subprograma: sentenzas do mesmo
- Argumentos: entradas e saídas do subprograma
- Poden estar no mesmo arquivo *.f90* ou en arquivos distintos

Subprogramas (II)

- O corpo do subprograma debe estar fóra do programa principal
- A chamada ao subprograma está no programa principal
- O programa principal (chamador) non pode acceder ás variábeis do subprograma
- O subprograma (chamado) non pode acceder ás variábeis do programa principal
- A única comunicación entre o programa principal e o subprograma é mediante os argumentos e os valores retornados



Sentenza *return* (opcional): retorna ao programa chamador (pode haber varias *return* no mesmo subprograma)

Tipos de subprogramas

Principais

- **Subrutina**: non retorna ningún valor, so ten argumentos *in-out-inout*
- **Función externa**: retorna un único valor (*integer*, *real*, *complex*, *double*, *character*, vector ou matriz), tamén ten argumentos *in-out-inout*

- **Subprograma interno** (función ou subrutina): en programa principal, logo de *contains*. So se pode chamar dende o subprograma onde se define
- **Función de sentenza**: ten unha única sentenza, so se pode chamar dende o subprograma onde se define.

Cando usar funcións e cando subrutinas?

- **Función**: cando o subprograma so ten que calcular un único resultado (enteiro, real, complexo, lóxico ou carácter). A función retorna este resultado, almacenado na variábel co nome da función ou na variábel indicada no *result(...)*.
- **Subrutina**: cando o subprograma ten que calcular máis dun resultado, ou un único resultado pero éste é vector ou matriz. A subrutina debe ter algún argumento de saída ou entrada/saída (ver páxina seguinte) no que almacenar os resultados.
- Cando un subprograma ten que retornar un vector ou matriz, pódese empregar unha subrutina ou unha función, pero:

  a) A subrutina é máis fácil: o vector/matriz será un argumento *out* ou *inout*. Neste curso usaremos unha subrutina.

  b) A función é máis difícil: debe haber unha *interface* indicando que retorna un vector/matriz

Argumentos (I)

- Tipo dos argumentos: *integer*, *real,* vector, matriz...
  - *intent(in)*: só lectura, o subprograma non pode modificalo (argumento de entrada): hai que inicializalos antes da chamada ao subprograma
  - *intent(out)*: non se pode ler (o chamador non lle dou ningún valor), só escribir nel (argumento de saída)
    - Non se pode inicializar antes da chamada
    - Logo de chamar ao subprograma hai que usa-lo seu valor
  - *intent(inout)*: o subprograma pode ler o seu valor e tamén modificalo (argumento de entrada/saída)

Argumentos (II)

- Os argumentos poden ser constantes, variábeis ou expresións
- Cando se pasa un argumento constante, variábel, ou expresión, hai que ter en conta o que espera o subprograma (*intent in*, *out* ou *inout*):
  - Se espera un argumento *out* ou *inout*, na chamada hai que pasarlle unha **variábel**: non constante ou expresión (daría un erro de compilación), porque vai ser modificado
  - Se o subprograma espera un argumento *in*, pódeselle pasar unha **constante, variábel ou expresión** (porque non vai modificarse)
- Os argumentos deben coincidir en número, tipo e orde na chamada ao subprograma e no corpo do subprograma, pero poden ter nomes distintos nos dous sitios
- Os argumentos *intent*(*in*) son constantes dentro do subprograma, e poden ser usados como dimensión de arrais estáticos.

Subrutina

- Non retorna ningún valor ao chamador: as entradas e saídas son somentes a través dos argumentos. Axeitadas cando hai múltiples saídas

```
subroutine nome(arg1, ..., argN)
  tipo1, intent(...) :: arg1
  tipoN, intent(...) :: argN
  declaracións
  sentenzas executábeis

end subroutine nome
```

Variábeis locais do subprograma

- Chamada á subrutina: ***call*** *nome(arg1, ..., argN)*

Función externa

- Retorna un único valor, do *tipo* indicado:

```
tipo function nome(arg1, ..., argN) result(var)
tipo1, intent(...) :: arg1
tipoN, intent(...) :: argN
declaracións
sentenzas executábeis
var=expresión    !valor retornado: var
end function nome
```

Variábeis locais do subprograma

- A función chámase dende unha sentenza de asignación:
  tipo :: nome
  var=nome(arg1, ..., argN)
- Se non leva *tipo*, retorna un valor do tipo implícito de *nome*
- Se falta ***result***(*var*), a función retorna a variábel *nome*

Subprogramas e programa principal en arquivos distintos

- Os subprogramas e programa principal poden ir no mesmo arquivo *.f90*, ou en arquivos distintos, normalmente un arquivo distinto para cada subprograma
- Podes compilar co comando: *gfortran *.f90 -o executabel*
- Tamén podes compilar cada arquivo coa opción *-c*:

gfortran -c principal.f90
gfortran -c subprograma1.f90
...
gfortran *-c subprogramaN.f90*

- Para cada arquivo *.f90* crea un arquivo *.o* que non se executa
- Para crear o executábel:

*gfortran *.o -o executabel*

Paso de vectores e matrices como argumentos

- Na chamada ao subprograma se lle pasa o nome do vector ou matriz e a súa lonxitude ou número de filas e columnas: tamano explícito (*explicit-shape arrays*)
- Dentro do subprograma, a lonxitude ou nº de filas/columnas debe ser un argumento enteiro *intent(in)*.

```
program proba
integer :: x(10)
call sub(x,10)
end program proba
!-------------------------
subroutine sub(x,n)
integer,intent(out) :: x(n)
integer,intent(in) :: n
do i=1,n
      x(i)=i**2+i-1
end do
end subroutine sub
```

```
program proba
integer,allocatable :: a(:,:)
allocate(a(2,3))
y=fun(a,2,3)
end program proba
!-------------------------
function fun(a,n,m)
integer,intent(in) :: a(n,m)
integer,intent(in) :: n,m
do i=1,n
      do j=1,m
            a(i,j)=i**2+j-1
      end do
end do
fun=a(1,n)*a(n,m)
end function fun
```


Exemplo de **subrutina**: ordeamento dun vector de números:

```
subroutine ordea_seleccion(x,n)
real, intent(inout) :: x(n)
integer,intent(in) :: n
do i = 1, n
   aux = x(i); k = i
   do j = i + 1, n
      if(x(j) < aux) then
         aux = x(j); k = j
      end if
   end do
   x(k) = x(i); x(i) = aux
end do
return
end subroutine ordea_seleccion
```

vector: argumento *inout* xa que se os seus elementos lense e tamén se modifican

```
real :: x(5) = (/5,2,4,3,1/)
call ordea_seleccion(x,5)
print *, “ordeado=”, x
```

chamada ao subprograma

Exemplo de **función externa** sen cambiar o tipo do valor devolto

```
program proba
real :: x(5)=[1,2,3,4,5]
t=f(x,5)
print *,t
end program proba
```

Programa principal

Chamada á función

```
 function f(x,n)
real,intent(in) :: x(n)
integer,intent(in) :: n
f=0
do i = 1, n
     f=f+i*x(i)
end do
end function f
```

Subprograma

$$f = \sum_{i=1}^{n} i\, x_i$$

o valor calculado debe retornarse almacenándoo na variábel co mesmo nome ca función

Exemplo de **función externa** cambiando o tipo e nome do valor devolto

Cambiando so o tipo do valor devolto

```
program proba
real :: x(5)=[1,2,3,4,5]
integer :: f  ! valor devolto enteiro
n=f(x,5)
print *,n
end program proba

integer function f(x,n)
real,intent(in) :: x(n)
integer,intent(in) :: n
f=0
do i = 1, n
     f=f+i*x(i)
end do
end function f
```

$$f = \sum_{i=1}^{n} i\, x_i$$

Cambiando o tipo e nome do valor devolto

```
program proba
real :: x(5)=[1,2,3,4,5]
integer :: f  ! valor devolto
enteiro
n=f(x,5)
print *,n
end program proba

integer function f(x,n) result(y)
real,intent(in) :: x(n)
integer,intent(in) :: n
y=sum([(i,i=1,n)]*x) !vectorizado
end function f
```

Subprograma que ten que calcular un vector ou matriz

- Pódese facer cunha subrutina ou cunha función, pero é máis fácil cunha subrutina, porque coa función necesitas unha **interface**
- Polo tanto, usa unha **subrutina** cun argumento *out* (ou *inout*) para o vector ou matriz

```
program exemplo_vector
integer,allocatable :: x(:)
read *,n
allocate(x(n))
call sub(x,n)
print *,'x=',x
deallocate(x)
end program exemplo_vector
!-----------------------------------------
subroutine sub(x,n)
integer,intent(out) :: x(n)
integer,intent(in) :: n
do i=1,n
      x(i)=i**2
end do
end subroutine sub
```

```
program exemplo_matriz
integer :: a(2,3)
call sub(a,2,3)
print *,'a='
do i=1,2
      print *,(a(i,j),j=1,3)
end do
end program exemplo_matriz
!-----------------------------------------
subroutine sub(a,n,m)
integer,intent(out) :: a(n,m)
integer,intent(in) :: n,m
forall(i=1:2;j=1:3) a(i,j)=i**2+j**3
end subroutine sub
```


Paso de subprogramas como argumentos

- Subprograma *external*: pode pasarse como argumento doutro subprograma. Ex: integral definida

$$\int_0^1 \sin x\,dx \qquad \int_{-1}^1 \cos x\,dx$$

```
program integral_definida
f(x)=sin(x); h(x)=cos(x)
real :: integral
external :: f, h
print *, integral(f, 0., 1.)
print *, integral(h, -1., 1.)
end program integral_definida
```

```
real function integral(g,a,b)
real, intent(in) :: a, b
integral=0;x=a; h = 0.001
do
   integral=integral+g(x)
   x = x + h
   if(x > b) exit
end do
integral=integral*h
end function integral
```

```
real function f(t)
real,intent(in) :: t
f=sin(t)
return
end function f
```

```
real function h(t)
real,intent(in) :: t
f=cos(t)
return
end function f
```

Variábeis estáticas

- Son variábeis locais dos subprogramas que conservan o seu valor entre chamadas sucesivas a un subprograma
- Son estáticas porque se almacenan na mesma posición de memoria en tódalas chamadas ao subprograma
- As variábeis locais por defecto non son estáticas
- Para ser estática, dúas opcións alternativas:
  - Inicialización na declaración: *real :: x = 5*
  - Atributo *save: real, save :: x*
- Na 1ª chamada ao subprograma, o seu valor é 0 agás que se inicialice na declaración: *real :: x = 1*

Exemplo de uso das variábeis estáticas

```
program exemplo
do i=1,10
   call s()
end do
end program exemplo
```

```
subroutine s()
integer :: n=0
print *, “n=”, n
n=n+1
end subroutine s
```

Subprograma que mostra por pantalla o nº de veces que se leva executado usando unha variábel estática

Variábel local estática por inicializarse na declaración

Execución: imprime por pantalla os números do 0 ao 9 dende o subprograma (sen que se lle pasen argumentos)

Atención: se inicializamos na declaración unha variábel local dun subprograma:
• Pasa a ser **estática** (e conservar o seu valor entre chamadas a subprograma)
• A inicialización só vale para a 1ª chamada ao subprograma

Función de sentenza

- É unha función que so ten unha sentenza cunha expresión matemática.
- Sintaxe: *nome(arg1,...argN)=expresion*
- Exemplo:

```
f(x)=x**2+x-2
print *,x,y,f(x),f(y)
```

```
integer :: f
f(x)=x**2+x-2
print *,x,f(x)
```

- O seu resultado é do tipo definido implícitamente polo seu nome. Para cambialo:
- So se pode usar no subprograma no que se declara.
- Unha función de sentenza pode ser usada na definición doutra función de sentenza:

```
length(r)=2*pi*r
area(r)=pi*r*r
key(r)=length(r)*area(r)
```


Recursividade (I)

- Subprograma recursivo: subprograma que se chama a si mesmo. Pode ser función ou subrutina
- Debe ter o atributo *recursive*: se non, erro de compilación
- **Funcións recursivas**:

```
recursive function nome(args) result(var)
  x = nome(...)   ! chamada recursiva
  var = ...       ! valor retornado
```

- O nome da función non pode ser retornado, porque debe ser usado para chamarse a si mesma
- A función debe chamarse a si mesma nalgún lugar do corpo

Recursividade (II)

- **Subrutinas recursivas**:

```
recursive subroutine nome(args)
call nome(...)
```

- Dentro da subrutina debe chamarse a si mesma
- Subprogramas recursivos: debe haber sentenza(s) de selección para distinguir entre:
  - Caso recursivo (hai chamada recursiva)
  - Caso non recursivo (non hai chamada)
- Se non hai caso non recursivo, secuencia infinita de chamadas: *stack overflow* (erro de execución)

Exemplo de función recursiva: factorial dun número enteiro

```
program exemplo
integer :: factorial
fn = factorial(5)
end program exemplo

recursive integer function factorial(n) result(fn)
integer, intent(in) :: n
if(n <= 1) then
   fn = 1
else
   fn = n*factorial(n-1)
end if
end function factorial
```

$$n! = n(n-1)\cdot(n-2)...2\cdot1$$

$$n! = n(n-1)!$$

Caso non recursivo

Caso recursivo

Implementación iterativa:

```
fn=1
do i=2,n
   fn=fn*i
end do
```

Vectorizada:

```
fn=product([(i,i=2,n])
```

Versión con subrutina recursiva

```
program exemplo
call factorial(5,fn)
print *,fn
stop
end program exemplo
!-------------------------------------------------------------
recursive subroutine factorial(n,fn)
integer, intent(in) :: n
integer, intent(out) :: fn
if(n <= 1) then
   fn=1
else
   call factorial(n-1,m)
   fn=n*m
end if
end subroutine factorial
```

Argumento *out* para almacenar o resultado

Caso non recursivo

Caso recursivo

$n! = n(n - 1)!$

Argumentos nomeados

- Na chamada ao subprograma pódese especificar os nomes dos argumentos.
- Se na chamada se nomea un argumento, hai que nomealos todos.
- Pode haber chamadas con argumentos nomeados e outras sen nomealos.
- O subprograma debe declararse nun bloque *interface* no programa principal.

```
program argumentos_nomeados
interface
   subroutine nomeados(x,y)
      real,intent(in) :: x,y
   end subroutine
end interface
call nomeados(y=3.2,x=2.1)
call nomeados(1.1,2.2)
end program
argumentos_nomeados
!-------------------------------------------
subroutine nomeados(x,y)
real,intent(in) :: x,y
print *,'x=',x,'y=',y
end subroutine nomeados
```

Argumentos opcionais

- Teñen o atributo *optional*, de modo que se pode chamar ao subprograma indicando este argumento ou non.
- Non pode haber ningún argumento obrigatorio logo dun argumento opcional.
- O subprograma debe declararse nun bloque *interface* no programa principal.
- Función intrínseca *present(x)*: retorna *.true.* se o argumento *x* está presente, e *.false.* en caso contrario.

```
program argumentos_opcionais
interface
   subroutine opcionais(x)
      real,intent(in),optional :: x
   end subroutine
end interface
call opcionais(2.1)
call opcionais()
end program argumentos_opcionais
!-----------------------------------------------
subroutine opcionais(x)
real,intent(in),optional :: x
if(.not.present(x)) then
   print *,'argumento y non presente'
else
   print *,'argumento y presente',y
end if
end subroutine opcionais
```


Evelyn Berezin (1925-2018)

- Inventora (1957) do primeiro ordenador de oficina, gardaba libros e contas e automatizaba un sistema bancario nacional
- Creadora (1971) do primeiro software procesador de texto
- Desenvolvedora (1962) do primeiro sistema software de reservas de pasaxeiros (60 cidades) para a liña aérea United Airlines, o sistema informático máis grande construído ata entón

Formatos de E/S (I)

- Os datos (enteiros, reais, carácter, ...) pódense ler / escribir (teclado/pantalla ou arquivos) de distintas formas (ancho, nº de decimais, modo exponencial ou non, etc.)
- Sentenza *format*:

  print 2, x, y, z ! escritura en pantalla con formato 2
  read 2, x, y, z ! lectura por teclado co formato 2
  print '(códigos de formato)', x, y, z
  read '(códigos de formato)', x, y, z
  2 format(códigos de formato)

- Os códigos de formato indican como se le / escribe cada dato, e débense axustar en nº aos datos (constantes / variábeis / expresións) a ler ou escribir.

Formatos de E/S (II)

- Ancho de campo: nº de caracteres máximo que pode ocupar un dato

Códigos de formato

- **Enteiros**: código I: **3i5**: 3 enteiros con 5 caracteres Ex: *print '(i2)', n; read '(i5)', m*

  O mellor: *print '(i0)',n*: escribe co ancho necesario.

- **Reais sen expoñente**: código F: **2f6.1**: 2 reais sen expoñente con 6 cifras e 1 decimal
- **Reais con expoñente**: código E: **4e10.3**: 4 reais con expoñente, con 10 cifras (incluíndo signo, parte enteira e fraccionaria, E do expoñente, signo e expoñente) e 3 decimais

Formatos de E/S (III)

- **Reais de dobre precisión**: código D: **2d10.4**: 2 datos de dobre precisión, igual que con código F
- **Caracteres**: código A: **2a10**: 2 cadeas de caracteres, cada unha con 10 caracteres. Se pos o código **a** sen ancho nun *print*, escribe a cadea co seu ancho
- **Espazos en branco**: código X: **5x**: 5 espazos en branco
- **Supresión de nova liña**: código $: *print* '*(“n? ”,$)*'
- **Tabuladores**: código T: **t10**: tabulador que chega até a columna 10-1=9

print 1, x, y, n, a
1 format(f6.2,e10.1,i7,a20)

1 real de ancho 6 e 2 decimais
1 real con expoñente, ancho 10 e 1 decimal
1 enteiro de ancho 7
1 carácter de ancho 20

Apertura de arquivos: *open* (I)

Sentenza *open*: permite abrir un arquivo e asocialo a unha unidade (representada por un nº enteiro):

open(1,file=’datos.dat’)

open(1,file=’datos.dat’,status=‘old’,err=1)

- *status=’old’*: o arquivo xa debe existir (p.ex. para ler): se non existe, *open* da erro para evitar ler dun arquivo baleiro
- *status=’new’*: o arquivo non debe existir (p.ex. para crealo e escribir nel): se xa existe, *open* da erro para evitar sobrescribilo

Apertura de arquivos: *open* (II)

- En caso de erro, salta á sentenza con etiqueta 1:

```
open(1, file='datos.dat', status='new', err=1)
...
stop
1 print *,'erro en open abrindo datos.dat'
end program
```

- Neste exemplo, a unidade é a nº 1. Hai unidades reservadas (non se poden usar):
  - Erro estándar (terminal): 0
  - Entrada estándar (teclado): 5, *
  - Saída estándar (terminal): 6, *

Lectura de arquivos: *read* (I)

- Sentenza *close*: desvincula arquivo e unidade:

```
close(1)
```

- Lectura dende un arquivo: *read*

```
read (1, *) x, y, z
read (1, fmt=1,end=2) a, b, c
1 format(3f5.2)
read (1,'3f5.2',end=2) a,b,c
...
2 close(1)
```

moi importante: variábeis nas que se almacenan os datos lidos (hai que saber previamente a forma do arquivo): fácil de esquecer

- Le unha liña do arquivo e almacena os datos lidos nas variábeis indicadas
- Unidade (1), formato: * (por defecto), etiqueta ou códigos de formato

Lectura de arquivos: *read* (II)

- Cada *read* faise nunha liña distinta do arquivo
- Opción *end=2*: salta á sentenza con etiqueta 2 cando atopa o final do arquivo
- Exemplo: le un arquivo até que atopa o final:

```
character(100) :: a
open(1, file='datos.dat', status='old', err=1)
do
   read (1, *, end=2) a
end do
2 close(1)
stop
1 print *,'erro lendo de datos.dat'
```


Exemplo: lectura dun arquivo dato a dato

```
program lectura_arquivo_dato_a_dato
open(1,file='arquivo_dato_a_dato.dat',status='old',err=1)
do
    read (1,'(i2,$)',end=2) n
    print '(i0," ",$)',n
end do
2 close(1)
print *,''
stop
1 stop 'arquivo_dato_a_dato non existe'
end program lectura_arquivo_dato_a_dato
```

Lectura dun enteiro sen pasar á seguinte liña

Remate do bucle ao chegar á fin de arquivo

arquivo_dato_a_dato.dat

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

Uso de *read* para extraer variábeis de cadeas de texto formatadas

- Supón que tes unha cadea de texto na cal hai valores numéricos (enteiros/reais) e caracteres: ‘x= 5.32 n=512 s=ola caracola’
- Para extraer os valores 5.32, 512 e ‘ola caracola’ a variábeis real, enteira e carácter debes usar a sentenza *read*:

```
character(100) :: s='x= 5.32 n=512 s=ola caracola'
character(50) :: t,u,v,w
print *,s
read (s,'(a3,f4.2,a3,i3,a3,a)') u,x,v,n,w,t
print *,x,n,t
```

Extrae os valores 5.32. 512 e ‘ola caracola’ ás variábeis *x*,*n* e *t*

Os a3 correspóndense con ‘*x*= ’, ‘*n*= ’ e ‘*s*= ‘

O *read* ten un carácter (*s*) no canto dunha unidade de arquivo

Escritura de arquivos: *write*

write (1,) x, y, z*

- Escribe no arquivo da unidade 1 as variábeis indicadas co formato indicado (neste caso, por defecto, *)
- Para escribir na saída estándar (terminal), poñer unidade=5 ou *; ou *print *, x, y, z*
- Pódese engadir a opción *err=n* (*n*=etiqueta de sentenza á que salta se hai erro na escritura)
- Posíbeis erros: tentar escribir en arquivos nos que non hai permiso de escritura, ou en arquivos de directorios que non existen

Exemplos de formatos de E/S

```
real :: x=-3.2,y=-1.1234
integer :: n=-10,m=19
write (1,fmt=1) x,n,y,m
1 format(e10.3, t15, i4, f8.2, 6x, i8)
```

```
-0.320E+01    -10   -1.12          19
```

Saída do *write*

```
write(1,fmt=2) n,m,x
2 format(2(i5,4x), e8.1)
```

```
 -10       19    -0.3E+01
```

Saída do *write*

```
write(1,fmt=3) x,y
3 format(2f3.1)
```

non hai ancho de campo dabondo para os datos

```
***  ***
```

Saída do *write*

Exemplo: creación e escritura dun arquivo

```
open(1,file='datos.dat',status='new',err=1)
do i = 1, 10
   write (1, *) i
end do
close(1)
stop
1 print *,'erro en open abrindo datos.dat'
```

- Se non lle poñemos *status='new'*, non nos daría erro en caso de xa existi-lo arquivo. Neste caso, sobrescribiría o arquivo, se este existe, **perdéndose**. Isto é perigoso

Uso de *write* para crear cadeas de texto formatadas a partir de variábeis

- A sentenza *write* tamén permite escribir en cadeas de texto, non so en unidades de arquivo.
- Supoñámos que tes as variábeis *x* (real), *n* (enteira) e *s* (cadea de caracteres).
- Para crear unha cadea de caracteres *t* da forma '*x=X n=N s=S*', onde X, N e S son os valores das variábeis *x* (con 5 cifras decimais), *n* (con 7 cifras) e *s*, tes que facer o seguinte:

```
character(20) :: s='ola caracola'
character(100) :: t
x=3.141592;n=1830
!t debe ter lonxitude dabondo para todo (x,n,s)
write (t,'(a,f9.5,a,i7,a,a)') 'x=',x,' n=',n,' s=',s
print *,t
```

Escribe en *t* a cadea formatada 'x=*X* n=*N* s=*S*'

Escritura de arquivos: pregunta antes de sobrescribir

```
program exemplo
character(5) :: s
open(1,file='datos.txt',status='new',err=1)
2 write(1,*) 'ola'
close(1)
stop
1 print '("datos.txt existe: sobrescribir?(s/n) ",$)';read *,s
if(s/='n') then
   open(1,file='datos.txt');goto 2
end if
stop
end program exemplo
```

Vai a sentenza con etiqueta 2

Escritura ao final dun arquivo (I)

- Tes que empregar a subrutina:

fseek(unidade, desprazamento, onde)

- Move o cursor polo arquivo, en función do argumento *onde*: *onde=0* (comezo do arquivo), *1* (posición actual no arquivo), *2* (final do arquivo);
- Argumento *desprazamento*: nº de bytes (ou caracteres) logo do comezo / actual / final.
- Exemplos:

call fseek(1,0,0): sitúa o cursor ao comezo do arquivo

call fseek(1,10,1): sitúa o cursor 10 bytes logo da posición actual

call fseek(1,-20,2): sitúa o cursor 20 bytes antes do final do arquivo

Escritura ao final dun arquivo (II)

```
open(1, file='datos.dat', status='old', err=1)
call fseek(1,0,2)  ! sitúase ao final do arquivo
write (1, *) x, y, z  ! engade ao final do arquivo
close(1)
stop
1 print *, "erro en open abrindo datos.dat"
```

- Función *ftell*: retorna a posición (en nº de bytes ou caracteres dende o comezo do arquivo) do cursor:

pos = ftell(unidade)

- Ex: *print *,'pos=', ftell(1)*
- Función *rewind(unidade)*: vai ao comezo do arquivo

Función que le un vector columna dende un arquivo

Hai que reservar memoria para o vector na función e definir a interface axeitada:

```
program proba
interface
    function le_vector(nf) result(x)
        character(len=100),intent(in) :: nf
        real,allocatable :: x(:)
    end function
end interface
character(len=100) :: nf
real,allocatable :: x(:)
print '(a,$)','nf? '
read *,nf  ! usa 'vector.dat'
x=le_vector(nf)
print *,'x=',x
deallocate(x)
stop
end program proba
```

```
function le_vector(nf) result(x)
character(len=100),intent(in) :: nf
real,allocatable :: x(:)
open(1,file=nf,status='old',err=1);n=0
do
    read (1,*,end=2);n=n+1
    print *,n
end do
2 allocate(x(n))
rewind(1)
do i=1,n
    read (1,*) x(i)
end do
return
1 print *,'open: arquivo',nf,'non existe'
stop
end function le_vector
```

Le arquivo e conta datos

Reserva memoria

Le datos e almacena no vector

Arquivo vector.dat:

```
1
2
3
4
5
```


Subrutina que escribe un vector nun arquivo

```
subroutine escribe_vector(v,n)
real, intent(in) :: v(:)
integer, intent(in) :: n
open(1, file = “vector.dat”, status=’new’)
write (1, *) (v(i), i = 1, n)
close(1)
end subroutine escribe_vector
```

```
real, dimension(3) :: v=(/1,2,3/)
call escribe_vector(v,3)
...
```

Inicialízase o vector no programa principal

Función que le unha **matriz cadrada** dende un arquivo

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

Arquivo matriz_cadrada.dat

```
program principal
interface
    function le_matriz(f) result(b)
        integer,dimension(:,:),allocatable :: b
        character(100),intent(in) :: f
    end function le_matriz
end interface
integer,dimension(:,:),allocatable :: a
character(100) :: f='matriz_cadrada.dat'
a=le_matriz(f)
print *,'a='
n=size(a,1)
do i=1,n
    print *,(a(i,j),j=1,n)
end do
stop
end program principal
```

```
function le_matriz(f) result(a)
character(100),intent(in) :: f
integer,dimension(:,:),allocatable :: a
character(100) :: s
open(1,file=f,status='old',err=1)
n=0
do
    read (1,*,end=2)
    n=n+1
end do
2 allocate(a(n,n))
rewind(1)
do i=1,n
    read (1,*) (a(i,j),j=1,n)
end do
close(1)
return
1 print *,'erro: arquivo ',f,'non existe'
stop
end function le_matriz
```

Le arquivo e conta liñas

Reserva memoria

Le datos e almacena en matriz

Programa que le unha matriz **non cadrada** dende un arquivo

```
1 2 3
4 5 6
```

Arquivo matriz.dat

```
program proba
character(100) :: s
integer,allocatable :: a(:)
open(1,file='matriz.dat',status='old',err=1)
read (1,'(a)') s
nf=1;nc=1
do i=1,len_trim(s)
    if(s(i,i)==' ') nc=nc+1
end do
do
    read (1,*,end=2)
    nf=nf+1
end do
2 rewind(1)
allocate(a(nf,nc))
do i=1,nf
    read (1,*) (a(i,j),j=1,nc)
end do
print *,'a='
do i=1,nf
    do j=1,nc
        print '(i5,$)',a(i,j)
    end do
    print *,''
end do
close(1);deallocate(a)
stop
1 print *,'erro: matriz.dat non existe'
end program proba
```

Conta as columnas da matriz

Conta as filas da matriz

Le a matriz

Mostra a matriz por pantalla

O nº de columnas é o nº de espazos da columna máis 1

len_trim(*s*): lonxitude de *s*

Subrutina que escribe unha matriz nun arquivo

```
program principal
real, dimension(3,4) :: a
forall(i=1:3,j=1:4) a(i,j)=i*j+j/2
call escribe_matriz(a,3,4)
end program principal
```

Inicialízase a matriz no programa principal

```
subroutine escribe_matriz(a,nf,nc)
real, intent(in) :: a(nf,nc),nf,nc
open(1, file='matriz.dat',status='new',err=1)
do i = 1, nf
    write (1, *) a(i, :)
end do
close(1)
return
1 print *,'erro escribindo matriz.dat'
stop
end subroutine escribe_matriz
```


Exemplo: lectura de arquivo en formato *write.table* de R

```
program le_arquivo
character(10),allocatable :: entrada(:),saida(:)  !vectores de strings
real,allocatable :: a(:,:)   !matriz cos datos
nf=2;ne=4 ◄
allocate(entrada(ne),a(nf,ne),saida(nf))
open(1,file='arquivo.dat',status='old',err=1)
read (1,*) (entrada(i),i=1,ne)  ! descarta a última cadea de caracteres ('Saida')
do i=1,nf
    read (1,*) j,(a(i,j),j=1,ne),saida(i)
end do
close(1)
print *,'a='
do i=1,nf
    print *,(a(i,j),j=1,ne)
end do
print *,'saida=',(saida(i),i=1,nf)
deallocate(entrada,a,saida)
stop
1 print *,'erro en open abrindo arquivo.dat'
end program le_arquivo
```

Pre-define o nº de filas e de columnas

arquivo.dat: columnas separadas por tabuladores

	E1	E2	E3	E4	Saida
1	0.25	0.33	1.23	-0.51	Branco
2	-0.34	1.3E5	0.22	4.3	Negro

Le arquivo en formato R sen pre-definir o nº de filas e columnas

```
program le_arquivo_formato_R
character(100) :: fich='arquivo.dat'
real,allocatable :: x(:,:)
character(100),allocatable :: y(:),none(:)
call conta_filas_columnas(fich,nf,nc)
allocate(x(nf,nc),y(nf),name(nc))
open(1,file=fich,status='old',err=1)
read (1,*) (nome(i),i=1,nc)
do i=1,nf
      read (1,*) t,(x(i,j),j=1,nc),y(i)
end do
close(1)
do i=1,nc
      print '(a10,$)',nome(i)
end do
print *,'saída'
do i=1,nf
      do j=1,nf
            print '(f10.2,$)',x(i,j)
      end do
      print *,y(i)
end do
deallocate(x,y)
stop
1 print *,fich,'non atopado'
end program le_arquivo_formato_R
```

Le datos do arquivo

Mostra datos por pantalla

```
subroutine conta_filas_columnas(fich,nf,nc)
character(100),intent(in) :: fich
integer,intent(out) :: nf,nc
character(100) :: s
open(1,file=fich,status='old',err=1)
read (1,'(a)') s
nf=0;nc=0
do i=2,len_trim(s)
      j=i-1;n=ichar(s(j:j));m=ichar(s(i:i))
      if(n/=69.and.m==69) nc=nc+1
end do
do
      read (1,*,end=2);nf=nf+1
end do
2 close(1)
return
1 print *,fich,'non atopado';stop
end subroutine conta_filas_columnas
```

Conta columnas na cabeceira

Conta as filas de datos

69=código do carácter tabulador

Erros comúns de programación en Fortran (I)

- Poñer caracteres especiais na sentenza *program*. Ex: *program programa.f90* (o . é un carácter especial)
- Usar nomes incorrectos para variábeis ou subprogramas. Ex: chamarlle *1x* a unha variábel no canto de *x1*
- Declaración de variábel fóra de sitio

```
x=x+i
real :: k
```

- Sobrescritura de variábel

```
x=i+1
x=i**2
```

- Acceder a elementos dun vector ou matriz dinámico antes do *allocate* ou despois do *deallocate* (erro de execución)
- Poñer unha expresión ou unha chamada a función na beira esquerda dunha asignación. Ex: *count*(*x*)=*n*, *x*+5=*y* (erro de compilación)
- Usar variábel enteira/real como vector/matriz, ou viceversa

```
x=5
print *,x(i)
```

```
real :: a(2,2),y=3
a=y+2
```

Erros comúns (II)

- Usar variábeis sen inicializar, especialmente índices dun vector ou matriz, por exemplo poñer *x*(*i*) fóra dun bucle *do i=1,n*

```
program proba
print *,i
```

```
do i=1,n
      print *,x(i)
end do
x(i)=6
```

- Cambiar o valor do índice dentro dun bucle *do* (erro de compilación)
- Uso de índice real, non enteiro, no elemento dun vector

```
real :: i=1
print *,x(i)
```

- Esquecer o *then* nun bloque *if* ou *if-else*
- Usar a función *sum*(·), ou *minval*(·), etc., cun único número. Deben aplicarse a un vector ou matriz, ou a unha parte. Ex: *sum*(*x*(*i*)) está mal, debe ser *sum*(*x*) ou *sum*(*x*(*1*:*i*))

```
x=5
print *,sum(x)
```

- Modificar un argumento de tipo *intent*(*in*) dentro dun subprograma
- Ler ou escribir nun arquivo logo de pechalo con *close*(·)

Radia Perlman (1951)

- Experta en comunicacións
- Deseñadora de Internet
- Creadora do *Spanning Tree Protocol* (STP), fundamental para as pontes entre redes de ordenadores
- Realizou grandes avances nos métodos de transmisión de datos en rede
- Pertencente á *National Academy of Engineering* de EEUU

Módulo

- Bloque de código que contén datos e subprogramas: ***use*** *modulo*

```
module nome
   tipo :: var
   interface
      tipo subprograma(...)
      ...
      end fipo
   end interface
contains:
   tipo subprograma(...)
   ...
   end tipo
end module
```

```
module proba
     integer :: x
contains
     subroutine sub(y)
          integer,intent(in) :: y
          print *,y
     end subroutine sub
end module
```

```
program modulo
use proba
call sub(5)
end program modulo
```

- Pode estar en arquivo distinto do programa principal
- Se está en arquivo distinto: *f95 modulo.f90 principal.f90*

antes do arquivo que o usa!

Clase e obxecto

- Un dato pode ser:

1) Atómico (indivisible): *integer*, *real*, etc.
2) Agregado: colección de datos do mesmo tipo (vectores e matrices)
3) Heteroxéneo: colección de datos de distintos tipos (libro= título, autor, ISBN,editorial,ano,...)

- Unha *clase* é unha agrupación de:

1) Variables, en xeral de distintos tipos (dato heteroxéneo)
2) Subprogramas (funcións ou subrutinas)

- Un *obxecto* é unha variable dunha clase. Exemplo: defino a clase *persoa* e dous obxectos (variábeis) de tipo *persoa* chamados *p* e *q*

Clase e obxecto en Fortran

- É un bloque *type*: inclúe variábeis e nomes de subprogramas
- Os subprogramas da clase van no mesmo módulo que o *type*
- Declaración:

  type(*persoa*) :: *p*=*persoa*(*nome*,idade)
- Acceso a variables ou subprogramas: *p%nome*, *p%idade*, *p%mostra*()
- *persoa* é unha clase, *p* é un obxecto desa clase

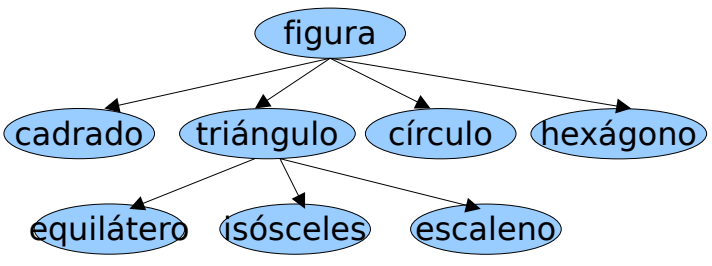
```
module modulo_persoa
type persoa
      character(10) :: nome
      integer :: idade
contains
      subroutine mostra()
end type persoa
subroutine mostra()
      print *,'nome=',nome
end subroutine mostra
end module modulo_persoa
```

Alternativa a *p%mostra*()

```
program exemplo_tipos
use modulo_persoa
type(persoa) :: p=persoa('carlos',14)
p%mostra()
print *,'nome=',p%nome,'idade=',p%idade
end program exemplo_tipos
```


Herdanza

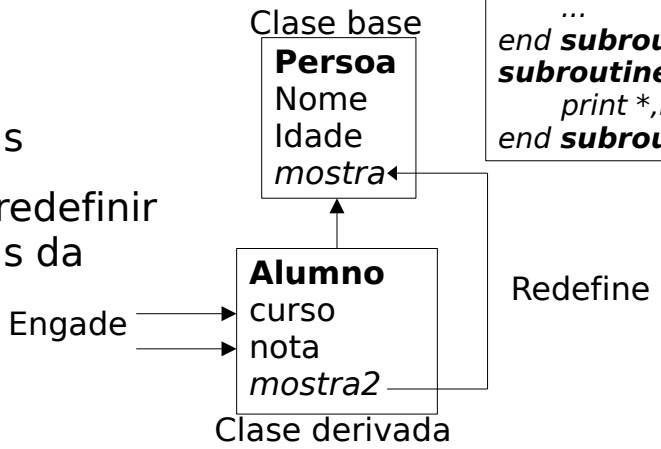
- Defínese unha clase (derivada) que herede de outra (base) as variables e subprogramas.
- Isto evita ter que redefinir cousas e permite reutilizar o código
- A clase derivada pode redefinir subprogramas
- Pode haber varias clases derivadas da mesma base, definindo unha xerarquía de clases que pode ter máis de dous niveis
- Cada clase derivada pode redefinir un subprograma da clase base ou heredalo
- Isto chámase *sobrecarga* do subprograma



Herdanza en Fortran

- Palabra clave *extends*: define unha clase derivada que extende a outra clase base
- A clase *alumno* hereda de *persoa*, engade a variábel *curso* e o subprograma *nota*(), e redefine o subprograma *mostra*() de *persoa*
- Pode engadir variábeis e subprogramas
- Tamén pode redefinir subprogramas da clase base

```
type,extends(persoa) :: alumno
    integer :: curso
contains
    procedure mostra=>mostra2()
    subroutine nota()
end type alumno
subroutine nota()
    ...
end subroutine nota
subroutine mostra2()
    print *,nome,idade,curso
end subroutine mostra2
```



Polimorfismo

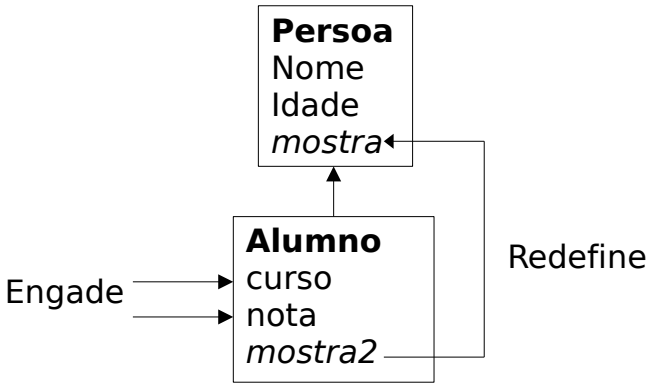
- Se hai varias clases derivadas e chamamos a un método da clase derivada, será distinto para cada unha
- Podes chamar a subprogramas distintos co mesmo nome sen ter que comprobar que programa executar
- Isto chámase *polimorfismo*: chamar da mesma forma a varios subprogramas distintos
- Vantaxe: cando chamas a un subprograma dun obxecto derivado, non hai que comprobar de que tipo é cada obxecto
- Polo tipo de obxecto, Fortran xa sabe a que subprograma chamar
- Para conseguir polimorfismo en Fortran declaras un obxecto da clase base e inicializas este obxecto coa clase derivada

Polimorfismo en Fortran

- A sentenza *procedure* indica o subprograma da clase derivada que sobrescribe o subprograma da clase base

procedure subprograma=>subprograma2

- *subprograma*: nome do subprograma na clase base
- *subprograma2*: en clase derivada




```
type,extends(persoa) :: alumno
    integer :: curso
contains
    procedure mostra=>mostra2(a)
    subroutine nota()
end type alumno
subroutine nota()
    ...
end subroutine nota
subroutine mostra2(a)
    class(persoa),intent(in) :: a
    print *,a%nome,a%idade,a%curso
end subroutine mostra2
!-------------------------------------------------------------
program principal
type(persoa) :: p=alumno('Carlos',16,2)
p%mostra()  ! chama a mostra2() de alumno
end program principal
```


Sobrecarga de operadores aritméticos, relacionais ou lóxicos

- Define un operador a medida para unha clase
- Bloque *interface operator,* indicando o operador a sobrecargar e a función que o sobrecarga:

```
interface operator (+)
    module procedure suma
end interface operator(+)
```

- Chámase á función *suma* cando se executa *p+q* sendo *p* e *q* obxectos da clase *persoa*. É como se *suma* substituíse a +
- O operador pode ser aritmético, relacional ou lóxico.
- Se o operador é aritmético, a función debe retornar un obxecto do mesmo tipo que os operandos.

Sobrecarga de operador aritmético (+)

```
module mod_persoa
    type persoa
        character(10) :: nome
        real :: peso,altura
    end type persoa
    interface operator (+)
        module procedure suma
    end interface operator(+)
    contains
    type(persoa) function suma(x,y)
        class(persoa),intent(in) :: x,y
        character(10) :: nome
        real :: peso,altura
        nome=cat(x%nome,y%nome)
        peso=x%peso+y%peso
        altura=x%altura+y%altura
        suma=persoa(nome,peso,altura)
    end function suma
    character(10) function concat(x,y) result(s)
        character(10),intent(in) :: x,y
        character(10) :: x2,y2
        x2=trim(x);nx=len(x2);y2=trim(y);ny=len(y2)
        do i=1,nx
           s(i:i)=x2(i:i)
        end do
        do j=1,ny
           s(i:j)=y2(j:j);i=i+1
        end do
    end function concat
end module mod_persoa
```

```
program principal
use mod_persoa
type(persoa) :: x,y,s
x=persoa('alba',65.2,1.84)
y=persoa('clara',70.1,1.98)
s=x+y
print *,'s=',s%nome,s%peso,x%altura
stop
end program principal
```

Función *concat*: concatena os nomes de *x* e *y*

Sobrecarga de operador relacional (>)

Se o operador que se sobrecarga é relacional ou lóxico, o subprograma que sobrecarga ao operador debe retornar un dato de tipo *logical*

```
program principal
use mod_persoa
type(persoa) :: x=persoa('alba',65.2,1.84)
type(persoa) :: y=persoa('clara',70.1,1.98)
if(x>y) then
    print *,x%nome,'>',y%nome
else
    print *,x%nome,'<=',y%nome
end if
stop
end program principal
```

```
module mod_persoa
    type persoa
        character(10) :: nome
        real :: peso,altura
    interface operator (>)
        module procedure maior
    end interface operator(+)
    end type persoa
    contains
    logical function maior(x,y)
        type(persoa),intent(in) :: x,y
        if(x%peso/x%altura > y%peso/y%altura) then
            maior=.true.
        else
            maior=.false.
        end if
    end function maior
end module mod_persoa
```

Mary Allen Wilkes (1937)

- Creadora (1965) del primer ordenador personal para trabajo en casa
- Desarrolladora de sistemas operativos e linguaxe ensamblador (LAP6)
- Traballou no MIT e na Univ. Washington

Conversión entre tipos de datos

- De enteiro a real: *x=i* ou *real(i)*
- De real a enteiro: *i=x*, pero podes perder información (p.ex. de 3.2 a 3).
- Podes redondear ao enteiro mais cercano con *i=nint(x)*, por defecto con *i=floor(x)*, por exceso con *i=ceiling(x)*.
- De carácter (p.ex. '32') a enteiro: *s='32';read (s,*) i*. So funciona se o carácter ten un número enteiro. Se *s='ola'*, o *read* da erro de entrada/saída.
- De carácter (p.ex. '3.2') a real: *s='3.2';read (s,*) x*. So funciona igual que no anterior.
- De enteiro a carácter: *i=32; write (s,'(i0)') i*
- De real a carácter: *x=3.2; write (s,'(f3.1)') x*

Creación dunha libraría en Fortran

- Libraría: código máquina de moitos subprogramas (en Fortran) empaquetado nun arquivo
- Non contén o código fonte (non se pode depurar ou ver como opera).
- Proporciónan subprogramas que permiten facer operacións (p.ex., determinantes, resolución de sistemas de ecuacións, ...)
- Para usar unha libraría, hai que enlazar (*link*) o noso programa coa libraría
- O compilador *gfortran* xa usa algunhas librarías incluídas co compilador (p.ex. funcións intrínsecas)

Ficheiros para crear a libraría

media.f90

```
real function media(x,n)
real,intent(in) :: x(n)
integer,intent(in) :: n
media=0
do i=1,n
     media=media+x(i)
end do
media=media/n
return
end function media
```

Podes descargalos dende estes enlaces:
media.f90
desviacion.f90
principal.f90

principal.f90

```
program principal
real,allocatable :: x(:),media
print *,‘n? ‘
read *,n
allocate(x(n))
print *,‘x[]? "
read *,x
y=media(x,n)
d=desviacion(x,n,y)
print *,‘media=‘,m
print *,‘desviacion=‘,d
end program principal
```

desviacion.f90

```
function desviacion(x,n,media)
real,intent(in):: x(n)
integer, intent(in) :: n
real, intent(in) :: media
desviacion=0
do i=1,n
     y = x(i)-media
     desviacion=desviacion+y*y
end do
desviacion=sqrt(desviacion/n)
return
end function desviacion
```

Libraría estática

- É un ficheiro con extensión *.a*: *libestatistica.a*
- Libraría *libestatistica.a* que proporcione subprogramas para calcula-la media e desviación dun vector
- Comando de compilación (sen enlazado):

  gfortran -c media.f90 desviacion.f90
- Creación da libraría: *ar qv libestatistica.a media.o desviacion.o*
- Compilación de programa principal e enlazado con libraría:

  gfortran -L. principal.f90 -lestatistica
- Execución: *a.exe*

Libraría dinámica

- Ficheiro con extensión *.so*: *libestatistica.so*
- Comando de compilación (sen enlazado):

  gfortran -fpic -c media.f90 desviacion.f90
- Empaquetado: *gfortran -shared -o libestatistica.so *.o*
- Listado de arquivos *.o*: *nm libestatistica.so* ou *readelf -s X.so*
- Uso da libraría dende programa *principal.f90*:

  gfortran -L. principal.f90 -lestatistica
- Execución: *a.exe*

Interface

- Bloque no que se indican subprogramas (tipo, nome, argumentos, valores retornados).

```
interface
    bloque subprograma1
    ...
    bloque subprograma N
end interface
```

```
interface
    function fun(x,y,n) result(z)
        integer,intent(in) :: x(n),y(n)
        real :: z
    end function fun
    subroutine sub(x,y)
        real,intent(in) :: x
        real,intent(out) :: x
    end subroutine sub
end interface
```

- Empréganse para cousas especiais:

  Retornar arrais dinámicos
  Pasar argumentos arrai sen a súa dimensión
  Pasar argumentos nomeados ou opcionais
  Sobrecarga de operadores

Exemplo de uso de módulos e interfaces: paso de vectores como argumentos

- Pasar o nome do arrai (*assumed-shape arrays*) sen a súa lonxitude como argumento a un subprograma:
- Cun subprograma interno
- Cun módulo
- Cunha interface

```
program proba
integer,allocatable :: x(:)
...
call sub(x)
...
contains
    subroutine sub(x)
    integer,intent(out) :: x(:)
    n=size(x)
    ...
    end subroutine sub
end program proba
```

```
module aux
contains
    subroutine sub(x)
    integer,intent(out) :: x(:)
    n=size(x)
    ...
    end subroutine sub
end module
program proba
use aux
integer,allocatable :: x(:)
...
call sub(x)
...
end program proba
```

```
program proba
interface
    subroutine sub(x)
        integer,intent(out) :: x(:)
    end subroutine
end interface
integer,allocatable :: x(:)
call sub(x)
end program proba
subroutine sub(x)
integer,intent(out) :: x(:)
n=size(x)
...
end subroutine sub
```

Exemplo de uso de módulos e interfaces: paso de matrices como argumentos

- Cun subprograma interno
- Cun módulo
- Cunha interface

```
program proba
integer,allocatable :: a(:,:)
...
y=fun(a)
...
contains
      function fun(a)
      integer,intent(...) :: a(:,:)
      nf=size(a,1);nc=size(a,2)
      ...
      end function fun
end program proba
```

```
module aux
contains
      function fun(a)
      integer,intent(...) :: a(:,:)
      nf=size(a,1);nc=size(a,2)
      ...
      end function fun
end module
```

```
program proba
use aux
integer,allocatable :: a(:,:)
...
y=fun(a)
...
end program proba
```

```
program proba
interface
      function fun(a)
            integer,intent(...) :: a(:,:)
      end function
end interface
integer,allocatable :: a(:,:)
y=fun(a)
end program proba
```

```
function fun(a)
integer,intent(...) :: a(:,:)
nf=size(a,1);nc=size(a,2)
...
end function fun
```


Exemplo de interface: subrutina que reserva un vector ou matriz *intent*(*out*) cunha interface

Con vector

```
program exemplo
interface
      subroutine sub(x)
         integer,allocatable,intent(out) :: x(:)
      end subroutine sub
end interface
integer,allocatable :: x(:)
call sub(x)
print *,'x=',(x(i),i=1,size(x))
deallocate(x)
end program exemplo
!---------------------------------------------
subroutine sub(x)
integer,allocatable,intent(out) :: x(:)
allocate(x(5))
do i=1,5
      x(i)=i*i
end do
return
end subroutine sub
```

Con matriz

```
program exemplo
interface
      subroutine sub(a)
            integer,allocatable,intent(out) :: a(:,:)
      end subroutine sub
end interface
integer,allocatable :: a(:,:)
call sub(a)
do i=1,size(a,1)
      print *,(a(i,j),j=1,size(a,2))
end do
deallocate(a)
end program exemplo
!---------------------------------------------
subroutine sub(a)
integer,allocatable,intent(out) :: a(:,:)
allocate(a(3,3))
do i=1,3
      do j=1,3
            a(i,j)=i*j+i-j
      end do
end do
return
end subroutine sub
```

Exemplo de interface: función externa que retorna un vector ou matriz reservado dinámicamente

Con vector

```
program exemplo
interface
      function func(x) result(y)
         integer,intent(in) :: x(:)
         integer,allocatable :: y(:)
      end function func
end interface
integer :: x(5)=(/1,2,3,4,5/)
integer,allocatable :: y(:)
print *,'x=',x
y=func(x)
print *,'y=',y
deallocate(y)
end program exemplo
!-----------------------------------------------
function func(x) result(y)
integer,intent(in) :: x(:)
integer,allocatable :: y(:)
n=size(x);allocate(y(n))
do i=1,n
      y(i)=x(n-i+1)
end do
end function func
```

Con matriz

```
program exemplo
interface
      function func(a) result(b)
            integer,intent(in) :: a(:,:)
            integer,allocatable :: b(:,:)
      end function func
end interface
integer :: a(5,5)
integer,allocatable :: b(:,:)
b=func(a)
deallocate(b)
end program exemplo
!-----------------------------------------------
function func(a) result(b)
integer,intent(in) :: a(:,:)
integer,allocatable :: b(:,:)
nf=size(a,1);nc=size(a,2)
allocate(b(nf,nc))
do i=1,nf
      do j=1,nc
            b(i,j)=a(nf-i+1,nc-j+1)
      end do
end do
end function func
```


Exemplo de interface: función que retorna un vector dinámico: *find*

- Atopa os índices dos elementos dun vector que cumpren unha condición (expresión lóxica)
- Moi útil para vectorizar expresións
- Retorna un vector reservado dinámicamente na función

```
function find(x) result(y)
logical,intent(in) :: x(:)
integer,allocatable :: y(:)
n=count(x);m=size(x)
allocate(y(n));j=1
do i=1,m
      if(x(i)) then
            y(j)=i;j=j+1
      end if
end do
end function find
```

```
program proba
interface
      function find(x) result(y)
         logical,intent(in) :: x(:)
          integer,allocatable :: y(:)
      end function find
end interface
integer :: x(5)=(/1,2,3,4,5/)
print *,count(x>2) !nº elementos >2
print *,find(x>2)    ! índices de elementos >2
print *,pack([(i=1,5)],x>2) !alternativa
print *,x(find(x>2))  ! valores de elementos >2
print *,pack(x,x>2) !alternativa
end program proba
```

Programación estruturada en Fortran
Exercicios clases interactivas

Semana 1

Instruccións para instalar o compilador Gfortran e o entorno Visual Studio Code no teu ordenador

Descarga o compilador **Gfortran** dende:

https://www.equation.com/ftpdir/gcc/gcc-13.2.0-32.exe

Executa o instalador e instala Gfortran no teu ordenador. Cando remate, reinicia o ordenador. Logo, descarga o entorno **Visual Studio Code** (VSCode) no teu ordenador dende este enlace (pulsa no botón **Windows**):

https://code.visualstudio.com/download (Pulsa no botón Windows(Windows 10,11))

Instala o VSCode executando o instalador. Executa o VSCode buscando "Visual Studio Code" no menú de inicio de Windows. Podes engadir a aplicación á barra de tarefas de Windows pulsando no botón dereito do rato no menú onde aparece VSCode e seleccionando "Anclar á barra de tarefas". Así, poderás executalo a seguinte vez pulsando no seu botón da barra de tarefas no canto de abrir o menú de inicio. Vai á barra da esquerda, pulsa no botón **Extensións**, situado abaixo nesa barra, e busca e instala as extensións "**Modern Fortran**" e "**cppdebug**".

Tes instruccións máis detalladas neste enlace:

https://github.com/fran-pena/met-num-for/blob/main/vscode_gfortran_windows/instalacion.md

Traballo en clase

No VSCode, vai á barra da esquerda, pulsa no botón **Extensións**, situado abaixo nesa barra, e busca e instala as extensións "**Modern Fortran**" e "**cppdebug**".

Vai ao menú **File→Open Folder** e sitúate no **Escritorio**. Alí, crea unha carpeta chamada **fortran**, na cal almacenaremos tódolos programas.

Para executar programas que se atopan na carpeta actual, vai ao menú de inicio, botón **Configuración** e na ventá de configuración busca "**Editar variables de ambiente desta conta**". Selecciona **Path**, pulsa no botón **Editar** e engade o directorio **.** que representa ao directorio actual. Pulsa no botón **Aceptar**. Pecha o VSCode e volve a abrir, para que a modificación do **Path** teña efecto no VSCode.

1. **Variábeis. Expresións aritméticas. Entrada/saída básicas**. Crea un programa no VSCode no directorio **fortran** anterior dende o menú **File→New File...** do VSCode, ou pulsando en **New File...** na pestana **Welcome**, e chámalle `expresions.f90`. O programa debe ler un número real x por teclado e mostrar por pantalla $3x-1$, $x^2+\sqrt{x-2}$ e $(\operatorname{sen} x-3)/(\ln x+\mathrm{e}^x-1)$. Gárdao no directorio **fortran**. Para compilar o programa, vai ao menú **Terminal→New Terminal**. Ábrese así unha terminal na ventá de VSCode. Nela, executa o comando:

   ```
   gfortran expresions.f90
   ```

   Deste modo xérase o programa executable `a.exe`. Para executalo, teclea **a.exe** ou `.\a.exe` na terminal, ou vai ao menú **Run → Run without debugging**, ou pulsa as teclas **Control+F5**.

   ```
   program expresions
   print '("x? ",$)'
   read *, x

   print *, 'Os valores son: '
   print '("3x-1=",f6.3)',3*x-1
   print '("x^2+sqrt(x-2)=",f6.3)',x**2+sqrt(x-2)
   print '("(sin(x)-3)/(ln(x)+exp(x)-1)=",f6.3)',(sin(x)-3)/(log(x)+
       exp(x)-1)

   end program expresions
   ```

Se queres que o executable se chame, por exemplo, `expresions`, hai que executar:

```
gfortran expresions.f90 -o expresions
```

Executa o programa introducindo por teclado o valor 1 para a variábel x. Logo execútao novamente usando o valor 2. Copia o programa `expresions.f90` á memoria flash, unidade **D:**.

Para evitar ter que gardar o programa cada vez que o modifiques (Ctrl+S ou menú **File→Save**), pódese activar no VSCode a opción de autogardado (Autosave). Para isto, vai ao menú **File→Auto Save**.

2. **Ecuación de 2º grao. Sentenzas de selección**. Escribe un programa chamado `ec2grao.f90` que lea os coeficientes (reais) dunha ecuación de segundo grao $ax^2 + bx + c = 0$ e calcule as súas solucións segundo a fórmula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

Se $a = 0$, temos unha ecuación de primeiro grao. Neste caso, se $b = 0$ a ecuación redúcese a $c = 0$, sendo independente de x. Se o valor c introducido por teclado é $c = 0$, entón a ecuación cúmprese para todo $x \in \mathbb{R}$. Se, polo contrario, o c lido por teclado é $c \neq 0$, entón a ecuación non se cumpre para ningún x, de modo que a solución é o conxunto baleiro. Se $b \neq 0$, entón pódese resolver a ecuación de 1^o grao e $x = -c/b$.

Se $a \neq 0$, temos unha ecuación de segundo grao. Sexa $d = b^2 - 4ac$ o discriminante. Se $d = 0$, entón hai dúas solucións reais iguais $x = \frac{-b}{2a}$. Se $d < 0$, entón hai dúas solucións complexas conxugadas $x = \frac{-b}{2a} \pm i\frac{\sqrt{-d}}{2a}$, onde i é a unidade imaxinaria $i = \sqrt{-1}$. Finalmente, se $d > 0$ hai dúas solucións reais distintas $x = \frac{-b \pm \sqrt{d}}{2a}$.

Para comprobar que o programa funciona correctamente, proba cos exemplos da táboa 1:

a	b	c	Nº solucións	Solucións
0	0	0	∞	$x = \mathbb{R}$
0	0	1	0	$x = \varnothing$
0	1	-1	ec. 1^o grao	$x = 1$
1	-2	1	2 reais iguais	$x = 1$
1	1	1	2 complexas	$x = -\frac{1}{2} \pm i\frac{\sqrt{3}}{2}$
1	0	-1	2 reais distintas	$x = \pm 1$

Cuadro 1: Valores dos coeficientes a, b, c e das solucións que debe obter o programa.

```
program ec2grao
print '("a,b,c? ",$)';read *, a,b,c
if(a==0) then
    if(b==0) then
        if(c==0) then
            print *, 'infinitas solucions reais: x=IR'
        else
            print *, 'non existen solucions'
        endif
    else
        print *, 'solucion= ', -c/b
    endif
else
    d=b*b-4*a*c;a2=2*a;rd=sqrt(abs(d));u=-b/a2;v=rd/a2
    if(d<0) then
        print *,'2 solucions complexas conxugadas: x=',y,'+/-I*',abs(v)
    else if(d==0) then
        print *,'2 solucions reais iguais: x=',u
    else
        print *,'2 solucions reais: x=',u-v,u-v
    endif
endif
end program ec2grao
```

3. **Sentenza de iteración definida. Acumulador. Constantes con nome**. Escribe un programa en Fortran chamado `multiplicatorio.f90` que lea por teclado un número enteiro positivo n e un valor real x e calcule o seguinte multiplicatorio:

$$2^{n-1}\prod_{k=1}^{n}\left[x-\cos\left(\frac{(2k-1)\pi}{2n}\right)\right] \tag{2}$$

Executa o programa usando $n=9$ e $x=2$. O programa debe mostrar 70225.9531 na terminal.

```
program multiplicatorio
real,parameter :: pi=3.141592
print '("n,x? ",$)'
read *,n,x
p=2**(n-1);t=pi/(2*n)
do k=1,n
    p=p*(x-cos((2*k-1)*t))
end do
print *,"p=",p
end program multiplicatorio
```

Exercicios propostos

1. Escribe un programa `plano.f90` que calcule a distancia entre un punto $\mathbf{v}=(x_0,y_0,z_0)$ e un plano π dado pola ecuación $ax+by+cz+d=0$. Esta ecuación tamén se pode escribir como $\mathbf{w}^T\mathbf{x}+d=0$, onde $\mathbf{w}=(a,b,c)$ é o vector director do plano, $\mathbf{w}^T$ é o trasposto de $\mathbf{w}$, perpendicular ao plano π, e $\mathbf{x}=(x,y,z)$ é un punto pertencente ao plano. A distancia entre $\mathbf{v}$ e π pódese calcular como:

$$d(\mathbf{v},\pi)=\frac{|ax_0+by_0+cz_0+d|}{|\mathbf{w}|} \tag{3}$$

O valor absoluto é coa función `abs(...)`; ademáis, $|\mathbf{w}|=\sqrt{a^2+b^2+c^2}$. Usa a función `sqrt()` para calcular a raíz cadrada. O programa debe ler por teclado os valores a,b,c,d,x_0,y_0,z_0. Usa $a=b=c=x_0=y_0=z_0=1$ e debes obter d=2.0394.

```
program plano
print '("a,b,c,d? ",$)'
read *,a,b,c,d
print '("x0=(x,y,z)? ",$)'
read *,x,y,z
print *,'distancia=',abs(a*x+b*y+d*z+d)/sqrt(a**2+b**2+c**2)
end program plano
```

2. Escribe un programa `mediaprod.f90` que lea n números $x_1,\ldots,x_n$ por teclado e calcule a súa media $\frac{1}{n}\sum_{i=1}^{n}x_i$ e o seu producto $\prod_{i=1}^{n}x_i$ sen usar arrais. Usa n=5 e os números 1,2,3,4,5.

```
program mediaprod
integer :: media=0,prod=1
print '("n? ",$)'
read *,n
do i=1,n
    print '(i0," x? ",$)',i
    read *,x
    media=media+x;prod=prod*x
end do
print '("media=",i0," prod=",i0)',media,prod
end program mediaprod
```

Semana 2

Traballo en clase

1. **Números de Fibonacci**. Escribe un programa chamado `fibonacci.f90` que lea por teclado un número enteiro n maior que 1 (proba con $n = 10$) e calcule os números de Fibonacci F_i para $i = 1, \ldots, n$. Estes números están definidos como:

$$F_1 = 0, \quad F_1 = 1, \quad F_i = F_{i-1} + F_{i-2}, \text{ para } i > 1 \tag{4}$$

   Podes atopar máis información sobre estes números neste enlace:

   https://es.wikipedia.org/wiki/Sucesión_de_Fibonacci

   Executa o programa para n=8, e debes obter 0, 1, 2, 3, 5, 8, 13, 21 e 34.

```
program fibonacci
integer :: f0,f1,f2
print '("n(>1)? ",$)'
read *,n
f0=0;f1=1
print '(a5," ",a10)','i','F(i)'
print '(i5," ",i10)',0,f0
print '(i5," ",i10)',1,f1
do i=2,n
    f2=f0+f1
    print '(i5," ",i10)',i,f2
    f0=f1;f1=f2
end do
end program fibonacci
```

2. **Sumatorio dobre. Vectores dinámicos**. Escribe un programa chamado `sumatorio.f90` que lea por teclado un número enteiro n e logo dous vectores n-dimensionais $\mathbf{v}$ e $\mathbf{w}$, reservados dinámicamente. O programa debe calcular, usando vectores:

$$s = \sum_{i=1}^{n} \sum_{j=1}^{i} v_i w_j \tag{5}$$

   Proba con $n = 3$, $\mathbf{v} = (1, 2, 1)$ e $\mathbf{w} = (-1, 0, 1)$ e tes que obter $s = -3$.

```
program sumatorio_dobre

real,allocatable :: v(:),w(:)

print '(a,$)', 'n? ';read *, n
allocate(v(n),w(n))
print '("v? ",$)';read *,v
print '("w? ",$)';read *,w

! v=[1,2,1]  ! inicializacion no programa (non necesita allocate)

print *,'v=',v  ! formato por defecto

suma=0
do i=1,n
    do j=1,i
        suma=suma+v(i)*w(j)
    end do
end do

! alternativa simple vectorizada
!suma=0
!do i=1,n
!    suma = suma + v(i)*sum(w(1:i))
```

```
!end do

! alternativa optima
!suma=0;s=0
!do i=1,n
! s=s+w(i);suma=suma+v(i)*s
!end do

print*, "O resultado e: ", suma

deallocate(v, w)

end program sumatorio_dobre
```

3. **Cálculo iterativo de media e desviación típica**. Escribe un programa `mediadesv.f90` que lea iterativamente por teclado números x_i, ata que se lea un 0 e calcule iterativamente a súa media m, varianza v e desviación típica d, definidas por:

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad v = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - m)^2, \quad d = \sqrt{v} \tag{6}$$

sendo n o número de valores de x_i. Para isto, usa as seguintes fórmulas, onde m_i, v_i e d_i son a media, varianza e desviación típica de $x_1, \ldots, x_i$ para $i = 1, \ldots, n-1$, sendo $m_1 = x_1$ e $v_1 = w_1 = d_1 = 0$:

$$m_{i+1} = m_i + \frac{x_{i+1} - m_i}{i+1}, \quad w_{i+1} = w_i + (x_{i+1} - m_i)(x_{i+1} - m_{i+1}) \tag{7}$$

$$v_{i+1} = \frac{w_{i+1}}{i}, \quad d_i = \sqrt{v_i}, \quad i = 1, \ldots, n-1 \tag{8}$$

Usa o vector `[8 3 1 2 7 9 5]` que ten n=7 elementos, que ten media=5, varianza=9.6667 e desviación=3.1091. Podes atopar máis información neste **enlace**:

https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm

```
program mediadesv
integer,parameter :: n=7
real :: x(n)=[8,3,1,2,7,9,5],m(n),w(n),v(n),d(n),m0
m0=sum(x)/n;v0=sum(((x-m0)**2)/(n-1));d0=sqrt(v0)
print *,'media=',m0,' varianza=',v0,' desviacion=',d0
print *,'-----------------------------------------'
print *,'   dato     media      varianza      desviacion'
m(1)=x(1);
print *,x(1),m(1),v(1),d(1)
do i=1,n-1
    j=i+1
    m(j)=m(i)+(x(j)-m(i))/j
    w(j)=w(i)+(x(j)-m(i))*(x(j)-m(j))
    v(j)=w(j)/i;d(j)=sqrt(v(j))
    print *,x(j),m(j),v(j),d(j)
end do
end program mediadesv
```

Este programa implementa exactamente as fórmulas anteriores, pero non é computacionalmente eficiente. Unha versión máis rápida e curta, que non require tantos vectores, é a seguinte:

```
program mediadesv
real :: x(n)=[8,3,1,2,7,9,5],m
print *,'   Dato     Media       Varianza'
m=0;v=0  !m=media v=varianza
do i=1,n
    y=x(i)  ! y=dato actual
    t=y-m   ! t=diferencia
```

```
    m=m+t/i
    v=v+t*(y-m)
    print *,y,m,v
end do
d=sqrt(v/(n-1))
print *,'media=',m,' var=',v,' desv=',d
end program mediadesv
```

Exercicios propostos

1. Escribe un programa `escalar.f90` que lea un número enteiro n e dous vectores $\mathbf{v}$ e $\mathbf{w}$ n-dimensionais con valores reais (usa vectores reservados dinámicamente). O programa debe calcula-lo producto escalar (ou interior) $\mathbf{v}^T\mathbf{w} = \sum_{i=1}^{n} v_i w_i$ de ambos. Usa n=5, $\mathbf{v}$=[1,2,3,4,5] e $\mathbf{w}$=[5,4,3,2,1]:

```
program escalar
integer,allocatable :: v(:),w(:)
integer :: p=0
print '("n? ",$)'
read *,n
allocate(v(n),w(n))
print '("v[]? ",$)'
read *,v ! introducir 1 2 3 4 5 na mesma linha
print '("w[]? ",$)'
read *,w
do i=1,n
    p=p+v(i)*w(i)
end do
print '("v*w=",i0)',p
! print '("v*w=",i0)',dot_product(v,w)
deallocate(v,w)
end program escalar
```

2. Escribe un programa `norma.f90` que lea un vector estático $\mathbf{v} = (v_1, \ldots, v_5)$ con 5 valores reais. Usa bf v=[1,2,3,4,5]. O programa debe calcula-la norma ou módulo $|\mathbf{v}| = \sqrt{\sum_{i=1}^{n} v_i^2}$ de $\mathbf{v}$. Usa a función **sqrt()** para calcular a raíz cadrada.

```
program norma
integer,parameter :: n=5
real :: v(n)
print '("v[]? ",$)'
read *,v ! usa 1 2 3 4 5
x=0
do i=1,n
    x=x+v(i)**2
end do
print '("norma=",f6.3)',sqrt(x)
end program norma
```

Semana 3

Traballo en clase

1. **Producto vector-matriz-vector. Matrices dinámicas**. Escribe un programa chamado `producto.f90` que lea por teclado un número enteiro n, dous vectores $\mathbf{v}$ e $\mathbf{w}$ e unha matriz $\mathbf{A}$ de orde n, e calcule o producto

$$p = \mathbf{v}^T\mathbf{A}\mathbf{w} = [v_1 \ldots v_n] \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \ldots \\ w_n \end{bmatrix}$$

onde $\mathbf{v}^T$ denota o vector trasposto de $\mathbf{v}$ (os vectores considéranse por defecto vectores columna). Proba con $n = 3$, $\mathbf{v} = (1, 2, 1), \mathbf{w} = (-1, 0, 1)$ e $\mathbf{a}$ =(1 2 3; 4 5 6; 7 8 9) (filas separadas por ;) e tes que obter $p = 8$.

```
program producto

real,allocatable :: v(:),w(:),a(:,:),p(:)

print '("n? ",$)'; read *,n
allocate(a(n,n), v(n), w(n), p(n))

print '("v? ",$)'; read *,v
print '("w? ",$)'; read *,w
print *,'a? '
do i=1,n
    read *, (a(i,j),j=1,n)
end do

! inicializacion no programa (non necesita allocate)
! a=reshape([1,2,3,4,5,6,7,8,9],shape(a))

print *,'a='  ! imprime matriz con formato por defecto
do i=1,n
    print *,a(i,:)
end do

!p=vA
do i=1,n
    p(i)=0
    do j=1,n
        p(i)=p(i)+v(i)*a(j,i)
    end do
end do

!r=pw
r=0
do i=1,n
    r=r+p(i)*w(i)
end do

print*, "O resultado e: ", r
deallocate(a, v, w, p)


end program producto
```

Versión optimizada, que non necesita o vector $\mathbf{p}$:

```
program producto

real,allocatable :: a(:,:)
real,allocatable :: v(:),w(:)

print '("n? ",$)'; read *,n
allocate(a(n,n),v(n),w(n))
print '("v? ",$)'; read *,v
print '("w? ",$)'; read *,w
print 'a? '
do i=1,n
    read *, (a(i,j),j=1,n)
end do

r = 0
do i=1,n
    s = 0
    do j=1,n
```

```
            s = s + v(i)*a(j,i)
        end do
        r = r + s*w(i)
    end do
    print*, "O resultado e: ", r

    deallocate(a,v,w)

    end program producto
```

Versión usando funcións intrínsecas `dot_product` e `matmul` de Fortran:

```
program producto
real,allocatable :: v(:),w(:),a(:,:)

print '("n? ",$)'; read *,n
allocate(v(n),w(n),a(n,n))

print '("v? ",$)'; read *,v
print '("w? ",$)'; read *,w
print *,'a? '
do i=1,n
    read *,(a(i,j),j=1,n)
end do
print *, "vAw'=",dot_product(v,matmul(a,w))
! print *, "vAw'=",dot_product(matmul(v,a),w)  ! alternativa
deallocate(v,w,a)

end program producto
```

2. **Búsqueda dos elementos comúns a dous vectores**. Escribe un programa **comun.f90** que defina dous vectores **x**=[1,8,2,9,0,3] e **y**=[5,1,8,2,7] e mostre por pantalla os elementos comúns a ambos.

```
program comun
integer :: x(6)=[1,8,2,9,0,3],y(5)=[5,1,8,2,7]
integer,allocatable :: z(:)
allocate(z(0))
do i=1,6
    if(any(x(i)==y)) z=[z,x(i)]
end do
print *,'elementos comuns=',z
end program comun
```

3. **Aprendizaxe cooperativa na aula**. Este exercicio explicarase na clase interactiva.

Exercicios propostos

1. Escribe un programa `ocurrencia.f90` que lea por teclado un número n e un vector **v** enteiro de lonxitude n. Usa n=10 e **v**=[1,2,1,4,5,0,1,9,1,7]. O programa debe ler outro número enteiro m e mostrar por pantalla os índices das ocurrencias de m en **v**.

```
program ocurrencia
integer,allocatable :: v(:)
print '("n? ",$)'
read *,n ! n=10
allocate(v(n))
print '("v[]? ",$)'
read *,v ! usa v=1 2 1 4 5 0 1 9 1 7
print '("m? ",$)'
read *,m
print '("Ocurrencias de ",i0,": ",$)',m
k=0
do i=1,n
    if(v(i)==m) then
```

```
            print '(i0," ",$)',i
            k=k+1
        end if
    end do
    print '(": ",i0," ocurrencias")',k
    deallocate(v)
    end program ocurrencia
```

2. Escribe un programa `matricial.f90` que lea por teclado catro números enteiros n, m, p e q e dúas matrices A e B de orde $n \times m$ e $p \times q$ respectivamente, e calcule o producto matricial de ambas. O programa debe comprobar que son multiplicábeis. Usa n=2, m=3, p=3, q=2, **a**=[1 2 3;4 5 6] e **b**=[1 2;3 4;5 6], filas separadas por ";".

```
program matricial
integer,allocatable :: a(:,:),b(:,:),c(:,:)
integer :: p,q,s
print '("n,m,p,q? ",$)'
read *,n,m,p,q
if(m/=p) stop 'm debe ser igual a p'
allocate(a(n,m),b(p,q),c(n,q))
print *,'matriz a? '
do i=1,n
    read *,a(i,:)
end do
print *,'matriz b? '
do i=1,p
    read *,b(i,:)
end do
do i=1,n
    do j=1,q
        s=0
        do k=1,m
            s=s+a(i,k)*b(k,j)
        end do
        c(i,j)=s
    end do
end do
print *,'matriz a*b='
do i=1,n
    print *,c(i,:)
end do
deallocate(a,b,c)
end program matricial
```

Semana 4

Traballo en clase

1. **Análise dunha matriz. Variábeis lóxicas. Bucles nomeados. Sentenza `exit`. Vectorización. Función `all`.** Escribe un programa chamado `matriz.f90` que lea por teclado un número enteiro n e unha matriz **A** cadrada de orde n e calcule:

   - A súa traza (suma dos elementos da diagonal principal), definida pola ecuación:

$$tr(\mathbf{A}) = \sum_{i=1}^{n} a_{ii} \tag{9}$$

   - A suma dos elementos do seu triángulo superior (sen a diagonal).
   - Determine se a matriz é simétrica, é decir, se $a_{ij} = a_{ji}$ para $i, j = 1, \ldots, n$.

```
program matriz
integer,allocatable :: a(:,:)
logical :: simetrica

print '("n? ",$)';read *, n
allocate(a(n,n))
print *,"a?"
do i=1,n
    read *, (a(i,j),j=1,n)
end do

!Calculo da traza menos eficiente
! m = 0
! do i=1,n
!    do j=1, n
!        if(i==j) m = m + a(i,j)
!    end do
! end do

! calculo mais eficiente
m = 0
do i=1,n
    m = m + a(i,i)
end do
print *, "traza=", m

!suma do triangulo superior menos eficiente
!m = 0
!do i = 1, n
!   do j = 1, n
!      if(j > i) m = m + a(i, j)
!   end do
!end do
!print *, "sts=", m

!suma do triangulo superior mais eficiente
!m = 0
!do i = 1, n-1
!   do j = i + 1, n
!      m = m + a(i, j)
!   end do
!end do
!print *, "sts=",m

!suma do triangulo superior ainda mais eficiente con funcion sum()
m = 0
do i=1,n-1
   m = m + sum(a(i,i+1:n))
end do
print *, "sts= ", s

! E a matriz simetrica
simetrica=.true.
filas: do i=1,n
    do j=i+1, n
        if(a(i,j)/= a(j,i)) then
            simetrica=.false.
            exit filas
        end if
    end do
end do filas
if(simetrica) then
    print *, 'simetrica'
else
    print *, 'non simetrica'
```

```
end if

! mellor usar transpose() para transpor e all() para comprobar igualdade
!if(all(a==transpose(a))) then
! print *,'simetrica'
!else
! print *,'non simetrica'
!end if

deallocate(a)
end program matriz
```

2. **Mínimo común múltiplo de dous números enteiros. Vectores estáticos. Iteración indefinida. Subrutina. Función externa. Resto da división de dous números enteiros. Paso de vector como argumento**. Escribe un programa chamado `mcm.f90` que presente na pantalla o mínimo común múltiplo (mcm) de dous números enteiros positivos. O mcm calcúlase como o producto dos factores primos de ambos números, procedendo da seguinte maneira: se un factor primo está presente nunha das factorizacións e non na outra, inclúese no cálculo do mcm; se un factor primo está presente nas dúas factorizacións, tómase aquel que ten un expoñente maior. Usa unha función `mcm(x,y)` para calcular o mínimo común múltiplo de dous números `x` e `y`, e unha subrutina `factores(...)` para descompoñer un número en factores primos.

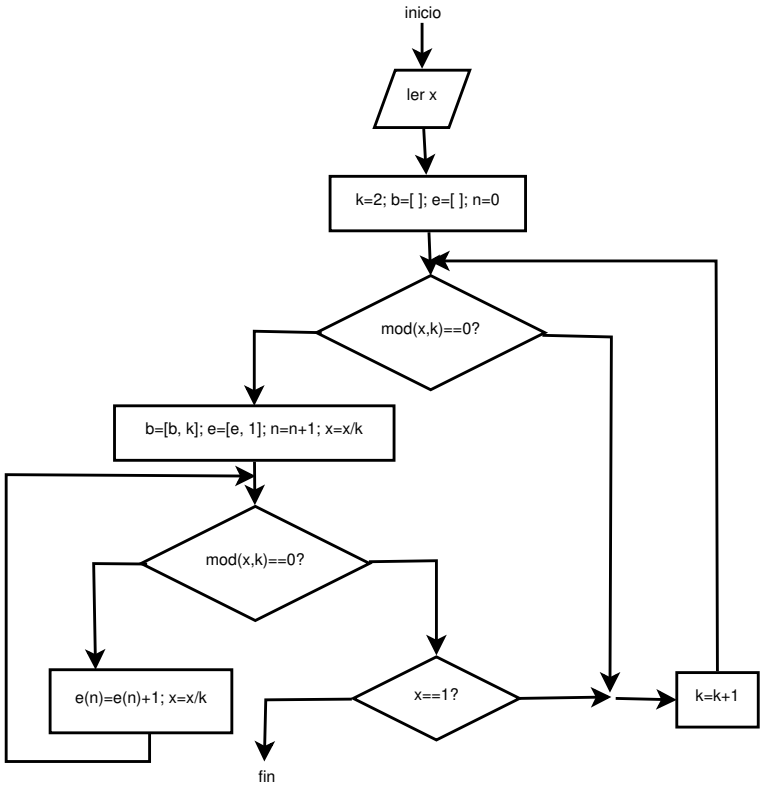


Figura 1: Diagrama de fluxo da factorización dun número enteiro.

```
! Proba con x = 120 e y = 252
! factores de 120= 2^3 * 3^1 * 5^1
! factores de 252= 2^2 * 3^2 * 7
! Factores comuns= 2^3 * 3^2 * 5^1 * 7^1
! O mcm e 2520
program principal
integer :: x,y
print '("x,y? ",$)';read *,x,y
m=mcm(x,y)
print '("mcm= ",i0)',m
end program principal
!---------------------------------
function mcm(x,y)
integer,intent(in) :: x,y
```

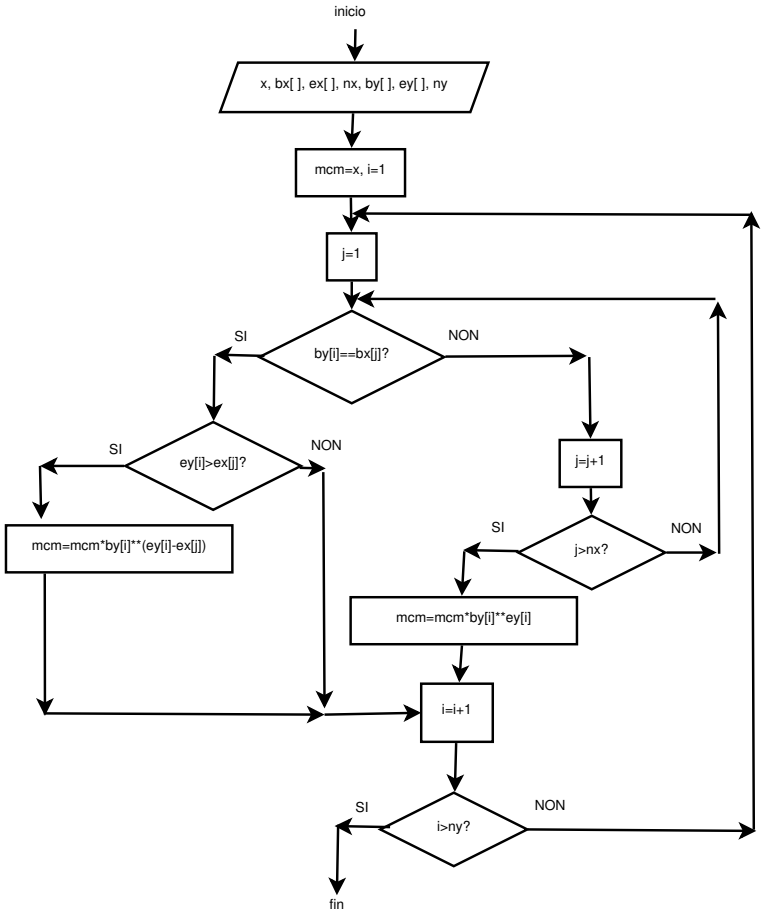


Figura 2: Diagrama de fluxo do cálculo do mcm de dous enteiros.

```
integer :: bx(100),ex(100),by(100),ey(100)
call factores(x,bx,ex,nx)
call factores(y,by,ey,ny)
mcm=x
do i=1,ny
  k=by(i);l=ey(i)
  do j=1,nx
    if(k==bx(j)) exit
  end do
  if(j<=nx) then
    if(l>ex(j)) mcm=mcm*k**(l-ex(j))
  else
    mcm=mcm*k**l
  end if
end do
end function mcm
!---------------------------------
subroutine factores(x,b,e,nf)
integer,intent(in) :: x
integer,intent(out) :: b(100),e(100),nf
k=2;nf=0;m=x
do
  if(mod(m,k)==0) then
    nf=nf+1;b(nf)=k;e(nf)=1;m=m/k
    do while(mod(m,k)==0)
      e(nf)=e(nf)+1;m=m/k
    end do
  end if
  if(m==1) exit
  k=k+1
```

```
end do
print '("factores de ",i0,"= ",$)',x
do i=1,nf
  print '(i0,"^",i0," ",$)',b(i),e(i)
end do
print *,''
end subroutine factores
```

A función externa `mcm` pódese vectorizar usando a función `any()` para ver se un factor de `y` é común e a función `findloc(vector,valor,1)` para obter o índice deste factor en `x`, aforrando así o bucle en `j`:

```
function mcm(x,y)
integer,intent(in) :: x,y
integer :: bx(100),ex(100),by(100),ey(100)
call factores(x,bx,ex,nx)
call factores(y,by,ey,ny)
mcm=x
do i=1,ny
    k=by(i);l=ey(i)
    if(any(k==bx(1:nx))) then
        j=findloc(bx(1:nx),k,1)
        if(l>ex(j)) mcm=mcm*k**(l-ex(j))
    else
        mcm=mcm*k**l
    end if
end do
end function mcm
```

3. **Máximo común divisor e mínimo común múltiplo de dous números enteiros usando o algoritmo de Euclides**. Escribe un programa en Fortran chamado `euclides.f90` que lea por teclado dous números enteiros n e m e mostre por pantalla o seu máximo común divisor (mcd) e mínimo común múltiplo (mcm). Para calcular o mcd, usa o método de Euclides, descrito neste enlace:

   https://es.wikipedia.org/wiki/Algoritmo_de_Euclides

   Supoñemos que $n > m$. Se non é así, intercambia n e m. O método de Euclides baséase en: 1) que o mcd de n e m é o mesmo que o mcd(m,r) sendo r o resto de dividir n entre m; e 2) que o mcd(n,0)=n. Deste modo, o proceso repetirase e en cada paso substituimos n por r, ata que r sexa 0, que rematas dando como mcm o valor de n. Dado que o producto do mcd e mcm é o producto $n \cdot m$, coñecido o mcd o mcm pódese calcular como $n \cdot m$/mcd. Proba con n=120 e m=252, debes obter mcm=2520 e o mcd=12.

```
program euclides
print '("n,m? ",$)'
read *,n,m
if(n<m) then
    aux=n;n=m;m=aux
end if
i=n*m
do
    r=mod(n,m)
    n=m
    if(r==0) exit
    m=r
end do
print '("mcd=",i0," mcm=",i0)',n,i/n
end program euclides
```

4. **Progreso dun programa. Formatos. Código ASCII dun carácter non imprimíbel**. Escribe un programa chamado `progreso.f90` que mostre por pantalla o progreso dun bucle como un porcentaxe na mesma liña da terminal. Podes descargar este programa desde este **enlace**.

```
program progreso
integer,parameter :: n=10000000
do i=1,n
```

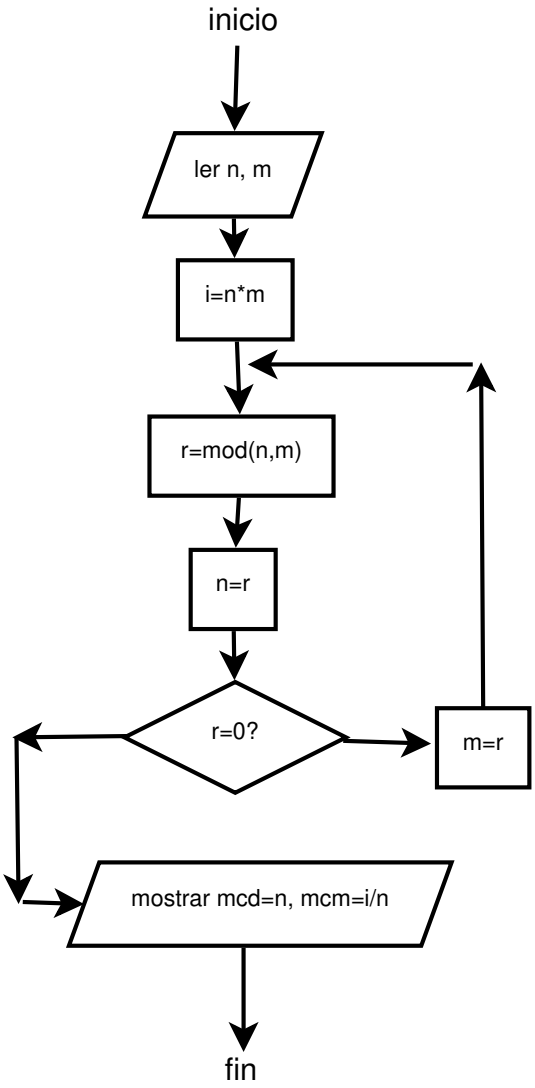


Figura 3: Pseudocódigo do cálculo de mcd e mcm usando o algoritmo de Euclides.

```
    write (*,'(1a1,f6.2,"%",$)') char(13),100.*i/n
end do
write (*,*) ''

end program progreso
```

Ampliando este programa podemos estimar o tempo que queda para que remate (**enlace**):

```
program progreso2
real(8) :: t0,t1,dt,i=1,n=50000000.  !50000000.
character(40) :: strtime
print '(a10,"   ",a)','Progreso','Tempo restante'
call cpu_time(t0)
do
    call cpu_time(t1)
    dt=(t1-t0)*(n-i)/i
    !char(13): codigo para retorno de carro
    write (*,'(1a1,f10.2,"%   ",a,$)') char(13),100.*i/n,strtime(dt)
    i=i+1
    if(i>n) exit
end do
write (*,*) ''
end program progreso2
!---------------------------------
character(40) function strtime(t) result(str)
```

```
real(8),intent(in) :: t
integer :: year,month,d,h,m,s
if(t<60) then  ! seconds in a minute
    s=floor(t)
    write(str,'(i2," s")') s
else if(t<3600) then  ! seconds in an hour
    m=floor(t/60);s=floor(t-60*m);
    write(str,'(i2," m ",i2," s")') m,s
else if(t<86400) then  ! seconds in a day
     h=floor(t/3600);m=floor((t-3600*h)/60);s=floor(t-3600*h-60*m);
     write(str,'(i2," h ",i2," m ",i2," s")') h,m,s
else if(t<2592000) then  ! seconds in a month
     d=floor(t/86400);h=floor((t-86400*d)/3600)
     m=floor((t-86400*d-3600*h)/60);s=floor(t-86400*d-3600*h-60*m)
     write(str,'(i2," d ",i2," h ",i2," m ",i2," s")') d,h,m,s
else if(t<31536000) then  ! seconds in a year
     month=floor(t/2592000);d=floor((t-2592000*month)/86400)
     h=floor((t-2592000*month-86400*d)/3600)
     m=floor((t-2592000*month-86400*d-3600*h)/60)
     s=floor(t-2592000*month-86400*d-3600*h-60*m);
     write(str,'(i2," month ",i2," d ",i2," h ",i2," m ",i2," s")')
         month,d,h,m,s
else
    y=floor(t/31536000);month=floor((t-31536000*y)/2592000)
    d=floor((t-31536000*y-2592000*month)/86400)
    h=floor((t-31536000*y-2592000*month-86400*d)/3600)
    m=floor((t-31536000*y-2592000*month-86400*d-3600*h)/60)
    s=floor(t-31536000*y-2592000*month-86400*d-3600*h-60*m)
    write(str,'(i2," y ",i2," month ",i2," d ",i2," h ",i2," m ",i2,
        " s")') month,d,h,m,s
end if
return
end function strtime
```

Exercicios propostos

1. Escribe un programa `raices.f90` que lea por teclado o grao n dun polinomio $p(x) = a_n x^n + \ldots a_2 x^2 + a_1 x + a_0$ e os seus coeficientes $a_n, \ldots, a_0$ e calcule as raíces enteiras do polinomio, tendo en conta que as posíbeis raíces son divisores do termo independente a_0. Proba con $n = 5$ e o polinomio $x^5 + 3x^4 + 2x^3 - x^2 - 3x - 2$ que ten raíces enteiras $x = \pm 1$ e $x = -2$.

```
program raices
integer,allocatable :: a(:)
print '("n? ",$)'
read *,n
allocate(a(0:n))
do i=0,n
    print '("coeficiente de x^",i0,"? ")',n-i
    read *,a(n-i)
end do
print '("Polinomio: ",$)'
print '("(",i0,a,i0,$)',a(n),')x^',n
do i=n-1,0,-1
    print '("+(",i0,")x^",i0,$)',a(i),i
end do
print *,''
print '("Raices: ",$)'
m=abs(a(0))
do i=-m,m
    if(i==0) cycle
    if(mod(m,i)==0) then
```

```
            p=0
            do j=0,n
                p=p+a(j)*i**j
            end do
            if(p==0) print '(i0," ",$)',i
        end if
    end do
    print *,''
    deallocate(a)
    end program raices
```

2. Escribe un programa `intervalo.f90` en Fortran que calcule para $x \in [-1, 4]$ os valores da seguinte función definida por intervalos:

$$f(x) = \begin{cases} 1+x & x \leq 0 \\ x & 0 < x < 1 \\ 2-x & 1 \leq x \leq 2 \\ 3x - x^2 & x > 2 \end{cases}$$

```
program intervalo
x=-1
do
    if(x<=0) then
        print *,x,1+x
    else if(x<1) then
        print *,x,x
    else if(x<=2) then
        print *,x,2-x
    else if(x<3) then
        print *,x,3*x-x*x
    else
        exit
    end if
    x=x+0.01
end do
end program intervalo
```

Semana 5

Traballo en clase

1. **Validación de datos lidos por teclado**. Escribe un programa chamado `valida.f90` que pida por teclado un número enteiro maior que 2 e valide o valor introducido, voltando a pedilo se éste non cumpre a condición.

```
program valida
do
  print '("n(>2)? ",$)'
  read *,n
  if(n>2) exit
  print *,'valor non aceptado'
end do
print '("valor ",i0," aceptado")',n
end program valida
```

2. **Cálculo de límite dunha función nun punto finito. Bucle indefinido**. Escribe un programa chamado `limite.f90` que calcule o límite:

$$\lim_{x \to 2} \frac{x^2 + x - 6}{x^2 - 4} \tag{10}$$

```
program limite
!------------------------
! version con variabel real: da warning por variable e paso non enteiros
! do x = 1, 3, 0.05
!      print *, x, (x*x+x-6)/(x*x-4)
! end do
!------------------------
! version con bucle indefinido
x=1
do
    print *, x, (x*x+x-6)/(x*x-4)
    x=x+0.05
    if (x>3) exit ! para evitar pasar de x=3
end do
end program limite
```

No VSCode, redirixe a saída do programa anterior a un arquivo executando o comando:

```
a.exe >limite.txt
```

E logo representa gráficamente esta saída co `Octave`. Para isto, vai ao menú de inicio de Windows e executa "Octave". Dentro do octave, vai á carpeta **fortran** e executa os seguintes comandos:

```
load limite.txt
plot(limite(:,1), limite(:,2))
grid on
```

Isto debe abrir unha ventá na que se mostra a gráfica de $f(x)$ para $x \in [1,3]$ tal como se mostra na figura 4. Finalmente, sae do Octave.

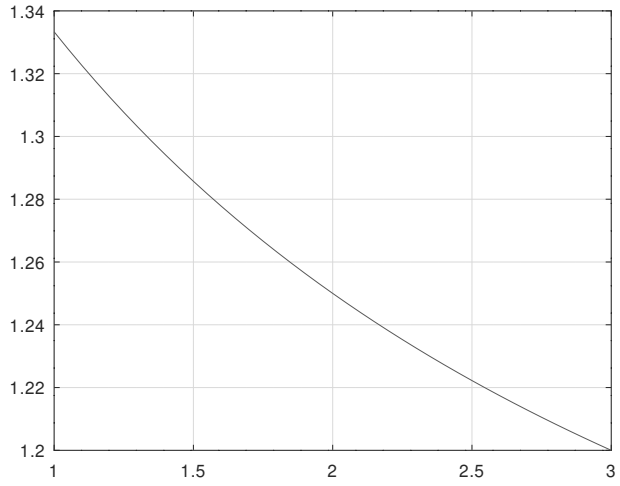


Figura 4: Representación gráfica dunha función dunha variábel usando Octave.

3. **Cálculo da derivada dunha función. Función de sentenza. Escritura en arquivo.**. Escribe un programa chamado `derivada.f90` que calcule e represente gráficamente a derivada da función $f(x) = \mathrm{e}^{-x}\,\mathrm{sen}\,2x$ no intervalo $[0, 10]$. Para isto, ten en conta, pola definición de derivada dunha función nun punto, que, usando $h = 0^+$:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \simeq \frac{f(x+h) - f(x)}{h} \tag{11}$$

```
program derivada
real, parameter :: a = 0, b = 10
f(x)=exp(-x)*sin(2*x)  ! funcion de sentenza
fp(x)=-exp(-x)*sin(2*x)+2*exp(-x)*cos(2*x)  ! derivada analitica
open(1, file="derivada.txt", status="new", err=1)
h=0.01;x=a;fx=f(x)
do
    xh=x+h;fxh=f(xh);df=(fxh - fx)/h
    write (1, *) x, fx, df, fp(x)
```

```
    if(xh > b) exit
    x=xh;fx=fxh
end do
close(1)
stop
1 stop "derivada.txt xa existe"
end program derivada
```

Para representar a función e a derivada, executa no VSCode o comando:

```
a.exe >limite.txt
```

Logo, executa o **Octave**, sitúate no directorio **fortran** e teclea os seguintes comandos:

```
x=load('derivada.txt');
subplot(2,1,1)
plot(x(:,1),x(:,2),';f(x);','linewidth',5)
subplot(2,1,2)
plot(x(:,1),x(:,3),';df(x);','linewidth',5)
hold on
plot(x(:,1),x(:,4),'r;fp(x);','linewidth',5)
```

4. **Cálculo de integrais indefinidas**. Escribe un programa chamado `primitiva.f90` que calcule a integral indefinida (primitiva) dunha función *f(x)* no intervalo $[a, b]$. Sabes que se $p(x) = \int_a^x f(t)dt$, con $a \leq x \leq b$, é unha primitiva de $f(x)$, entón $p'(x) = f(x)$. Pola definición de derivada temos que:

$$f(x) = p'(x) = \lim_{h\to 0} \frac{p(x+h) - p(x)}{h} \tag{12}$$

Se tomamos $h \simeq 0^+$ podemos aproximar:

$$f(x) \simeq \frac{p(x+h) - p(x)}{h} \tag{13}$$

e despexar na ec. anterior $p(x+h) \simeq p(x) + hf(x)$. Como coñecemos $f(x)$ e queremos a súa integral indefinida (é dicir, $p(x)$ tal que $p'(x) = f(x)$), fixando un valor inicial $p(a)$ podemos calcular $p(x), \forall x > a$. Este valor inicial $p(a)$ prefixado é equivalente á constante C que se lle pode sumar á función primitiva $p(x)$. A fórmula anterior indica que o valor novo $p(x+h)$ da primitiva calcúlase como o valor en $x+h$ da liña recta que pasa polo punto $(x, p(x))$ e ten pendente $f(x)$. Deste modo, a derivada da primitiva $p(x)$ é a función orixinal $f(x)$, como se mostra na figura 5. O valor de h debe verificar que para $x \in [a, b]$ a función $f(x)$ pode aproximarse entre x e $x+h$ por unha liña recta con pendente $f(x)$. No programa, calcula $p(x) = \int_a^x f(t)dt$ no intervalo $[a, b]$ usando $a = 0, b = 1, f(t) = t$, $p(a) = 0$. Repite o cálculo para $a = 0, b = \pi, f(t) = \operatorname{sen} t, p(a) = 0$. Se queres calcular outra integral indefinida, so tes que cambiar a, b, $p(a)$ e $f(x)$.

```
program primitiva
f(x)=x;a=0;b=1;px=0
!f(x)=sin(x);a=0;b=3.141592;px=0
open(1, file="integral.txt", status="new", err=1)
x=a;h=0.01
do
    fx=f(x)
    write (1, *) x,fx,px
    x=x+h;px=px+h*fx
    if(x > b) exit
end do
close(1)
stop
1 stop "integral.txt xa existe"
end program primitiva
```

Tamén se pode calcular a primitiva dunha función nun intervalo sen que se coñeza a súa expresión analítica pero si os seus valores nese intervalo. Supón que a separación h entre dous valores consecutivos é h=0.01. O seguinte exemplo calcula a primitiva lendo os valores dende o arquivo `valores_funcion.txt`, que podes descargar dende este **enlace**.

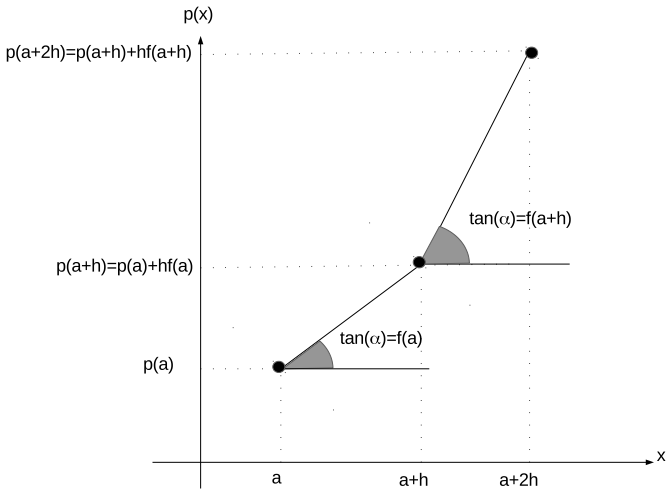


Figura 5: Aproximación numérica á primitiva $p(x)$ dunha función $f(x)$.

```
program primitiva2
open(1, file="integral.txt", status="new", err=1)
open(2, file="valores_funcion.txt", status="old", err=2)
a=0;b=1;px=0;x=a;h=0.01
do
    read (2,*,end=3) fx
    write (1,*) x,fx,px
    x=x+h;px=px+h*fx
end do
3 close(2)
close(1)
stop
1 stop "integral.txtt xa existe"
2 stop "valores_funcion.txt non existe"
end program primitiva2
```

5. **Cálculo dunha integral definida. Reais de dobre precisión**. Escribe un programa chamado `integral.f90` que calcule a integral definida funha función $f(x)$ no intervalo $[a, b]$. Prueba con $\int_{-1}^{1} \frac{\arccos x}{1+x^2} dx$. Usa reais de dobre precisión.

```
program integral
real(8) :: a=-1,b=1,h=1d-004,s=0,x,f  ! modificar a,b,h para cada caso
f(x)=acos(x)/(1+x*x)   ! modificar para cada caso
x=a
do
    s=s+f(x);x=x+h
    if(x > b) exit
end do
s=h*s
print *,'h=',h
print *,'integral=', s
print *,'valor correcto= 2.46740110027234'
print *,'diferencia=',abs(s-2.46740110027234)
end program integral
```

Compara o resultado co proporcionado polo Octave. Para isto, executa:

```
f=@(x) acos(x)/(1+x*x)
format long
quad(f,-1,1)
quit
```

Derivada, integral indefinida e integral definida dunha función dada como un vector de puntos.

A partir dos tres exercicios anteriores, consideremos unha función $f(x)$ definida no intervalo $[a,b]$ por un vector de n valores $\mathbf{f} = (f_1, \ldots, f_n)$, onde $f_i = f(x_i)$ con $x_i = a + h(i-1)$ con $i = 1 \ldots n$ e $h = \dfrac{b-a}{n-1}$. Consideraremos que o número n de puntos é suficientemente elevado como para que a función $f(x)$ poda aproximarse con precisión por unha recta entre x_i e x_{i+1} ou, equivalentemente, que h é suficientemente pequeno. Entón temos que:

- A súa derivada $d(x) = f'(x)$ pode describirse polo vector $\mathbf{d} = (d_1, \ldots, d_{n-1})$, onde $d_i = \dfrac{f_{i+1} - f_i}{h}$, con $i = 1 \ldots n-1$. É decir, a derivada calcúlase como a diferencia de dous valores consecutivos de f.
- A súa primitiva $p(x) = \int f(x)dx$ pode describirse polo vector $\mathbf{p} = (p_1, \ldots, p_n)$, onde $p_1 = p(a)$ (prefixado por nós arbitrariamente, p.ex. $p(a) = 0$) e $p_{i+1} = p_i + hf_i$ con $i = 1 \ldots n-1$. É decir, a primitiva p_i é a suma acumulativa dos valores f_i dende 1 ata i multiplicada por h.
- A súa integral definida $\int_a^b f(x)dx$ pode aproximarse pola suma $h\sum_{i=1}^{n} f_i$. É decir, é a suma de tódolos valores de f no intervalo $[a,b]$ multiplicada por h.

Exercicios propostos

1. Escribe un programa `armonico.f90` que calcule a posición $x(t)$, velocidade $v(t)$ e aceleración $a(t)$ dun móvil en movemento armónico, dado por:

$$x(t) = b\,\mathrm{sen}(\omega t + \theta) \tag{14}$$
$$v(t) = b\omega\cos(\omega t + \theta) \tag{15}$$
$$a(t) = -b\omega^2\,\mathrm{sen}(\omega t + \theta) \tag{16}$$

sendo ω= 0.1 radiáns/s, $\theta = \pi/2$ radiáns, $b = 2.5$ m para tempos $0 \leq t \leq 100$ segs. separados 1 seg. entre si. Representa gráficamente x, v, a usando Octave.

```
program armonico
real,parameter :: pi=3.141592
n=100;w=0.1;theta=pi/2;b=2.5
open(1,file='armonico.txt',status='new')
do i=1,n
    x=b*sin(w*i+theta)
    v=b*w*cos(w*i+theta)
    a=-b*w**2*sin(w*i+theta)
    print *,x,v,a
    write (1,*) x,v,a
end do
close(1)
end program armonico
! en Octave:
! datos=load('armonico.txt');
! figure(1);plot(x)
! figure(2);plot(v)
! figure(3);plot(a)
```

2. Escribe un programa `distancia.f90` que lea dous vectores $\mathbf{x}$ e $\mathbf{y}$ de dimensión n por teclado (usa vectores dinámicos) e calculen a súa distancia $|\mathbf{x} - \mathbf{y}|$ definida como:

$$d = |\mathbf{x} - \mathbf{y}| = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{17}$$

Proba con n=5, $\mathbf{x}$=[1,2,3,2,1] e $\mathbf{y}$=[5,2,1,3,4] e debes obter d=5.4772.

```
program distancia
real,allocatable :: x(:),y(:)
print '("n? ",$)'
read *,n
allocate(x(n),y(n))
print '("x[]? ",$)'
read *,x
print '("y[]? ",$)'
read *,y
print *,'distancia=',sqrt(sum((x-y)**2))
deallocate(x,y)
end program distancia
```

Semana 6

Traballo en clase

1. **Determinante dunha matriz cadrada de orde n. Subprogramas recursivos. Paso de matrices a subprogramas. Arquivos**. Escribe un programa chamado `determinante.f90` que lea dende un arquivo de texto unha matriz cadrada de orde n. Logo, o programa principal debe chamar a un subprograma recursivo `det(...)` que calcule o determinante da matriz lida usando o desenvolvemento por adxuntos da primeira fila da matriz. Proba cun arquivo chamado `matriz3.txt` que conteña a matriz [0 2 3; 4 5 6; 7 8 9] (filas separadas por ;) con determinante 3. Proba logo con outro arquivo `matriz4.txt` coa matriz [1 0 2 -1; 1 1 1 1; 3 2 0 1; 5 3 1 0], que ten determinante 4.

   Arquivo `matriz3.txt`:

   ```
   3
   0 2 3
   4 5 6
   7 8 9
   ```

   Arquivo `matriz4.txt`:

   ```
   4
   1 0 2 -1
   1 1 1 1
   3 2 0 1
   5 3 1 0
   ```

   Programa `determinante.f90`:

   ```
   program determinante
   integer,allocatable :: a(:,:)
   integer :: det
   character(100) :: nf='matriz3.txt'
   open(1,file=nf,status='old',err=1)
   read (1,*) n
   allocate(a(n,n))
   do i=1,n
       read (1,*) a(i,:)
   end do
   call imprime(a,n)
   close(1)
   m=det(a,n)
   print '("det(a)=",i0)',m
   deallocate(a)
   stop
   1 print *,'erro open ',nf
   end program determinante
   !-----------------------------------------
   recursive integer function det(a,n) result(d)
   integer,intent(in) :: a(n,n),n
   integer,allocatable :: b(:,:)
   select case(n)
   case(:0)
       print *,'erro: matriz de orde <=0';stop
   case(1)
       d=a(1,1)
   ```

```
case(2)
    d=a(1,1)*a(2,2)-a(1,2)*a(2,1)
case default
    d=0;k=1;m=n-1
    do i=1,n
        b=a(2:n,[(j,j=1,i-1),(j,j=i+1,n)])
        call imprime(b,m)
        d=d+k*a(1,i)*det(b,m);k=-k
    end do
end select
end function det
!-----------------------------------------------
subroutine imprime(a,n)
integer,intent(in) :: a(n,n),n
do i=1,n
    print *,a(i,:)
end do
print *,'-------------'
end subroutine imprime
```

2. **Persistencia dun número enteiro (descomposición en cifras)**. Escribe un programa chamado `persistencia.f90` que lea por teclado un número enteiro e calcule a súa persistencia. Para isto, o programa debe separar o número nas súas cifras e multiplicalas entre si. Este producto dividirase novamente nas súas cifras, e éstas multiplicaranse entre si, continuando o proceso ata obter un resultado dunha única cifra. A **persistencia** será o número de veces que se repetiu o proceso. `Exemplo: o número 715 ten persistencia 3 (715 ->35 ->15 ->5)`

```
program persistencia
print '("n? ",$)'; read *,n
m=n;k=0
do
  i=1
  do
    i=i*mod(m,10);m=m/10
    if(m==0) exit
  end do
  print *,i
  k=k+1
  if(i<10) exit
  m=i
end do
print '("persistencia de ",i0,": ",i0)',m,k
end program persistencia
```

Versión usando cadeas de caracteres:

```
program persistencia
character(100) :: n
print '("n? ",$)'
read *,n
k=0
do
    i=1;k=k+1
    do j=1,len_trim(n)
        read (n(j:j),'(i1)') l
        i=i*l
    end do
    print '("i=",i0)',i
    if(i<10) exit
    write (n,'(i0)') i
end do
print '("persistencia=",i0)',k
end program persistencia
```

3. **Valores únicos nun vector**. Escribe un programa **unico.f90** que defina o vector **x**=[1,2,1,3,9,0,0,9,1,9] e mostre por pantalla os seus elementos sen repeticións. Escribe outro programa **unico2.f90** que mostre os mesmos elementos, pero ordeados.

   Programa **unico.f90**:

```
program unico
integer :: x(10)=[1,2,1,3,9,0,0,9,1,9],y
n=size(x)
print *,'x=',x
print '("unique(x)=",$)'
do i=1,n
    y=x(i)
    if(all(x(i+1:n)/=y)) print '(i0," ",$)',y
end do
print *,''
end program unico
```

   Programa **unico2.f90**:

```
program unico2
integer,parameter :: n=10
integer :: x(n)=[1,2,1,3,9,0,0,9,1,9],y(n)
do i=1,n
    y(i)=NaN !para que se inserte o valor 0
end do
j=0
print *,'x=',x
print '("unique(x)=",$)'
do i=1,n
    m=x(i)
    if(all(y/=m)) then
        do k=1,j !busco o primeiro k con y(k) maior que n
            if(y(k)>m) exit
        end do
        do l=j,k,-1 !move os y(l) con l>=k unha posicion a dereita
            y(l+1)=y(l)
        end do
        y(k)=m;j=j+1 !inserto n en y(k)
    end if
end do
print *,y(1:j)
end program unico2
```

Exercicios propostos

1. **Produto escalar e matricial usando funcións intrínsecas**. Crea o programa `escalar.f90`, que defina un vector **v** e unha matriz cadrada **a**, ambos de orde 3, e calcule o producto escalar $\mathbf{v}^T\mathbf{v}$ e o producto matricial **aa**.

```
program exemplos_funcions
integer :: v(3) = [1,2,3]
integer :: a(3,3) = reshape([1,2,3,4,5,6,7,8,9],shape(a)), b(3,3)
interface
  subroutine imprime_matriz(a)
    integer,intent(in) :: a(:,:)
  end subroutine imprime_matriz
end interface
print '("v=",3(i0," "))', v
print *, "a="
call imprime_matriz(a)
print '("dot(v,v)=",i0)', dot_product(v,v)
b = matmul(a,a)
print *, "a*a="
call imprime_matriz(b)
```

```
b = transpose(a)
print *, "a^T="
call imprime_matriz(b)
end program exemplos_funcions
!---------------------------------
subroutine imprime_matriz(a)
integer,intent(in) :: a(:,:)
n=size(a,1)
do i=1,n
  do j=1,n
    print '(i0," ",$)',a(i,j)
  end do
  print *,''
end do
end subroutine imprime_matriz
```

2. Escribe un programa `vector.f90` que lea por teclado un número enteiro n, un vector **v** e unha matriz **A**, ambos de orde n. O programa principal debe chamar a un subprograma `producto(...)` (debes decidir o seu tipo e argumentos) que calcule o resultado do producto matricial **vA** (sendo **v** un vector fila). Proba con n=5, **v**=[1 2 3 4 5] e **a**=[1 2 3 4 5;6 7 8 9 8;7 6 5 4 3;2 1 2 3 4;5 6 7 8 9], filas separadas por ";".

```
program vector
integer,allocatable :: v(:),a(:,:),p(:)
print '("n? ",$)'
read *,n
allocate(v(n),a(n,n),p(n))
print '("v[]? ",$)'
read *,v
print *,'matriz a? '
do i=1,n
    read *,a(i,:)
end do
call producto(v,a,p,n)
print *,'producto v*a: ',p
deallocate(v,a,p)
end program vector
!---------------------------------------
subroutine producto(v,a,p,n)
integer,intent(in) :: v(n),a(n,n),n
integer,intent(out) :: p(n)
integer :: s
do i=1,n
    s=0
    do j=1,n
        s=s+v(j)*a(j,i)
    end do
    p(i)=s
end do
end subroutine producto
```

3. Escribe un programa `externa.f90` que xeralice `integral.f90` para calcular integrais definidas de modo que, usando funcións `external`, poida calcular a integral de calquer función (definida como función externa no programa).

```
program externa
real :: integral
external :: f,h
print *,integral(f,0.,1.)
print *,integral(h,-1.,1.)
end program externa
!---------------------------------
real function integral(g,a,b)
real,intent(in) :: a,b
integral=0;x=a;h=0.001
do
```

```
        integral=integral+g(x)
        x=x+h
        if(x>b) exit
    end do
    integral=integral*h
    end function integral
    !-------------------------------------
    real function f(t)
    real,intent(in) :: t
    f=sin(t)
    return
    end function f
    !-------------------------------------
    real function h(t)
    real,intent(in) :: t
    h=cos(t)
    return
    end function h
```

Semana 7

Traballo en clase

1. **Módulo, clase, herdanza, polimorfismo e sobrecarga de operador aritmético**. Escribe un programa `punto.f90` que defina o módulo **obxecto** cunha clase **punto** con dúas coordenadas x e y e un subprograma **mostra()** que mostre por pantalla x e y. No mesmo módulo, define unha clase **masa** derivada de **punto** que incorpore o valor m da masa e redefina o subprograma **mostra()** de modo que mostre por pantalla x, y e m. Dende o programa principal:

   a) Declara tres obxectos p, q e r da clase **punto**.

   b) Inicialíza p a un obxecto da clase **punto** e chama ao seu subprograma **mostra()**. Comproba que se executa **mostra()** da clase **punto**.

   c) Inicialíza q a un obxecto da clase **masa** e chama ao seu subprograma **mostra()**. Comproba que se executa o **mostra()** de **masa** e non da clase **punto**.

   d) Suma p e q e almacena o resultado en r. Chama ao subprograma **mostra()** de r e comproba que r é a suma de p e q.

```
module obxecto
    type :: punto
        integer :: x,y
    contains
        procedure :: mostra
    end type punto
    !-------------------------------------
    type,extends(punto) :: masa
        integer :: m
    contains
        procedure :: mostra=>mostra_masa
    end type masa
    !-------------------------------------
    interface operator(+)
        procedure suma
    end interface operator(+)
    !-------------------------------------
    contains
    !-------------------------------------
    subroutine mostra(a)
        class(punto),intent(in) :: a
        print *,'punto: x=',a%x,' y=',a%y
    end subroutine mostra
```

```
    !-------------------------------------
    subroutine mostra_masa(a)
        class(masa),intent(in) :: a
        print *,'masa: x=',a%x,' y=',a%y,' m=',a%m
    end subroutine mostra_masa
    !-------------------------------------
    type(punto) function suma(a,b)
        class(punto),intent(in) :: a,b
        x=a%x+b%x
        y=a%y+b%y
        suma=punto(x,y)
    end function suma
end module obxecto
!-----------------------------------------
program principal
use obxecto
class(punto),allocatable :: p,q,r
!-------------------------------------------------
print *,'polimorfismo'
p=punto(1,3)
print *,'p:'
call p%mostra
q=masa(4,5,6)
print *,'q:'
call q%mostra
!-------------------------------------------------
print *,'sobrecarga de operador (+) punto'
print *,'p:'
call p%mostra
print *,'q:'
q=punto(2,1)
call p%mostra
print *,'q:'
call q%mostra
r=p+q
print *,'r:'
call r%mostra
end program principal
```

2. **Sobrecarga de operador relacional**. Escribe un programa `sobrecarga.f90` que defina un módulo **mod_persoa** cun dato **persoa** que teña un carácter **nome** de lonxitude 10 e dous reais **peso** e **altura**. Sobrecarga o operador > para que unha persoa sexa maior que outra ten un meirande índice de masa corporal (peso/altura). No programa principal, declara dous obxectos da clase **persoa**, Alba con 62.5 kg e 1.84 m, e Clara con 70.1 kg e 1.98 m, e mostra por pantalla se Alba>Clara ou viceversa.

```
module mod_persoa
  type persoa
    character(10) :: nome
    real :: peso,altura
  end type persoa
  interface operator (>)
    module procedure maior
  end interface
  contains
  logical function maior(x,y)
    type(persoa),intent(in) :: x,y
    if(x%peso/x%altura > y%peso/y%altura) then
      maior=.true.
    else
      maior=.false.
    end if
  end function maior
end module mod_persoa
```

```
!----------------------------------------
program sobrecarga
use mod_persoa
type(persoa) :: x=persoa('alba',65.2,1.84)
type(persoa) :: y=persoa('clara',70.1,1.98)
if(x>y) then
  print *,x%nome,'>',y%nome
else
  print *,x%nome,'<=',y%nome
end if
stop
end program sobrecarga
```

3. **Xerador de números aleatorios**. Descarga o programa `aleatorio.f90` dende este **enlace**. Este programa imprime por pantalla 10 números aleatorios no intervalo $[a, b)$ empregando a subrutina `random_number()` de `gfortran`. Usa $a = -10, b = 10$. O programa usa a subrutina `init_random_seed()` para inicializar o xerador de números aleatorios co reloxio do sistema. Proba con e sen a chamada a esta subrutina.

```
program aleatorio
real :: x(10),m(3,3),s(3)=[0,0,0]
print '(a,$)', "introduce a,b: "
read *, a, b
rango = b - a
! call init_random_seed_clock()     ! non-reproducible
call init_random_seed_default()  ! reproducible
call random_number(x)
print '("real: ",10f8.4)', rango*x + a
call random_number(x)
print '("enteiro: ",10i5)', int(rango*x + a)
call random_number(m)
m=int(rango*m+a)
print *,'matriz de enteiros:'
do i=1,3
  print *,(int(m(i,j)),j=1,3)
end do
stop
end program aleatorio
!-------------------------------------
subroutine init_random_seed_clock()
integer :: i, n, clock
integer, allocatable :: seed(:)
call random_seed(size = n)
allocate(seed(n))
call system_clock(count = clock)
seed = clock + 37 * [ (i - 1, i = 1, n) ]
call random_seed(put = seed)
deallocate(seed)
return
end subroutine
!-------------------------------------
subroutine init_random_seed_default()
integer :: i, n
integer,allocatable :: seed(:)
call random_seed(size=n)
allocate(seed(n))
seed=0
call random_seed(put=seed)
deallocate(seed)
return
end subroutine
```

4. **Medida do tempo consumido por un programa en Fortran**. Descarga o programa `tempo.f90` dende este **enlace**. Este programa executa un bucle de 10^8 iteracións e mostra o tempo consumido:

```
program tempo
real(8) :: inicio, fin, n, i
n=1e8; i=0
print '("medindo tempo consumido por ",d8.1," iteracions ...")',n
call cpu_time(inicio)
do
    i=i+1
    if(i>n) exit
end do
call cpu_time(fin)
print '("n=",d8.1, " tempo= ",f10.4," s.")',n,fin-inicio
end program tempo
```

5. **Derivada dun polinomio**. Escribe un programa chamado `polider.f90` que lea por teclado a orde n e os coeficientes $a_0, \ldots, a_n$, dun polinomio $p(x)$. Usa un vector dinámico de $n+1$ compoñentes con índices $0, \ldots, n$. O programa debe crear o arquivo `polider.txt` (inicialmente baleiro). Logo, debe chamar n veces a un subprograma `derivada(...)`: na chamada k-ésima (con $k = 1, \ldots, n$), este subprograma debe calcula-los coeficientes da derivada k-ésima de $p(x)$. Para isto hai que ter en conta que:

$$p(x) = \sum_{i=0}^{n} a_i x^i, \quad p'(x) = \sum_{i=1}^{n} i a_i x^{i-1}$$
$$p''(x) = \sum_{i=2}^{n} i(i-1) a_i x^{i-2} \quad p'''(x) = \sum_{i=3}^{n} i(i-1)(i-2) a_i x^{i-3}$$

E, polo tanto, a derivada k-ésima do polinomio está dada por:

$$p^{(k)}(x) = \sum_{i=k}^{n} \left[\prod_{j=0}^{k-1} (i-j) \right] a_i x^{i-k}, \qquad k = 1, \ldots, n \tag{18}$$

Deste modo, o coeficiente de x^{i-k} en $p^{(k)}(x)$ para $i = k, \ldots, n$, está dado por:

$$a_i \prod_{j=0}^{k-1} (i-j) \tag{19}$$

O subprograma `derivada(...)` anterior debe engadir ao arquivo `polider.txt` os coeficientes dos polinomios derivados (un polinomio en cada liña do arquivo).

EXEMPLO: dado o polinomio $p(x) = x^4 + x^3 + x^2 + x + 1$, resulta que $n = 4$ e as derivadas do polinomio son:

$$p'(x) = 4x^3 + 3x^2 + 2x + 1$$
$$p''(x) = 12x^2 + 6x + 2$$
$$p^{(3)}(x) = 24x + 6$$
$$p^{(4)}(x) = 24$$

e polo tanto o arquivo `polinomio.txt`, logo de executa-lo programa, debe almacena-lo seguinte contido:

```
1  2  3  4
2  6  12
6  24
24
```

```
program polider
real,allocatable :: a(:)
print '("n? ",$)'; read *,n
allocate(a(0:n))
print '("a(0:n)? ",$)'; read *,a
```

```
open(1,file='polider.txt',status='new',err=1)
do k=1,n
    call derivada(a,n,k)
end do
close(1)
deallocate(a)
stop
1 stop 'erro: polider.txt xa existe'
end program polider
!----------------------------
subroutine derivada(a,n,k)
real,intent(in) :: a(0:n)
integer,intent(in) :: n,k
do i=k,n
    d=a(i)
    do j=0,k-1
        d=d*(i-j)
    end do
    write (1,'(f5.1,$)') d
end do
write (1,*) ''
end subroutine derivada
```

Exercicios propostos

1. Codificar un programa `minmax.f90` que lea dende un arquivo **minmax.txt** un número enteiro n e unha matriz **A** enteira cadrada de orde n e calcule, usando subprogramas: 1) cal das súas filas ten menor valor medio; 2) o valor máximo da dita fila. Usa n=5 e a matriz [15 19 12 19 13; 18 19 3 5 5;8 3 8 6 14; 13 11 8 15 15; 3 8 15 19 16].

```
program minmax
integer,allocatable :: a(:,:)
open(1,file='minmax.txt',status='old',err=1)
read (1,*) n
allocate(a(n,n))
do i=1,n
    read (1,*) a(i,:)
end do
close(1)
print *,'a:'
do i=1,n
    print *,a(i,:)
end do
call fila_menor_media(a,n,i,j)
print '("fila con menor media=",i0," valor maximo=",i0)',i,j
deallocate(a)
stop
1 stop 'erro open minmax.txt'
end program minmax
!-----------------------------------------------
subroutine fila_menor_media(a,n,i,j)
integer,intent(in) :: a(n,n),n
integer,intent(out) :: i,j
real(8),parameter :: inf=huge(dbl_prec_var)
s_min=inf
do k=1,n
    s=sum(a(k,:))/n
    if(s<s_min) then
        s_min=s;i=k;j=a(k,1)
        do l=2,n
            if(a(k,l)>j) j=a(k,l)
        end do
    end if
```

```
end do
end subroutine fila_menor_media
```

2. Escribe un programa `transformado.f90` que lea por teclado un número enteiro n e un vector $\mathbf{v}$ de dimensión n (usar vectores reservados dinámicamente), e calcule o vector transformado $\mathbf{w}$, tamén de dimensión n, definido por:

$$w_i = \sum_{j=1}^{i} v_j, \quad i = 1, \ldots, n \tag{20}$$

Usa n=10 e $\mathbf{v}$=[1,2,3,4,5,6,7,8,9,10].

```
program transformado
integer,allocatable :: v(:),w(:)
print '("n? ",$)'
read *,n
allocate(v(n),w(n))
print '("v[]? ",$)'
read *,v
w(1)=v(1)
do i=2,n
    w(i)=w(i-1)+v(i)
end do
print *,'w= ',w
deallocate(v,w)
end program transformado
```

3. Escribe un programa `exterior.f90` que lea por teclado un número enteiro n e logo dous vectores n-dimensionais $\mathbf{v}$ e $\mathbf{w}$. O programa debe invocar a unha subrutina chamada `calcula_producto_exterior(...)`, que calcule e proporcione como saída a matriz A resultante de multiplicar o vector columna $\mathbf{v}$ polo vector fila $\mathbf{w}$: $a_{ij} = v_i w_j; i, j = 1, \ldots, n$. O programa debe mostra-la matriz A por pantalla dende o programa principal. Usa n=5, $\mathbf{v}$=[1,2,3,4,5] e $\mathbf{w}$=[5 4 3 2 1].

$$\mathbf{v}'\mathbf{w} = \begin{bmatrix} v_1 \\ \ldots \\ v_n \end{bmatrix} [w_1 \ldots w_n] = \begin{bmatrix} v_1 w_1 & \ldots & v_1 w_n \\ \ldots & \ldots & \ldots \\ v_n w_1 & \ldots & v_n w_n \end{bmatrix}$$

```
program exterior
integer,allocatable :: v(:),w(:),a(:,:)
print '("n? ",$)'
read *,n
allocate(v(n),w(n),a(n,n))
print '("v[]? ",$)'
read *,v
print '("w[]? ",$)'
read *,w
call calcula_producto_exterior(v,w,a,n)
print *,'producto exterior v*w='
do i=1,n
    print *,a(i,:)
end do
deallocate(v,w,a)
end program exterior
!------------------------------
subroutine calcula_producto_exterior(v,w,a,n)
integer,intent(in) :: v(n),w(n),n
integer,intent(out) :: a(n,n)
do i=1,n
    do j=1,n
        a(i,j)=v(i)*w(j)
    end do
end do
end subroutine calcula_producto_exterior
```

Semana 8

Traballo en clase

1. Escribe un programa chamado `vida.f90` que codifique o **algoritmo da vida**. Mediante unha matriz cadrada de orde n=10 representarase unha poboación aleatoria inicial de individuos. Un "1" nunha compoñente da matriz representará a existencia dun individuo nesa posición, mentres ca un "_" representará a non existencia de individuo nesa posición. O número de veciños dun individuo é o que determina o seu destino na seguinte xeración. O programa debe pedir por teclado un número enteiro m entre 1 e 100 (usa o 54). Entón debe crear a matriz inicial, inicializando o xerador de números aleatorios co valor m, e xerando n^2 veces un número x real aleatorio en $[0, 1]$ de modo que cada elemento da matriz sexa "1" se $x > 0{,}5$ ou "_" en caso contrario. As regras que gobernan a evolución das sucesivas xeracións dunha poboación inicial son as seguintes:

   - Un individuo con mais de 3 veciños nas posicións máis próximas morre por superpoboación.
   - Un individuo con menos de 2 veciños máis próximos morre por aillamento.
   - Aparece un individuo en calquer posición baleira que ten exactamente 3 veciños próximos.

   Estas regras aplícanse sobre a poboación inicial para determina-la seguinte xeración, e así sucesivamente, determinando a evolución das seguintes xeracións. O programa deberá presentar no monitor a poboación inicial e as sucesivas xeracións obtidas aplicando as regras anteriores. Para visualiza-la seguinte xeración será necesario que o usuario pulse unha tecla.

```
program vida
integer,parameter :: n=10
integer :: xeracion,v !v=num. vecinhos
character(1) :: a(n,n),b(n,n),aleatorio,c
call inicializa_aleatorio(a,n)
print *,'Matriz inicial: '
call mostra(a,n)
xeracion=1; c='s'
do while(c=='s')
  do i=1,n
    do j=1,n
      v=0
      do k=-1,1
        do l=-1,1
          if(i+k>=1.and.i+k<=n.and.j+l>=1.and.j+l<=n) then
            if(a(i+k,j+l)=='1'.and.(k/=0.and.l/=0)) v=v+1
          end if
        end do
      end do
      if(a(i,j)=='1') then
        if(v>3.or.v<2) then
          b(i,j)='_'
        else
          b(i,j)=a(i,j)
        endif
      else
        if(v==3) then
          b(i,j)='1'
        else
          b(i,j)=a(i,j)
        endif
      endif
    end do
  end do
  a=b;xeracion=xeracion+1 ! copia de matriz b a matriz a
  print '("xeracion ",i0,":")',xeracion
  call mostra(a,n)
  print *, 'continuar? (s/n)'
  read *, c
end do
```

```
end program vida
!-----------------------------------------
subroutine mostra(a,n)
character(1),intent(in) :: a(n,n)
integer,intent(in) :: n
do i=1,n
  print *,a(i,:)
end do
end subroutine mostra
!-----------------------------------------
subroutine inicializa_aleatorio(a,n)
character(1),intent(out) :: a(n,n)
integer,intent(in) :: n
print *, 'Introduce un numero entre 1 e 100:'
read *, i
call srand(i)
do i=1,n
  do j=1,n
    x=rand()
    if(x>0.5) then
      a(i,j)='1'
    else
      a(i,j)='_'
    endif
  end do
end do
end subroutine inicializa_aleatorio
```

2. Escribe un programa chamado `covarianza.f90` que calcule a matriz de covarianza $\mathbf{\Sigma}$ dun conxunto de n vectores m-dimensionais $\{\mathbf{x}_i, i = 1, \ldots, n\}$, sendo $\mathbf{x}_i = (x_{i1}, \ldots, x_{im})$. O elemento ij da matriz de covarianza $\mathbf{\Sigma}$ (cadrada de orde m) defínese como:

$$\Sigma_{ij} = \frac{1}{n}\sum_{k=1}^{n}(x_{ki} - \langle x_i\rangle)(x_{kj} - \langle x_j\rangle), \quad j = 1..m \tag{21}$$

Onde $\langle x_i\rangle$ é o valor medio da compoñente i dos vectores $\mathbf{x}_k$:

$$\langle x_i\rangle = \frac{1}{n}\sum_{k=1}^{n} x_{ki} \tag{22}$$

O programa debe, dados $n = 12$ e $m = 5$, debe chamar a un subprograma onde abra o arquivo e lea os vectores (cada vector está almacenado nunha liña distinta no arquivo). Logo, debe chamar a outro subprograma que calcule as medias $\langle x_i\rangle, i = 1, \ldots, m$. Por ltimo, debe chamar a un subprograma que calcule cada elemento $\Sigma_{ij}, i, j = 1, \ldots, m$, mediante a fórmula 3. Finalmente, debe chamar a outro subprograma que imprima a matriz Σ (fila a fila). Emprega o seguinte arquivo `vectores.txt` ($n = 12$, $m = 5$).

```
1.3 0.4 1.5 0.4 1.2
1.9 0.4 1.5 0.7 1.3
1.2 0.4 0.9 0.5 1.2
1.5 0.4 2.1 0.8 1.1
1.1 0.4 2.2 0.9 1.0
1.0 0.4 2.3 0.2 0.9
0.6 0.4 2.5 0.1 0.8
1.1 1.4 1.9 0.4 0.7
0.4 0.3 1.5 0.3 0.6
0.5 1.2 1.3 0.2 0.5
0.8 1.4 1.6 0.4 0.4
1.3 0.9 1.2 0.7 0.2
```

Programa `covarianza.f90`:

```
program covarianza
integer,parameter :: n=12,m=5
real :: v(n,m),c(n,m),media(m)
call le_vectores(v,n,m)
print *,'datos='
call imprime_matriz(v,n,m)
do i=1,m
  media(i)=sum(v(i,:))/n
end do
do i=1,m
  do j=1,m
    x=0
    do k=1,n
      x=x+(v(k,i)-media(i))*(v(k,j)-media(j))
    end do
    c(i,j)=x/n
  end do
end do
print *, 'matriz de covarianza='
call imprime_matriz(c,m,m)
end program covarianza
!-------------------------------------------------------------
subroutine le_vectores(v,n,m)
real,intent(out) :: v(n,m)
integer,intent(in) :: n,m
open(1,file='vectores.txt',status='old',err=1)
do i=1,n
  read (1,*) v(i,:)
end do
close(1)
return
1 stop 'erro en open vectores.txt'
end subroutine le_vectores
!-------------------------------------------------------------
subroutine imprime_matriz(a,n,m)
real,intent(in) :: a(n,m)
integer,intent(in) :: n,m
do i=1,n
  do j=1,m
    print '(f10.3," ",$)',a(i,j)
  end do
  print *,''
end do
end subroutine imprime_matriz
```

3. **Conversión entre tipos de datos** Escribe un programa `conversion.f90` que defina n=5, x=1.23 e unha cadea de caracteres c=`'456'`. O programa debe convertir e mostrar por pantalla: 1) n a real; 2) x a enteiro; 3) redondear x a enteiro por exceso; 3) redondear x a enteiro por defecto; 4) redondear x ao enteiro máis cercano; 5) c a enteiro; 6) c a real; n a carácter; 7) x a carácter.

```
program conversion
character(10) :: c='456'
n=5;x=1.23
print *,'n->real: ',real(n)
print *,'x->enteiro: ',int(x)
print *,'x->enteiro defecto: ',floor(x)
print *,'x->enteiro exceso: ',ceiling(x)
print *,'x->enteiro mais cercano: ',nint(x)
read (c,*) i
print *,'c->enteiro: ',i
read (c,*) y
print *,'c->real: ',y
write (c,'(i0)') n
```

```
print *,'n->caracter: <',c,'>'
write (c,'(f5.2)') x
print *,'x->caracter: <',c,'>'
end program conversion
```

4. **Creación dunha libraría**. Descarga os programas **media.f90**, **mediana.f90**, **desviacion.f90**, **ordea.f90** e **principal.f90**.

   a) **Libraría estática**. Para crear unha libraría estática `libstat.a`, executa na terminal do VSCode os comandos:

   ```
   gfortran -c media.f90 mediana.f90 desviacion.f90 ordea.f90
   ar qv libstat.a media.o desviacion.o mediana.o ordea.o
   ```

   Para listar os arquivos `*.o` contidos na libraría `libstat.a` executa `ar tv libstat.a`. Para compilar o programa `principal.f90` enlazado coa libraría `libstat.a`, usa o comando:

   ```
   gfortran -L. principal.f90 -lstat
   ```

   b) **Libraría dinámica**. Para crear unha libraría dinámica `libstat.so`, executa os comandos:

   ```
   gfortran -fpic -c media.f90 desviacion.f90 mediana.f90 ordea.f90
   gfortran -shared -o libstat.so media.o desviacion.o mediana.o ordea.o
   ```

   Para listar os arquivos `*.o` contidos na libraría `libstat.so` executa `nm libstat.so`. Para compilar o programa `principal.f90` enlazado coa libraría `libstat.so`, usa o comando:

   ```
   gfortran -L. principal.f90 -lstat
   ```

   Executa o programa co comando **a.exe**.

   c) **Compilación separada**. Tamén se pode compilar separadamente, sen crear ningunha libraría, cos comandos:

   ```
   gfortran -c media.f90 mediana.f90 desviacion.f90 ordea.f90
   gfortran principal.f90 *.o
   ```

Exercicios propostos

1. Escribir un programa en Fortran chamado `gauss.f90` que resuelva un sistema de n ecuacións lineais con n incógnitas empregando o Método de Eliminación Gaussiana. Dado o sistema seguinte:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \tag{23}$$
$$\ldots \tag{24}$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \tag{25}$$

   O método de eliminación transforma este sistema no seguinte:

$$x_1 + a'_{12}x_2 + \ldots + a'_{1n}x_n = b'_1 \tag{26}$$
$$0 + x_2 + \ldots + a'_{1n}x_n = b'_1 \tag{27}$$
$$\ldots \tag{28}$$
$$0 + 0 + \ldots + x_n = b'_n \tag{29}$$

   Onde se pode despexar directamente x_n, sustituir na $(n-1)$-ésima ecuación e despexar x_{n-1} e así sucesivamente ata calcula-las n incógnitas. As únicas transformacións permitidas son:

   - Dividir tódolos elementos dunha fila polo mesmo número.
   - Sumar a tódolos elementos dunha fila o producto dun escalar polo elemento correspondente doutra fila

   Arquivo **sistema.txt**:

```
3
1 0 2 1
-1 1 2 -2
0 1 1 3
```

O programa `gauss.f90` é o seguinte:

```
! x +   + 2z = 1
!-x + y + 2z = -2
!     y + z  = 3
! x = 5, y = 5, z = -2
program gauss
real,allocatable  :: a(:,:),x(:)
print *, "sistema inicial:"
open(1,file='sistema.txt',status='old')
read (1,*) n
m=n+1
allocate(a(n,m),x(n))
do i=1,n
  read (1,*) a(i,:)
  print *,a(i,:)
end do
close(1)
call verifica(a,n)
do i=1,n
  do j=i,n
    if(a(j,i)/=0) then
      t=a(j,i)
      do k=1,m
        a(j,k)=a(j,k)/t
      end do
    end if
  end do
  do j=i+1,n
    if(a(j,i)/=0) then
            a(j,:)=a(j,:)-a(i,:)
    end if
  end do
  print *, "pasada", i, "-esima"
  do k=1,n
    print *,a(k,:)
  end do
end do
do i=n,1,-1
  x(i)=a(i,m)
  do j=i+1,n
    x(i)=x(i)-a(i,j)*x(j)
  end do
  print *, "x(", i, ") = ", x(i)
end do
stop
end program gauss
!-------------------------------------------
subroutine verifica(a, n)
real,intent(inout) :: a(n,n)
integer, intent(in) :: n
m=n+1
do i=1,n
  if(a(i,i)==0) then
    do j=1,n
      if(a(j,i)/=0) exit
    end do
    if(j==m) then
      print *, "incognita", i, "ten todolos coeficientes nulos"
      stop
    end if
    a(i,:)=a(i,:)+a(j,:)
  end if
    print *,a(i,:)
```

```
end do
return
end subroutine verifica
!-----------------------------------------------
subroutine le_sistema(a,n)
real,intent(out) :: a(10,11)
integer,intent(in) :: n
print *, "sistema inicial:"
open(1,file='sistema.txt',status='old')
do i=1,n
  read (1,*) (a(i,j),j=1,n+1)
  print *,(a(i,j),j=1,n+1)
end do
close(1)
return
end subroutine le_sistema
```

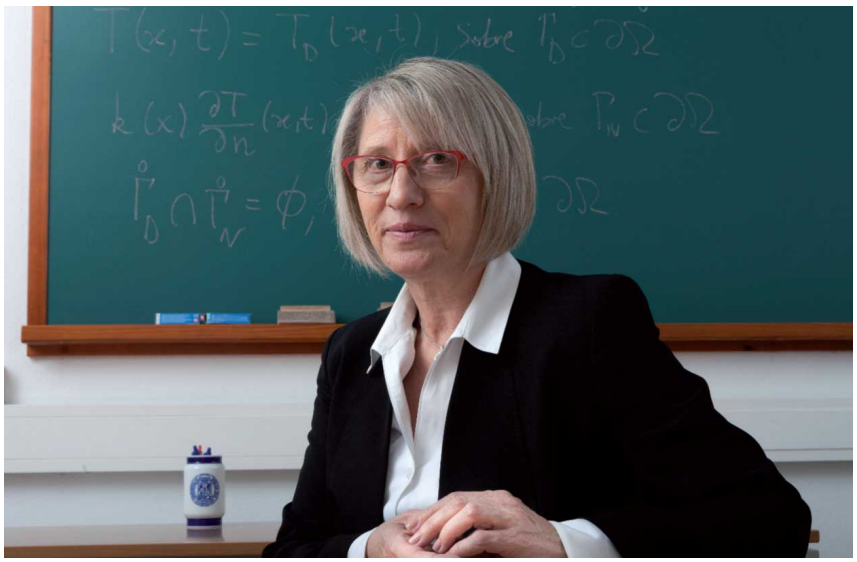
2. Escribe un programa `matvec.f90` que lea por teclado un número enteiro n, un vector $\mathbf{v}$ e unha matriz $\mathbf{A}$, ambos de orde n. O programa debe calcula-lo resultado do producto matricial $\mathbf{Av}$ (sendo $\mathbf{v}$ un vector columna). Usa n=5, $\mathbf{v}$=[1 2 3 4 5] e $\mathbf{a}$=[1 2 3 4 5;6 7 8 9 8;7 6 5 4 3;2 1 2 3 4;5 6 7 8 9].

```
program matvec
integer,allocatable :: v(:),a(:,:),p(:)
print '("n? ",$)'
read *,n
allocate(v(n),a(n,n),p(n))
print '("v[]? ",$)'
read *,v
print *,'a[;]? '
do i=1,n
    read *,a(i,:)
end do
do i=1,n
    j=0
    do k=1,n
        j=j+a(i,k)*v(k)
    end do
    p(i)=j
end do
print *,'p=Av=',p
deallocate(v,a,p)
end program matvec
```

CÁLCULO NUMÉRICO

EN OCTAVE/MATLAB

Peregrina Quintela Estévez (1960)

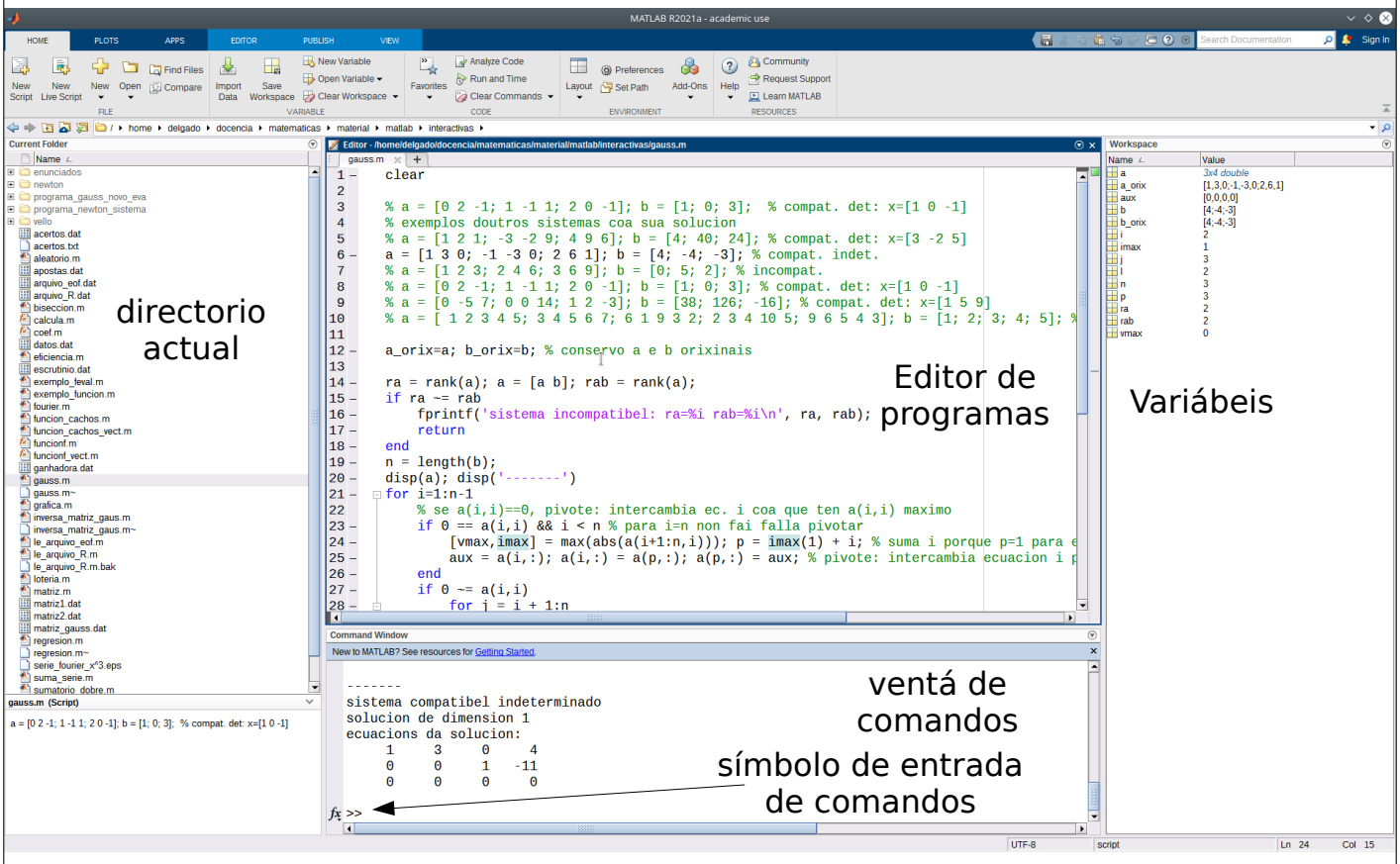


- Catedrática de Matemática Aplicada da USC
- Directora do Instituto Tecnolóxico de Matemática Industrial da USC
- Escritora de varios libros sobre Matlab

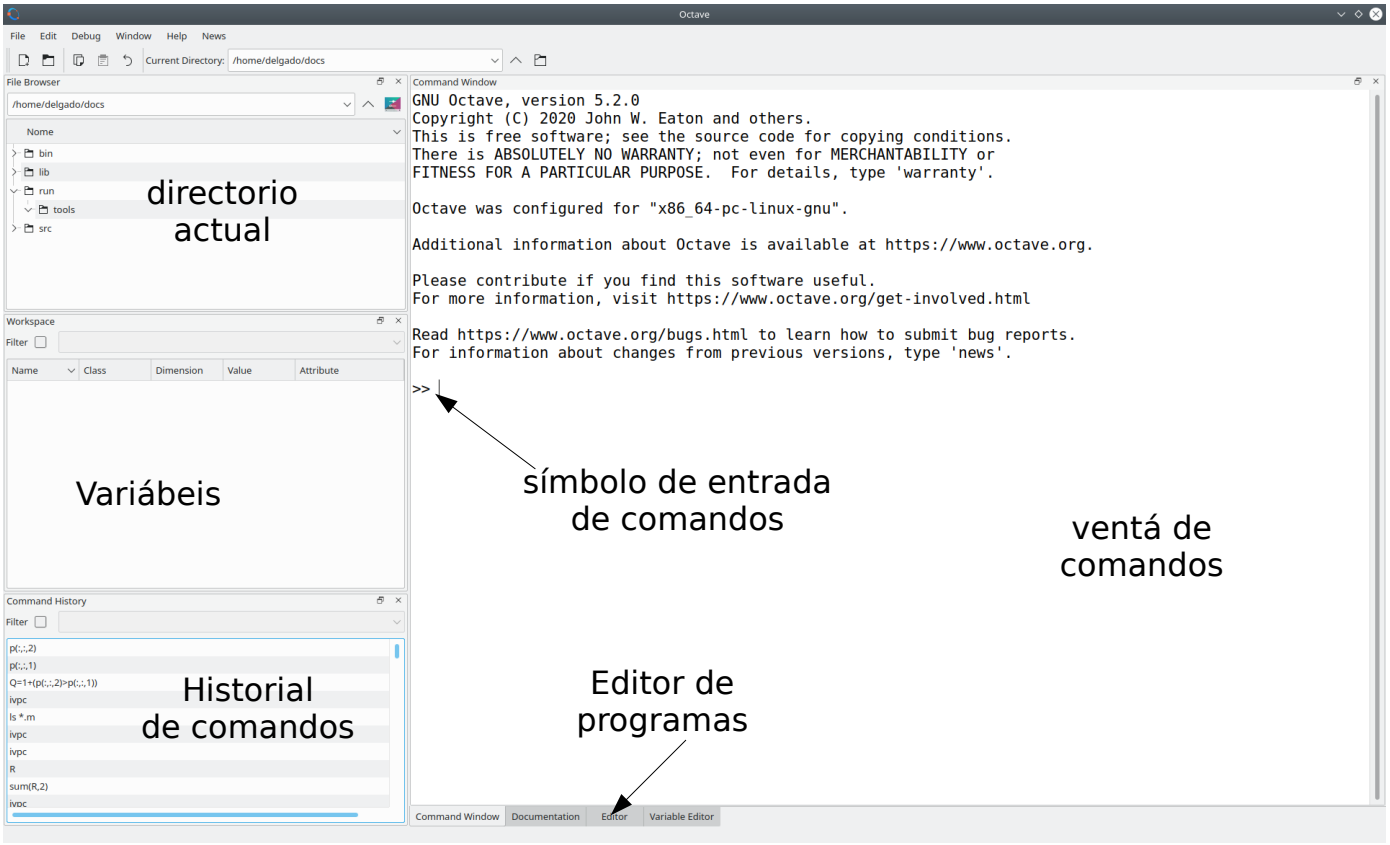
Características de Matlab

- Linguaxe de cálculo científico e numérico, visualización e programación
- Octave: versión libre de Matlab
- Librerías de funcións moi amplas
- Cálculos matemáticos
- Desenvolvemento de algoritmos
- Análise e representación gráfica de datos
- Simulación
- Desenvolvemento de interfaces de usuario

Interface gráfica de Matlab



Interface gráfica de Octave



Comandos básicos

- Execución de operacións: *ans* é unha variábel predefinida que almacena o resultado da última operación (se éste non se almacena noutra variábel)
- Comando rematado en ; non mostra o resultado
- Repetición de comandos anteriores: ↑
- *clc*: limpia a ventá de comandos
- *clear*: borra a memoria (workspace)
- Pódense encadear varias ordes con ;

  x=-1:0.1:1;plot(sin(x))

Variábeis (I)

- Variábeis: non hai declaración, só hai que asignarlle un valor; antes desta asignación, non existe, e non pode ser referenciada (erro)
  - Enteiras e reais (con / sen expoñente)
  - Complexas: i,j= unidade imaxinaria: 2+2*i;
- Os nomes poden conter letras, números e o signo "_", pero só poden comezar por letras. Non poden ter signos especiais (+&%$(/?*, etc.). Matlab distingue entre maiúsculas e minúsculas
- Almacénanse internamente como reais de dobre precisión (8 bytes, 16 cifras decimais, rango $\pm 10^{\pm 308}$)

Variábeis (II)

- Comando *diary*: almacena a historia de comandos
- *diary ficheiro.txt*: comeza a almacenar en *ficheiro.txt*
- *diary off*: remata o almacenamento
- Variábeis predefinidas: *ans*, *pi*, *eps* (menor diferencia entre números=-1.2E-16), *inf* (∞), *i*, *j*, *NaN* (Not a Number: 0/0), *realmax/realmin* (nº real máximo e mínimo)
- Asignación de valor a unha variábel: *x = 5.4;*
- Cadeas de caracteres: entre comiñas simples: s = 'cadea de caracteres'

```
>> whos s
  Name      Size                Bytes  Class
  s         1x5                    10  char array
```

Funcións básicas (I)

- *sqrt*, *abs* (valor absoluto), *exp*, *log*, *log10*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *sinh*, *cosh*, *tanh*, *asinh*, *factorial*
- Redondeo de real a enteiro: *round* (cara enteiro máis cercano), *fix* (ídem cara 0), *floor*(ídem cara -inf), *ceil* (ídem cara +inf)
- Exemplo: a= [-1.9 –0.2 3.4 5.6 7 2.4+3.6i]
  round(a) →[-2 0 3 6 7 2 +4 i]
  fix(a) → [-1 0 3 5 7 2 +3 i]
  floor(a) →[-2 -1 3 5 7 2 +3 i]
  ceil(a) → [-1 0 4 6 7 3 +4 i]
- *conj(z)*: conxugado dun nº complexo z

Funcións básicas (II)

- *real*(z), *imag*(z): partes real e imaxinaria de z
- *factorial*(x): factorial de nº enteiro
- *rem*(x, y): resto de división enteira x/y
- *rats*(x): aproxima x polo nº racional máis cercano
- *factor*(x): factores primos dun nº enteiro
- *isprime*(x): determina se x é primo
- *primes*(x): números primos menores que x
- Poden operar sobre vectores e matrices (operan elemento a elemento)

Formatos e operacións aritméticas

- Formatos: comando *format*:
  - *short* (*short e*): 5 decimais (exponencial)
  - *long* (*long e*): 15 decimais (exponencial)
  - *compact*: suprime liñas en branco
- Operacións aritméticas: +-*/^ (*power(x,y)*=x^y; *nthroot(x,y)*=$\sqrt[y]{x}$).
- Prioridades: as usuais: ^ */ +-
- Axuda: *help/doc comando*, tecla F1
- Tempos: *tic* (inicializa reloxio) e *toc* (mide o tempo transcurrido dende *tic*); *cputime*, *etime*, *clock*

Vectores

- Almacenamento de comandos en ficheiro: *diary ficheiro.txt*; *diary on*; *diary off*.
- Definición entre corchetes: *v = [1 2 3]*: *e*lementos separados por espacios ou comas. Separación entre filas mediante ;
- Vector columna: *v = [1;2;3]*
- Trasposición dun vector: *v*'
- Definición con compoñentes equiespaciadas (v_{i+1}-v_i=*cte*) nun intervalo [a,b]: *v=a:paso:b* (por defecto *paso*=1): *v= 0:0.1:1*: elementos de 0 a 1 separados 0.1
- *linspace(a,b,n)* *n*=lonxitude
- Vector con compoñentes logarítmicamente espaciadas: *v = logspace(a,b,n)*: *n* mostras logarítmicamente equiespaciada entre 10^a e 10^b: ($log_{10}x_{i+1} - log_{10}x_i$=*cte* independente de *i*):

$$\log_{10} v_i = \frac{(b-an)+i(a-b)}{1-n}; i=1,\ldots,n$$

Acceso e edición dun vector

- Acceso a elementos dun vector:
  - v(1) elemento nº 1
  - v(*end*) último elemento
  - v(1:5) elementos de 1 a 5
  - v(1:2:10) elementos de 1 a 10 de 2 en 2
  - v(:) o vector completo
  - v(*1*:*end*~=*k*): o vector menos o elemento *k*-ésimo
- Adición / supresión de elementos:
  - Adición de elementos: *v* = [*v* 5 6]. Tamén: *v*=1:3; *v*(6)=9
  - Concatenación de vectores; *v*=[1 2 3];*w*=[4 5 6]; *z=[v w]* ou *z=[v' w']*
  - Borrar elementos: *v*(5:8)=[]; *v*(*v*>2)=[]; *v*(*rem*(*v*,2)==0)=[];

Funcións con vectores (I)

- Lonxitude dun vector: *length(v)*; nº elementos: *numel(v)*
- Producto escalar de 2 vectores: *dot(v,w)*, *v***w*' ou *sum(v.***w)*
- Lectura de vector/matriz dende arquivo: *load datos.dat;* ou ben v=*load('datos.dat')*.
- Almacena en vector/matriz *datos* ou *v*. O arquivo debe conter unha matriz numérica (non *char*). Tódalas liñas coa mesma cantidade de valores.
- Suma/producto de elementos dun vector: *sum(v)*, *prod(v)*
- *min(v)* e *max(v)*: valores mínimo e máximo dun vector.

[vmax imax] = max(v): valor máximo e índice do máximo

[~,imax]=max(v): só o índice do máximo

Funcións con vectores (II)

- *sort(v): ordea un vector por orde crecente (con matrices, ordea cada columna); sort(v, 'descend') ->* orde decrecente;
- [v2,i]=sort(v): v2=vector ordeado, i=vector cos índices dos elementos de v ordeados
- *mean(v)*, *var(v)*, *std(v), median(v)*: media, varianza, desviación típica e mediana.
- *unique(v)*: elementos non repetidos de *v* ordeados (crecente).
- Invertir un vector: *flip*(1:4) -> 4 3 2 1
- Intersección entre dous vectores: *intersect*([1 2 3 4],[1 3 7]) -> [1 3]
- Unión de dous vectores: *union*([1 3],[1 2]) -> [1 2 3]

Matrices

matriz=[elem 1ª fila; ...; elems. nª fila]

- *a = [1 2 3; 4 5 6; 7 8 9]* : matriz 3x3: columnas separadas por ;
- Tipo da matriz (nº filas e cols): *whos a*

Name	*Size*	*Bytes*	*Class*
a	*3x3*	*72*	*double array*

Ollo: podes escribir a(1,2) ou a(5), onde o índice incrémentase por columnas

- *eye(m, n)*: matriz identidade; *eye(n)*: cadrada identidade
- *zeros(m, n)*: matriz *mxn* con ceros; *3+zeros*(*m*,*n*): con 3s
- *ones(m, n)*: matriz *mxn* con 1s; *5*ones(m, n)*: con 5s
- *rand(m, n)*: con valores reais aleatorios en [0, 1]

  *a + (b – a)*rand(m, n)* : aleatorios en [a, b]
- *randi([m n],nf,nc)*: matriz *nf* x *nc* con valores enteiros aleatorios entre *m* e *n; randi(n,nf,nc): valores entre 1 e n*
- Inicializa xerador de números aleatorios:
  - *rng('default')* ← Sempre igual (reproducible)
  - *rng('shuffle')* ← Distinto (co tempo) (non reproducible)

Octave:
rand('*seed*',0)
rand('*seed*','*reset*')

Acceso a elementos dunha matriz e inserción / borrado de filas e columnas

- a(1,2): elemento 1ª fila e 2ª columna
- a(5): elemento 5º percorrendo por columnas
- a(1,:): elementos da 1ª fila
- a(:, 2): elementos da 2ª columna
- *a(1:2,2:3)*: [a_{12} a_{13}; a_{22} a_{23}]
- *a = rand(10,10); a(1:2:5, 2:3:10)*

$$\begin{bmatrix} a_{12} & a_{15} & a_{18} \\ a_{32} & a_{35} & a_{38} \\ a_{52} & a_{55} & a_{58} \end{bmatrix}$$

- Adición dunha fila: *a=ones(3);a=[a; [1 2 3]]*. Ou ben: a=ones(3);a(5,:)=[1 1 1]
- Adición de columna: *a=[a [1;2;3]]* ou a(:,6)=zeros(3,1)
- Supresión dunha fila: *a(1,:)=[]*
- Supresión dunha columna: *a(:,1)=[]*

Funcións con matrices (I)

- Tamano: *[nf nc] = size(a); nf = size(a,1); nc=size(a,2);*
- *numel(a)*: nº elementos de matriz; *length(a)*: *max*(*nf*,*nc*)
- Matriz diagonal cun vector: *v=[1 2 3];a= diag(v)*
- Vector coa diagonal dunha matriz: *v = diag(a)*
- *diag(diag(a))*: matriz diagonal coa diagonal de *a*
- *magic(n)*: matriz cadrada máxica de orde *n* (igual suma de filas e columnas).
- Suma de elementos por columnas: *sum(a)* ou *sum(a,1); por filas*: *sum(a,2)* ou *sum(a')*; suma completa: *sum(a(:))* ou *sum(sum(a))*
- Producto de elementos por columnas/filas: *prod(a)/prod(a,2)*
- Triángulo superior / inferior: *triu(a)* / *tril(a)*

Transformación de matriz en vector ou matriz doutra orde

- Útil para transformar unha matriz nun vector e procesar os seus elementos cun so bucle, evitando bucles dobres.
- Conversión de matriz *a* en vector fila por columnas: *v=a(:)'*. Se queres por filas: *b=a';b*(:)'
- Función *reshape(matriz,nf,nc)*: transforma a matriz noutra de orde *nf* x *nc* por columnas. O número *nf* x *nc* debe ser igual ao nº de elementos de *a*.
- Se queres que sexa por filas: *reshape(a',nf,nc);*
- Transformar de matriz a vector por filas: *v=reshape(a,1,nf * nc);* Se queres por columnas: *v=reshape(a',1,nf * nc);*
- Transformar *a* -> *b* (con *n* filas): *b=reshape(a,n,[]);* o número *n* debe ser divisor do nº de elementos de *a*; o nº de columnas será o necesario para almacenar os *nf* x *nc* elementos de *a* nunha matriz *b* de *n* filas.

Repetición dunha matriz

- Función *repmat*(*a*,*n*,*m*): repite a matriz a *n* veces verticalmente e *m* veces horizontalmente
- Ex: *a=[1 2;3 4]*

  repmat(a,2,3): repite a matriz dúas veces verticalmente e 3 horizontalmente:

  1 2 1 2 1 2
  3 4 3 4 3 4
  1 2 1 2 1 2
  3 4 3 4 3 4

Operacións con matrices (I)

- Operacións por compoñentes: punto antes do operador: *a.*b, a./b, a.^b*: ambas matrices deben coincidir en nº de filas e de columnas
- Operacións matriz-escalar:
  - Suma / resta / produto / cociente con escalar: tódolos elementos da matriz se operan co escalar
  - Cociente escalar-matriz por compoñentes: $b=k./a \rightarrow b_{ij} = k/a_{ij}$
  - Potencia escalar-matriz por compoñentes: $b=k.\hat{}a \rightarrow b_{ij} = k^{aij}$
  - Potencia matriz-escalar por compoñentes: $b=a.\hat{}k \rightarrow b_{ij} = a_{ij}^{k}$
  - Potencia matriz-escalar matricial: $b=a\hat{}k$ ($a \cdot \ldots \cdot a$, a debe ser cadrada)

Operacións con matrices (II)

- Operacións entre matrices:
  - Suma $a + b$ e resta $a - b$: a e b deben coincidir en nº de filas e de columnas
  - Producto matricial: $a*b$: o nº de columnas de a debe coincidir co nº de filas de b
  - Producto por compoñentes: $c=a.*b$: a e b deben coincidir en nº filas e de columnas, e $c_{ij} = a_{ij} \cdot b_{ij}$
  - División matricial a esquerda: $a\backslash b \rightarrow a^{-1} \cdot b \rightarrow pinv(a)*b$
  - División matricial a dereita: $a/b \rightarrow a \cdot b^{-1} \rightarrow a*pinv(b)$
  - Cociente por compoñentes: $c=a./b \rightarrow c_{ij} = a_{ij}/b_{ij}$
  - Potencia por compoñentes: $c=a.\hat{}b \rightarrow c_{ij} = a_{ij}\hat{}b_{ij}$

Funcións con matrices (II)

- *unique(a)*: elementos non repetidos de *a* ordeados (crecente) como vector columna.
- Determinante dunha matriz cadrada: *det(a)*.
- Inversa dunha matriz cadrada: *inv(a)*, so cando $det(a) \neq 0$.
- Pseudoinversa de Moore-Penrose $a^{\dagger}$: *pinv(a)*, existe para matrices non cadradas e cadradas con $det(a)=0$.
- Autovalores dunha matriz cadrada: *v=eig(a)*
- Autovectores: *[v d] = eig(a)*

  v= matriz con autovectores de matriz *a* por columnas

  d=matriz diagonal con autovalores de matriz *x*: $det(a - d_{ii}\mathbf{1}) = 0$; $a \cdot v_i = d_{ii} v_i$ (v_i= columna *i* de *v*), $i=1,\ldots,n$

Funcións con matrices (III)

- Mínimo e máximo:
  - Por columnas: *min(a)* e *max(a)*
  - Por filas: *min(a,[],2)* ou *min(a')*, menos eficiente
  - Matriz completa: *min(a(:))* ou *min(min(a))*
- Valores mínimos/máximos e índices dos elementos mín/máx:
  - Por columnas: *[v,i]=min(a)*
  - Por filas: *[v,i]=min(a,[],2)*
  - Matriz completa: *[v,i]=min(a(:))*
  - *v*: vector con valores mínimos
  - *i*=vector con índices de elementos mínimos
- Índices de fila e columna do elemento mínimo/máximo dunha matriz:

  [~,i]=min(a(:)); [f,c]=ind2sub(size(a),i)

Funcións con matrices (IV)

- Media, varianza desviación típica e mediana:
  - Por columnas: *mean(a), var(a)*, *std(a*), *median(a)*:
  - Por filas: *mean(a,2), var(a,[],2)*, *std(a,[],2*), *median(a,2)*
  - Matriz completa: *mean(a(:)), var(a(:)), std(a(:)), median(a(:))*
- Ordeamento:
  - Por columnas: *sort(a)*, *sort(a,'descend')*
  - Por filas: *sort(a,2), sort(a,2,'descend')*
  - Matriz completa: *sort(a(:))*, *sort(a(:),'descend')*
- Matrices dispersas (moitos elementos nulos): *a = sparse(i, j, c, m, n); full(a)*: mostra matriz; i(j) = índices de filas (columnas) de elementos non nulos; c= vector con valores de elementos non nulos; m(n)= nº filas(columnas)

Sistema de ecuacións lineais

- Sexa o sistema en forma matricial **b** = **Ax**, con *n* ecuacións e *n* incógnitas
- Resolución en Matlab:

  $a = [a_{11} \ldots a_{1n}; \ldots; a_{n1} \ldots a_{nn}];$
  $b = [b_1; \ldots; b_n];$
  rank(a)
  rank([a b])
  x = a\b % se rank(a)==rank([a b])
  *x = inv(a)*b % alternativa*

  rango dunha matriz: se *rango(a)==n*, o sistema é compatíbel determinado; se *m=rango(a)<n*, o sistema é compatíbel indeterminado (se *rango([a b])=m*) ou incompatíbel (se *rango([a b])≠m*)
- Se o sistema é **incompatíbel**, a pseudoinversa *x=pinv(a)*b* permite atopar unha solución de norma mínima, é dicir, *norm(a*x-b)* é mínima, aínda que ≠0 e pode ser elevada

x+2y+3z=0
2x+4y+6z=5
3x+6y+9z=2

rank(*a*)=1
rank([*a b*])=3

Non existe **x** con **Ax**-**b**=**0**
x=**A**$^{+}$**b** verifica |**Ax**-**b**| é mínima

Sistema compatíbel indeterminado (I)

- As infinitas solucións pódense escribir como unha **solución individual do sistema** *(x0)* máis unha **combinación linear de solucións do sistema homoxéneo** *(k·c)* asociado.
- **Solución individual**: *x0=pinv(a)*b*. Podes comprobar que é solución calculando *norm(A*x0-b)*.
- **Solucións do sistema homoxéneo**: cerne da aplicación linear asociada á matriz dos coeficientes: *k=null(a)* retorna unha matriz onde cada columna é un vector dunha base ortonormal deste subespazo linear (nulo).
- **Solución xeral** do sistema indeterminado: *x0+k*c*, sendo *c* o vector de coeficientes da combinación linear (p.ex. *c=ones(r,1)*, sendo *r=size(k,2)* a dimensión do espazo solución (nº columnas de *k*).

Sistema compatíbel indeterminado (II)

- Sistema *ax=b c*on *a=[1 2 3;4 5 6;7 8 9]; b=[1;2;3]*
- *rank(a)=2,rank(b)=2<3: s*istema compatíbel indeterminado
- A solución ten dimensión 3-2=1 (é unha recta en $\mathbb{R}^3$)
- *x0=pinv(a)*b* : solución individual
- *k=null(a)*: *k*=vector columna director da recta
- Solucións da forma *x=x0+c*k* onde *c*=escalar
- O vector *x* é solución porque *norm(a*x-b)* $\simeq 0$

Exercicios

1) Define un vector x con 10 compoñentes espaciadas logarítmicamente entre 1 e 100; suprímelle as compoñentes 3-5; engádelle o vector [3 4 5] polo comezo; selecciona os elementos de índice múltiplo de 3; calcula a lonxitude de x
2) Calcula a suma, producto, máximo e mínimo, media e desviación típica do vector x do exercicio anterior
3) Crea co editor de Matlab un arquivo de *datos.dat*. Cárgao en Matlab ao vector x e representa as dúas columnas de x
4) Define os vectores (1,2,3,4,5) e (5,4,3,2,1) e calcula o seu producto escalar

Exercicios

5) Crea unha matriz 3x3 con elementos=1 e outra 2x2 con elementos=5. Logo pégaas e obtén a seguinte matriz:
6) Define unha matriz de orde 3x4 con números aleatorios no intervalo [−1, 1]:
7) Dada unha matriz A cadrada de orde 5: selecciona a submatriz de A coas filas 2-3 e as columnas 1-3; amplía a matriz engadíndolle unha fila ao comezo da matriz; bórralle as filas 1 e 4
8) Dadas as matrices A e B:
   calcular A·B, A^{-1}·B, A·B^{-1}, |A|, suma,min e max por columnas de A; triángulo superior e inferior de B

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 0 & 0 & 5 & 5 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 4 & 5 \\ -3 & 2 & 1 \\ 0 & 5 & 4 \end{bmatrix}$$

Exercicios

9) Define unha matriz dispersa 10x10 con valores non nulos (8,1)=-3 e (3,4)=-9

10) Define unha matriz 5x5 con valores aleatorios no intervalo [-3,1]

11) Partindo da matriz identidade 7x7 e usando o : obtén a seguinte matriz:

$$\begin{bmatrix} 2 & 2 & 2 & 0 & 5 & 5 & 5 \\ 2 & 2 & 2 & 0 & 5 & 5 & 5 \\ 3 & 3 & 3 & 0 & 5 & 5 & 5 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 4 & 4 & 7 & 0 & 9 & 9 & 9 \\ 4 & 4 & 7 & 0 & 9 & 9 & 9 \\ 4 & 4 & 7 & 0 & 9 & 9 & 9 \end{bmatrix}$$

12) Crea unha matriz 5x7 coa 1ª fila 1 2 3 4 5 6 7, 2º fila 8 9 10 11 12 13 14, 3ª fila 15-21, etc. A partir dela crea outra matriz 3x4 coas filas 2-4 e columnas 3-6 da matriz orixinal

13) Resolve este sistema de ecuacións:

$$\begin{aligned} -44x+10y+16z&=20 \\ 10x-43y+6z+12t&=0 \\ 16x+6y-30z+8t&=12 \\ 12y+8z-34t&=-40 \end{aligned}$$

Solucións aos exercicios (I)

1) x=logspace(1,2,10);x(3:5)=[];x=[3 4 5 x]; x(3:3:10);length(x) ou size(x,2)

2) sum(x);prod(x);max(x);min(x);mean(x); std(x)

3) x=load('datos.dat');plot(x(:,1),x(:,2),'o-')

*4) x=[1 2 3 4 5]; y=[5 4 3 2 1];dot(x,y);x*y'*

*5) a=ones(3,3);b=5*ones(2);[[a zeros(3,2)] ; [zeros(2,3) b]]*

*6) a=-1+2*rand(3,4)*

7) a=magic(5);a(2:3,1:3);a=[ones(1,5); a];a(1:3:4,:)=[];

Solucións aos exercicios (II)

*8)a=[1 2 3; 0 1 2; 0 0 1]; b=[1 4 5; -3 2 1; 0 5 4];a*b; a\b ou inv(a)*b; a/b ou a*inv(b); det(a);sum(a);min(a);max(a);a-tril(a);a-triu(a)*

9)a=sparse([8 3],[1 4],[-3 -9],10,10); full(a)

*10)a=-3+4*rand(5)*

11)a=eye(7);a(1:2,1:3)=2; a(3,1:3)=3; a(1:3,5:7)=5; a(5:7,1:2)=4; a(5:7,3)=7; a(5:7,5:7)=9

12)a=zeros(5,7);x=1;for i=1:5; for j=1:7; a(i,j)=x; x=1+1;end; end; b=a(2:4,3:6)

13)a = [-44 10 16 0; 10 -43 6 12; 16 6 -30 8; 0 12 8 -34]; b = [20;0;12;40];a\b

Margaret Hamilton (1936)

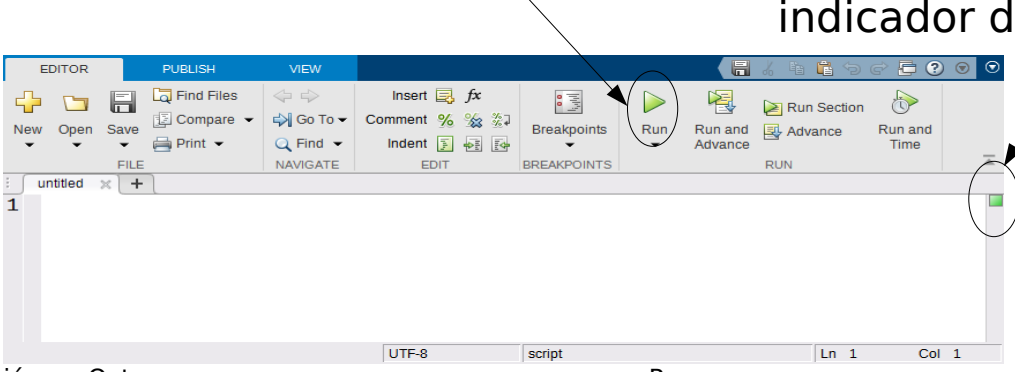
- Primeira enxeñeira do software
- Desenvolveu o **software de navegación** para o programa espacial EEUU (años 60)
- Xefa da equipa de programación da NASA para a viaxe á lúa
- Medalla Presidencial de Liberdade (2016)

Programas

- Ficheiros coa extensión .m: conteñen comandos que se executan secuencialmente
- Execución: escribe o seu nome (sen .m) na ventá de comandos
- O arquivo debe estar no directorio actual (ou nun directorio incluido na variábel *path*, que se pode consultar co comando *path*)
- Pódese engadir directorios a *path* no menú *File* submenú *Set Path* ou co comando *addpath(dir)*
- Execución alternativa: na ventá de directorio, seleccionar arquivo e *Run* (F5) no menú contextual
- Ou dende o editor de Matlab, menú *Debug -> Run*

Edición do programa

- Botón *New* en barra botóns: abre o editor de Matlab (dende o cal tamén se pode executa-lo programa)
- Execución: botón *Run* (F5) no editor ou escribi-lo nome do programa + Intro na ventá de comandos
- Permite depura-lo programa durante a execución
- Comentarios con %



Execución do programa (I)

- Matlab: linguaxe interpretado (o programa necesita ao Matlab para executarse)
- Non hai erros de compilación (non hai compilación): só erros de execución e lóxicos
- Matlab só atopa un erro de sintaxe cando executa o programa e chega ao erro
- Nembargantes, o editor de Matlab indica con cores vermello, laranxa e verde se o programa ten erros (vermello), advertencias (laranxa) ou é correcto (verde) antes de executalo
- Pode haber erros aínda que o indicador sexa verde

Execución do programa (II)

- Co Matlab dende a terminal de comandos de Linux (ponlle *exit* ao final do programa para que o Matlab retorne á terminal):

  matlab -nosplash -nodesktop -r programa

- Dentro do Octave: escribe o nome do programa sen a extensión *.m*
- Co Octave dende a terminal de comandos de Linux:

  octave programa.m

- Outra forma co octave dende a terminal de Linux:

  1) Engade a liña *#!/usr/bin/octave* ao comezo do programa para indicarlle ao *bash* que interprete o programa co Octave.
  2) Na terminal (*bash*), executa *chmod u+x programa.m* para darlle permiso de execución.
  3) Executa *programa.m* dende a terminal.

Depurador de Matlab

- No editor de Matlab, abre o programa a depurar
- Establece un punto de ruptura (breakpoint) pulsando no marxe esquerdo na liña desexada
- Executa o programa (F5 ou menú Debug->Run no editor): detense a execución no punto de ruptura
- Podes inspecciona-las variábeis poñendo o rato sobre o variábel no programa (ou no workspace)
- Podes executa-lo programa sentenza a sentenza con F10 (ou menú Debug->Step)
- Entra nunha función: F11 ou Debug->Step into
- Continua a execución: F5 ou Debug->Continue

Variábeis e entrada de datos

- O programa usa as variábeis globais (as do workspace)
- Recomendábel executar *clear* ao comezo do programa (borra as variábeis existentes)
- Entrada de datos por teclado:

```
var = input('introduce un valor: ');
```

```
cadea = input('introduce unha cadea: ', 's');
```

- Exemplo:

```
x = input('introduce x: ');
```

- Podes introducir un vector/matriz entre corchetes.

Saída de información por pantalla

- Comando *disp*:

  disp(var); disp('mensaxe de texto');

- Comando *fprintf*:

  fprintf('formato', var1, ..., varN);

  - 'formato': cadea con caracteres, e tamén ...
  - códigos de formato (ver páx. seguinte)
  - secuencias de control: \n para nova liña, \t para tabulador, \r para retorno de carro, \b para borrar un carácter impreso, \\ (carácter '\'), %% (carácter '%'), \” (carácter “);

Códigos de formato

- **%c**: carácter simple
- **%s**: cadea de caracteres
- **%i** ou **%d**: enteiro; **%5i** para enteiro de ancho 5.
- **%f**: nº real sen expoñente; %.6f: con 6 decimais; %10.3f: con ancho 10 (incluindo o punto decimal) e 3 cifras decimais
- **%e**: nº real en formato exponencial; %10.2e: real exponencial con ancho 10 e 2 decimais (tamén con %n.dE, neste caso E no expoñente)
- **%g**: nº real na forma máis compacta entre e/f
- Exemplo: *x=3.5;t=5; s='ola';*
  fprintf('x= %.3f x=%.2e t= %4i s=%s\n', x, x, t, s);

Función *fprintf*

- *fprintf* está vectorizada: se unha variábel é vector ou matriz, repítese a función até que se imprimen tódolos elementos (por columnas) na mesma liña, agás que se poña \n.
- Ex: *x = [1 2; 3 4]; fprintf('x=%i\n', x);*
  x=1 % en cada liña por ter \n
  x=3
  x=2
  x=4
- Ex: *fprintf('%i ', x);*
  1 3 2 4 % na mesma liña por non ter \n

Función *sprintf*

- *fprintf* tamén permite almacenar nun arquivo
- A función *sprintf* opera igual que *fprintf* pero non mostra a cadea por pantalla, senón que retorna a cadea formateada, para logo facer cousas con ela:

  s=sprintf('cadea formato', var1, ..., varn)
- Útil cando se quere manipular cadeas (concatenar con outras, etc.)
- Ex: *s = sprintf('x= %i y= %f\n', x, y);*
  mensaxe=[s ' ' sprintf('a= %c\n', a)];
  disp(mensaxe)

Estrutura de selección básica

- Similar a IF de Fortran
- Avalía unha condición definida polos operadores >, >=, <, <=, ==, ~=

```
if x<=0
    disp('baixo);
elseif x<=1
    disp('medio');
else
    disp('alto');
end
```

- Sentenza IF-ELSE IF:

```
if condición1
    sentenzas1;
elseif condición2
    sentenzas2;
...
else
    sentenzasN;
end
```


Estrutura iterativa básica

- Definida:

```
for var = ini:paso:fin
    sentenzas;
end
```

Similares a *do* definido e indefinido e *do-while* de Fortran

```
for k = 1:10
    fprintf('k= %i\n', k);
end
```

- Indefinida:

```
while condición
    sentenzas;
end
```

```
n=0; suma=0;
while n ~= -1
    n=input('introduce n (-1 para rematar)');
    suma = suma + n;
end
```

Remate dun programa

- *return*: remata a execución do programa (ou retorna dende unha función)
- *error*('*mensaxe*'): remata a execución e mostra a *mensaxe* de erro en cor vermella (que pode estar formateada como con *fprintf*):
  - Ex: *error('erro: x=%i < 0!\n', x)*
- *break*: remata un bucle *for*/*while* (análogo a *exit* en Fortran) cando se cumpre unha condición
- *break* remata a execución se está fora dun bucle
- Se usas *exit,* remata o Matlab (isto é útil se executas o programa dende a terminal de comandos)

Frances Allen (1932-2020)

- Pioneira na optimización de compiladores, optimización automática de código e programación paralela
- Creadora de linguaxes de programación e códigos de seguridade para a NSA
- Premio Turing de Informática en 2006

Operadores relacionais

- Operadores relacionais: > (maior) >= (maior ou igual) < (menor) <= (menor ou igual) == (teste igualdade), ~= (teste desigualdade)
- Se comparamos escalares, o resultado é 1 (certo) se se cumpre o teste ou 0 (falso) se non se cumpre
- Se comparamos matrices (deben ser da mesma dimensión en filas e columnas), a comparación faise elemento a elemento
- O resultado é unha matriz de 0s (nos elementos onde falla o teste) e 1s (nos elementos onde se cumpre) coa mesma dimension cas orixinais
- Precedencia: todos teñen a mesma, e avalíanse de esquerda a dereita

Operadores lóxicos

- NOT Lóxico: ~: P. ex: ~*x* da 1 se *x* é 0, e 0 se x é distinto de 0
- AND lóxico: & (para vectores/matrices), && (para escalares)
- OR lóxico: | (para vectores/matrices), || (para escalares)
- Operandos numéricos (0 é falso, $\neq 0$ é certo)
- Con escalares, dan 0 ou 1; con matrices, operan elemento a elemento e dan unha matriz da mesma orde
- Se actúan cun escalar e unha matriz, cada elemento da matriz opérase co escalar

Función lóxica *all*

- *all(x)*: retorna 1 se tódolos elementos do vector *x* son non nulos, 0 se algún elemento de *x* é nulo
- *all*(*a*), *all*(*a*,*1*): vector de lonxitude *size*(*a*,*2*), ou sexa, nº de columnas de *a*, con valores 1 nas columnas de *a* con tódolos elementos non nulos e 0 nas restantes
- *all*(*a*,*2*): vector de lonxitude *size*(*a*,*1*), nº de filas de *a*, con valores 1 nas filas de *a* con tódolos elementos non nulos e 0 nas restantes
- *all*(*all*(*a*)) ou *all*(*a*(:)): retorna un número, que é 1 se tódolos elementos de *a* son non nulos, ou 0 se a ten algún elemento nulo
- *all* pode aplicarse a unha expresión: *all(rem(a,2)==0)* vale 1 se tódolos elementos de *a* son pares

Funcións lóxicas *xor* e *any*

- *xor(a,b)*: retorna 1 se un dos operandos é 0 e o outro non, ou viceversa; con vectores/matrices opera elemento a elemento
- *any(x)*: retorna 1 se algún elemento do vector *x* é non nulo
- *any*(*a*), *any*(*a*,1): vector de lonxitude *size*(*a*,*2*) con valores 1 nas columnas de *a* con alomenos un elemento non nulo e 0 nas restantes
- *any(a,2)*: vector de lonxitude *size*(*a*,*1*) con valores 1 nas filas de *a* con alomenos un elemento non nulo e 0 nas restantes
- *any(any(a))* ou *any(a(:))*: actúa para toda a matriz *a*
- *xor* ou *any* poden aplicarse sobre expresións: *any(a>2)*

Función *find*

- *find(v)*: retorna os índices dos elementos non nulos do vector *v*
- *find(v>0 & v<5)*: índices dos elementos no intervalo [0,5]
- *v(v>0)* ou *v(find(v))* => elementos non nulos
- *[i,j]=find(rem(a,2)==0)*: índices de fila (*i*) e columna (*j*) dos elementos pares da matriz *a*
- *find(v>0,1,'first')*: índice do primeiro elemento positivo de vector *v*
- *find(isprime(v),3,'last')*: índices dos 3 últimos elementos primos de *v*

Sentenzas de selección (I)

As condicións elabóranse con operadores relacionais e lóxicos

- Sentenza IF:

```
if condición
    sentenzas;
end
```

- Cunha soa sentenza:
  if condición; sentenza; end

- Sentenza IF-ELSE

```
if condición
   sentenzas1;
else
   sentenzas2;
end
```

- Sentenza IF-ELSE IF:

```
if condición1
    sentenzas1;
elseif condición2
    sentenzas2;
...
else
    sentenzasN;
end
```

if condición; sentenza1; else; sentenza2; end

Sentenzas de selección (II)

- Sentenza *switch*:
- Compara a expresión cos distintos valores *val1*... e executa as sentenzas asociadas ao valor co cal coindice
- Se non coincide con ningún valor, execútanse *sentenzasN* (isto é opcional, pero permite aforrar un caso)

```
switch expresión
case val1
    sentenzas1;
case val2
    sentenzas2;
...
otherwise
    sentenzasN;
end
```


Sentenza de iteración definida (I)

- Se *x* é un vector fila, *a* é unha matriz:

```
for i=x
    sentenzas
end
```

```
for i=a(:)'
    sentenzas
end
```

Poño a(:)' para que sexa un vector fila, porque a(:) é un vector columna

- Nas sentenzas, *i* percorre os elementos do vector *x* ou da matriz *a* (por columnas)

```
for i=x
    disp(i)
end
```

```
for i=a(:)'
    disp(i)
end
```

- Para percorrer *a* por filas:

```
b=a';
for i=b(:)'
    disp(i)
end
```

Sentenza de iteración definida (II)

- O vector x pode ser da forma *ini:paso:fin*

```
for var = ini:paso:fin
    sentenzas;
end
```

Cando so é unha sentenza

```
for var=ini:paso:fin; sentenza; end
```

- Pódense usar variábeis *i*, *j* (por defecto son a unidade imaxinaria: *i*^2=-1)
- Inicializa *var = ini*
- En cada iteración executa as sentenzas
- Se *var + paso > fin* rematan as iteracións.
- En caso contrario, executa *var=var+paso* e continúa coa seguinte iteración

Sentenza de iteración definida (III)

- Se *paso > 0*, entón debe ser *ini <= fin* para que haxa iteracións; se *paso < 0*, debe ser ini >= *fin*
- A *var* pódenselle asignar valores específicos. Ex:

```
for k = [1 3 6 -4]
    fprintf('k=%i\n', k);
end
```

- Non se lle debe cambia-lo valor a *var* dentro do bucle *for* (Matlab non o detecta)
- Exemplo: suma da serie $\sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$

```
x=input('x?');n=input('nº de elementos? '); suma=0;
for k=0:n
    suma=suma+(-1)^k*x^(2*k+1)/factorial(2*k+1);
end
```

Bucle *for* con varios índices

- O *for* pódese poñer con dous ou tres índices (deben ser vectores fila)

```
i=1:3; j=4:6;
for k=[i;j]
      disp(k)
end
```

→

```
Iter. 1: k=[1;4]
Iter. 2: k=[2;5]
Iter. 3: k=[3;6]
```

```
i=1:3; j=4:6;
for k=[i;j]
      fprintf('%i %i\n',k(1),k(2))
end
```

→

```
1 4
2 5
3 6
```

- Con tres índices (caracteres):

```
x='abc'; y='def'; z='ghi';
for k=[x;y;z]
      fprintf('%c %c %c\n',k(1),k(2),k(3))
end
```

→

```
a d g
b e h
c f i
```


Exemplos de estruturas iterativas definidas

- Mostrar por pantalla un vector na mesma liña:

```
v=randi([vmax vmin],1,n);
fprintf('%7.3f ',v);fprintf('\n')
```

← Estrutura iterativa implícita (vectorizada)

- Mostrar por pantalla unha matriz (unha fila en cada liña):

```
a=randi([vmin vmax],n,m);
for i=1:n
     fprintf('%10.2f ',a(i,:)); fprintf('\n')
end
```

- Cálculo do máximo dunha serie de números (para o mínimo debes inicializar *m=inf*):

```
m=-inf;
for i=1:n
      x=input('x? ');m=max(m,x);
end
```

Se os números están almacenados nun vector *x*: *min*(*x*), *max*(*x*)

Sentenza de iteración indefinida

- Sentenza *while* (iteración indefinida): ten unha condición para continuar coa iteración

```
while condición
    sentenzas;
end
```

```
while condición;sentenza; end
```

- A condición debe ter alomenos unha variábel (se é nula, a condición é falsa, e certa en caso contrario)
- As variábeis da condición deben inicializarse antes (en caso contrario, erro de execución)
- Dentro das sentenzas débese modifica-lo valor de alomenos unha das variábeis da condición: en caso contrario, entrará nun bucle infinito

Sentenza de iteración indefinida

- As modificacións nestas variábeis deben garantir que a iteración acade un final: en caso contrario, iteración infinita (isto non o detecta Matlab)
- Non poñer condicións de igualdade estricta entre variábeis con valores reais, xa que o redondeo poden levar a que nunca se cumpran
- Ex: suma de serie $\sum_{n=0}^{\infty} \frac{x^n}{n!}$ con sumandos > 0.0001

```
suma = 0;sumando = 1;n = 0;x=1;
while sumando > 0.0001
   sumando = x^n/factorial(n);
   suma = suma + sumando; n = n + 1;
end
```

```
n=1:1000;
sum(x.^n./factorial(n))
```

Versión vectorizada

Sentenzas *break* e *continue*

- Sentenza *break*:
  - Provoca o remate inmediato da estrutura iterativa, se está dentro dun bucle *for* ou *while*
  - Se non está dentro dun bucle, remátase o programa (p.ex., se se introducen datos inválidos)
- Sentenza *continue*: provoca o paso inmediato á seguinte iteración, saltando a execución do que resta da iteración actual
- O *continue* debe estar dentro dunha estrutura de selección (para que so se execute cando se cumpra a condición)

Exemplo de *break* en bucle dobre con matrices

- ¿Cómo rematar un bucle dobre?
- Exemplo: busca un elemento impar nunha matriz:

```
clear all
% a=matriz nxn enteira aleatoria en {1..10}
n=5;a=randi(10,n,n);impar=0;
for i=1:n
    for j=1:n
        if rem(a(i,j),2)==1
            impar=1;break
        end
    end
    if impar; break end
end
fprintf('elemento (%i,%i) con valor %g é impar\n',i,j,a(i,j));
```

Podes vectorizalo: *any(rem(a(:),2)==1)*
disp(find(rem(a(:),2)==1,1,'first'))

Sentenzas de iteración híbrida

- Teñen un número máximo de iteracións (parte definida) pero poden rematar antes se se cumpre unha condición (parte indefinida).
- Pódese facer cun *for+break* ou cun *while+and* lóxico.
- Exemplo: executa *n* iteracións pero remata cando *s>10*:

```
s=0;
for i=1:n
   s=s+x(i);
   if s>10
      break
   end
end
```

```
s=0;i=1;
while s<=10 && i<=n
   s=s+x(i);i=i+1;
end
```

Outro exemplo: método de Newton

```
x1=input('x1? ');i=1;dif=inf;
while dif>=1e-5 && i<=n
   x2=x1-f(x1)/df(x1);i=i+1;
   dif=abs(x2-x1);x1=x2;
end
```

```
x1=input('x1? ');
for i=1:n
   x2=x1-f(x1)/df(x1);
   if abs(x2-x1)<1e-5
      break
   end
   x1=x2;
end
```

- O *while+and* é útil para controlar os índices dun vector ou matriz, evitando superar os rangos dos seus índices.

Exemplo: Conversión dun vector por filas/columnas a unha matriz

- Manualmente (conversión por filas): vector *v* de *n* elementos, matriz *a* de orde *mxm*, con *m=ceil(sqrt(n))*

```
clear all
n=10;v=randi(10,1,n);m=ceil(sqrt(n));a=zeros(m);k=1;
for i=1:m
      for j=1:m
            a(i,j)=v(k);k=k+1;
            if k>n; break; end   % para evitar sairse do vector
      end
      if k>n; break; end
end
disp(v);disp(a)
```

- Para convertir por columnas: *a(j,i)=v(k)*
- Automáticamente: *reshape([1 2 3 4],2,2)* por columnas; *reshape([1 2 3 4],2,2)'* por filas. O vector debe ter exactamente o mesmo nº de elementos que a matriz

Exemplo: Conversión dunha matriz nun vector

- Manualmente: matriz *a* de orde *n*x*n* a vector *v* de $m=n^2$ elementos (conversión por filas):

```
clear all
n=5;a=randi(10,n,n);m=n^2;v=zeros(1,m);k=1;
for i=1:n
    for j=1:n
        v(k)=a(i,j);k=k+1;
    end
end
disp(a);disp(v)
```

- Para convertir por columnas: *v(k)=a(j,i)*
- Automáticamente: *a*(:), como vector columna; *a*(:)', como vector fila; *reshape(a,1,m)* por columnas; *reshape(a',1,m)* por filas.

Ann Zeilinger Karakristi (1921-2016)

- Especialista en criptografía con ordenadores da NSA (axencia de seguridade nacional dos EEUU)
- 2ª guerra mundial (contra Japón) e a guerra fría
- Directora adxunta da NSA (1980)
- Premio de servizos civís distinguidos dos EEUU

Funcións

- Pode estar nun arquivo *.m* co mesmo nome que a función:

```
function [ret1,...,retN] = nome(arg1,...,argM)
%nome: liña de axuda (liña H1)
%texto da axuda
sentenzas da función;
ret1=...;...;retN=...;
end
```

Se so hai un valor devolto, os corchetes sobran

- Pode chamarse dende calquera programa se o directorio onde se atopa está no path
- *arg1, ..., argM*: argumentos de entrada
- *ret1, ..., retN*: valores devoltos pola función
- A función pode estar tamén logo do programa principal. Esta función é **local**, é decir, non se pode chamar dende outros arquivos *.m*

```
clear
v=randi(20,1,10);
w=f(v);disp(w)

function y=f(x)
y=x.^2;
end
```


Estructura dunha función

- Cando chamamos a unha función, Matlab búscaa no arquivo actual, nos arquivos da carpeta actual e nos arquivos das carpetas do *path*
- Debe haber sentenzas que asignen valores a *ret1*, ..., *retN*
- En **octave**, a función tamén pode estar no programa principal, pero antes da chamada á función
- Se a función comeza na primeira liña do arquivo, éste será considerado un arquivo de función, e non un programa

```
clear

function y=f(x)
y=x.^2;
end

v=randi(20,1,10);
w=f(v);disp(w)
```

Exemplo de función

Función que calcule $\frac{1}{n}\sum_{k=1}^{n}\cos(kx)$ dados *x*, *n*

```
function y = f(x, n)
y = 0;
for k=1:n
    y = y + cos(k*x);
end
y=y/n;
%y = mean(cos((1:n)*x));
end
```

Título da función, variábel devolta e argumentos

Dáselle o valor á variábel devolta

Versión vectorizada

Exemplo de función (varios valores devoltos)

```
function [b x np ni] = funcionf(a)
[nf nc]=size(a);
b = zeros(nf,nc); x = zeros(1,nf);
for i=1:nf
   for j=1:nc
      b(i,j)=a(i,j)*a(j,i); % b_ij=a_ija_ji
   end
   x(i) = a(i,:)*a(:,i); %prod. escalar fila i por columna i
end
i=mod(a(:),2);
np = sum(i==0); %nº elementos pares
ni = sum(i==1); % nº elementos impares
end
```

Función con varios valores devoltos

- Se non imos empregar algún dos valores devoltos pola función, na chamada á función podemos non almacenalos en ningunha variábel. Ex:
  - Corpo da función:

```
function [z t] = calcula(x,y)
   z=x+y;t=x*y;
end
```

  - Chamada a función: *[~, b]=calcula(3,4);*
- Neste exemplo, o primeiro valor devolto non se almacena en ningunha variábel.

Chamada á función

- A función pode ser chamada dende a ventá de comandos, dende outra función ou dende un programa (*script*)
- Chamada á función:

[var1 ... varN] = f(arg1, ..., argM);

Ex: a = randi(10,4,4);
[b x np ni] = f(a);

- Tamén se pode chamar á función dende unha expresión ou dende a chamada a outra función:

z = y + f(arg1, ..., argM);

fprintf('y = %f\n', sin(f(arg1, ..., argM)));

Función con varios puntos de retorno

- Sentenza *return*: provoca o retorno inmediato da función:

```
function y=le_arquivo(fich)
f=fopen(fich,'r');
if -1==f
    y=NaN;return
end
n=fscanf(f,'%g',1);y=zeros(n);
fclose(f);
end
```

- Pode haber varias sentenzas *return* (que poden estar asociadas a diferentes valores retornados) na mesma función.

Función con argumentos opcionais

- Matlab almacena o número de argumentos na variábel *nargin* (non tes que definila).

```
function z=exemplo(x,y)
   if nargin==2
      z=x+y;
   else
      z=2*x;
   end
end
```

```
function z=exemplo(varargin)
   if numel(varargin)==2
      z=varargin(1)+varargin(2);
   else
      z=2*varargin(1);
   end
end
```

- Chamada á función: *z=exemplo(2)* ou *z=exemplo(2,3)*
- Alternativa: argumento (vector) chamado *varargin*: tes que comprobar a súa lonxitude dentro da función.

Variábeis globais

- As variábeis dunha función son **locais** (non se coñecen fóra da función)
- Na función non se pode acceder ás variábeis do *workspace* nin doutras funcións
- Para acceder a unha variábel do *workspace* na función, hai que declarala **global** na función:
- Para acceder a unha variábel da función dende fóra (*workspace* ou outra función), hai que poñela como **global** onde se quera usar

```
x=5;y=6;
z=f(y)

function z=f(y)
global x
z=f+x
end
```

```
global z
x=5;
y=f(x)

function y=f(x)
global z
z=5;y=z+x
end
```


Paso dunha función como parámetro a *feval*

- Dende a mesma sentenza (que debería estar dentro dun bucle), *feval* permite chamar a unha función distinta en cada iteración do bucle
- O nome da función chamada por *feval* pode ser un carácter ou unha referencia a función (ver máis adiante)

[ret1 ... retM]=feval('nome función', arg1, ..., argN);

```
function y=nome(f,x)
...
y=eval(f, x);
end
```

nesta función o argumento *f* é unha función que en cada chamada a *nome(...)* pode ser distinta

Permite facer o mesmo que as funcións *external* en Fortran

Exemplo de *feval*: integral definida

Cálculo da integral indefinida dun vector de funcións (varias funcións á vez) usando *feval*:

```
clear all
f = {@sin, @cos};
for i = 1:2
   a=0;b=pi;x=a;h=0.001;integral=0;
   for x=a:h:b
      integral=integral+h*feval(f{i}, x);
   end
   fprintf('integral de %s en [%g, %g]= %g\n', char(f{i}),
     a, b, integral);
end
```

Vector de celdas de Matlab

Referencia a función

Transforma a referencia a función nunha cadea de caracteres

Función anónimas

- Funcións simples (dunha única liña, análogas a **funcións de sentenza** en Fortran). Pódense crear en calquer parte (dentro dunha función, arquivo ou liña de comando).
- Definición: *f=@(argumentos) expresión;*
- Chamada: *f(argumentos)*
- Aínda que non se lles pasen argumentos teñen que levar os parénteses tanto na creación como na chamada. Exemplos:
  - Definición: *f=@(x) x^2+1, f=@(x,y) sin(x+y)*
  - Chamada: *f(4), f(pi/2,pi/3)*

Referencias a función

1) Co ***operador @*** (“at” ou “arroba”), precedendo o operador @ ó nome dunha función predefinida de Matlab: *f=@sin*

2) Coa función ***str2func('funcion')*** que toma como argumento o nome de función e devolve a referencia á función: *f=str2func('sin'); f(pi)*

   Esta función permite transformar unha **expresión** (cadea de caracteres) en referencia a función:

   expr=’x^2’;f=str2func(sprintf(‘@(x) %s’,expr))

Funcións *inline* (I): obsoletas

- Funcións simples (dunha única liña, análogas a **funcións de sentenza** en Fortran) para cálculos matemáticos que requiren computación extensiva e deben ser eficientes, xa que se executan moitas veces
- Defínense dentro do programa (non nun arquivo separado)

  nome = inline('expresión matemática')
  nome = inline('expresión', 'var1',...,'varN')

- Pode incluír funcións de Matlab ou propias
- As *var1...varN* son as variábeis independentes
- Debe respeta-la dimensión do argumento (escalar / vector/matriz). Se é vector/matriz, hai que facer as operacións compoñente a compoñente (.*, .^, ./)

Funcións *inline* (II)

- A expresión pode ter varias variábeis independentes (non *i* ou *j*, unidade imaxinaria)
- Ex: *f = inline('exp(x^2)/sqrt(x^2+5)');*
  f(2)
  ans=18.1994
- Ex: *f=inline('x^2+y^2+z^2','x','y','z');*
  f(1,2,1) => resultado: 6
- *Ex: f=inline('exp(-x.^2)./(x.^2+5)') % con vectores*
  x=-1:0.1:1; f(x)
- Se as variábeis independentes non se indican, Matlab asume que son as letras da expresión por orde alfabética
- Son equivalentes ás funcións anónimas

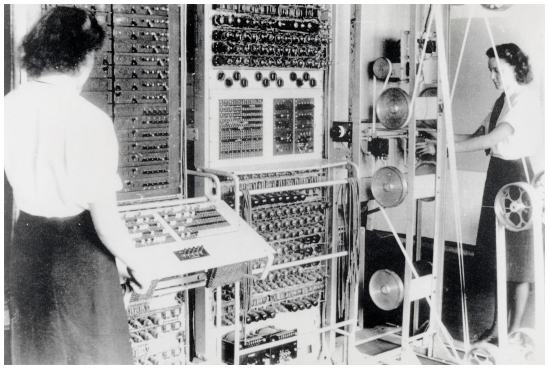
Resumen de referencias a función

- Función anónima: f=@(x) x^2
- Operador @: f=@*sin*
- Inline: f=*inline*(‘x^2’);
- Con *str2func*: f=*str2func*(‘@(x) sin(x)^2’)

  s=’x*sin(1+x)’;f=*str2func*(*sprintf*(‘@(x) %s’,s));
- Todas poden ser chamadas: f(5)
- Agás *inline*, podes obter a súa expresión como cadea de caracteres coa función *func2str*(f):

  printf(‘f=%s\n’,*func2str*(*f*))

Programadoras dos ordenadores Colossus (1943)



- Primeiras calculadoras electrónicas
- Programadas por 273 mulleres programadoras do Women's Royal Naval Service
- Entre outras (2016): Irene Dixon, Lorna Cockayne, Shirley Wheeldon, Joanna Chorley and Margaret Mortimer
- Usados por Inglaterra para descifrar comunicacións alemanas na 2ª guerra mundial

Lectura de datos dende arquivo

- Se o arquivo so ten números (sen letras nin símbolos) e tódalas liñas teñen o mesmo número de elementos (arquivo regular): comando ***load***.
- ***Non*** hai que abrir/pechar o arquivo:

  load datos.dat => carga os datos á matriz *datos*

  x=load('datos.dat'); => carga os datos á matriz *x*

  Mellor a segunda forma con nomes de arquivos longos.
- Se o arquivo so ten números, aínda que sexa irregular (p.ex. 2 números na 1ª liña e 3 números na 2ª liña):

  f=fopen(‘*arquivo.txt*’); *x=fscanf*(*f*,’*%i*’);

  Le tódolos números ao vector columna *x*

Entrada e saída a arquivos

- Para arquivos irregulares con letras e números.
- Apertura: *f=fopen('arquivo.dat', 'permisos');*
- Retorna *f=-1* en caso de erro, *f>0* noutro caso.
- Permisos:
  - '*r*': abre o arquivo para lectura (por defecto)
  - '*w*': escritura: borra o arquivo se xa existe
  - '*a*': abre o arquivo para escribir ao final (conserva o que xa está)

```
nf='arquivo.txt';f=fopen(nf,'r');
if -1==f; error('fopen %s',nf); end
```

- Peche do arquivo: *fclose(f);*

Escritura / lectura en arquivos

- **Escribir**: función *fprintf*:

fprintf(f, 'formato', datos);

  - A mesma función para saída por pantalla pero con *f* para enviar a arquivo. Ex: *fprintf(f,'n=%i x=%f\n',n,x);*
  - Se mostra en pantalla, non retorna nada; se almacena en arquivo, retorna o nº de bytes escritos (rematar en **;**)
  - Vectorizada: *x=randi(100,1,20); fprintf(f,'%i ',x);*
- **Ler**: Función *fscanf*: ten dúas formas:

a=fscanf(f, 'formato');
[a m]=fscanf(f, 'formato', n);

Le *n* datos. Só con arquivos nos que as liñas teñen distintos números de elementos ou conteñen texto (noutro caso, usa *load*)

Se non lle indicas *n*, le tódolos datos do arquivo

Lectura dende un arquivo con *fscanf*

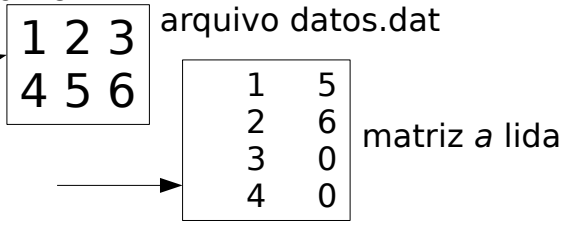
[a,m] = fscanf(f, 'formato', n)

- *f* : identificador de arquivo retornado por *fopen*
- '*formato*': cadea de formato: igual que en *fprintf*
- Se non ten nada que ler (p. ex., no final do arquivo) retorna *a=[]*
- *m: nº de datos realmente lidos*
- Argumento *n* (opcional, se non está vale *inf*): nº de datos a ler: pode ser:
  - Un enteiro: neste caso, le *n* datos (ou m<n se non hai máis datos no arquivo), que se almacenan no vector columna *a*
  - *inf*: Le tódolos datos do arquivo e almacénaos no vector columna *a* (igual que se non se especifica *n*).
  - *[nf nc]:* Le *nf* x *nc* datos do arquivo e méteos por columnas nunha matriz *a* de orde *nf* x *nc*.

Lectura dende un arquivo con *fscanf*

- *Se hai menos de nf x nc datos no arquivo, le os que haxa e enche os elementos restantes con ceros*
- *O valor nc* pode ser *inf*, de modo que *a* ten *nf* filas e o número mínimo de columnas para mete-los datos lidos en *a*
- *O valor nf* non pode ser *inf*
- Se *n*=[*nf nc*], entón *m* pode ser menor que *nf* x *nc* cando hai menos de *nf* x *nc* datos no arquivo
- Exemplo:

```
f=fopen('datos.dat');
a=fscanf(f,'%d', [4 inf])
fclose(f);
```



- *A matriz a* é 4x2 porque en *datos.dat* hai 6 elementos e non collen nunha matriz 4x1, necesita 2 columnas

Exemplos de lectura/escritura

- Lectura dunha matriz [*nf* x *nc*] dende un arquivo: ***a = load('arquivo.txt');***

```
f = fopen('arquivo.txt', 'r');
a = fscanf(f, '%g', [nf nc]);
fclose(f);
```

- Escritura nun arquivo:

```
f = fopen('arquivo.txt', 'w');
for i=1:nf
    fprintf(f, '%g ', a(i,:));
    fprintf(f,'\n');
end
fclose(f);
```

• Lectura dun arquivo irregular completo:

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

```
f= fopen('arquivo.txt', 'r');
x=fscanf(f, '%g');
fclose(f);
```

• Adición ao final dun arquivo:

```
x = [1 2 3 4];
f = fopen('arquivo.txt', 'a');
fprintf(f, '%g ', x);
fclose(f);
```


Lectura de cadeas de caracteres con *fgetl* e *strsplit*

- Sintaxe: *s=fgetl(f);* le unha liña como cadea de caracteres
- Se a liña está baleira, ou está ao final do arquivo, retorna -1 como número
- Divides en palabras con *strsplit*(*s*): *s*{*1*},...,*s*{*n*}
- Se a liña ten números, hai que dividir a cadea en palabras e convertir os números en double (función *str2double*)
- Se a cadea de caracteres non é un número, *str2double* retorna *NaN*, e podes comprobalo coa función *isnan*

```
clear all
f=fopen('datos.dat');
if -1==f; error('fopen datos.dat'); end
while ~feof(f)
   s=fgetl(f);
   fprintf('liña= <%s>\n',s);
   t=strsplit(s);n=numel(t);
   fprintf('palabras:');
   for j=1:n
      x=str2double(t{j});
      if ~isnan(x)
         fprintf('%f\n',x);
      else
         fprintf('%s\n',t{j});
      end
   end
end
fclose(f);
```

Divide a cadea e convirte a números se procede

Funcións *feof* (fin de arquivo) e *frewind* (rebobinado)

- Cando lemos, é importante saber cando chegamos á fin de arquivo
- A función *fscanf* retorna 0 bytes cando chega á fin do arquivo
- A función *feof*(*f*) retorna 1 se atopou a fin do arquivo, ou 0 en caso contrario
- Función *frewind*(*f*): retorna ao comezo do arquivo

Exemplo: programa que le todo o arquivo e mostra liña a liña:

```
f=fopen('datos.dat');
if -1==f
   error('fopen datos.dat')
end
while ~feof(f)
   s=fgetl(f);fprintf('%s\n',s);
end
fclose(f);
```


Funcións *fseek*, *ftell, str2double, isnan*

- *fseek*(*f*,*n*,*w*): sitúase *n* bytes (caracteres) á dereita (se *n*>0) ou esquerda (se *n*<0) de *w*, que pode ser:
  - 'bof' ou -1: comezo do arquivo (*beginning of file*)
  - 'cof' ou 0: posición actual no arquivo (*current position on file*)
  - 'eof' ou 1: final do arquivo (*end of file*)
- *fseek*(*f*,10,'*bof*'): 10 caracteres logo do comezo do arquivo
- *fseek*(*f*,-5,'*cof*'): 5 caracteres antes da posición actual
- *fseek*(*f*,-20,'*eof*'): 20 caracteres antes do final de arquivo
- *ftell*(*f*): retorna a posición actual (onde se vai ler ou escribir) en caracteres dende o comezo do arquivo
- Distinguir se *s* é número ou palabra: *x*=*str2double*(*s*), *isnan*(*x*)=1 se *x* é palabra, =0 se *x* é número (real ou enteiro).

Exemplo de lectura de arquivo en formato *write.table* de R con *fscanf*

- Arquivo co seguinte contido:

	E1	E2	E3	E4	Saida
1	0.25	0.33	1.23	-0.51	Branco
2	-0.34	1.3E5	0.22	4.3	Negro

```
clear all
f=fopen('arquivo.dat','r');
if -1==f
    error('erro abrinfo arquivo.dat')
end
s=strsplit(fgetl(f));s(1)=[];nc=numel(s)-1;nf=0;
while ~feof(f)
    fgetl(f);nf=nf+1;
end
dato=zeros(nf,nc);saida=cell(1,nf);
frewind(f);nomes=fscanf(f,'%s',nc+1);
for i=1:nf
    fscanf(f,'%i',1);  % le e descarta (non almacena) o nº de liña
    dato(i,:)=fscanf(f,'%g',nc);  % le as ne entradas (tamén formato exponencial)
    saida{i}=fscanf(f,'%s',1);  % le a saída (cadea de caracteres)
end
fclose(f);
for i=1:nf
    fprintf('dato %i: ',i);fprintf('%g ',dato(i,:));fprintf('saida=%s\n',saida{i});
end
```

Le o nº de filas e columnas

Jean Sammet (1928-2017)

- Matemática e informática
- Desenvolveu (1962) a primeira linguaxe de programación simbólica FORMAC (formula manipulation compiler) en IBM
- Primeira persoa que escribiu extensamente (1969) sobre a historia e clasificación das linguaxes de programación

Función *find* e operadores relacionais

- Obter elementos que cumpren unha condición: *vector(condición)*. Ex: *a(rem(a,2)==0), v(v>3 && v<=5)*

  Elementos con valores pares

  Elementos con valores no intervalo (3,5]

- Obter índices de elementos: *find(condición)*. Ex: *find(rem(5*v-3,4)==3)*

  Índices de elementos que dan resto 3 cando divido 5v-3 entre 4

- Obter nº de elementos que cumpren unha condición: *length(v(condición))* ou *sum(condición).*

  Índices de elementos con valores impares

  Ex: *length(v(rem(v,2)==1))*, ou *sum(rem(v,2)==0)*

- Modificar elementos que cumpren unha condición: *v(cond)=F(v(cond)).*

  Incrementa en 5 os elementos de v maiores ca 3

  Ex: *v(v>3)=v(v>3)+5*, ou *t=v>3;v(t)=v(t)+5*

Vectorización de expresións (I)

- Executa un comando que cree unha matriz *a* de orde 5 e poña os elementos con valores pares a 7 e os elementos con valores impares a -1:

  a = magic(5);

  a(rem(a,2)==1)=-1

  a(rem(a,2)==0)=7

- Se fago *a=magic(5);rem(a,2)==0* devólveme unha matriz de 1s nos elementos pares e 0 nos restantes
- Hai que executar primeiro *rem(a,2)==1* e logo *rem(a,2)==0*: se facemos primeiro *rem(a,2)==0*, poñemos os elementos pares a valores impares

Vectorización de expresións (II)

- Dada unha matriz cadrada, manipúlaa de modo que os elementos a_{ij} que verifiquen que $i \cdot j$ é par pasen a valer -1, e os elementos con $i \cdot j$ impar pasen a valer 3

  a=magic(5)

  *i=1:5;j=i; b=i'*j;*

  a(rem(b,2)==1)=3

  a(rem(b,2)==0)=-1

- Matriz *b=i'*j,* de orde *nxn*: $b_{kl}=i(k)*j(l)$

Vectorización de expresións (III)

- Crea un vector con 10 elementos enteiros aleatorios no rango [-10,10]: *v=round(-10+20*rand(1,10))*
- Mostra os índices dos elementos positivos: *find(v>0)*
- Mostra so os elementos positivos: *v(v>0)*
- Mostra tódolos elementos positivos do vector e ceros nos negativos: *v(v<0)=0* ou *v.*(v>0)*
- Crea un vector con valores -3 onde *v>0* e 5 onde *v<=0*: *-3*(v > 0) + 5*(v<=0)*

Vectorización de expresións (IV)

- Crea dous vectores *v* e *w* de orde 10: *a=magic(10); v=a(1,:); w=a(2,:);*
- Atopa os índices dos elementos de *v* maiores que os seus correspondentes de *w* : *find(v > w)*
- Atopa os elementos de *v* maiores que o seu correspondente de *w* : *v(v > w)*
- Crea un vector que valia 4 nos elementos *i* nos que $v_i > w_i$ e -6 nos restantes: *4*(v > w) - 6*(v <= w)*
- Selecciona os elementos de *v* nas posicións *i* nas que v_i e w_i sexan meirandes ca 5: *v(v > 5 & w > 5)*

Carol Shaw (1955)

- Icono da industria dos videoxogos
- Programadora pioneira de videoxogos en Atari (1978)
- Enxeñeira microprocesadora de videoxogos
- Creadora dos videoxogos River Raid (1982, Activision; 1983, Atari 800), Happy Trails (1983, Intellivision)
- Industry Icon Award (2017) polas súas contribucións á industria dos videoxogos

Vectores e matrices de celdas (*cell arrays*)

- Vectores/matrices onde cada elemento pode ser dun tipo distinto.
- Creación: utilizando chaves { }
  - *s={'ola',1:3,17,1+3*i}; disp(s{1})*: define un vector de 4 celdas
  - Define un vector de 3 celdas:

    1)vc(1)={[1 2 3]} ou *vc{1}=[1 2 3]*

    2)vc(2)={'unha cadea'} ou *vc{2}='unha cadea'*

    3)vc(3)={ones(3)} ou *vc{3}=ones(3)*
- Acceso a un elemento: s{1} para imprimir: *fprintf*('%s', s{1}); s(1) para eliminar: s(1)=[ ]; s{1}=[ ] asigna a matriz baleira

Vectores e matrices de celdas

- Funcións de manipulación:
  - *cell(m,n)*: crea un cell array baleiro de m filas e n columnas
  - *celldisp(ca)*: mostra o contido de tódalas celdas de ca
  - *iscell(ca)*: indica se ca é un vector de celdas
  - *num2cell(v)*: convirte un vector numérico **v** nun cell array
  - *cellstr(s)*: crea un vector de celdas a partires dun vector de caracteres.

Cadeas de caracteres

- Definición: *str='son a cadea 450'*
- Se queremos ter varias cadeas (p.ex. un vector de cadeas) non podemos facer *s=['ola','adeus']* xa que entón *s='olaadeus'*.
- Non podemos facer *s=['ola' ; 'adeus']*, xa que as dúas filas (ou cadeas) teñen que ter a mesma lonxitude
- Por iso o usual é usar unha celda de cadeas: *s={'ola', 'adeus'}*
- Para acceder á 1ª cadea: *disp(s{1})* ou *disp(s(1))*
- O elemento *s{1}* é de tipo *char*, pero *s(1)* é *cell*:
  - *fprintf('%s\n',s{1})* non da erro
  - *fprintf('%s\n',s(1))* si da erro
- Outra opción: s=*char*({'ola','adeus'}); s(1,:)->'ola'; s(2,:)->'adeus'

Funcións de manipulación de cadeas de caracteres (I)

- Función **strsplit**(s,t): divide unha cadea *s* en palabras usando o delimitador *t* (espazo, por defecto). Retorna un vector de celdas, cada elemento é unha cadea de caracteres (palabra)
- O nº de palabras é o nº de elementos do vector de celdas
- Exemplo: s='valor 3.65'; p=*strsplit*(s); p{1}='valor' p{2}='3.65';
- *str2num*(p{2}) da 3.65 como número para poder facer operacións. Se *p*{2} non é un número, *str2num*(*p*{2}) da *[ ]* (matriz baleira)
- Outra forma de convertir de char a número: *x=str2double*(*s*). Se *s* non ten un número, *str2double* da *NaN*
- Función *isnan*(*x*): retorna 1 se *x=NaN*, e 0 en caso contrario.

Funcións de manipulación de cadeas de caracteres (II)

- *ischar(str):* devolve 1 se *str* é unha cadea de caracteres e 0 se non o é.
- *isletter(str)*: devolve un vector de igual dimensión a str con 1 se é unha letra do abecedario e 0 en caso contrario.
  - Ex: *s='ola que tal'; isletter(s) -> 1 1 1 0 1 1 1 0 1 1 1*
    (Nota: -> denota a resposta que da Matlab á función *isletter*)
- isdigit(str): igual pero con 1 se son díxitos e 0 en caso contrario
- *isspace(str)*: igual que *isletter()* pero con caracteres de espacio.
  - Ex: *s='ola que tal'; isspace(s) -> 0 0 0 1 0 0 0 1 0 0 0*
- *char(x), char(c), char(t1, t2, t3, ...):* devolve un vector de caracteres a partires de: *x* un vector de enteiros (códigos Unicode), *c* un vector de celdas de caracteres, as cadeas *t1, t2, t3* ....
  - Ex: *c={'ola','adeus'};s=char(c); s(1,:)->ola; s(2,:)->adeus*

Funcións de manipulación de cadeas de caracteres (III)

- *num2str(x,n):* convirte o número *x* con *n* (opcional) cifras nunha cadea de caracteres.
- *str2num(s):* convirte a cadea de caracteres *s* a un número (da *[ ]* se *s* non é un número).
- *str2double*(*s*): convirte *s* a número (da *NaN* se *s* non é un número)
- *strcmp('str1','str2'), strcmp('str1',C2), strcmp(C1, C2):* compara as cadeas *str1* e *str2* e devolve 1 se son iguais e senon 0. Se temos o vector de celdas *C1*, compara *str1* con tódolos elementos de *C1*. No último caso compara os dous vectores de celdas. *strcmp* distingue entre maiúsculas e minúsculas; a función *strcmpi* non.
  - Ex: *strcmp('ola','ola') -> 1; strcmp('ola',{'ola','adeus'}) -> 1 0; strcmp({'ola','pepe'},{'ola','adeus'}) -> 1 0*
- *strncmp('str1', 'str2', n), strncmp('str', C, n), strncmp(C1, C2, n)*: igual que strcmp pero comparando so os *n* primeiros caracteres.

Funcións de manipulación de cadeas de caracteres (IV)

- *strmatch('str', C)* ou *strmatch('str', C, 'exact')*: devolve un vector columna cos índices de *C* onde se atopa *str*.
  - Ex: *C={'N', 'H', 'O', 'He', 'Ca'}; strmatch('H', C)* -> vector columna cos índices 2 e 4 (que corresponden ó '*H*' e '*He*')
  - Ex: *strmatch('H', C, 'exact')*-> vector columna co índice 2
- Lonxitude dunha cadea: *s='ola caracola'; numel(s),length(s) -> 11*
- Concatenación de cadeas: *strcat(s1,s2,s3,...); strvcat(s1,s2,s3,...)*
  - Horizontal: *strcat('ola','adeus') -> 'olaadeus'*
  - Vertical: *strvcat('ola','adeus') -> ola*
    adeus
  - Tamén se pode facer concatenando vectores (supoñendo que unha cadea de caracteres é un vector de caracteres).

Funcións de manipulación de cadeas de caracteres (V)

- *strfind(str, patrón)* e *strfind(cellstr, patrón)***:** devolve un vector coas posicións de comezo onde se atopou a cadea *patrón* na cadea *str* ou vector de celdas *cellstr*.
  - Ex: *strfind('ola caracola', 'ol') -> 1 10* (o patrón *'ol'* aparece nas posicións 1 e 10 da cadea *'ola caracola'*)
  - Ex: *strfind({'ola','caracola'},'ol') -> [1] [6]* (o patrón *'ol'* aparece nas posicións 1 e 6 do 1º e 2º elementos do arrai de celdas)
- *regexprep('str', 'expr', 'repstr'):* reemplaza a expresión *expr* por *repstr* na cadea de caracteres *str* e devolve a cadea resultante.
  - Ex: *cad='H2(g)+O2(g)=H20'; regexprep(cad, '\(g\)', '') -> H2+O2=H20*: substitúe '(g)' pola cadea baleira, é dicir, elimina '(g)'.

Función *textscan*

- Le un arquivo aberto con *fopen* e almacena nun vector de celdas os datos lidos (números ou cadeas de caracteres):
- Sintaxe:
  f=fopen('arquivo.dat','r');
  c=textscan(f,formato); *% le todo o arquivo*
  c=textscan(f,formato,n); *% le n datos*
- O formato é como en *fprintf*: *%i,%s,%f,%g*
- O valor *c* é un vector de celdas: *c{1},...,c{n}*

Función *textscan*: exemplo

```
clear all;
fid=fopen('taboa.txt');
t=textscan(fid, '%s', 4); % ler primeira liña
celldisp(t);
datos=textscan(fid, '%d %s %s %f'); % ler resto arquivo
celldisp(datos);
fclose(fid);
```

posición	símbolo	elemento	pesoAtómico
1	H	HIDRÓXENO	1
6	C	CARBONO	12
7	N	NITRÓXENO	14.01
8	O	OSÍXENO	16
17	Cl	CLORO	35.45

```
t{1}{1} =posición
t{1}{2} =símbolo
t{1}{3} =elemento
t{1}{4}=pesoAtómico
```

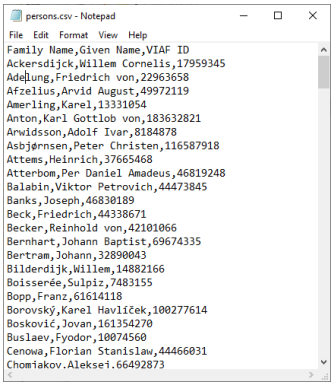
```
datos{1}' = [   1     6          7          8         17] (vector)
datos{2}' = {   'H'    'C'    'N'    'O'    'Cl'} (vector celdas)
datos{3}' ={'HIDRÓXENO' 'CARBONO'  'NITRÓXENO'
 'OSÍXENO'  'CLORO'} (vector celdas)
datos{4}' =[1.0000   12.00  14.010  16.000   35.450] (vector)
```

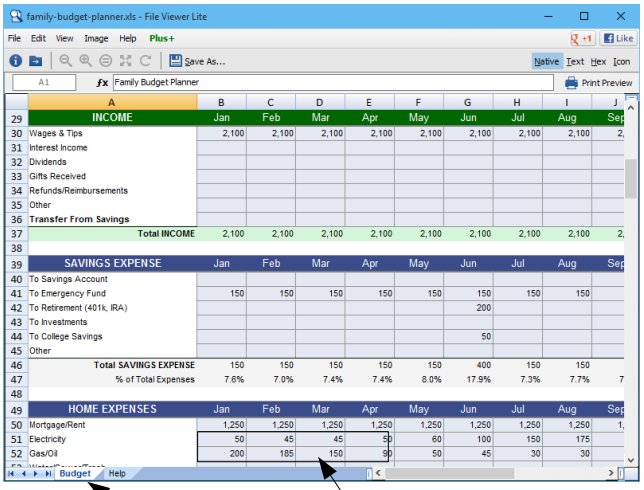
Ollo:
datos{1}(2) -> 6
datos{2}{3} -> 'N'

Outras funcións de lectura de arquivos

- Arquivos CSV (comma separated values):

x=readtable(‘arquivo.csv’)





- Arquivos XLS ou XLSX (follas de cálculo):

x=xlsread(‘arquivo.xls’,’páxina’,’rango’)

rango: ‘A2:C3’

Estructuras (*struct*)

- Conxunto de datos heteroxéneo (ten datos de distintos tipos, p.ex. real e carácter)

a=struct(‘nome’,’Carlos’,’idade’,19);

- *a* é unha estructura con dúas variábeis: *a.nome* é unha cadea, e *a.idade* é un número. Podes imprimir os seus campos con:

fprintf(‘nome=%s idade=%i\n’,a.nome,a.idade)

- Normalmente se usa un vector ou matriz de estructuras:

```
v=cell(1,10);
for i=1:10
    v{i}=struct(‘nome’,nome(i),’idade’,idade(i));
end
```

- Logo podes facer *v{i}.nome* (accedes ao nome i-ésimo) ou *[v{:}].nome* (accedes a todos os nomes): *fprintf(‘%s ‘,[v{:}].nome)*
- É máis cómodo que usar un vector distinto para cada campo.

Susan Kare (1954)

- Deseñadora de iconos, elementos gráficos e tipografías dixitais en ordenadores, móviles, ... para as maiores empresas de informática (Microsoft, Apple, IBM, Facebook, ...)
- Pioneira do *pixel art*
- A máis incluínte deseñadora de iconos informáticos segundo o MoMA (Museo de Arte Moderna) de Nova Iorque

Clase en Octave

- Bloque *classdef*
- En arquivo co mesmo nome que a clase (*punto.m*)
- Seccións *properties* e *methods*
- Declaración de obxecto:

  p=punto(1,3);
- Acceder a variable: *p.x*
- Chamar a función: *p.mostra*()

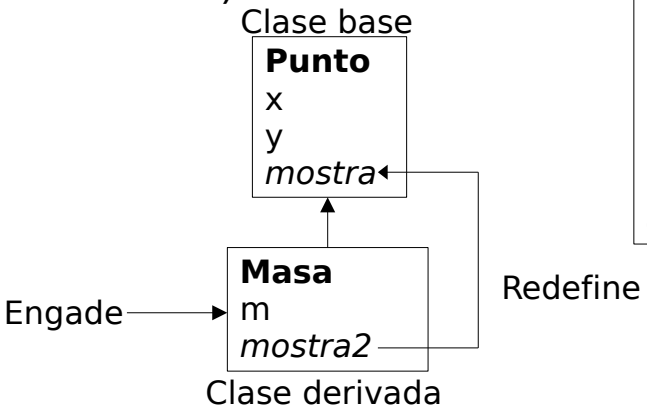
```
classdef punto
    properties
        x
        y
    end
    methods
        function p=punto(x,y)
            p.x=x;p.y=y;
        end
        function mostra(p)
            printf('punto: x=%g y=%g\n',p.x,p.y)
        end
    end
end
```

Herdanza en Octave

- Clase derivada: *masa*
- Hereda de *punto*
- Engade a variábel *m*
- Redefine a función *mostra*() de *punto* (sobrecarga da función)

Isto é o que fai que *masa* herede de *punto*

```
classdef masa < punto
    properties
        m
    end
    methods
        function ms=masa(x,y,m)
            ms=ms@punto(x,y);
            ms.m=m;
        end
        function mostra(p)
            printf('masa: x=%g y=%g m=%g\n',
              ms.x,ms.y,ms.m)
        end
    end
end
```



Polimorfismo en Octave

- Chamar a funcións distintas co mesmo nome sen ter que comprobar no programa qué función se executa
- *p.mostra*(): chama á función *mostra*() da clase base (*punto*)
- *m.mostra*(): chama á *mostra*() da clase derivada (*masa*)
- Se hai varias clases derivadas que redefinen *mostra*(), unha variable *x* pode chamar a distintas funcións *mostra*() dependendo de a que clase se refiran

Chamando manualmente a cada función

```
x=punto(1,2);
x.mostra()
x=masa(2,3,1);
x.mostra()
```

Chamando a cada función automáticamente nun bucle

```
p=punto(1,2);
m=masa(2,1,4);
y={p,m};
for i=1:2
    y{i}.mostra()
end
```

En cada iteración executa a función *mostra*() de cada obxecto

Sobrecarga de operadores

- Para redefinir (sobrecargar) un operador aritmético, relacional ou lóxico hai que definir unha función cun nome específico para cada operador
- Por exemplo, para a suma (operador +) a función débese chamar *plus*()
- As prioridades dos operadores redefinidos non cambian

a+b(plus)	a-b(minus)	a*b(mtiles)	a.*b(times)	a./b(rdivide)
a.\b(ldivide)	a/b(mrdivide)	a\b(mldivide)	a^b(power)	a.^b(mpower)
a<b(lt)	a>b(gt)	a<=b(le)	a>=b(ge)	a==b(eq)
a&b(and)	a|b(or)	~a(not)	a'(ctranspose)	a~=b(ne)

Sobrecarga de operador +

Ficheiro *sobrecarga.m*

```
clear all;clc
p=punto(1,3);
m=masa(4,5,1);
printf('sobrecarga de operador +:\n')
s=p+m;
s.mostra()
```

Ficheiro *punto.m*

```
classdef punto
   properties
      x
      y
   end
   methods
      function p=punto(x,y)
         p.x=x;p.y=y;
      end
      function mostra(p)
         printf('punto: x=%g y=%g\n',p.x,p.y)
      end
      function r=plus(a,b)
         x=a.x+b.x;y=a.y+b.y;
         r=punto(x,y);
      end
   end
end
```

Cálculo numérico e simbólico

- Resolución **numérica** de ecuación non linear dunha variábel: *f(x)=0*:

Función anónima

f=@(x) expresión; x = fzero(f, x0)

  - *x0* é un punto de inicio
  - Ex: $xe^{-x}=0.2$: *f=@(x) x*exp(-x)-0.2; fzero(f, 0.7)*

- Mínimo dunha función:

f=@(x) expr
xmin=fminbnd(f,a,b);
[xmin vmin] = fminbnd(f, a,b)

  - Busca o valor mínimo de *f* no intervalo *[a,b]*
  - Retorna punto e valor mínimo *xmin* e *vmin*
  - Exemplo: *f=@(x) x^3-12*x^2+3*x-1;*
    [x v]=fminbnd(f,0,10)

Integración numérica

x = quad('función', a, b)

- A función pode ser unha expresión, función predefinida de Matlab ou función definida polo usuario. Ídem en *fzero()* e *fminbnd()*
- A función debe escribirse considerando que *x* é un vector (operandos compoñente a compoñente)
- Ex: *quad('x.*exp(-0.8*x) + 0.2', 0, 8)* → $\int_0^8 (xe^{-0.8x}+0.2)dx$
- Para integral de función definida polos puntos $\{(x_i,y_i),i=1...n\}$ mediante o método dos trapecios: *trapz(x, y)*

```
x=0:0.01:8;
y=x.*exp(-0.8*x)+0.2;
trapz(x, y)
```

- Simbólicamente: *syms x; eval(int(x*exp(-0.8*x)+0.2,x,0,8))*

Cálculo simbólico (I)

- Definición de variábeis simbólicas: *syms v1 ... vn*
- En **Octave**: antes de definilas, executa *pkg load symbolic*.
- Límites: *limit(expresión, variábel, valor, lado)*
  - *var*, *valor* e *lado* son opcionais (*var=x,valor=0* por defecto; *lado* = '*left*' ou '*right*', por defecto calcúlase o límite por ambos lados)
  - Ex: *syms x; limit(1/x); limit(1/x,inf); limit(1/x, x, 0, 'left');*
- Derivación: *diff(expresión, var, orde)*
  - *var* e *orde* son opcionais (*var=x* e *orde=1*)
  - Ex: *syms x; diff(x^2); diff(x^2, x); diff(x^2,x,2)*
    syms x y; diff(x^2+y^2,x,y)

Cálculo simbólico (II)

- Integración indefinida: *int(expresión, var)*
- Integración definida: *int(expresión, var, ini, fin)*

$$\int_0^\infty e^{-x^2} dx$$

  - Ex: *int(cos(x), x); int(exp(-x^2), 0,inf); eval(ans)*
- Series numéricas: *symsum(expresión, var, ini, fin)*
  - Ex: *syms n; symsum(1/(n^2-1),n,2,inf)* $\longrightarrow \sum_{n=2}^{\infty} \frac{1}{n^2-1}$
- Substitución de variábeis simbólicas por valores numéricos:

  res = subs(expresión, var, valor(es))

  - Ex: *subs(x^2+exp(-x), x, pi/2)*
- *Avaliación de expresión simbólica en punto flotante: eval(expr)* ou double(expr). En octave: *double(expr)*.

Conversión entre cadeas de caracteres, expresións simbólicas e referencias a función

- Conversión de cadea de caracteres (cunha expresión matemática) a expresión simbólica:

  syms x;f=str2sym('x^2')

  Non podes chamala (p.ex. f(5)), pero si podes derivala: *diff(f,x)*, e calcular o seu valor con *subs(f,x,5)*

  Non funciona *diff('x^2')*: *diff(str2sym('x^2'))*

- Podes facer operacións de cálculo simbólico:

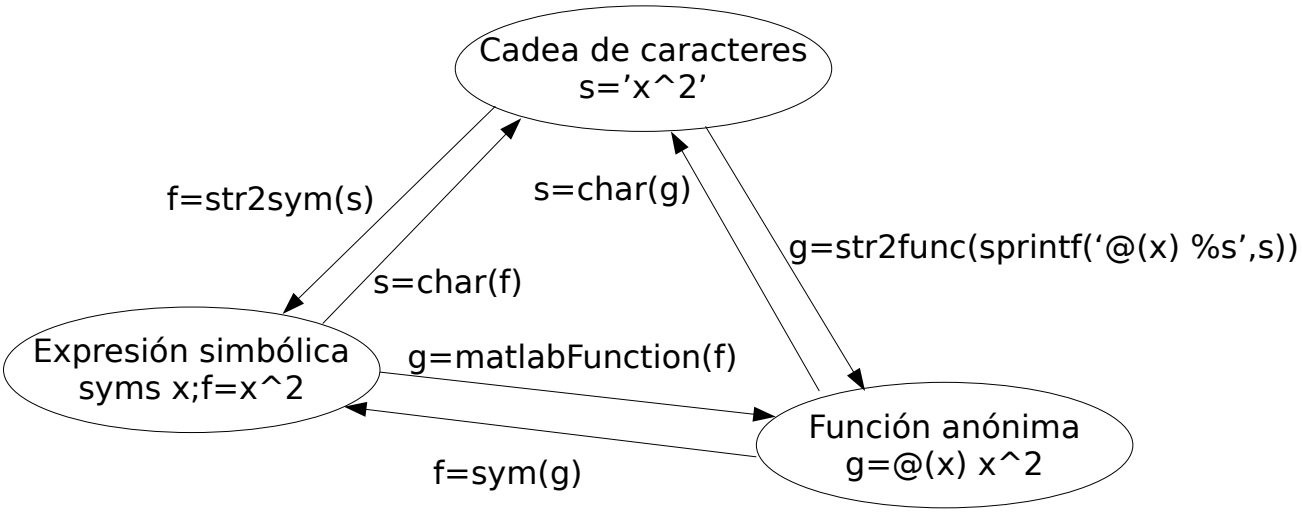
  con funcións *inline*: *f=inline('x^2'); diff(f(x))*

  con funcións anónimas: *f=str2func('@(x) x^2'); diff(f(x))*

- Conversión de expresión simbólica a función anónima:

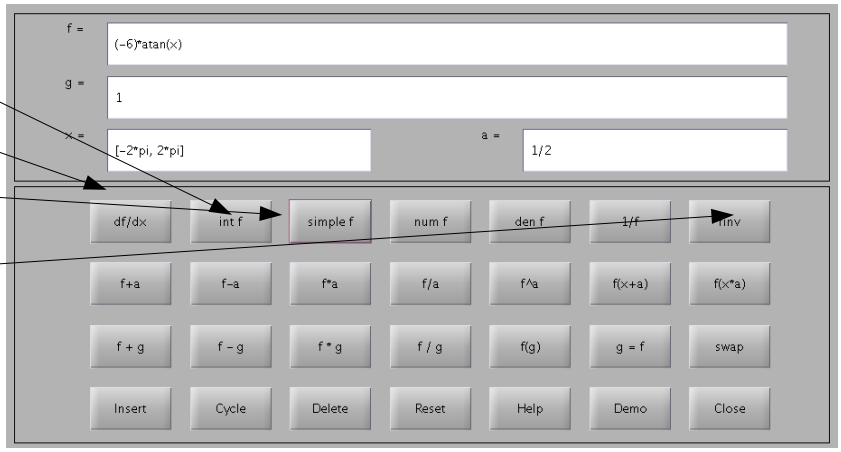
  syms x;f=x^2;g=matlabFunction(f)

Esquema de conversión: cadea de caracteres ↔ función anónima ↔ expresión simbólica



Cálculo simbólico (III)

- *funtool*: calculadora de funcións
  - Integrar
  - Derivar
  - Simplificar
  - Invertir
  - *help funtool*

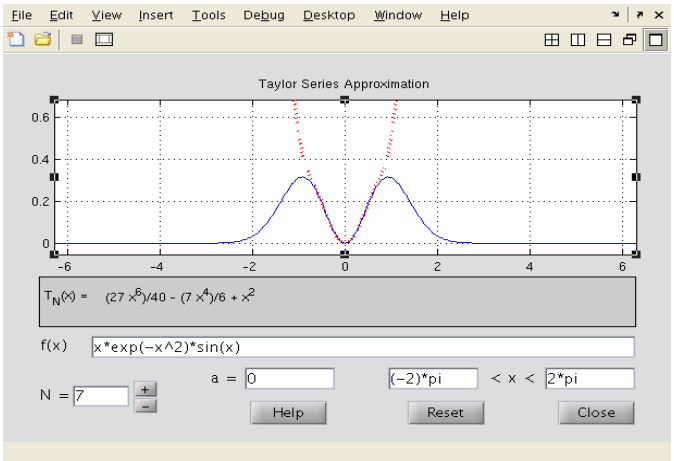


Cálculo simbólico (IV)

Polinomio de Taylor orde *n-1* dunha función *f* en *x=a*:

taylor(f,x,'ExpansionPoint',a,'Order',n)

taylor(f): en x=0,orde 5
taylor(f,x):en x=0,orde 5
taylor(f,x,'ExpansionPoint',1);
en x=1, orde 5
taylor(f,x,'Order',7);
en x=0, orde 6
Ex: syms x; f=(x-1)/(x+1);
taylor(f,x,'ExpansionPoint',7,
'Order',1)



taylortool: calcula e representa graficamente a función e o seu polinomio de Taylor

Cálculo simbólico (V)

- Resolución **simbólica** de ecuacións e sistemas de ecuacións:

  syms x; h = solve(expresión, var)

  - A ecuación é da forma *expresión=0*
  - Ex: *syms x; h = solve(x*exp(2*x)-5)*
- *Resolución simbólica dun sistema de varias ecuacións con varias variábeis:*

  syms x1...xn; [v1 ... vn]=solve(eq1,...,eqn, x1, ..., xn)

  - Ex: *syms x,y,z; [xz yz]= solve(10*x+12*y+16*z, 5*x-y-13*z,x,y)* %x,y función de z
  - Ex: *syms x, y; [x y] = solve(x*exp(y)-3,y*exp(x)-2,x,y*)

Combinatoria

- Función *nchoosek(n, k)*: calcula $\frac{n!}{(n-k)!\,k!}$
- **Combinacións** de *n* elementos dun vector *v* tomados en grupos de *k* (*n* < 15): *nchoosek(v, k)*, *nchoosek('abcde',3)*, *nchoosek({'a','b','c','d','e'},5)*
- Retorna unha matriz de *n*!/((*n* – *k*)! *k*!) filas e *k* columnas; *n* < 15 (evitar explosión combinatoria)
- **Permutacións** de *n* elementos: *perms(1:n)*, *perms('abcd')*, *perms({'a' 'b' 'c'})*. Retorna unha matriz de *n*! filas e *n* columnas; *n* debe ser < 15
- Selección aleatoria dunha permutación de *n* números de 1 a *n*: *randperm(n)*. Ex: *i=randperm(n);v(randperm(n))*: barallamento aleatorio dos elementos do vector *v* con *n* elementos.

Exercicios

1) Representa gráficamente $f(x)=e^{-x^2/10}\sin(x^2)$ e atopa o punto $(x_0, f(x_0))$ que minimiza f en $[-10, 10]$

2) Resolve as ecuacións $4\cos 2x - e^{x/2}+5=0$; $2\,sen\,x-\sqrt{x}=-2.5$ $\cos x = 2x^3$

3) Calcula θ tal que $92=88/(\cos\theta + 0.45 sen\,\theta)$

4) Calcula numéricamente a integral $\longrightarrow \int_0^5 \frac{1}{0.6x^2+0.5x+2}\,dx$

5) Calcula simbólicamente:

$$\lim_{x\to\infty} \frac{\ln x}{\sqrt{x}} \qquad \lim_{x\to 1}\left(\frac{x}{x-1}-\frac{1}{\ln x}\right)$$

$$\frac{d}{dx}\left[\frac{\sqrt{x^2+x+2}}{x-1}\right] \qquad \int_0^{\pi/2}\left(1+\frac{1}{2}sen^2 x\right)dx \qquad \sum_{n=1}^{\infty}\frac{2^n+3^n}{n^2+\log n+5^n}$$

Solucións aos exercicios (I)

1) Represento e atopo o mínimo en [-10, 10]
*fplot('exp(-x^2/10)*sin(x^2)', [-10,10])*
*[xmin fmin] = fminbnd('exp(-x^2/10)*sin(x^2)', -10, 10)*
x =2.1477, fmin =-0.6274

*2) a) fzero('4*cos(2*x)-exp(x/2)+5', 0) -> 1.2374*
*comprobación: subs('4*cos(2*x)-exp(x/2)+5', 1.2374) -> 2.4960e-04*
*syms x; solve(4*cos(2*x)-exp(x/2)+5, x)*
*comprobación: subs(4*cos(2*x)-exp(x/2)+5, ans)*
*b) Solución numérica: fzero('2*sin(x)-sqrt(x)+2.5', 2) -> 3.4664*
*comprobación: eval(subs(2*sin(x)-sqrt(x)+2.5, 3.4664))*
*c) Solución simbólica: fzero('cos(x)-2*x^3', 0) -> 0.7214*
*comprobación: subs(cos(x)-2*x^3, 0.7214)*

Solucións aos exercicios (II)

3) *fzero('92-88/(cos(x)+ 0.45*sin(x))', 0) -> -0.0881*

4) *quad('1./(0.6*x.^2+0.5*x+2)', 0, 5) -> 0.9596*
*x=0:0.01:5; y=1./(0.6*x.^2+0.5*x+2); trapz(x, y) -> 0.9596*
*syms x; eval(int(1/(0.6*x^2+0.5*x+2), x, 0, 5)) -> 0.9596*

5) *syms x; limit(x/(x-1)-1/log(x), x, 1) -> ½*
syms x; diff(sqrt(x^2 + x + 2)/(x-1), x, 1)
*syms x: int(1 + sin(x)^2/2, x, 0, pi/2) -> 5/8*pi*
syms n; symsum((2^n + 3^n)/(n^2+log(x)+5^n), n, 1, inf) -> sum((2^n+3^n)/(n^2+log(x)+5^n),n = 1 .. Inf) (non a resolve, pero converxe)

Versión vectorizada:

```
n=1:100;
sum((2.^n + 3.^n)./(n.^2 + log(n)+5.^n))
```

Resultado: 1.8918

```
clear all
suma = 0;sumando = inf; n = 1
while sumando > 1e-5
      sumando = (2^n + 3^n)/(n^2 + log(n) + 5^n);
      suma = suma + sumando; n = n + 1;
end
fprintf('n= %i suma= %g\n', n, suma);
```


Gráficos 2D

- Sentenza *plot*: varias formas
- Representa os elementos do vector x fronte ao nº de compoñente: *plot(x)*
- Se x e y son vectores co mesmo nº de elementos, representa y frente a x: *plot(x, y)*
- Indicando propiedades do gráfico:

plot(x, y, 'r-')*

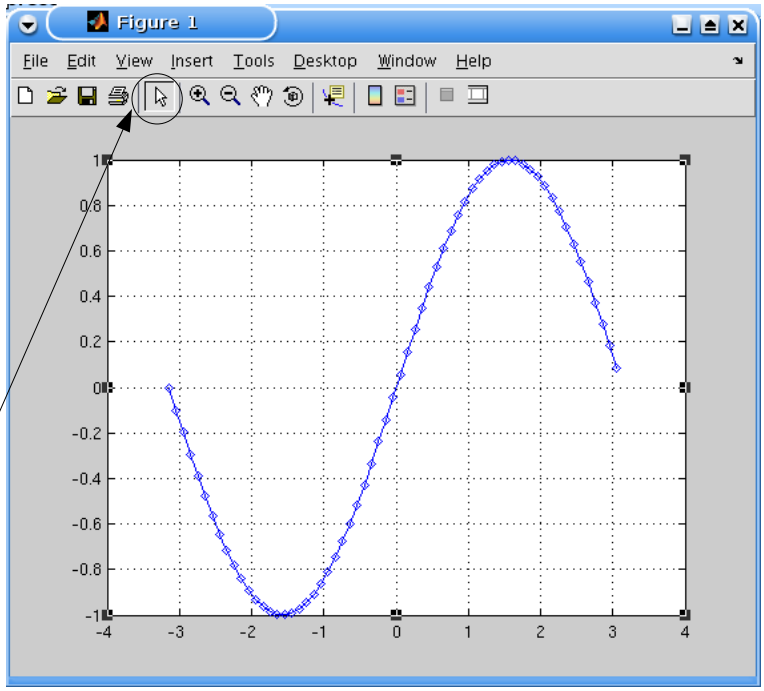
r=red, -= liña, *=punto con forma de asterisco

- Se *a* é unha matriz, *plot(a)* representa cada columna como unha gráfica distinta (e cores distintas)
- Podes representar varias gráficas no mesmo *plot*:

plot(x,sin(x),'bs-',x,cos(x),'or-',x,x.^2,'vg-')

Ventá de figuras

- Como en Maple, non é necesario nin eficiente saber tódalas opcións (cores, tipos de liña e punto, propiedades dos eixos, etc.). Pódense modificar na ventá do plot



Exportación de figuras a ficheiros de imaxe

- Manualmente: gardar ou imprimir en ventá de figura.
- Automáticamente dende ventá de comandos ou dende programa: función *print*.
- A arquivo en formato encapsulado postscript (EPS): *print('-depsc','arquivo.eps');*
- Formato portable network graphics (PNG):

  print('-dpng','arquivo.png');
- Formato PDF: *print('-dpdf','arquivo.pdf);*
- Formato TIFF: *print('-dtiff','arquivo.tif')*
- Formato JPEG: *print('-djpeg','arquivo.jpg')*

Representación de funcións (expresión analítica)

- Función *ezplot:*

  ezplot('función', [min, max])

  ezplot('función'): en [-2π, 2π]

  Ex: *ezplot('exp(-x^2)*sin(x)', [0,2*pi]), ezplot('sin(x)')*

- Función *fplot:*

  fplot(@(x) x.^2,[min max])

  - Función anónima con operacións compoñente a compoñente: .* ./ .^

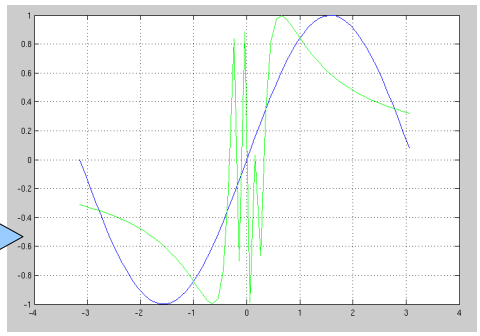
Comandos relacionados

- *xlabel('título eixo X'),* ídem con *ylabel*
- *title('título do gráfico')*
- *clf: borra a gráfica actual*
- *figure(2)*: crea unha nova ventá de figura
- *hold on: permite representar unha gráfica mantendo a(s) anterior(es); hold off: desactiva isto*
- *axis([xmin xmax ymin ymax])*: establece rangos en ambos eixos
- *axis equal*: igual lonxitude para a unidade en X e Y
- *axis square*: figura cadrada e non rectangular
- *grid on (off)*: pon (quita) o enreixado; *grid*: conmuta

Múltiples gráficas na mesma ventá

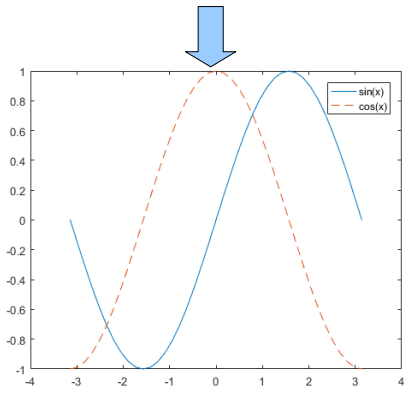
- Con *hold on*:

```
clf
x=-pi:0.1:pi;
y=sin(x);
z=sin(1./x);
plot(x,y)
grid on
plot(x,z,'g')
plot(x,y)
grid on
hold on
plot(x,z,'g')
```



- Comando *legend*:

legend({'gráfica1' 'gráfica2'}, 'location', 'northwest')
Outras *location*: *east*, *southwest*,...



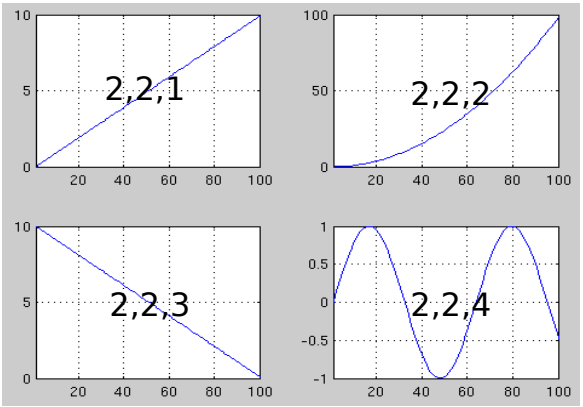
Múltiples gráficas en distintas sub-ventás

- Crea unha matriz de figuras (2x2, 3x2, ...)

subplot(filas,columnas,actual)

- A 1ª execución de subplot crea a matriz (sen figuras) e crea a figura (1,1). Execútase unha vez por cada figura, e crea a figura actual (1 ≤ *actual* ≤ *filas*columnas*)

```
x=0:0.1:10;y=x.^2;z=10-x;
t=sin(x);
subplot(2,2,1); plot(x)
subplot(2,2,2); plot(y)
subplot(2,2,3); plot(z)
subplot(2,2,4);plot(t)
```



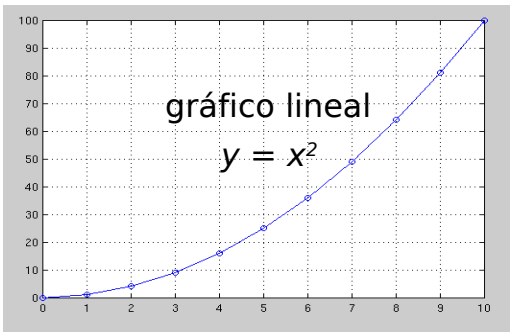
Outros comandos

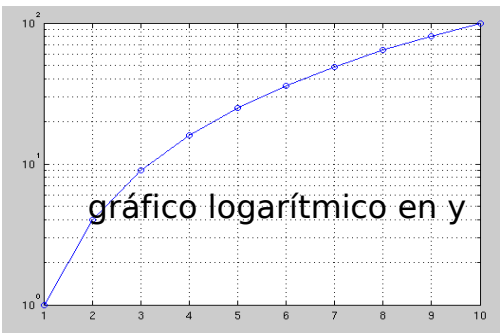
- Engadir texto na figura: *text(x,y,'texto no gráfico en (x,y)'); gtext('texto en posición indicada co rato');*

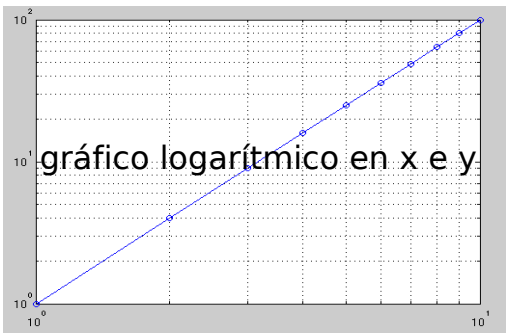
Gráficos logarítimos

- Ás veces é bon usar unha escala logarítmica (isto é, representar *log(x)* no canto de y, ou o mesmo para y) xa que o rango de valores é moi grande.
- *semilogx(x, y)*: escala log só en x
- *semilogy(x, y)*: escala log só en y
- *loglog(x, y)*: escala log en x e y
- Os valores nulos ou negativos non se poden representar nestos gráficos.

Gráficos logarítmicos





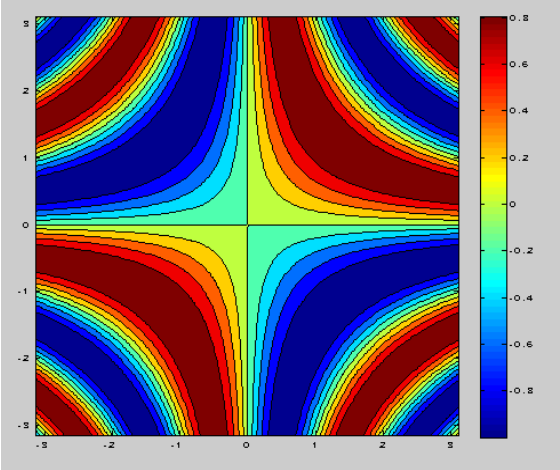


Mapa de calor

- Representa unha función *f(x,y)* cun código de cores (temperaturas: vermello=alto, azul = baixo, verde-marelo=medio).

```
[x y]=meshgrid(-3:0.05:3);
z=sin(x.*y);
contourf(x,y,z)
colorbar
```

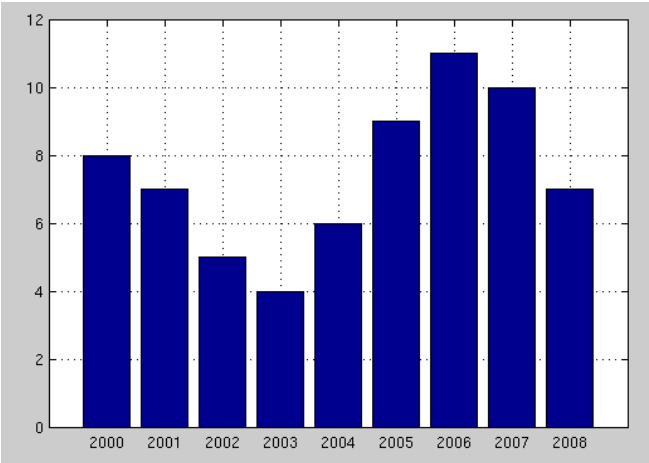
Engade a barra de cores na dereita



Gráficos especiais

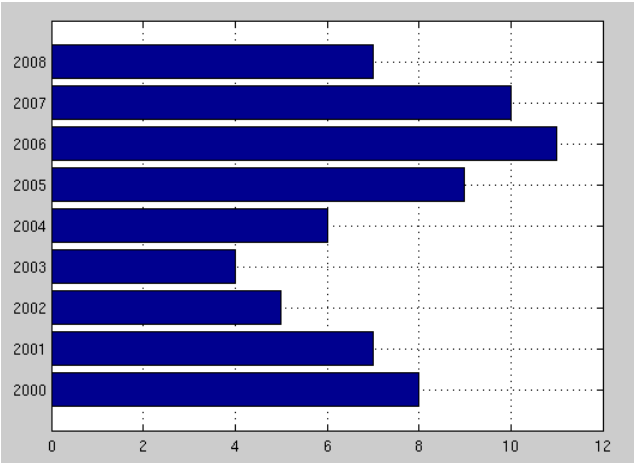
- Barras verticais:

x = [2000:2008]; y = [8 7 5 4 6 9 11 10 7];
bar(x, y); % x crecente



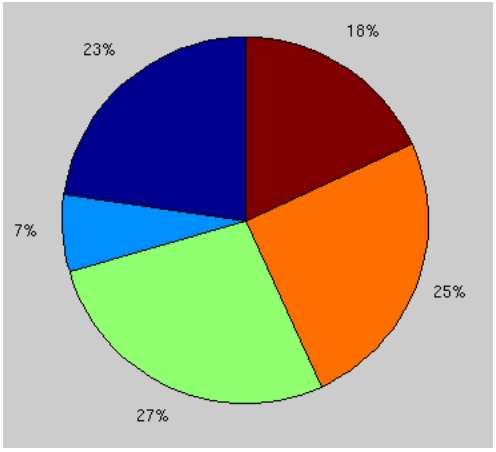
- Barras horizontais:

x = [2000:2008]; y = [8 7 5 4 6 9 11 10 7];
barh(x, y); % x crecente

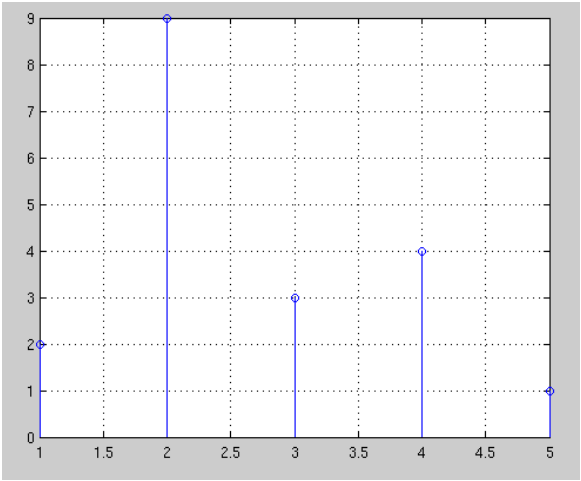


Gráficos especiais

- Tartas:
  x=[10 3 12 11 8];
  pie(x) % calcula os porcentaxes;sentido antihorario

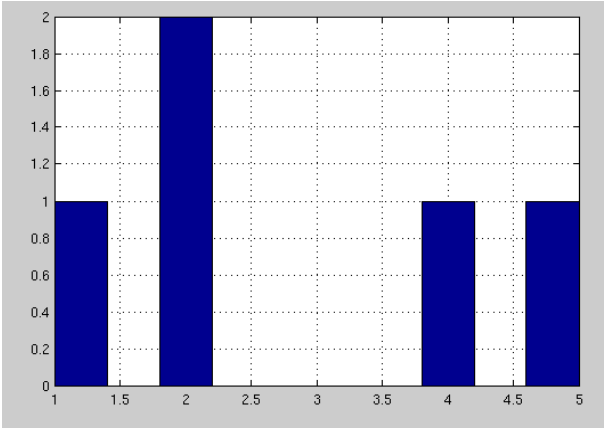


- Diagrama de troncos:
  x=[1:5];y=[2 9 3 4 1];
  stem(x,y) %x crecente

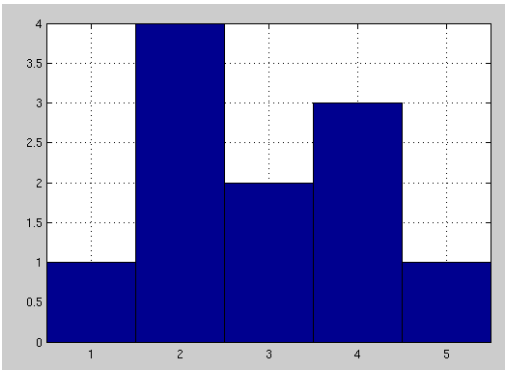


Histogramas (I)

- Cada barra vertical está asociada a un intervalo dos datos
- A altura de cada barra representa o nº de datos que se atopan no seu intervalo asociado
- Divide o rango dos elementos do vector en intervalos (10, por defecto)
- Ex: *y=[1 2 4 5 2];*
  hist(y)
- Con nº de intervalos:
  hist(y, 5)



Histogramas (II)



- Usando 2 vectores, para X e Y:

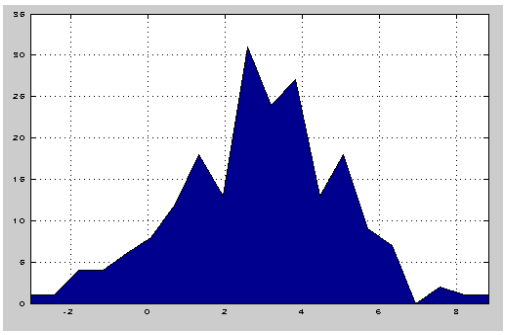
x=[1 2 3 4 5];y=[1 2 3 2 3 4 4 2 5 4 2];

hist(y, x)

- O 2º vector debe ser crecente (se non, erro)
- Para que o histograma sexa cunha liña continua:

med=3;desv=2;n=20;
x=normrnd(med,desv,1,200);
t=linspace(min(x),max(x),n);
area(t,hist(x,n));grid on

Números aleatorios seguindo unha distribución gausiana

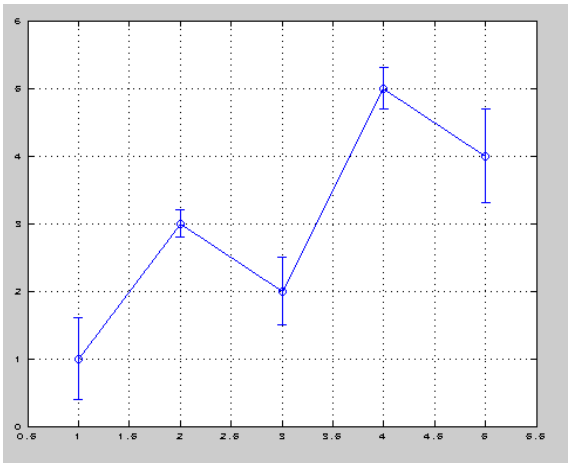


Gráficas con barras de erros

- Gráficas con barras de erros (p.e. para desviacións típicas de valores promedio):

errorbar(datos, erro, opcións)

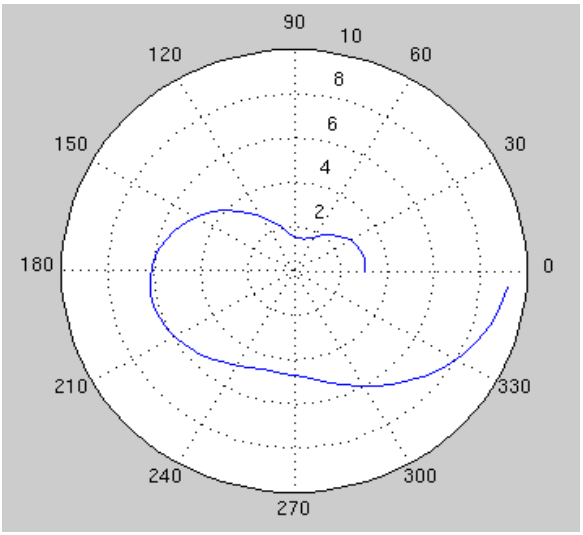
Ex: *x = [1 3 2 5 4];*
erro = [0.6 0.2 0.5 0.3 0.7];
errorbar(x, erro, 'o-')



Curvas en coordenadas polares

- Ecuación da curva en coordenadas polares: *radio = radio(ángulo)*: $\rho=\rho(\theta)$
- Ex: *polar(ángulo, radio)*

ángulo: vector cos valores do ángulo; *radio*: vector cos valores do radio




```
t=0:0.1:2*pi;
r=3*cos(t).^2+t;polar(t,r)
```

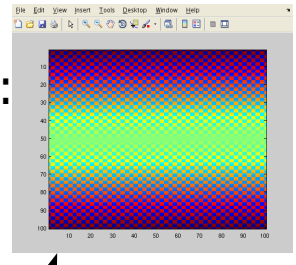
- *Función ezpolar($\rho(\theta)$,[θ_{min} θ_{max}])*

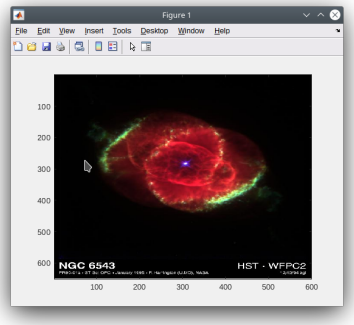
*ezpolar('3*cos(t)^2 + t', [0 2*pi])*

Máis funcións gráficas

- Gráficas para matrices:

  image(a): non escala
  imagesc(a): escala
  pcolor(a)

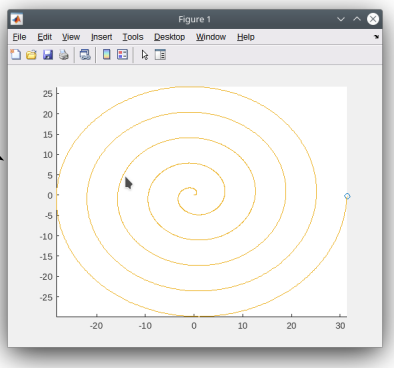




- Ex: *a=magic(10); image(a)*

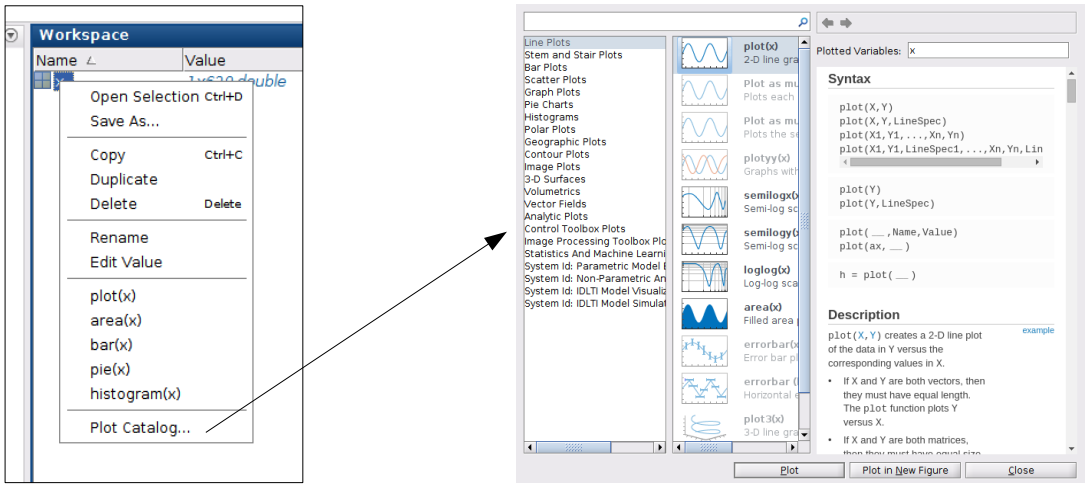
  x=imread('ngc6543a.jpg');image(x)

- Animacións: *comet(x), comet(x,y)*
- Ex: *t=0:0.01:10*pi; x=t.*cos(t);y=t.*sin(t); comet(x,y)*

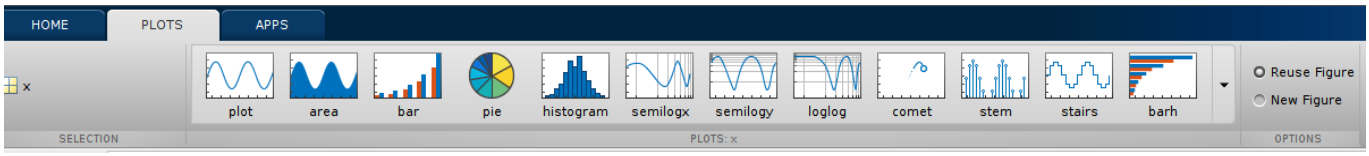


Asistente para gráficos

- Na ventá *workspace* pódese seleccionar un vector/matriz e o tipo de gráfico para representalo



- Tamén na barra de menú *Plots*



Exercicios

Representa as seguintes curvas e superficies:

1)Espiral de Arquímedes (polares): $r(\theta) = a\theta$

2)Bruxa de Agnesi: $y=\frac{8a^3}{x^2+4a^2}$

3)y=f(x) = tan x, en [-π/2, π/2] (usa `ezplot()`)

4)y=f(x) = $e^{-x/2}$ sen 20x, en [0, 10] (usa `fplot()`)

5) $y=\frac{x^2-x+1}{x^2+x+1}, x\in[-10,10]$

6)Representa os datos da táboa xunto coa función que os modela

$$y=320\left[\left(\frac{x}{210}\right)^{0.16}+1\right]$$

x	y
7E-5	345
2E-4	362
0.05	419
0.8	454
4.2	485
215	633
3500	831

Solucións aos exercicios

1) t = 0:0.1:pi; r = t; polar(t, r); (supoño a = 1)

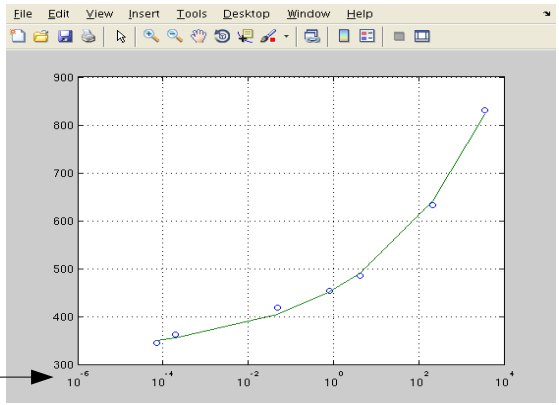
2) (supoño a = 1) fplot('8/(x^2 + 4)', [-10, 10])

3) ezplot('tan x', [-pi/2, pi/2])

*4) fplot('exp(-x/2)*sin(20*x)', [0, 10])*

5) fplot('(x^2 – x + 1)/(x^2 + x + 1)', [-10, 10])

6)
```
a=load('datos.dat');
s = 320*((a(:, 1)/210).^0.16 + 1);
semilogx(a(:, 1), a(:, 2), 'bo-', a(:, 1), s, 'gs-')
```

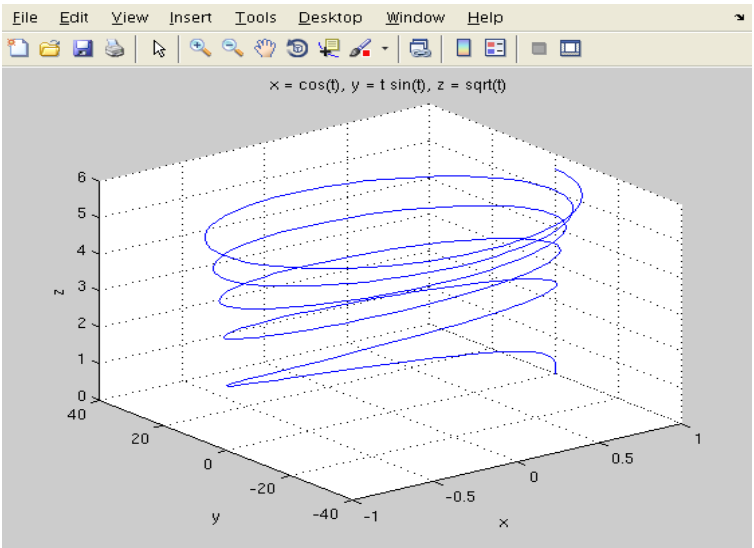


Curvas paramétricas en 3D: *ezplot3*

- Función *ezplot3('x(t)', 'y(t)', 'z(t)', [ini fin])*

  *ezplot3('cos(t)', 't*sin(t)','sqrt(t)', [0 10*pi])*

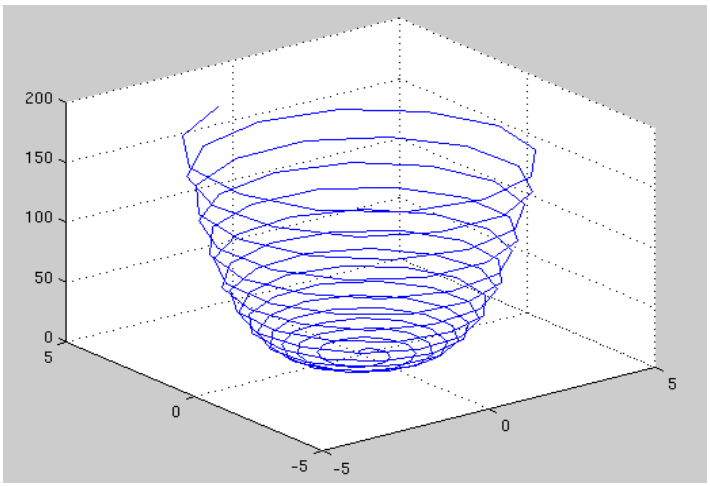
- Por defecto: rango *0..2*pi*



Curvas paramétricas en 3D: *plot3*

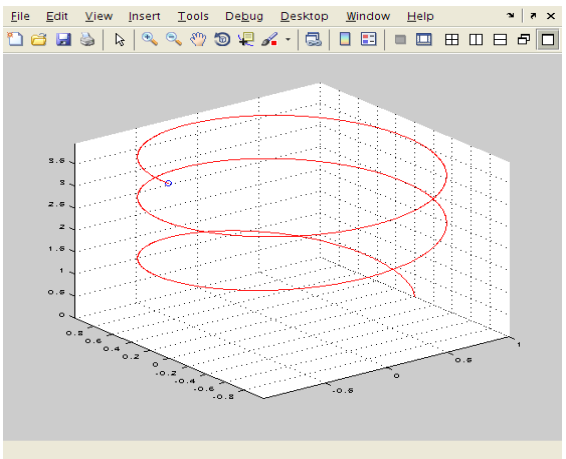
- Función *plot3(x,y,z,opcións)*
- *x, y, z*: vectores coas coordenadas dos puntos (ecuacións paramétricas); opcións: as mesmas que *plot*
- Exemplo:

```
t=0:0.1:6*pi;
x=sqrt(t).*sin(5*t);
y=sqrt(t).*cos(5*t);
z=t.^2./2;
plot3(x,y,z)
```



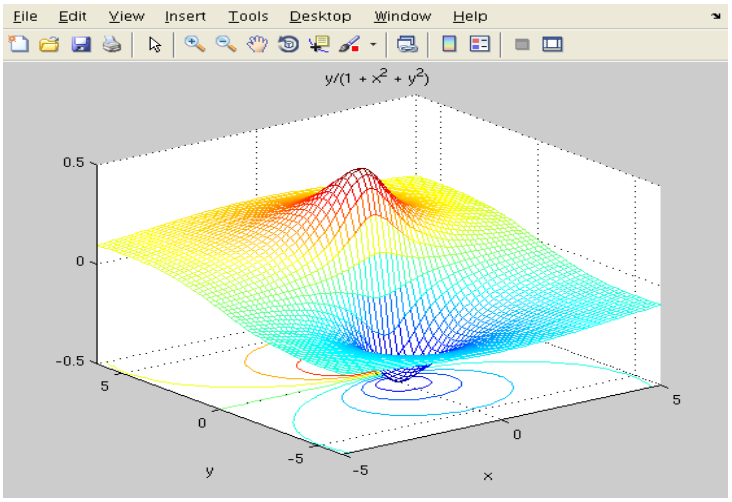
Curva animada en 3D: comet3

- Sintaxe: *comet3(x,y,z)*
- Vectores *x*,*y*,*z* da mesma dimensión
- Lonxitude elevada para que a animación sexa lenta
- Exemplo: *t=0:0.0001:10*pi; comet3(sin(t),cos(t),sqrt(t))*
- Moi útil para visualizar movementos en 3D



Superficies en 3D: *ezmesh, ezsurf*

- Función *ezsurf('f(x,y)',[xmin,xmax,ymin,ymax])*, e *ezmesh*
- Ex: *ezsurf('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi])*
- Tamén funcións *ezmesh(...)* e *ezmeshc(...)*, con contornos no plano XY



Superficies en 3D: *mesh*

Ecuación normal (explícita): *z = f(x, y)*

1. *[X Y]=meshgrid(x, y)* ou *[X Y]=meshgrid(x)*
   - *x,y*: vectores cos puntos en coordenadas. Ex: se os intervalos de representación son [0,1] para x e [-1,1] para y, podemos ter: *x=0:0.1:1* e *y=-1:0.1:1;*
   - *X,Y*: matrices coas coordenadas de tódolos puntos do plano XY para os cales se calcula *z = f(x,y)*

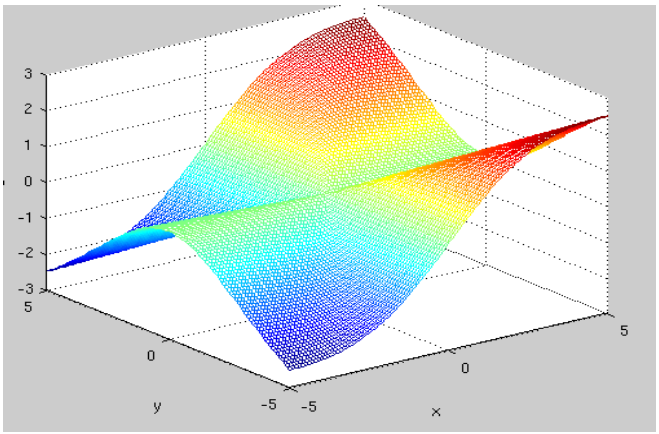
2. Cálculo de Z: *Z = f(X, Y)*. A expresión *f* debe estar vectorizada (operacións .* ./ .^)

3. Representación: *mesh(X, Y, Z)*

Superficies en 3D: *mesh*

- Exemplo: función $z=\frac{xy^2}{x^2+y^2}$ en [-5,5]x[-5,5]

```
x=[-5:0.1:5];y=[-5:0.1:5];
[X Y] = meshgrid(x, y);
Z = X.*Y.^2./(X.^2 + Y.^2 + eps);
mesh(X, Y, Z)
xlabel('x');ylabel('y');
zlabel('z')
```

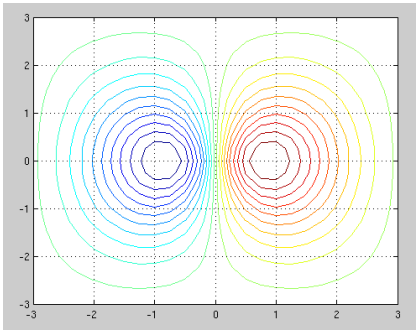


Superficies en 3D: *surf*, *contour*

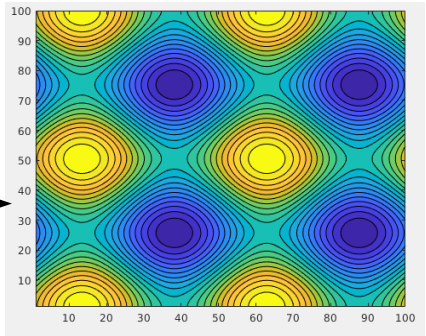
- Tamén se pode emprega-la función *surf(X,Y,Z)*, que rechea a superficie
- Contorno: *contour(X,Y,Z,n)*

  contorno da superficie 3D sobre o plano XY; n=nº niveis do contorno (opcional)

- Mapa de calor: *contourf(Z,n)*



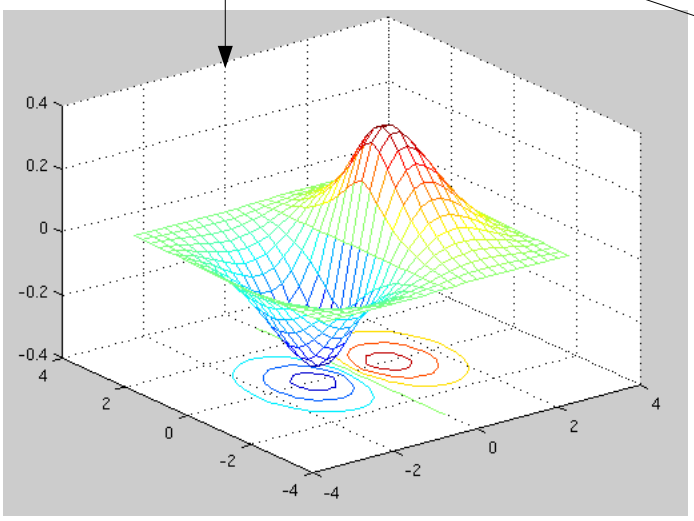

```
x = linspace(-2*pi,2*pi);
[X,Y] = meshgrid(x);
Z = sin(X)+cos(Y);
contourf(Z)
```

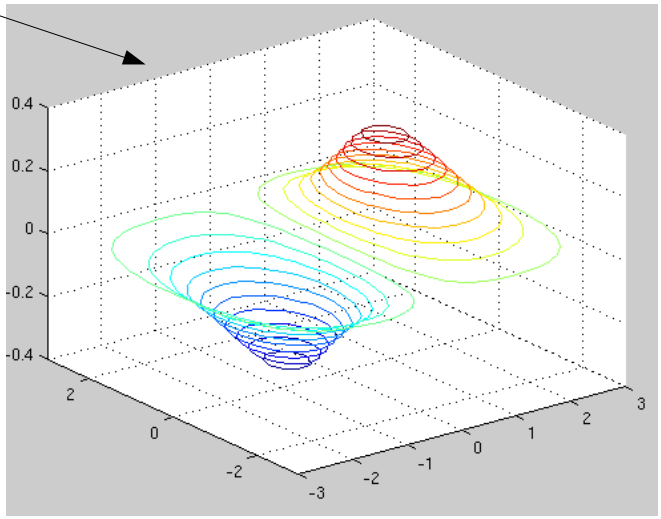


Superficies en 3D

$$z = f(x,y) = 1.8^{-1.5\sqrt{x^2+y^2}} \cdot \sin x \cdot \cos y$$

```
x=-3:0.25:3
[X Y] = meshgrid(x);
Z=1.8.^(-1.5*sqrt(X.^2+Y.^2)).*cos(0.5*Y).*sin(X);
meshc(X,Y,Z); contour3(X,Y,Z,20)
```

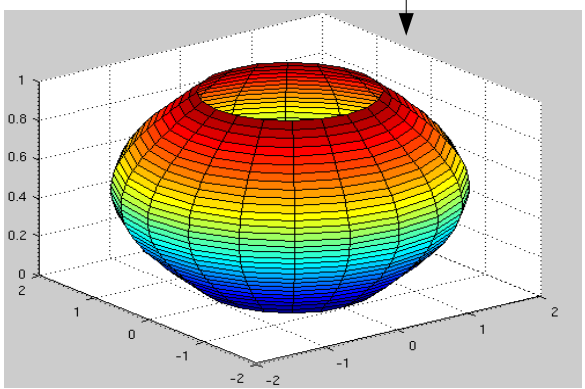


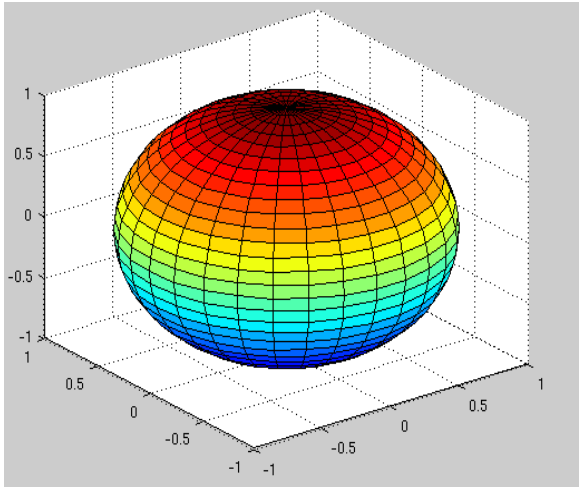


Outras gráficas

- Esfera: *sphere(50)* % nº de puntos
- Cilindro: *cylinder(r)* %r= vector cos radios

```
t=0:0.1:pi;
r=1+sin(t);
cylinder(r);
```





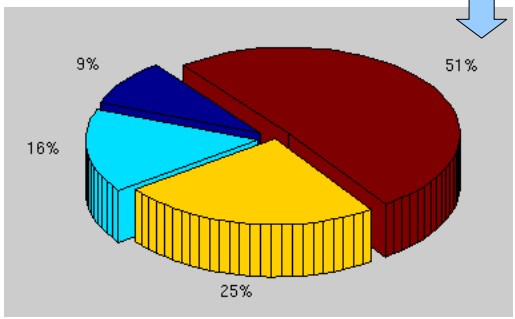
Outros diagramas 3D

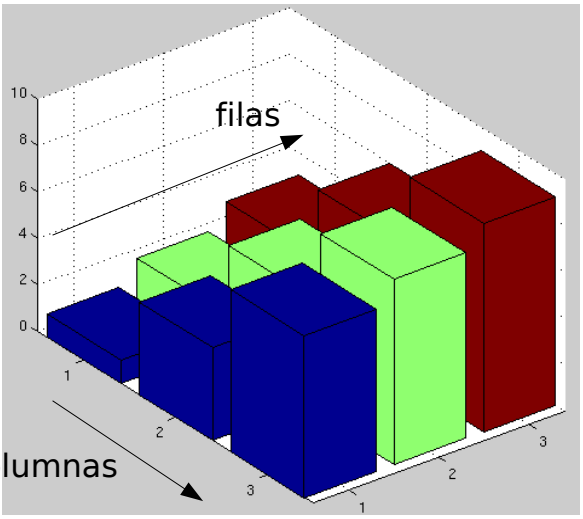
- Diagrama de barras 3D

  y=[1 2 3; 4 5 6; 7 8 9]; bar3(y)

- Tarta 3D (sentido antihorario): *x* e *s* deben ter igual lonxitude

  x = [5 9 14 29];
  s=[1 2 0 1]; %separación
  pie3(x); pie3(x, s)





Exercicios

Representa as seguintes curvas e superficies 3D:

1) $f(x,y)=\frac{sen\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}}\ x,y\in[-20,20]$

2) *x(t) = cos t – t sen t; y(t) = sen t – t cos t; z(t) = t^2*

3) *x(θ) = 1 + cos θ; y(θ) = 1 + sen θ; z(θ)=4θ*

4) $z=f(x,y)=\frac{1}{\pi}\sum_{n=1}^{10}\sin[(2n-1)\pi\ x]\sinh[(2n-1)\pi\ y],x,y\in[-0.1,0.1]$

5) *z=-x^2/4-y^2/4, x,y ∈[-4,4]* (fai o contorno 3D)

6) *z=(y+3)²+1.5x^2-x^2y, x,y∈[-3,3]* (ídem)

7) *x(t)=2cos(2πt), y(t)=2sen(2πt), z(t)=t*

8) $z=\frac{x}{x^2+y^2},x,y\in[-0.1,0.1]$

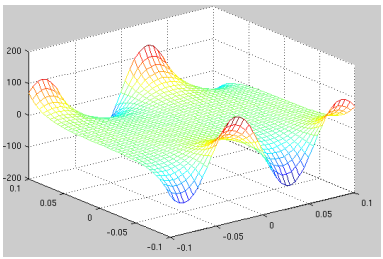
Solucións aos exercicios (I)

1) *ezmesh('sin(x^2 + y^2)/(x^2 + y^2)')*; ou ben: *[X Y] = meshgrid(-20:0.1:20); Z = sin(X.^2 + Y.^2)./(X.^2 + Y.^2) mesh(X, Y, Z)*

2) *ezplot3('cos(t)-t*sin(t)','sin(t)-t*cos(t)','t^2',[0 10*pi]);* ou ben: *t = 0:0.1:10*pi; x= cos(t) – t.*sin(t); y = sin(t) – t.*cos(t); z= t.*t; plot3(x, y, z)*

3) *ezplot3('1+cos(t)','1+sin(t)','4*t',[0 10*pi])* ou ben: *t = 0:0.1:10*pi; x= 1+cos(t); y=1+sin(t); z=4*t; plot3(x, y, z)*

4) *[X Y] = meshgrid(-0.1:0.005:0.1); Z = f(X, Y); mesh(X, Y, Z)*



```
function z = f(x, y)
z = 0;
for n = 1:10
   z = z + sin((2*n - 1)*pi*x).*sinh((2*n - 1)*pi*y);
end
z = z/pi;
end
```


Solucións aos exercicios (II)

5) ezmesh('-x^2/4 – y^2/4',[-4 4])
ou ben: [X Y] = meshgrid(-4:0.1:4); Z=-X.^2/4 -Y.^2/4; meshc(X, Y, Z)

*6) ezmesh('(y+3)^3 + 1.5x^2*y',[-3 3])*
ou ben: *[X Y] = meshgrid(-3:0.1:3); Z = (Y + 3).^2 + 1.5*X.^2 – X.^2.*Y; meshc(X, Y, Z)*

*7) ezplot3('2*cos(2*pi*t)','2*sin(2*pi*t)','t',[0 pi])*
ou ben: *t = 0:0.1:2*pi; x = 2*cos(2*pi*t); y = 2*sin(2*pi*t); z = t; plot3(x, y, z)*

8) ezmesh('x/(x^2 + y^2)',[-1 1])
ou ben: *[X Y] = meshgrid(-1:0.1:1); Z = X./(X.^2 + Y.^2); mesh(X, Y, Z)*

Programación en Octave
Exercicios clases interactivas

Semana 10

Traballo en clase

1. **Entrada/Saída básica. Vectores. Bucles definidos. Vectorización**. Escribe un programa chamado `sumatorio.m` que lea por teclado dous vectores **v** e **w**. Debes comprobar que teñen a mesma lonxitude n. O programa debe calcular:

$$s = \sum_{i=1}^{n} \sum_{j=1}^{i} v_i w_j \tag{1}$$

   Proba con $\mathbf{v} = (1, 2, 1)$ e $\mathbf{w} = (-1, 0, 1)$ e tes que obter $s = -3$.

```
clear
while 1
    v=input('vector v: ');n=numel(v);  % introducir vector entre corchetes
    w=input('vector w: ');
    if n==numel(w); break; end
end

% como en fortran
r=0;
for i=1:n
    t=0;
    for j=1:i
        t=t+w(j);
    end
    r=r+v(i)*t;
end
fprintf('r=%g\n',r);

% alternativa simple
r=0;
for i=1:n
    r=r+v(i)*sum(w(1:i));
end
fprintf('r=%g\n',r);

% alternativa mais curta
r=0;t=0;
for i=1:n
    t=t+w(i);r=r+v(i)*t;
end
fprintf('r=%g\n',r);
```

2. **Matrices**. Escribe un programa chamado `matriz.m` que lea por teclado unha matriz **A**, cadrada de orde n, e calcule:

   - A súa traza (suma dos elementos da diagonal principal), definida pola ecuación:

$$tr(A) = \sum_{i=1}^{n} a_{ii} \tag{2}$$

   - A suma do seu triángulo superior.
   - Determine se a matriz é simétrica: $A_{ij} = A_{ji}, i, j = 1, \ldots, n$.

```
clear
% introducir elementos entre corchetes, filas separadas por punto e coma
while 1
  a=input('matriz a [;]? ');[n,m]=size(a);
  if n==m; break; end
end
disp('a='); disp(a)

% traza
tr = trace(a);
%tr = sum(diag(a));
fprintf('tr = %g\n', tr);

% suma do triangulo superior
% sts = sum(sum(triu(a)) - diag(diag(a)));
sts = sum(sum(a - tril(a)));
fprintf('sts = %g\n', sts);

% matriz simetrica / non simetrica
if all(all(a == a'))
    disp('matriz simetrica');
else
    disp('matriz non simetrica');
end
```

3. **Números aleatorios**. Escribe un programa chamado `aleatorio.m` que defina $n = 10$ e cree un vector $\mathbf{x}$ con n números enteiros aleatorios entre 0 e 100. Queremos poñer estos valores nos triángulos superior e inferior dunha matriz cadrada $\mathbf{a}$. É decir, queremos poñer tódolos elementos de $\mathbf{x}$ no triángulo superior da matriz, e queremos poñer tódolos elementos de $\mathbf{x}$ tamén no triángulo inferior, de modo que $a_{ij} = a_{ji}$, con $j \neq i$. Polo tanto, a orde m da matriz $\mathbf{a}$ debe verificar que $\frac{m^2-m}{2}$, que é o número de elementos do triángulo superior ou inferior, sexa o mínimo número enteiro igual ou maior que n. Tomando a igualdade temos que $m^2 - m - 2n = 0$. Polo tanto, o programa debe crear unha matriz cadrada de orde $m = \left\lceil \frac{1+\sqrt{1+8n}}{2} \right\rceil$. No caso $n = 10$ temos que $m = 5$. Esta matriz debe ter valores nulos na diagonal ($a_{ii} = 0, i = 1 \ldots, m$) e os elementos do vector $\mathbf{x}$ nos triángulos superior e inferior. É dicir:

$$a_{12} = a_{21} = x_1, a_{13} = a_{31} = x_2, \ldots, a_{1m} = a_{m1} = x_m,$$
$$a_{23} = a_{32} = x_{m+1}, \ldots, a_{(m-1)m} = a_{m(m-1)} = x_n$$

```
clear
% para inicializar de forma distinta o xerador de numeros aleatorios
%  en cada execucion: rng('shuffle')
% para inicializar sempre da mesma forma: rng('default')
n=10;x=randi([0  100],1,n)
disp('x='); disp(x)
m=ceil((1+sqrt(1+8*n))/2);a=zeros(m);k=1;
for i=1:m
    for j=i+1:m
        t=x(k);a(i,j)=t;a(j,i)=t;k=k+1;
        if k>n; break; end
    end
    if k>n; break; end
end
disp('a='); disp(a)
```

Alternativa más corta usando `return`:

```
clear
n=10;x=round(100*rand(1,n));
disp('x='); disp(x)
m=floor((1+sqrt(1+8*n))/2);a=zeros(m);k=1;
```

```
for i=1:m
    for j=i+1:m
        t=x(k);a(i,j)=t;a(j,i)=t;k=k+1;
        if k>n
            disp('a=');disp(a);return
        end
    end
end
```

4. **Medida de tempos. Bucle indefinido**. Descarga o programa `tempo.m` desde este **enlace**. Este programa pide por teclado a introducción dun número natural n, comprobando que o número é positivo, e define un vector **x** con n elementos onde $x_i = i$, $i = 1 \ldots n$. Proba distintos xeitos de crear o vector **x** e calcula o tempo computacional que necesita cos comandos de Octave `tic` e `toc` para distintos valores de n. Usa $n = 10^5$.

```
% Eficiencia computacional
clear;n=0;
while n <= 0  % Ler o valor de n positivo
    n=round(input('Numero de iteracions (> 0): '));  % Usa n=1e5
end
%-------------------------------------------------
tic;x=[]; % inicia o reloxio e crea un vector baleiro
for i=1:n
    x=[x i];
end
% tempo transcorrido desde que se executou tic
fprintf('Tempo agregando elementos o vector=%g s.\n',toc)
%-------------------------------------------------
tic
for i=1:n
    y(i)=i;
end
fprintf('Tempo con vector baleiro= %g s.\n',toc)
%-------------------------------------------------
tic; z=zeros(1,n);  % con reserva de memoria
for i=1:n % declarando un vector de tamanho n
    z(i)=i;
end
fprintf('Tempo con reserva de memoria e for= %g s.\n',toc)
%-------------------------------------------------
tic; t=zeros(1,n); i=1;  % con while
while i<=n
    t(i)=i;i=i+1;
end
fprintf('Tempo con while= %g s.\n',toc)
%-------------------------------------------------
tic; u=1:n;  % vectorizado
fprintf('Tempo vectorizado %g s.\n',toc)
```

5. **Progreso dun programa**. Descarga o programa `progreso_octave.m` desde este **enlace**. Este programa imprime o seu progreso (en %).

```
n=1000000;
for i=1:n
    fprintf('%10.1f%%\r',100*i/n)
end
```

Este programa funciona en Octave ou executando Matlab dende a terminal de comandos (neste caso, debes engadir un comando `exit` ao final do programa. Se é dentro do entorno de Matlab, a secuencia de escape `r` non funciona e hai que facer o seguinte (arquivo **progreso_matlab.m**):

```
n=100000;
fprintf(repmat(' ',1,7));
for i=1:n
```

```
    fprintf(repmat('\b',1,7));fprintf('%6.2f%%',100*i/n);
end
fprintf('\n');
```

Descarga o programa `progreso2.m` desde este **enlace**. Este programa amplía o anterior para que mostre, en cada iteración, o tempo estimado que queda de execución. Usa para isto unha función chamada `strtime(t)` que transforma un tempo `t` numérico nunha cadea de caracteres da forma `Y y MM m DD d HH h MM m SS s`.

```
n=1000000;t=tic;
for i=1:n
    t2=toc(t);t3=t2*(n-i)/i;
    fprintf('%10.1f%% %40s\r',100*i/n,strtime(t))
end

function str=strtime(t)
    if t<60  % seconds in a minute
        str=sprintf('%2d s',floor(t));
    elseif t<3600  % seconds in an hour
        m=floor(t/60);s=floor(t-60*m);
        str=sprintf('%2d m %2d s',m,s);
    elseif t<86400  % seconds in a day
        h=floor(t/3600);m=floor((t-3600*h)/60);s=floor(t-3600*h-60*m);
        str=sprintf('%2d h %2d m %2d s',h,m,s);
    elseif t<2592000  % seconds in a month
        d=floor(t/86400);h=floor((t-86400*d)/3600);m=floor((t-86400*d-3600*h)/60);
        s=floor(t-86400*d-3600*h-60*m);str=sprintf('%2d d %2d h %2d m %2d s',h,m,s);
    elseif t<31536000  % seconds in a year
        month=floor(t/2592000);d=floor((t-2592000*month)/86400);
        h=floor((t-2592000*month-86400*d)/3600);
        m=floor((t-2592000*month-86400*d-3600*h)/60);
        s=floor(t-2592000*month-86400*d-3600*h-60*m);
        str=sprintf('%2d month %2d d %2d h %2d m %2d s',h,m,s);
    else
        y=floor(t/31536000);month=floor((t-31536000*y)/2592000);
        d=floor((t-31536000*y-2592000*month)/86400);
        h=floor((t-31536000*y-2592000*month-86400*d)/3600);
        m=floor((t-31536000*y-2592000*month-86400*d-3600*h)/60);
        s=floor(t-31536000*y-2592000*month-86400*d-3600*h-60*m);
        str=sprintf('%1d y %2d month %2d d %2d h %2d m %2d s',h,m,s);
    end
end
```

Exercicios propostos

1. Escribe un programa `producto.m` que lea por teclado dous vectores **v** e **w** e unha matriz **A**, todos eles da mesma orde n, e calcule o produto:

$$\mathbf{vAw}' = [v_1 \dots v_n] \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} w_1 \\ \dots \\ w_n \end{bmatrix}$$

```
clear
v = input('vector v: '); % introducir vector entre corchetes
w = input('vector w: '); % introducir vector entre corchetes
a = input('matriz a: '); % elementos entre corchetes, filas separadas por punto e coma
p = v*a*w';
fprintf('prod = %g\n', p);
```

2. Escribe un programa `fibonacci.m` que imprima os $n+1$ primeros termos da serie de Fibonacci, dados por $F_0 = 1; F_1 = 1; F_i = F_{i-1} + F_{i-2}, i = 2, ...n$. Executa o programa para n=8, e debes obter 0, 1, 2, 3, 5, 8, 13, 21 e 34.

```
clear
n=input('n(>1)? ');
f0=0;f1=1;
printf('%10s %10s\n','i','F(i)')
printf('%10i %10i\n',0,f0)
for i=1:n
    f2=f0+f1;
    fprintf('%10i %10i\n',i,f2)
    f0=f1;f1=f2;
end
```

3. Escribe un programa `cadrado.m` que defina unha matriz **a** 5×8 onde cada elemento con fila e columna par sexa a raíz cadrada da suma dos índices do elemento. Nos restantes elementos, o valor debe se-lo cadrado da suma dos índices.

```
clear all;clc
n=5;m=8;a=zeros(n,m);
for i=1:n
    for j=1:m
        if rem(i,2)==0 && rem(j,2)==0
            a(i,j)=sqrt(i+j);
        else
            a(i,j)=(i+j)^2;
        end
    end
end
disp('a=');disp(a)
```

4. Escribe un programa `serie.m` en Octave que calcule a suma dos m primeiros termos da serie:

$$\sum_{n=0}^{m} \frac{(-1)^n}{2n+1} \tag{3}$$

Proba con $m = 10$ e $m = 100$, e compara o resultado con $\pi/4$ (suma da serie infinita).

```
clear all;clc
m=input('m? ');
s=0;
for n=0:m
    s=s+(-1)^n/(2*n+1);
end
printf('m=%i s=%g serie infinita=%g\n',m,s,pi/4)
```

Semana 11

Traballo en clase

1. **Funcións propias. Vectorización**. Escribe un programa chamado `intervalo.m` cunha función `funcionf(x)` en Octave que calcule $f(x)$ definida por:

$$f(x) = \begin{cases} 4e^{x+2} & -6 \le x < -2 \\ x^2 & -2 \le x < 2 \\ (x+6.5)^{1/3} & 2 \le x < 6 \end{cases}$$

O programa debe calcular os valores de $f(x)$ para $x \in [-10, 10]$ e representalos gráficamente.

```
clear
x = -10:0.1:10; n = length(x); y = zeros(1,n);
for i = 1:n
    y(i) = funcionf(x(i));
```

```
end
plot(x, y)
grid on
%-------------------------
function y = funcionf(x)
  if x < -6
    y = 0;
  elseif x < -2
    y = 4*exp(x+2);
  elseif x <= 2
    y = x*x;
  elseif x < 6
    y = (x+6.5)^(1/3);
  else
    y = 0;
  end
end
```

Versión vectorizada:

```
x = -10:0.1:10;
y = funciony(x);
plot(x,y)
%-------------------------
function y = funcionf(x)
n = numel(x); y = zeros(1, n);
t = find(x >= -6 & x < -2); y(t) = 4*exp(x(t)+2);
t = find(x >= -2 & x < 2); y(t) = x(t).^2;
t = find(x >= 2 & x <= 6); y(t) = nthroot(x(t) + 6.5, 3);
```

2. **Chamada a funcións anónimas con `feval()`. Gráficas simultáneas con lendas**. Escribe un programa chamado `grafica.m` que represente gráficamente na mesma gráfica e no intervalo $[-\pi, \pi]$, con 100 puntos, as seguintes funcións: $f_1(x) = \sin x \cos x$, como función inline (en cor azul); $f_2(x) = \sin\cos 2x$ como función anónima (en cor vermella); $f_3(x) = \sin x$ como referencia a función con `str2func` (en cor verde); e $f_4(x) = \cos x$ como referencia a función con `@` (en cor negra). Engade etiquetas de texto coas expresións das distintas curvas.

```
clear
f{1}=inline('sin(x).*cos(x)'); % funcion inline
f{2}=@(x) sin(cos(2*x)); % funcion anonima
f{3}=str2func('@(x) sin(x)'); % referencia a funcion con str2func
f{4}=@cos;  % referencia con @
n=numel(f);m=100;x=linspace(-pi,pi,m);y=zeros(n,m);
c={'b','r','g','k'};clf;hold on
for i=1:n
    y=feval(f{i},x);plot(x,y,c{i})
%   plot(x,f{i}(x),c{i})   % alternativa
end
label={'sin(x)cos(x)'};  % func2str non funciona con inline
for i=2:n; label{i}=func2str(f{i}); end
legend(label,'location','bestoutside');grid on
```

3. **Resolución de sistema de ecuacións lineares mediante Eliminación Gaussiana**. Consideremos un sistema de n ecuacións lineais con n incógnitas:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= a_{1,n+1} \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= a_{2,n+1} \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n &= a_{3,n+1} \\
&\ldots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= a_{n,n+1}
\end{aligned}$$

O método de eliminación gausiana consiste en transformar as ecuacións sumando a tódolos elementos dunha fila o produto dun escalar polo elemento correspondente doutra fila, para transformar este sistema no seguinte:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= a_{1,n+1} \\ a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= a_{2,n+1} \\ a_{33}x_3 + \ldots + a_{2n}x_n &= a_{3,n+1} \\ \ldots & \\ a_{nn}x_n &= a_{n,n+1} \end{aligned} \tag{4}$$

onde os a_{ij} **non** son os iniciais. Concretamente, para cada incógnita x_i, con $i = 1, \ldots, n-1$ (coa última incógnita non hai que facer nada), substitúe a ecuación j, con $j = i+1, \ldots, n$, pola ecuación j menos a ecuación i multiplicada por $l = a_{ji}/a_{ii}$. Deste modo, o coeficiente da incógnita x_i na ecuación j (con $j = i+1, \ldots, n$) pasa a ser nulo. Dito doutra forma, a columna i da matriz ampliada pasa a valer 0 por debaixo da diagonal. No sistema transformado (ecuación 4 anterior) podemos despexar directamente x_n, sustituir na $(n-1)$-ésima ecuación e despexar x_{n-1} e así sucesivamente ata calcula-las n incógnitas, mediante a fórmula.

$$x_i = \frac{1}{a_{ii}} \left(a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j \right) \quad i = n, \ldots, 1 \tag{5}$$

Isto vale se $a_{ii} \neq 0$ para todo $i = 1, \ldots, n$. Pero se $\exists i$ tal que $a_{ii} = 0$, non se pode dividir por a_{ii} e resulta necesario realizar o denominado "pivote", consistente en intercambiar a ecuación i-ésima por aquela ecuación p posterior (é decir, $p > i$) que ten o coeficiente a_{pi} con valor absoluto máximo, coa intención de que $a_{pi} \neq 0$. Este cambio permitirá facer que a_{ii} pase a ser non nulo, a non ser que se trate dun sistema compatíbel indeterminado, porque neste caso unha ou varias das derradeiras ecuacións serán nulas e non será posíbel intercambiar filas para que $a_{ii} \neq 0$. Escribe un programa chamado `gauss.m` que implemente este método de resolución de sistemas de ecuacións lineares. Proba cos seguintes sistemas:

a) **Sistema compatíbel determinado sen pivote**. Solución: $x = 3, y = -2, z = 5$. Arquivo `sistema_compatibel_determinado.txt`:

$$\begin{matrix} 1 & 2 & 1 & 4 \\ -3 & -2 & 9 & 40 \\ 4 & 9 & 6 & 24 \end{matrix}$$

b) **Sistema compatíbel determinado con pivote** (coeficiente nulo na diagonal). Solución: $x = 1, y = 0, z = -1$. Arquivo `sistema_compatibel_determinado_pivote.txt`:

$$\begin{matrix} 0 & 2 & -1 & 1 \\ 1 & -1 & 1 & 0 \\ 2 & 0 & -1 & 3 \end{matrix}$$

c) **Sistema compatíbel indeterminado con solución de dimensión 1**. Ecuacións do subespazo vectorial solución de dimensión 1 (recta): $x + 3y = 4, z = -11$. Arquivo `sistema_compatibel_indeterminado_recta.txt` seguinte:

$$\begin{matrix} 1 & 3 & 0 & 4 \\ -1 & -3 & 0 & -4 \\ 2 & 6 & 1 & -3 \end{matrix}$$

d) **Sistema compatíbel indeterminado con solución de dimensión 2**. Ecuación do subespazo vectorial solución de dimensión 2 (plano): $x + 2y + 2z = 1$. Arquivo `sistema_compatibel_indeterminado_plano.txt` seguinte:

$$\begin{matrix} 1 & 2 & 3 & 1 \\ 2 & 4 & 6 & 2 \\ 3 & 6 & 9 & 3 \end{matrix}$$

e) **Sistema incompatíbel**. Solución de norma mínima x=0.081, y=0.163, z=0.244, con $|\mathbf{ax} - \mathbf{b}|$=3.27327. Arquivo `sistema_incompatibel.txt`:

$$\begin{matrix} 1 & 2 & 3 & 0 \\ 2 & 4 & 6 & 5 \\ 3 & 6 & 9 & 2 \end{matrix}$$

Emprega o depurador de Octave/Matlab para:

a) Deter a execución do programa nunha determinada sentenza (establecendo un punto de ruptura ou *breakpoint* co menú *Debug-Set/Clear breakpoint* ou coa tecla F12).

b) Executar as seguintes sentenzas paso a paso (menú *Debug-Step* ou tecla F10).

c) Entrar nunha función (menú *Debug-Step in* ou tecla F5).

d) Ver os valores das variábeis do programa (escribindo o seu nome na ventá de comandos, poñendo o rato enriba da variábel ou mirando o seu valor na ventá *Workspace*).

e) Continuar a execución (menú *Debug-Continue* ou coa tecla F5).

f) Ou deter a execución (menú *Debug-Exit debug mode* ou tecla MAY+F5), saíndo do modo de depuración.

```
clear
% sist_comp_det.txt, sist_comp_det_pivote, sist_comp_indet, sist_incomp
nf=input('arquivo? ','s');a=load(nf);[n m]=size(a); %a=matriz ampliada
if (n+1)~=m
    fprintf('#ecuacions~=#incognitas');return
end
disp('a=');disp(a);disp('--------------------');
c=a(:,1:n);b=a(:,m);rmc=rank(a(:,1:n));rma=rank(a);
if rmc~=rma
    fprintf('sistema incompatibel: rmc =% i rma =% i\n',rmc,rma);x=pinv(c)*b;
    fprintf('solucion de norma minima: ');disp(x);fprintf('|ax-b|=%g\n',norm(a*x-b))
    return
end
for i=1:n-1
    % se a(i,i)==0, pivote: intercambia ec. i coa que ten a(i,i) maximo
    if 0==a(i,i)
        % suma i a j porque p=1 para ec. i+1
        [~,j]=max(abs(a(i+1:n,i))); p=i+j;
        % pivote: intercambia ecuacion i por p
        aux=a(i,:);a(i,:)=a(p,:);a(p,:)=aux;
    end
    if 0~=a(i,i)  % se o sistema e indeterminado, pode ser a(i,i)==0
        t=a(i,i);u=a(i,:);
        for j=i+1:n
            l=a(j,i)/t;a(j,:)=a(j,:)-l*u;
        end
    end
    disp(a); disp('--------')
end
if ra<n
    fprintf('sistema compatibel indeterminado\n')
    d=n-rmc;fprintf('solucion de dimension % i:\n',d)
    fprintf('ecuacion implicita da solucion:\n');disp(a(1:rmc,:))
    fprintf('ecuacion parametrica da solucion:\nx=\n')
    c=a(:,1:n);b=a(:,end);x0=pinv(c)*b;k=null(c);v=1:size(k,2);
    fprintf('%4.2g ',x0(1));fprintf('+ c%i(%4.2g)',v,k(1,:));fprintf('\n')
    for i=2:n
        fprintf('%4.2g ',x0(i));fprintf('    (%4.2g)',k(i,:));fprintf('\n')
    end
    return
end
x=zeros(1,n);
for i=n:-1:1
    j=i+1:n;x(i)=(a(i,m)-a(i,j)*x(j)')/a(i,i);
end
fprintf('sistema compatibel determinado\n')
fprintf('x  =');disp(x)
fprintf('a\b=');  disp((c\b)')
```

4. **Escrutinio de lotería primitiva**. Escribe un programa `loteria.m` que realice o escrutinio de apostas de lotería primitiva. O programa debe pedir por teclado o nome dun arquivo de texto, e ler neste arquivo as apostas (en cada

fila, unha aposta, integrada por 6 números enteiros entre 1 e 49). Logo debe pedir por teclado o nome dun arquivo coa aposta ganhadora (unha liña con 6 números enteiros entre 1 e 49). Por último, debe almacenar nun arquivo (de nome introducido por teclado) tódalas apostas con 3 ou máis acertos: o n^o de aposta, o n^o de acertos e a súa combinación de números asociada. Empregar os seguintes arquivos de mostra:

```
apostas.txt
3   5  19  16  33  41
21   9   1  12  44  20
12  24   1  23  47  28
 3   5  41  25  19   1
18  12  11   1   8   9
11  33  21  45   1  12
ganhadora.txt
3 5 19 16 33 41
```

SOLUCIÓN:

```
clear all
f1=input('ficheiro apostas? ','s');
f2=input('ficheiro ganhadora? ','s');
f3=input('ficheiro acertos? ','s');
f=fopen(f3,'w');
if -1==f
    error('read %s\n',f3);
end
ap=load(f1);g=load(f2);
[nap n]=size(ap);
for i=1:nap
    acertos=0;
    for j=1:n
        if any(ap(i,j)==g)
            acertos=acertos+1;
        end
    end
    if acertos>3
        fprintf(f,'%i:',i);fprintf(f,'%i ',ap(i,:));
        fprintf(f,'acertos=%i\n',acertos);
    end
end
fclose(f);
```

Exercicios propostos

1. Escribe un programa `estatistica.m` que lea dende o arquivo `estatistica.txt` os seguintes datos:

```
31 26 30 33 33 39 41 41 34 33 45 42 36 39 37 45 43 36 41 37 32 32 35 42 38 33 40 37 50
37 24 28 25 21 28 46 37 36 20 24 31 34 40 43 36 34 41 42 35 38 36 35 33 42 42 37 26 31
```

O programa debe calcular, para cada fila: o valor medio, os valores por riba e por baixo da media, os valores dunha fila superiores ao seu correspondente da outra fila, os valores que coinciden co valor da outra fila na mesma posición, e os valores inferiores a 40.

```
clear all;clc
x=load('estatistica.txt');
%-------------------------------------------------------
y=x(1,:);m=mean(y);
printf('fila 1: media=%g\n',m)
printf('valores>media: ');printf('%i ',y(y>m));printf('\n')
printf('valores<media: ');printf('%i ',y(y<m));printf('\n')
printf('valores>outra fila: ');printf('%i ',y(y>x(2,:)));printf('\n')
printf('valores=outra fila: ');printf('%i ',y(y==x(2,:)));printf('\n')
printf('valores<40: ');printf('%i ',y(y<40));printf('\n')
%-------------------------------------------------------
```

```
y=x(2,:);m=mean(y);
printf('fila 2: media=%g\n',m)
printf('valores>media: ');printf('%i ',y(y>m));printf('\n')
printf('valores<media: ');printf('%i ',y(y<m));printf('\n')
printf('valores>outra fila: ');printf('%i ',y(y>x(1,:)));printf('\n')
printf('valores=outra fila: ');printf('%i ',y(y==x(1,:)));printf('\n')
printf('valores<40: ');printf('%i ',y(y<40));printf('\n')
```

2. **Función con varios valores retornados**. Escribe un programa `varios.m` que lea por teclado un número enteiro $n > 0$ e cree unha matriz **a** cadrada de orde n con valores enteiros aleatorios no conxunto $\{0, \ldots, n\}$. Logo, este programa debe chamar a unha función de Octave `calcula(...)` (debes decidir os seus argumentos), que retorne unha matriz **b** cadrada e un vector **x**, ambos de orde n, e un escalar y. A matriz **b** retornada debe ter tódolos elementos nulos agás os das diagonais principal e secundaria, que deben coincidir cos da matriz **a**. O vector **x** retornado debe verificar que x_i, con $i = 1, \ldots, n$, sexa a suma dos elementos iguais a i na columna i da matriz **a**. Pola súa banda, o escalar y retornado debe ser o número de elementos nulos na matriz **a**. Finalmente, o programa principal debe pedir por teclado o nome dun arquivo e almacenar neste arquivo as matrices **a** e **b**, o vector **x** e o escalar y.

   SOLUCIÓN:

```
clear
n=0;
while n<=0
    n=round(input('n? '));
end
a=round(n*rand(n));
[b v ceros]=calcula(a);
% gardar en arquivo
arquivo=input('Introduce nome arquivo: ','s');
f=fopen(arquivo,'w');
if f<0
    error('fopen %s\n', arquivo);
end
fprintf(f,'Matriz a:\n');
for i=1:n
    fprintf(f,'%d ',a(i,:));fprintf(f,'\n');
end
fprintf(f,'Matriz b: \n');
for i=1:n
    fprintf(f,'%d ',b(i,:));fprintf(f,'\n');
end
fprintf(f,'Vector v: ');fprintf(f,'%d ',v);
fprintf(f,'\nCeros= %d\n',ceros);
fclose(f);
%-------------------------------------------------------------
function [b, x, y]=calcula(a)
% calcula: fai calculos sobre unha matriz de enteiros a
% b e a matriz a con ceros fora da diagonal principal e secundaria
% xi e a suma dos elementos de a que son igual a i
% c e o numero de elementos de a que son igual a 0
    n=size(a,2); m = n + 1;
    b=diag(diag(a));x=zeros(1,n);
    for i=1:n
        b(i,m-i)=a(i,m-i);
        x(i)=i*sum(a(:,i)==i);
    end
    y=sum(sum(a==0));
end
```

Semana 12

Traballo en clase

1. **Lectura dende arquivo en formato R con fscanf**. Escribe un programa chamado `le_arquivo_R.m` que lea os datos (numéricos e caracteres) dende `arquivo_R.txt` seguinte, que tamén se pode ler en R coa función `read.table(...)`:

	E1	E2	E3	E4	Saida
1	0.25	0.33	1.23	−0.51	Branco
2	−0.34	1.3E5	0.22	4.3	Negro

```
clear
fn='arquivo_R.txt';f=fopen(fn);
if -1==f
  error('read %s',fn)
end
s=strsplit(fgetl(f));s(1)=[];m=numel(s)-1;n=0;
while ~feof(f)
    fgetl(f);n=n+1;
end
frewind(f);x=zeros(n,m);y=cell(1,n);fgetl(f);
for i=1:n
  fscanf(f,'%i',1);x(i,:)=fscanf(f,'%g',m);y{i}=fscanf(f,'%s',1);
end
fclose(f);
%-------------------------------
fprintf('%6s',s{:})
for i=1:nf
  fprintf('%6g ',x(i,:));fprintf('%6s\n',y{i})
end
```

2. **Mínimos cuadrados**. Crea o arquivo de texto `regresion.txt` co seguinte contido:

```
0.5 1.9 3.3 4.7 6.1 7.5
0.8 10.1 25.7 59.2 105 122
```

Escribe un programa chamado `regresion.m` que lea os datos do arquivo `regresion.txt` aos vectores $\mathbf{x}$ e $\mathbf{y}$, un en cada liña do arquivo, ambos da mesma lonxitude. O programa debe mostrar por pantalla ambos vectores e as constantes a e b que axustan os vectores $\mathbf{x}$ e $\mathbf{y}$ á recta de regresión $y = ax + b$, usando as seguintes expresións:

$$a = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \qquad b = \frac{\left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i\right) - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} x_i y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{6}$$

sendo n o tamaño dos vectores $\mathbf{x}$ e $\mathbf{y}$. Representa gráficamente os vectores $\mathbf{x}$ e $\mathbf{y}$ xunto cos valores de y calculados segundo a ecuación da recta.

Solución 1: un programa que codifique as expresións:

```
clear all;clc
datos=load('regresion.txt');
x=datos(1,:);y=datos(2,:);
n=numel(x);
disp('x=');disp(x)
disp('y=');disp(y)
sx=sum(x);sy=sum(y);
sxy=sum(x.*y);sx2=sum(x.^2);
a=(n*sxy-sx*sy)/(n*sx2-sx^2);
b=(sx2*sy-sx*sxy)/(n*sx2-sx^2);
fprintf('y=(%g)x+(%g)\n',a,b);
%representacion grafica-----------------------
x_recta=[min(x) max(x)];y_recta=a*x_recta+b;
```

```
figure(1)
hold on
plot(x,y,'bs','markerfacecolor','b')
plot(x_recta,y_recta,'r-');
hold off;grid on
xlabel('X');ylabel('Y')
title('Axuste a recta')
```

Solución 2: utilizando as funcións *polyfit()* e *polyval()* definidas en MATLAB:

```
clear all;clc
datos=load('regresion.txt');
x=datos(1,:);y=datos(2,:);
n=numel(x);
disp('x=');disp(x)
disp('y=');disp(y)
% coeficientes a=p(1) e b=p(2): y=a*x+b
p=polyfit(x,y,1);
fprintf('y=(%g)x+(%g)\n',p(1),p(2));
%representacion grafica
y_recta=polyval(p,x);
plot(x,y,'bs','markerfacecolor','b',x,y_recta,'r-');
grid on
xlabel('X');ylabel('Y')
title('Axuste a recta usando polyfit')
```

3. **Serie de Fourier dunha función. Cálculo simbólico e numérico de integrais definidas**. Dada unha función $f(x)$, a súa serie de Fourier $F(x)$ está dada por:

$$F(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos nx + b_n \operatorname{sen} nx), \quad a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos nx dx, \quad b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\operatorname{sen} nx dx \tag{7}$$

Escribe un programa en Octave chamado `fourier.m` que lea por teclado a expresión analítica dunha función (proba con x^2 e con x^3) chame á función `coef(...)`, pasándolle os argumentos axeitados, que calcule os coeficientes $a_n, n = 0,\ldots,m$ e $b_n, n = 1,\ldots,m$ para m=20. Calcula as integrais definidas usando os comandos `int` e `eval` de cálculo simbólico. Logo, o programa principal debe calcular o valor da función $f(x)$ e máis da súa serie de Fourier $F(x)$ para 100 valores de x no intervalo $[-\pi,\pi]$, representar gráficamente ambas e mostrar a media dos valores absolutos das diferencias e máis o tempo empregado.

```
clear;tic
s=input('f(x)? ','s');
f=str2func(sprintf('@(x) %s',s));
m=20;n=100;j=1:m;
[a,b]=coef(s,m);
x=linspace(-pi,pi,n);
y=zeros(1,n);Y=zeros(1,n);
for i=1:n
    z=x(i);y(i)=f(z);u=j*z;
    Y(i)=a(1)+sum(a(2:end).*cos(u)+b.*sin(u));
end
clf;plot(x,y,'b-',x,Y,'r-')
grid;legend({'f(x)','F(x)'})
fprintf('erro=%g tempo=%g s\n',mean(abs(y-Y)),toc)
%-------------------------------------------------
function [a,b]=coef(s,m)
a=zeros(1,m+1);b=zeros(1,m);
syms x;f=str2sym(s);
a(1)=eval(int(f,x,-pi,pi))/2;
for n=1:m
    a(n+1)=eval(int(f*cos(n*x),x,-pi,pi));
    b(n)=eval(int(f*sin(n*x),x,-pi,pi));
end
```

```
a=a/pi;b=b/pi;
end
```

Exercicios propostos

1. **Lectura con `fscanf` detectando fin de arquivo**. Escribe un programa chamado `le_arquivo_eof.m` que lea tódolos datos de `arquivo_eof.txt` seguinte:

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

```
clear
f=fopen('arquivo_eof.txt','r');
if -1==f
    error('arquivo_eof.txt non existe')
end
% x=fscanf(f,'% i',inf); % le todolos datos a vector x
while ~feof(f)
    x=fscanf(f,'% i',1); % le dato a dato e mostra por pantalla
    fprintf('% i ',x);
end
fprintf('\n')
fclose(f);
```

2. Escribe unha función en Octave chamada `maxoumin` que calcule o valor máximo ou mínimo dunha función cuadrática $f(x) = ax^2 + bx + c$. A función en Octave debe te-la forma `[x y w] = maxoumin(a,b,c)`, onde `x` é o valor de x que maximiza/minimiza $f(x)$, `y` é o valor máximo/mínimo de $f(x)$ e `w` sexa 1 se é un máximo e 2 se é un mínimo.

```
function [x y w]=maxoumin(a,b,c)
% maxoumin: busca o maximo / minimo de f(x) = ax^2 + bx + c
% nos extremos f'(x)=0: 2ax+b=0 -> x=-b/2a
if a~=0
    x=-b/(2*a);y=a*x^2+b*x+c;
    if a>0
        w=1;
    else
        w=2;
    end
else
    x=[];y=[];w=0;
end
end
```

3. Descarga o seguinte arquivo de texto cos datos dos elementos químicos da táboa periódica ordeados por número atómico (arquivo **taboaperiodica.txt**). As columnas deste arquivo teñen os seguintes significados: 1) abreviatura do elemento, 2) nome do elemento e 3) peso atómico en gramos por mol. Escribe un programa en Octave chamado `atomico.m` que pida repetidamente por teclado a abreviatura dun elemento químico e visualice na pantalla o seu nome completo e peso atómico (ten que atopar a información no arquivo anterior). O programa ten que rematar cando se introduza por teclado unha X. Usa a función `upper()`, que convirte unha cadea de caracteres en maiúsculas.

```
clear all;clc
f=fopen('taboaperiodica.txt', 'r');
if -1==f
  error('Erro abrindo taboaperiodica.txt');
end
taboa=textscan(f,'%s %s %f');
% funcion upper() convirte a maiusculas
abrev=upper(taboa{1}); % primeira columna (vector de celdas)
elementos=taboa{2}; % segunda columna
pa=taboa{3}; % terceira columna (vector de numeros)
while 1
```

```
    el=upper(input('elemento? ', 's'));
    if strcmp(el,'X'); break; end
    pos=find(strcmp(el,abrev));
    if length(pos) > 0
        fprintf('Elemento: %s con peso atomico %f\n',elementos{pos(1)},pa(pos(1)));
    else
        disp('Elemento non existe');
    end
end
```

Semana 13

Traballo en clase

1. Escribe un programa `hont.m` para realiza-lo reparto de escanos nas eleccións seguindo a Ley d'Hont. O programa debe ler dende o seguinte arquivo **votos.txt** os votos dos n partidos.

   ```
   2100 850 550 500 375
   ```

   Mostra por pantalla estos votos. Define o número m de escanos a 10. Para face-lo reparto segundo esta lei, utilízanse os seguintes criterios:

   - Os partidos que obteñan menos dun 10 % do total dos votos válidos quedan excluidos do reparto.
   - Para cada partido $i = 1..n$, calcúlase un "factor" f_i que inicialmente é igual ao seu número de votos v_i.
   - O escano k-ésimo, con $k = 1..m$, atribúese ó partido que ten o maior f_i. Unha vez atribuido ese escano, f_i divídese por 1 máis o número de escanos dese partido i.
   - O proceso finaliza no momento en que se repartiron os m escanos.

   O programa debe presentar por pantalla o partido ao que se asigna cada escano, e o número de escanos por partido.

   ```
   clear all;clc
   v=load('votos.txt');n=numel(v); % n=numero de partidos
   printf('votos=');printf('%i ',v);printf('\n')
   tv=sum(v);m=10; % m=numero de escanos
   mv=10*tv/100;esc=zeros(1,n); % mv=minimo numero de votos
   f=v;f(v<mv)=0;
   for k=1:m
       [~,i]=max(f);
       esc(i)=esc(i)+1;
       f(i)=v(i)/(1+esc(i));
       printf('escano %i/%i a partido %i\n',k,m,i)
   end
   printf('%10s %10s\n','Partido','Escanos')
   for i=1:n
       printf('%10i %10i\n',i,esc(i))
   end
   ```

2. Escribe un programa `suave.m` que realice defina n=10 e cree unha matriz **a** cadrada de orde n con valores enteiros aleatorios entre 1 e 10. O programa debe realizar iterativamente unha operación de "suavizado" dos elementos de **a**. Esta operación consiste en obter unha nova matriz **b** da mesma dimensión ca orixinal. Cada elemento b_{ij} da matriz transformada obtense como a media aritmética dos 9 elementos contidos nunha submatriz 3x3 centrada en a_{ij}. Para os elementos da primeira ou última fila ou columna so se deben usar os elementos existentes. En cada iteración, o programa debe mostrar **a** por pantalla, visualizala gráficamente co comando **imagesc** engadindo unha **colorbar**, e calcular a suma d das diferencias entre os valores absolutos dos elementos de **a** e **b**. Logo de cada iteración, usa o comando **input** para esperar e pulsa **intro** para continuar. Cando $d < 10$, ou logo de 20 iteracións, o programa debe rematar.

```
clear all;clc
n=10;a=randi([1 10],n,n);b=zeros(n);
disp('a inicial:');disp(a)
iter=1;niter=20;umbral=10;
for iter=1:niter
    for i=1:n
        k=max(1,i-1);l=min(i+1,n);
        for j=1:n
            p=max(1,j-1);q=min(j+1,n);
            u=a(k:l,p:q);b(i,j)=round(mean(u(:)));
        end
    end
    d=sum(abs(a(:)-b(:)));
    disp(a);printf('iter %i/%i dif=%g umbral=%g\n',iter,niter,d,umbral);
    imagesc(a);colorbar
    if d<umbral
        break
    end
    a=b;
    input('pulsa');
end
```

3. Crea o ficheiro de texto `matrices.txt` co seguinte contido:

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

Escribe un programa `matrices.m` que lea a matriz **a** dende o arquivo anterior e mostre o seguinte menú para executar operacións coas súas filas e columnas:

(1) Intercambiar de orde das filas de **a**. Usa a orde [6 2 7 1 3 5 4].
(2) Sumar unha das filas a todas as demais da matriz **a**. Usa a fila 5.
(3) Intercambiar de orde das columnas da matriz **a**. Usa a orde [4 1 5 3 2].
(4) Garda-la matriz **a** no arquivo **matrices2.txt**.
(5) Sair do programa.

```
clear all;clc
nf='matrices.txt';nf2='matrices2.txt';
printf('lendo matriz de arquivo %s ...\n',nf)
a=load(nf);[n,m]=size(a);disp('matriz=');disp(a)
printf('%i filas %i columnas\n',n,m)
while 1
    printf('Opcions:\n')
    printf('(1) Intercambiar de orde das filas.\n')
    printf('(2) Sumar unha das filas a todas as demais da matriz.\n')
    printf('(3) Intercambiar de orde das columnas da matriz actual.\n')
    printf('(4) Garda-la matriz no arquivo matrices2.txt.\n')
    printf('(5) Sair do programa.\n')
    c=input('Eleccion? ');
    switch c
    case 1
        x=input('nova orde de filas ([])? ');b=a;
        for i=1:n
            j=x(i);b(i,:)=a(j,:);
        end
        a=b;
```

```
        disp('nova matriz='); disp(a)
    case 2
        i=input('fila a sumar? ');
        for j=1:n
            if j==i; continue; end
            a(j,:)=a(j,:)+a(i,:);
        end
        disp('nova matriz='); disp(a)
    case 3
        x=input('nova orde de columnas? '); b=a;
        for i=1:m
            j=x(i); b(:,i)=a(:,j);
        end
        a=b;
        disp('nova matriz='); disp(a)
    case 4
        printf('gardando matriz en arquivo %s ...\n',nf2);
        f=fopen(nf2,'w');
        if f==-1; error('fopen %s',nf2); end
        for i=1:n
            fprintf(f,'%g ',a(i,:)); fprintf(f,'\n');
        end
        fclose(f);
    case 5
        printf('rematando ...\n')
        break
    otherwise
        printf('opcion incorrecta\n')
    end
end
```

4. **Clase, herdanza, polimorfismo e sobrecarga de operadores**. Escribe un programa `punto.m` que defina unha clase **punto** con dúas coordenadas x e y e un subprograma **mostra()** que mostre por pantalla x e y. Escribe outro programa `masa.m` que defina outra clase **masa**, derivada de **punto**, que incorpore o valor m da masa e redefina o subprograma **mostra()** de modo que mostre por pantalla x, y e m. Dende o programa principal `obxecto.m` fai o seguinte:

   a) Crea dous obxectos p e m das clases **punto** e **masa**, respectivamente.

   b) Chama á función **mostra()** do obxecto **punto**, e logo chama á función **mostra()** do obxecto **masa**. Comproba que se executan as dúas funcións **mostra()**.

   c) Crea un vector de celdas cos dous obxectos **p** e **m** e crea un bucle no que se chame ás dúas funcións **mostra()** da mesma forma, comprobando que se executan as dúas funcións.

   d) Sobrecarga o operador suma (+) e mostra por pantalla a suma de p e m.

   Programa **obxecto.m**:

```
clear all; clc
p=punto(1,3);
m=masa(4,5,1);
% chamada manual--------------
printf('polimorfismo: manual:\n')
p.mostra()
m.mostra()
% chamada nun bucle----------
printf('polimorfismo: bucle:\n')
y={p,m};
for i=1:2
    y{i}.mostra();
end
% sobrecarga de operador-----
printf('sobrecarga de operador +:\n')
s=p+m;
s.mostra()
```

Programa **punto.m**:

```
classdef punto
    properties
        x
        y
    end
    methods
        function p=punto(x,y)
            p.x=x;p.y=y;
        end
        function mostra(p)
            printf('punto: x=%g y=%g\n',p.x,p.y)
        end
        function r=plus(a,b)
            x=a.x+b.x;y=a.y+b.y;
            r=punto(x,y);
        end
    end
end
```

Programa **masa.m**:

```
classdef masa < punto
    properties
        m
    end
    methods
        function ms=masa(x,y,m)
            ms=ms@punto(x,y);
            ms.m=m;
        end
        function mostra(ms)
            printf('masa: x=%g y=%g m=%g\n',ms.x,ms.y,ms.m)
        end
    end
end
```

Exercicios propostos

1. Escribe un programa `molecular.m` que lea por teclado unha cadea de caracteres coa fórmula química dun composto e calcule o peso molecular do mesmo. Para simplificar:

   - Usa como abreviaturas dos elementos químicos os nomes que figuran no arquivo **taboaperiodica.txt** anterior.
   - Despois de cada elemento químico, debe haber un díxito especificando o número de átomos dese elemento na molécula de composto (ou de moles do elemento nun mol de composto). Por exemplo: C102 (para CO_2), H2O1 (para H_2O), C1O3Ca1 (para CO_3Ca), etc.
   - O peso atómico de cada elemento químico obterase do arquivo **taboaperiodica.txt** do exercicio anterior.

```
% Escribe un programa que lea do arquivo taboaperiodica.txt
% que conten a seguinte informacion:
% columna 1 a abreviatura dos elementos quimicos
% columna 2 o nome dos elementos
% columna 3 o peso atomico
% O programa pide o usuario a formula dun composto quimico
% e visualiza  o seu peso molecular
% a formula ten que ser elementoNumero como por exemplo Cl1Na1, H2O1
% mentres que o usuario non introduce unha x
clear all
fid=fopen('taboaperiodica.txt', 'r');
if -1 == fid
  error('Erro abrindo taboaperiodica.txt');
```

```
end
taboa=textscan(fid, '%s %s %f');
% funcion upper() convirte a maiusculas
abrev=upper(taboa{1}); % primeira columna (vector de celdas)
elementos=taboa{2}; % segunda columna
pa=taboa{3}; % terceira columna (vector de numeros)

el='' % cadea baleira

while ~strcmp(el, 'X')
  el=upper(input('Introduce molecula: ', 's'));
  d=isletter(el);  % vector con 1 na posicion de letra e 0 na posicion de numero
  pm=0; % peso molecular
  inicio=1;
  n=length(d);
  while inicio < n
      for j=inicio:n
          if d(j) == 0 % chegase a un numero
              break;
          end
      end
      elem=el(inicio:j-1);
      pos=strmatch(elem, abrev, 'exact'); % posicion elemento na taboa periorica
      inicio=j
      % calcula o numero de elementos
      for j=inicio:n
          if d(j)== 1 % chegase a unha letra
              fin=j-1;
              break;
          else
              fin=j;
          end
      end
      nel=str2num(el(inicio:fin))
      inicio=j;
      pm = pm + nel*pa(pos);
  end
end
```

2. Escribe un programa `divisor.m` que pida ó usuario un número enteiro n e mostre na pantalla a seguinte información (Probar o programa cos números 15, 1024, 14, 4e-12.):

   - Suma das cifras de n.
   - Números primos menores que n.
   - Suma dos enteiros da serie $1, 2, \ldots, n$.
   - Divisores de n.

```
clear all;clc
n=input('n? ');
%-------------------------------------------------------
c=num2str(n); % n como caracter
s=0;
for i=c
    s=s+str2num(i);
end
printf('suma de cifras=%i\n',s)
%-------------------------------------------------------
printf('numeros primos menores que n: ')
for i=1:n
    if isprime(i)
        printf('%i ',i)
```

```
    end
end
printf('\n')
%-------------------------------------------------------
printf('sum(1:%i)=%i\n',n,sum(1:n))
%-------------------------------------------------------
printf('divisores de %i: ',n)
for i=1:n
    if rem(n,i)==0
        printf('%i ',i)
    end
end
printf('\n')
```

Semana 14

Traballo en clase

1. **Resolución de ecuacións non lineares polo Método de Newton. Bucle híbrido definido-indefinido. Derivación simbólica**. O método de Newton resolve unha ecuación non linear da forma $f(x) = 0$ (con f non linear). Para isto, partimos dun punto x_0 e trazamos a recta tanxente á curva $f(x)$ en $x = x_0$. Esta recta tanxente ten a ecuación $y = mx + n$, onde $m = f'(x_0)$ por ser a recta tanxente a $f(x)$ en $x = x_0$. Pola outra banda, o feito de que a curva $f(x)$ e a recta $y = xf'(x_0) + n$ intersecten no punto $(x_0, f(x_0))$ fai que este punto cumpra a ecuación da recta, de modo que temos que $f(x_0) = x_0 f'(x_0) + n \Rightarrow n = f(x_0) - x_0 f'(x_0)$, e a ecuación da recta fica así:

$$y(x) = f(x_0) + f'(x_0)(x - x_0) \tag{8}$$

Para atopar o punto x_1 de corte desta recta co eixo horizontal, abonda con impor a condición $y(x_1) = 0$ e atopamos:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{9}$$

Repetindo o proceso iterativamente, de modo que a partir dun punto x_i calculamos o novo punto x_{i+1}, aproximámonos a unha solución x^* que verifica $f(x^*) = 0$ (se existe), a non ser que para algún valor x_i teñamos $f'(x_i) = 0$. Podes atopar unha ilustración animada do proceso de búsqueda iterativa da solución neste **enlace**. Escribe un programa chamado `newton.m` que lea por teclado a expresión analítica da función $f(x)$, o punto inicial x_0 e o número máximo n de iteracións e calcule os puntos sucesivos x_i empregando a seguinte fórmula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{10}$$

Esta operación iterativa deberá executarse ata que $|x_i - x_{i-1}| < 10^{-5}$. Antes de executar esta expresión, o programa debe avaliar que $f'(x_i) \neq 0$, e en caso contrario debe rematar cun erro. Como é posíbel que o proceso interativo non converxa, o programa debe rematar cando se supere o número máximo n de iteracións permitidas (usa $n = 100$). O programa debe almacenar os valores $(x_i, f(x_i)), i = 1, \ldots$ no ficheiro `newton.txt` (un par en cada liña).

EXEMPLO 1: dada a ecuación $x^3 - x^2 - x + 1 = 0$ ten raíces en $x = 1$ (dobre) e $x = -1$ (simple). Proba con $x_0 = 2$ (para calcular a raíz $x = 1$) e con $x_0 = -3$ (para calcular a raíz $x = -1$).

EXEMPLO 2: probando coa mesma ecuación e $x_0 = 1$ ten que dar derivada nula en x_0.

EXEMPLO 3: a ecuación $xe^{-x} - 0.2 = 0$ ten solución $x =$2.54264 (proba con $x_0 = 2$).

EXEMPLO 4: a ecuación $x^2 + 1 = 0$ non ten solución, así que o programa esgotará o número máximo de iteracións sen converxer a unha solución.

NOTA: a expresión `expr`, que é de tipo `char`, transfórmase en función anónima `f` usando `str2func()`, para poder chamar a `f()`. A función `diff()` emprégase para derivar `f` a respeito de `x` (variábel simbólica), e o resultado transfórmase tamén en función anónima `df` usando `matlabFunction()`, para poder chamar a `df()`.

Versión usando bucle híbrido `while`+`and` lóxico:

```
clear;syms x;dif=inf;i=0;n=100;
g=input('f(x)? ','s');xi=input('x0? ');
f=str2func(sprintf('@(x) %s',g));
df=matlabFunction(diff(f,x));
while dif>1e-5 && i<=n
    dfx=df(xi);
    if dfx==0; error('f''(%g)=0: proba con x0 distinto',xi); end
    xi1=xi-f(xi)/dfx;
    fprintf('%i: %g\n',i,xi1)
    dif=abs(xi1-xi);xi=xi1;i=i+1;
end
if i<n
    fprintf('x=%g\n',xi)
else
    fprintf('non hai solucion\n')
end
```

Versión usando bucle híbrido `for+return`:

```
clear;clc;niter=100;tol=1e-5;
pkg load symbolic  % so para octave
syms x;expr=input('f(x)? ', 's');xi=input('x0? ');
f=str2func(sprintf('@(x) %s',expr));df=matlabFunction(diff(f,x));
for i=1:n
    dfx=df(xi);
    if abs(dfx)<1e-10; fprintf('f''(%g)=0: usa outro x0\n', xi); end
    xi1 = xi - f(xi)/dfx;
    fprintf('i =%i x =%g\n', i, xi1);
    if abs(xi1-xi)<tol; fprintf('x=%g\n',xi1);return; end
    xi = xi1;
end
fprintf('non hai solucion\n')
```