Algoritmos matemáticos: Numerical Recipes en Fortran

- Objetivos:
  - → Resolver problemas aplicando algoritmos numéricos.
  - → Conocer la librería Numerical Recipes
- Problemas numéricos. Esquema de resolución:
  1. Planteamiento
  2. Especificación
  3. Implementación en pseudocódigo
  4. Análisis de la eficiencia
  5. Implementación en Fortran
- Librerías: Conjunto de módulos (funciones o subrutinas) listo para ser enlazado con otro programa, de manera que su código puede ser utilizado como si fuera parte de él.
  - → Ventajas:
    - Reutilización
    - Código correcto
    - No es necesario conocer el método
  - → Problemas que trataremos con Numerical Recipes:
    - Resolución de sistemas de ecuaciones lineales
    - Integración numérica de funciones

Sistemas de ecuaciones lineales

- Sea un sistema de *m* ecuaciones lineales

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ldots \qquad\qquad & \ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

- En forma matricial: $\boldsymbol{Ax} = \boldsymbol{b}$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

- Resolverlo consiste en encontrar un vector $\boldsymbol{x}$ que satisfaga la ecuación o, equivalentemente, encontrar la matriz $\boldsymbol{A}^{-1}$ de manera que

$$\boldsymbol{Ax} = \boldsymbol{b}$$

$$\boldsymbol{A}^{-1}\boldsymbol{Ax} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

Sistemas de ecuaciones lineales

- Métodos de resolución:
  - → nº incógnitas = nº ecuaciones independientes
    - ***A*** es no singular y tiene una única solución
    - Eliminación Gauss-Jordan o descomposición LU
  - → nº incógnitas > nº ecuaciones independientes
    - ***A*** es singular y puede tener o no solución, múltiple o simple.
    - Descomposición en valores singulares
  - → nº incógnitas < nº ecuaciones
    - Sistema sobredimensionado sin solución.
    - Descomposición en valores singulares o algún método de mínimos cuadrados
- Operaciones que no afectan al resultado
  1. Intercambiar dos filas de ***A*** y también de ***b***
  2. Intercambiar dos columnas de ***A*** y las correspondientes filas de ***b***
  3. Sustituir una fila de ***A*** y de ***b*** por una combinación lineal de ellas mismas y otra fila

Eliminación de Gauss-Jordan

- Método para resolver sistemas no singulares.
- Utiliza las operaciones anteriores para trasformar ***A*** hasta obtener la matriz identidad en su lugar. La solución coincidirá con el vector ***b*** obtenido tras las transformaciones.
- Si simultáneamente aplicamos a una matriz identidad las mismas operaciones que a ***A***, obtendremos la matriz inversa $\boldsymbol{A}^{-1}$.
- Variantes del método
  - → Eliminación sin pivote: sólo se utiliza la tercera operación. No funciona si algún elemento de la diagonal es 0 (división por 0).
  - → Eliminación con pivote: se utilizan las otras operaciones para evitar divisiones por 0. Se coloca en la diagonal un elemento distinto de 0 llamado pivote.
    - Pivote parcial: Se utilizan sólo la primera y la tercera operación
    - Pivote total: Se utilizan las tres operaciones
    - Pivote implícito: se elige como pivote el elemento de mayor valor relativo

Eliminación de Gauss-Jordan

- **Especificación**: realizar una abstracción funcional
  - → **Precondición**: Las variables libres sólo pueden ser los parámetros de entrada y entrada/salida.
    - ***A*** no singular
    - ***C*** identidad
    - Dimensiones no negativas
  - → **Postcondición**: Las variables libres pueden ser todos los parámetros
    - ***b*** debe ser solución
    - ***C*** debe ser $\boldsymbol{A}^{-1}$
    - ***A*** identidad
- **Formalmente**:

$P \equiv \{n \geq 0 \wedge |\boldsymbol{A}| \neq 0 \wedge \boldsymbol{A}=\boldsymbol{A}_{ini} \wedge \boldsymbol{b}=\boldsymbol{b}_{ini} \wedge \boldsymbol{C}=\boldsymbol{I}\}$

función GaussJordan (A: (**e/s**) **matriz** [1..n][1..n] **de real**, n: (**e**) **entero**, b: (**e/s**) **vector** [1..n] **de real**, C: (**e/s**) **matriz** [1..n] **de real**)

$Q \equiv \{\boldsymbol{A}_{ini}\cdot\boldsymbol{b}=\boldsymbol{b}_{ini} \wedge \boldsymbol{C}\cdot\boldsymbol{A}_{ini}=\boldsymbol{I} \wedge \boldsymbol{A}_{ini}\cdot\boldsymbol{C}=\boldsymbol{I} \wedge \boldsymbol{A}=\boldsymbol{I}\}$

Eliminación de Gauss-Jordan

- Implementación informal
  - → Para cada fila de la matriz:
    - Encontrar un pivote adecuado.
    - Colocar el pivote en la diagonal de A.
    - Dividir toda la fila de A, b y C entre el pivote
    - A las demás filas, restarle múltiplo de la fila actual
- Implementación en lenguaje algorítmico

```
función GaussJordan (A: (e/s) matriz [1..n][1..n] de real, n: (e) entero, b:
    (e/s) vector [1..n] de real, C: (e/s) matriz [1..n] de real)
  pivote, aux: real;fpivote, cpivote: entero;f, c, ff, cc: entero;
  para f:=1 hasta n hacer
      ElegirPivote (pivote, fpivote, cpivote);
      Intercambiar (f, fpivote, cpivote);
      para c:=1 hasta n hacer
        A[f,c] := A[f,c]/pivote;
        C[f,c] := C[f,c]/pivote;
      fin para
      b[f] := b[f]/pivote;
      para ff := 1 hasta n hacer
        aux := A[ff,cpivote];
        para cc:= 1 hasta n hacer
                (*) si (ff <> f) entonces
                        A[ff,cc] := A[ff,cc]-A[f][cc]*aux;
                        C[ff,cc] := C[ff,cc]-C[f][cc]*aux;
                fin si
        fin para
        b[ff] := b[ff] – b[f]*aux;
      fin para
  fin para
fin función
```

(*) Instrucción crítica

Eliminación de Gauss-Jordan

- **Eliminación de Gauss-Jordan con pivote total** ($\Theta(n^3)$)

```
      SUBROUTINE gaussj(a,n,np,b,m,mp)
      INTEGER m,mp,n,np,NMAX
      REAL a(np,np),b(np,mp)
      PARAMETER (NMAX=50)
      INTEGER i, icol, irow, j, k, l, ll, indxc(NMAX)
      INTGER indxr(NMAX), ipiv(NMAX)
      REAL big,dum,pivinv
      do 11 j=1,n
       ipiv(j)=0
11    continue
      do 22 i=1,n
          big=0.
          do 13 j=1,n
              if(ipiv(j).ne.1)then
                  do 12 k=1,n
                      if (ipiv(k).eq.0) then
                          if (abs(a(j,k)).ge.big)then
                              big=abs(a(j,k))
                              irow=j
                              icol=k
                          endif
                      else if (ipiv(k).gt.1) then
                          pause 'singular matrix'
                      endif
12                continue
              endif
13        continue
```


Numerical Recipes en Fortran
Informática II

Eliminación de Gauss-Jordan

```
          ipiv(icol)=ipiv(icol)+1
          if (irow.ne.icol) then
              do 14 l=1,n
                  dum=a(irow,l)
                  a(irow,l)=a(icol,l)
                  a(icol,l)=dum
14            continue
              do 15 l=1,m
                  dum=b(irow,l)
                  b(irow,l)=b(icol,l)
                  b(icol,l)=dum
15            continue
          endif
          indxr(i)=irow
          indxc(i)=icol
          if (a(icol,icol).eq.0.) pause 'sing. matrix'
          pivinv=1./a(icol,icol)
          a(icol,icol)=1.
          do 16 l=1,n
              a(icol,l)=a(icol,l)*pivinv
16        continue
          do 17 l=1,m
              b(icol,l)=b(icol,l)*pivinv
17        continue
          do 21 ll=1,n
              if(ll.ne.icol)then
                  dum=a(ll,icol)
                  a(ll,icol)=0.
```

Eliminación de Gauss-Jordan

```
                do 18 l=1,n
                    a(ll,l)=a(ll,l)-a(icol,l)*dum
18              continue
                do 19 l=1,m
                    b(ll,l)=b(ll,l)-b(icol,l)*dum
19              continue
            endif
21      continue
22  continue
    do 24 l=n,1,-1
        if(indxr(l).ne.indxc(l))then
            do 23 k=1,n
                dum=a(k,indxr(l))
                a(k,indxr(l))=a(k,indxc(l))
                a(k,indxc(l))=dum
23          continue
        endif
24  continue
    return
    END
```


Descomposición LU

- Método para resolver sistemas no singulares
- Descomponer la matriz ***A*** en ***A***=***L.U***, con ***L*** triangular inferior y ***U*** triangular superior.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix}$$

- Plantear dos sistemas que se resuelven por sustitución:

$$\boldsymbol{L.y = b};\quad y_1 = \frac{b_1}{\alpha_{11}} \qquad y_i = \frac{1}{\alpha_{ii}}\left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j\right];\quad i = 2,3,\ldots,n$$

$$\boldsymbol{U.x = y};\quad x_n = \frac{y_n}{\beta_{nn}} \qquad x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{n} \beta_{ij} x_j\right];\quad i = n-1,\ldots,1$$

- Especificación:
  - → Descomposición LU:
    - Precondición: ***A*** no singular
    - Postcondición: L triangular inferior, U triangular superior y el producto de L y U es A
  - → Sustitución:
    - Precondición: L triangular inferior, U triangular superior
    - Postcondición: vector de salida es solución

Descomposición LU

- Especificación de la descomposición LU:

$P \equiv \{n \geq 0 \wedge |\boldsymbol{A}| \neq 0\}$

función DescomposiciónLU (A: (**e**) **matriz** [1..n][1..n] **de real**, n: (**e**) **entero**, L: (**s**) **matriz** [1..n][1..n] **de real**, U: (**s**) **matriz** [1..n][1..n] **de real**)

$Q \equiv \{\boldsymbol{L}\cdot\boldsymbol{U}=\boldsymbol{A} \wedge \forall i,j \in \{1..n\}\ (i>j) \rightarrow (\boldsymbol{L}ij=0) \wedge \forall i,j \in \{1..n\}\ (i<j) \rightarrow (\boldsymbol{U}ij=0)\}$

- Especificación de la sustitución:

$P \equiv \{n \geq 0 \wedge |\boldsymbol{L}\cdot\boldsymbol{U}| \neq 0 \wedge \forall i,j \in \{1..n\}\ (i>j) \rightarrow (\boldsymbol{L}ij=0) \wedge \forall i,j \in \{1..n\}\ (i<j) \rightarrow (\boldsymbol{U}ij=0) \wedge \boldsymbol{b}=\boldsymbol{b}_{ini}\}$

función SustituciónLU (L: (**e**) **matriz** [1..n][1..n] **de real**, U: (**e**) **matriz** [1..n][1..n] **de real**, n: (**e**) **entero**, b: (**e/s**) **vector** [1..n] **de real**)

$Q \equiv \{(\boldsymbol{L}\cdot\boldsymbol{U})\cdot\boldsymbol{b}=\boldsymbol{b}_{ini}\}$

- Algoritmo de Crout para la descomposición LU
  - → Dar valor a αii=1, para i=1,...,n
  - → Para cada j=1,2,...n hacer
    - Para cada i=1,2,...n, $\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}$
    - Para cada i=j+1,j+2,...,n $\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj}\right)$

Descomposición LU

- Implementación descomposición: $\Theta(n^3)$

```
función DescomposiciónLU (A: (e) matriz [1..n][1..n] de real, n: (e) entero,
    L: (s) matriz [1..n][1..n] de real, U: (s) matriz [1..n][1..n] de real)
i, j, k: entero;
suma: real;
para j:=1 hasta n hacer
    L[i][i] := 1;
fin para
para j:=1 hasta n hacer
    para i:=1 hasta j hacer
        suma := 0;
        para k :=1 hasta i-1 hacer
            suma := suma + L[i][k]*U[k][j];
        fin para
        U[i][j] := A[i][j] – suma;
    fin para
    para i:=j+1 hasta n hacer
        suma := 0;
        para k:=1 hasta j-1 hacer
            suma := suma + L[i][k]*U[k][j];
        fin para
        pivote := ElegirPivote (A, j, i);
        L[i][j] := (1/pivote) * (A[i][j] – suma);
    fin para
fin para
fin función
```

Descomposición LU

- Implementación sustitución: $\Theta(n^2)$

función SustituciónLU (L: (**e**) **matriz** [1..n][1..n] **de real**, U: (**e**) **matriz** [1..n][1..n] **de real**, n: (**e**) **entero**, b: (**e/s**) **vector** [1..n] **de real**)
i, j: **entero**;
suma: **real**;
y: **vector** [1..n] **de real**;
Sustitución en la matriz L
y[1][1] := b[1]/L[1][1];
para i:=2 **hasta** n **hacer**
    suma := 0;
    para j:=1 **hasta** i-1 **hacer**
        suma := suma + L[i][j]*y[j];
    fin para
    y[i] := (1/L[i][i])*(b[i] – suma);
fin para
Sustitución en la matriz U
b[n] := y[n]/U[n][n];
para i:=n-1 **hasta** 1 **paso** -1 **hacer**
    suma := 0;
    para j:=i+1 **hasta** n **hacer**
        suma := suma + U[i][j]*b[j];
    fin para
    b[i] := (1/U[i][i])*(y[i] – suma);
fin para
fin función

Descomposición LU

- **Algoritmo de Crout para descomposición LU con pivote parcial e implícito**

```
 SUBROUTINE ludcmp(a,n,np,indx,d)
      INTEGER n,np,indx(n),NMAX
      REAL d,a(np,np),TINY
      PARAMETER (NMAX=500,TINY=1.0e-20)
      INTEGER i,imax,j,k
      REAL aamax,dum,sum,vv(NMAX)
      d=1.
      do 12 i=1,n
        aamax=0.
        do 11 j=1,n
          if (abs(a(i,j)).gt.aamax) aamax=abs(a(i,j))
11      continue
        if (aamax.eq.0.) pause 'singular matrix'
        vv(i)=1./aamax
12    continue
      do 19 j=1,n
        do 14 i=1,j-1
          sum=a(i,j)
          do 13 k=1,i-1
            sum=sum-a(i,k)*a(k,j)
13        continue
          a(i,j)=sum
14      continue
        aamax=0.
```

Descomposición LU

```
        do 16 i=j,n
          sum=a(i,j)
          do 15 k=1,j-1
            sum=sum-a(i,k)*a(k,j)
15        continue
          a(i,j)=sum
          dum=vv(i)*abs(sum)
          if (dum.ge.aamax) then
            imax=i
            aamax=dum
          endif
16      continue
        if (j.ne.imax)then
          do 17 k=1,n
            dum=a(imax,k)
            a(imax,k)=a(j,k)
            a(j,k)=dum
17        continue
          d=-d
          vv(imax)=vv(j)
        endif
        indx(j)=imax
        if(a(j,j).eq.0.)a(j,j)=TINY
        if(j.ne.n)then
          dum=1./a(j,j)
          do 18 i=j+1,n
            a(i,j)=a(i,j)*dum
18        continue
        endif
19    continue
      return
      END
```


Descomposición LU

- **Algoritmo de sustitución para descomposición LU**

```
SUBROUTINE lubksb(a,n,np,indx,b)
      INTEGER n,np,indx(n)
      REAL a(np,np),b(n)
      INTEGER i,ii,j,ll
      REAL sum
      ii=0
      do 12 i=1,n
        ll=indx(i)
        sum=b(ll)
        b(ll)=b(i)
        if (ii.ne.0)then
          do 11 j=ii,i-1
            sum=sum-a(i,j)*b(j)
11        continue
        else if (sum.ne.0.) then
          ii=i
        endif
        b(i)=sum
12    continue
      do 14 i=n,1,-1
        sum=b(i)
        do 13 j=i+1,n
          sum=sum-a(i,j)*b(j)
13      continue
        b(i)=sum/a(i,i)
14    continue
      return
      END
```

Mejora iterativa de las soluciones

- Mejora de los errores de redondeo acumulados
- Sea Ax=b. Su solución contiene un error δx, luego el b también es erróneo: A(x+δx) = b+δb.
- Restando las dos fórmulas anteriores obtenemos: Aδx=δb y sustituyendo de nuevo: Aδx=A(x+δx)-b, planteamos un nuevo sistema para calcular δx
- Especificación:
  - → Precondición: A no singular, L triangular inferior, U triangular superior, L.U=A y x solución aproximada.
  - → Postcondición: nueva solución mejor que la anterior.

$P \equiv \{n \geq 0 \wedge |\mathbf{A}| \neq 0 \wedge \boldsymbol{L}\cdot\boldsymbol{U}=\boldsymbol{A} \wedge \forall i,j \in \{1..n\}\ (i>j) \rightarrow (\boldsymbol{L}_{ij}=0) \wedge \forall i,j \in \{1..n\}\ (i<j) \rightarrow (\boldsymbol{U}_{ij}=0) \wedge \boldsymbol{A}\cdot\boldsymbol{x}-\boldsymbol{b}<\varepsilon \wedge \boldsymbol{x} = \boldsymbol{x}_{ini}\}$

función MejoraIterativaLU (A: (**e**) **matriz** [1..n][1..n] **de real**, n: (**e**) **entero**, L: (**e**) **matriz** [1..n][1..n] **de real**, U: (**e**) **matriz** [1..n][1..n] **de real**, b: (**e**) **vector**[1..n] **de real**, x: (**e/s**) **vector** [1..n] **de real**)

$Q \equiv \{\boldsymbol{A}\cdot\boldsymbol{x}_{ini}-\boldsymbol{b} \geq \boldsymbol{A}\cdot\boldsymbol{x}-\boldsymbol{b} \wedge \boldsymbol{A}(\boldsymbol{x}-\boldsymbol{x}_{ini})=\boldsymbol{A}\cdot\boldsymbol{x}_{ini}-\boldsymbol{b}\}$

Mejora iterativa de las soluciones

- Implementación informal

```
función MejoraIterativaLU (A: (e) matriz [1..n][1..n] de real, n: (e) entero, L:
    (e) matriz [1..n][1..n] de real, U: (e) matriz [1..n][1..n] de real, b: (e)
    vector[1..n] de real, x: (e/s) vector [1..n] de real)

derecha: vector [1..n] de real;
i, j: entero;
para i:=1 hasta n hacer
    derecha[i] := -b[i];
    para j:=1 hasta n hacer
        derecha[i] := derecha[i] + a[i][j]*x[j];
    fin para
fin para
Sustitución (L, U, n, derecha);
para i:=1 hasta n hacer
    x[i] := x[i] – derecha[i];
fin para

fin función
```

- Eficiencia: $\Theta(n^2)$

Mejora iterativa de las soluciones

- Implementación en Fortran

```
      SUBROUTINE mprove(a,alud,n,np,indx,b,x)

        INTEGER n,np,indx(n),NMAX
        REAL a(np,np),alud(np,np),b(n),x(n)
        PARAMETER (NMAX=500)
CU      USES lubksb
        INTEGER i,j
        REAL r(NMAX)
        DOUBLE PRECISION sdp
        do 12 i=1,n
          sdp=-b(i)
          do 11 j=1,n
            sdp=sdp+dble(a(i,j))*dble(x(j))
11        continue
          r(i)=sdp
12      continue
        call lubksb(alud,n,np,indx,r)
        do 13 i=1,n
          x(i)=x(i)-r(i)
13      continue
        return

        END
```


Descomposición en valores singulares

- Resuelve todo tipo de sistemas (tb. Singulares)
- Toda matriz $m \times n$ puede descomponerse en el producto $\boldsymbol{A}=\boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^{T}$
  - → $\boldsymbol{U}$ matriz $m \times n$ ortogonal por columnas
  - → $\boldsymbol{W}$ matriz $n \times n$ diagonal. Todos sus elementos ≥ 0
  - → $\boldsymbol{V}$ matriz $n \times n$ ortogonal
- Esta descomposición siempre existe y es única, salvo permutación o combinación lineal
- Algoritmos de Forsythe y de Golub y Reinsch
- La inversa de $\boldsymbol{A}$ es $\boldsymbol{A}^{-1}=\boldsymbol{V}[\mathrm{diag}(1/w_j)]\boldsymbol{U}^{T}$
- Cuando $\boldsymbol{A}$ es singular, algún w_j es 0, y no puede calcularse la inversa. Si sustituimos $1/w_j$ por 0 obtenemos la pseudoinversa:
  - → Ningún elemento de la diagonal de W es 0 y A es cuadrada ➪ A es no singular ➪ Pseudoinversa = Inversa
  - → Algún elemento de la diagonal de W es 0 ➪ A es singular o nº ecuaciones < nº incógnitas
    - Si el sistema tiene solución múltiple ➪ se obtiene la de menor módulo
    - Si el sistema no tiene solución ➪ obtiene el vector más cercano (mínimos cuadrados)
  - → Nº ecuaciones > nº incógnitas ➪ No hay 0 en la diag. de W y A degenerada ➪ Mín. cuadrados

Descomposición en valores singulares

- Especificación
  - → Descomposición
    - Precondición: no hay condiciones para $\boldsymbol{A}$
    - Postcondición: $\boldsymbol{U}$ ortogonal por columnas, $\boldsymbol{V}$ ortogonal, $\boldsymbol{W}$ diagonal con sus elementos ≥ 0
  - → Sustitución
    - Precondición: $\boldsymbol{U}$ ortogonal por columnas, $\boldsymbol{V}$ ortogonal, $\boldsymbol{W}$ diagonal con sus elementos ≥ 0
    - Postcondición: $\boldsymbol{x}$ solución del sistema
- Especificación de la descomposición:

$P \equiv \{m \geq 0 \wedge n \geq 0\}$

función DescomposiciónValoresSingulares (A: (**e**) **matriz** [1..m][1..n] **de real**, m: (**e**) **entero**, n: (**e**) **entero**, U: (**s**) **matriz** [1..m][1..n] **de real**, W: (**s**) **matriz** [1..n][1..n] **de real**, V: (**s**) **matriz** [1..n][1..n] **de real**)

$Q \equiv \{\boldsymbol{A}=\boldsymbol{U}\cdot\boldsymbol{W}\cdot\boldsymbol{V}^T \wedge \boldsymbol{U}^T\cdot\boldsymbol{U} = \boldsymbol{I} \wedge \boldsymbol{V}^T\cdot\boldsymbol{V}=\boldsymbol{V}\cdot\boldsymbol{V}^T=\boldsymbol{I} \wedge \forall i,j \in \{1..n\} ((i \neq j)\rightarrow(\boldsymbol{W}_{ij}=0) \wedge (i=j)\rightarrow(\boldsymbol{W}_{ij}\geq 0))\}$

- Especificación de la sustitución

$P \equiv \{ m \geq 0 \wedge n \geq 0 \wedge \boldsymbol{U}^T\cdot\boldsymbol{U} = \boldsymbol{I} \wedge \boldsymbol{V}^T\cdot\boldsymbol{V}=\boldsymbol{V}\cdot\boldsymbol{V}^T=\boldsymbol{I} \wedge \forall i,j \in \{1..n\} ((i \neq j)\rightarrow(\boldsymbol{W}_{ij}=0) \wedge (i=j)\rightarrow(\boldsymbol{W}_{ij}\geq 0))\}$

función SustituciónVS (U: (**e**) **matriz** [1..m][1..n] **de real**, W: (**e**) **matriz** [1..n][1..n] **de real**, V: (**e**) **matriz** [1..n][1..n] **de real**, m: (**e**) **entero**, n: (**e**) **entero**, b: (**e**) **vector** [1..m] **de real**, x: (**s**) **vector** [1..n] **de real**)

$Q \equiv \{(\boldsymbol{U}\cdot\boldsymbol{W}\cdot\boldsymbol{V}^T)\cdot\boldsymbol{x}=\boldsymbol{b}\}$

Descomposición en valores singulares

- Descomposición en valores singulares: Algoritmo de Forsythe, Golub y Reinsch: $\Theta(n^4)$
- Algoritmo de sustitución: $\Theta(n^2)$

```
función SustituciónVS (U: (e) matriz [1..m][1..n] de real, W: (e) matriz
    [1..n][1..n] de real, V: (e) matriz [1..n][1..n] de real, m: (e) entero, n: (e)
    entero, b: (e) vector [1..m] de real, x: (s) vector [1..n] de real)
aux: real;
tmp: vector [1..n] de real;
i, j, jj: entero;
para j:=1 hasta n hacer
    aux := 0;
    si (W[j]<>0) entonces
        para i:=1 hasta m hacer
            aux := aux+U[i][j]*b[i];
        fin para
        aux := aux/W[j];
    fin si
    tmp[j] := aux;
fin para
para j:=1 hasta n hacer
aux := 0;
para jj:=1 hasta n hacer
aux := aux + v[j][jj]*tmp[jj];
fin para
x[j] := aux;
fin para
fin función
```

Descomposición en valores singulares

- **Algoritmo de descomposición**

```
SUBROUTINE svdcmp(a,m,n,mp,np,w,v)
      INTEGER m,mp,n,np
      REAL a(mp,np),v(np,np),w(np)
```

- **Algoritmo de sustitución**

```
SUBROUTINE svbksb(u,w,v,m,n,mp,np,b,x)
      INTEGER m,mp,n,np,NMAX
      REAL b(mp),u(mp,np),v(np,np),w(np),x(np)
      PARAMETER (NMAX=500)
      INTEGER i,j,jj
      REAL s,tmp(NMAX)
      do 12 j=1,n
        s=0.
        if(w(j).ne.0.)then
          do 11 i=1,m
            s=s+u(i,j)*b(i)
11        continue
          s=s/w(j)
        endif
        tmp(j)=s
12    continue
      do 14 j=1,n
        s=0.
        do 13 jj=1,n
          s=s+v(j,jj)*tmp(jj)
13      continue
        x(j)=s
14    continue
      return
      END
```


Integración numérica de funciones

- Evaluación de la integral $I = \int_a^b f(x)dx$
- En términos de ecuaciones diferenciales, consiste en resolver $\frac{dy}{dx} = f(x)$ para I=y(b) con la condición y(a)=0
- Sea una secuencia de abcisas x_0, x_1, ..., x_n, x_{n+1} separadas una cantidad h: $x_i = x_0 + ih$, i=0,...,n+1, para las cuales la función $f(x)$ es conocida
- Tipos de fórmulas para integrar $f(x)$ entre a y b
  - → Cerradas: utilizan los valores a y b
  - → Abiertas: no utilizan los valores a y b
- Regla del trapezoide (área de un trapezoide)

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O\left(h^3 f''\right)$$

  - → El error depende de h y de la derivada segunda
  - → Es exacta para polinomios de hasta grado 1
- Regla del trapezoide extendida

$$\int_{x_1}^{x_n} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + \ldots + f_{n-1} + \frac{1}{2}f_n\right] + O\left(\frac{(b-a)^3 f''}{n^2}\right)$$

Regla del trapezoide

- Especificación:
  - → Precondición: al menos un intervalo
  - → Postcondición: debe cumplirse la regla del trapezoide

$P \equiv \{n \geq 1 \wedge a \leq b\}$

función ReglaTrapezoide (f: (**e**) **función** (**real**) **devolver real**, a: (**e**) **real**, b: (**e**) **real**, n: (**e**) **entero**) **devolver** (resultado: **real**)

$$Q \equiv \left\{ resultado = \frac{b-a}{n}\left(\frac{1}{2}\left(f(a)+f(b)\right)+\sum_{i=2}^{n-1} f\left(\frac{b-a}{n}i\right)\right)\right\}$$

- Implementación: $\Theta(n)$

función ReglaTrapezoide (f: (**e**) **función** (**real**) **devolver real**, a: (**e**) **real**, b: (**e**) **real**, n: (**e**) **entero**) **devolver** (resultado: **real**)
suma: **real**;
suma := 0;
Suma de las abcisas intermedias
para i:=1 **hasta** n-1 **hacer**
    suma := suma + f(i*(b-a)/n);
fin para
Resultado final
resultado := ((b-a)/n)*[(1/2)*(f(a)+f(b)) + suma];
fin función

Regla del trapezoide (incremental)

función ReglaTrapezoide (f: (**e**) **función** (**real**) **devolver real**, a: (**e**) **real**, b: (**e**) **real**, n: (**e**) **entero**) **devolver** (resultado: **real**)
resultado_parcial: **real estático**; x, suma: **real**;
si (n=1) **entonces**
    resultado_parcial := (1/2)*(b-a)*(f(a)+f(b));
    resultado := resultado_parcial;
si no
    x := a+(b-a)/2^(n-1); suma := f(x);
    para j:=1 **hasta** 2^(n-1)-1 **hacer**
        suma := suma + f(x+j*(b-a)/2^(n-2));
    fin para;
    resultado_parcial := (1/2)*(resultado_parcial + (b-a)/2^(n-2)*suma);
    resultado := resultado_parcial;
fin si;
fin función

$P \equiv \{a \leq b\}$

función TrapezoideEpsilon (f: (**e**) **función** (**real**) **devolver real**, a: (**e**) **real**, b: (**e**) **real**) **devolver** (resultado: **real**)
anterior, actual: **real**; n: **entero**;
n := 1; actual := ReglaTrapezoide (f, a, b, n); n := n+1;
repetir
    anterior := actual;
    actual := ReglaTrapezoide (f, a, b, n);
    n := n+1;
hasta (|actual-anterior| < ε) **fin repetir**
resultado := actual;
fin función

$$Q \equiv \left\{ \exists\, n>1 \left[resultado = \frac{b-a}{2^{n-2}}\left(\frac{1}{2}\left(f(a)+f(b)\right)+\sum_{i=2}^{2^{n-2}-1} f\left(\frac{b-a}{2^{n-2}}i\right)\right) \wedge resultado = \frac{b-a}{2^{n-3}}\left(\frac{1}{2}\left(f(a)+f(b)\right)+\sum_{i=2}^{2^{n-3}-1} f\left(\frac{b-a}{2^{n-3}}i\right)\right) < \varepsilon \right] \right\}$$

Regla del trapezoide

- n-ésimo paso de refinamiento

```
SUBROUTINE trapzd(func,a,b,s,n)
      INTEGER n
      REAL a,b,s,func
      EXTERNAL func
      INTEGER it,j
      REAL del,sum,tnm,x
      if (n.eq.1) then
        s=0.5*(b-a)*(func(a)+func(b))
      else
        it=2**(n-2)
        tnm=it
        del=(b-a)/tnm
        x=a+0.5*del
        sum=0.
        do 11 j=1,it
          sum=sum+func(x)
          x=x+del
11      continue
        s=0.5*(s+(b-a)*sum/tnm)
      endif
      return
      END
```


Regla del trapezoide

- **Cálculo de la integral de una función entre a y b por la regla del trapezoide**

```
SUBROUTINE qtrap(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
CU    USES trapzd
      INTEGER j
      REAL olds
      olds=-1.e30
      do 11 j=1,JMAX
        call trapzd(func,a,b,s,j)
        if (abs(s-olds).lt.EPS*abs(olds)) return
        if (s.eq.0..and.olds.eq.0..and.j.gt.6) return
        olds=s
11    continue
      pause 'too many steps in qtrap'
      END
```

Regla de *Simpson*

- Utiliza tres valores de abcisa

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O\left(h^5 f^{(4)}\right)$$

  → El error depende de la cuarta derivada

  → Es exacta para polinomios de hasta grado 3

- Regla de Simpson extendida

$$\int_{x_1}^{x_n} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \ldots + \frac{4}{3}f_{n-1} + \frac{1}{3}f_n\right] + O\left(\frac{1}{n^4}\right)$$

- Puede escribirse en función de la regla del trapezoide

$$\int_{x_1}^{x_n} f(x)dx = \frac{4}{3}S_{2n} - \frac{1}{3}S_n$$

  → S_{2n}: resultado de aplicar la regla del trapezoide, dividiendo el intervalo entre a y b 2n veces

  → S_n: aplicar la regla del trapezoide n veces

Regla de Simpson

- Especificación e implementación: $\Theta(2^n)$

$P \equiv \{a \leq b\}$

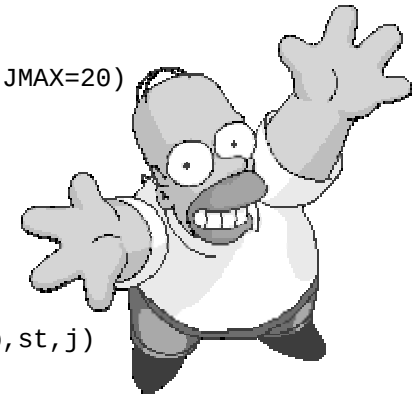
función SimpsonEpsilon (f: (**e**) **función** (**real**) **devolver** **real**, a: (**e**) **real**, b: (**e**) **real**) **devolver** (resultado: **real**)
anterior, actual, anteriorTr, actualTr: **real**;
n: **entero**;
n := 1;
actualTr := ReglaTrapezoide (f, a, b, n);
actual := ∞;
n := n+1;
repetir
    anteriorTr := actualTr;
    actualTr := ReglaTrapezoide (f, a, b, n);
    anterior := actual;
    actual := (4 / 3)*actualTr – (1 / 3)*anteriorTr;
    n := n+1;
hasta (| actual-anterior | < ε) **fin repetir**
resultado := actual;
fin función

$$Q \equiv \{\ \exists\, n>1\ resultado = \frac{b-a}{2^{n-2}}\left(\frac{1}{3}(f(a)+f(b)) + \frac{4}{3}\sum_{i=2}^{2^{n-2}-1} g(i) f\left(\frac{b-a}{2^{n-2}}i\right) + \frac{2}{3}\sum_{i=3}^{2^{n-2}-2}(1-g(i)) f\left(\frac{b-a}{2^{n-2}}i\right)\right) \wedge$$
$$\wedge resultado - \frac{b-a}{2^{n-3}}\left(\frac{1}{3}(f(a)+f(b)) + \frac{4}{3}\sum_{i=2}^{2^{n-3}-1} g(i) f\left(\frac{b-a}{2^{n-3}}i\right) + \frac{2}{3}\sum_{i=3}^{2^{n-3}-2}(1-g(i)) f\left(\frac{b-a}{2^{n-3}}i\right)\right) < \varepsilon\ \}$$

Regla de Simpson

- **Cálculo de la integral de una función entre a y b por la regla de Simpson**

```
 SUBROUTINE qsimp(func,a,b,s)
       INTEGER JMAX
       REAL a,b,func,s,EPS
       EXTERNAL func
       PARAMETER (EPS=1.e-6, JMAX=20)
CU     USES trapzd
       INTEGER j
       REAL os,ost,st
       ost=-1.e30
       os= -1.e30
       do 11 j=1,JMAX
         call trapzd(func,a,b,st,j)
         s=(4.*st-ost)/3.
         if (abs(s-os).lt.EPS*abs(os)) return
         if (s.eq.0..and.os.eq.0..and.j.gt.6) return
         os=s
         ost=st
11     continue
       pause 'too many steps in qsimp'
       END
```



Integración de Romberg

- Método general para fórmulas de cualquier orden
- Aproximación al límite de Richardson: Si aplicamos un algoritmo numérico para resolver un problema para valores cada vez menores de un parámetro, podemos utilizar técnicas de extrapolación para obtener una aproximación de la solución en el límite ($h \to 0$)
- Sea k el número de soluciones que utilizamos para la extrapolación
  - → El orden depende de k.
  - → El error depende de k: $O(1/n^{2k})$
- Como algoritmo de extrapolación utilizamos el método de Neville, basado en la interpolación / extrapolación de Lagrange

Integración de Romberg

- Especificación e implementación:

$P \equiv \{a \leq b\}$

función RombergEpsilon (f: (**e**) **función** (**real**) **devolver** **real**, a: (**e**) **real**, b: (**e**) **real**) **devolver** (resultado: **real**)

x, y: **vector** [1..∞] **de real**;
yy, dy: **real**;
n: **entero**;
n := 1;
x[n] := 1.0;
repetir
y[n] := ReglaTrapezoide (f, a, b, n);
si (n ≥ k) entonces
InterExtrapolación (Subvector(x,n-k,k), Subvector(y,n-k,k), 0, yy, dy);
x[n+1] := (1 / 4)*h[n];
n := n+1;
hasta (| dy | < ε) **fin repetir**
resultado := yy;
fin función

$$Q \equiv \left\{ resultado \approx \int_a^b f(x)dx \right\}$$

Integración de Romberg

```
SUBROUTINE qromb(func,a,b,ss)
      INTEGER JMAX,JMAXP,K,KM
      REAL a,b,func,ss,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20, JMAXP=JMAX+1,
     * K=5, KM=K-1)
CU    USES polint,trapzd
      INTEGER j
      REAL dss,h(JMAXP),s(JMAXP)
      h(1)=1.
      do 11 j=1,JMAX
        call trapzd(func,a,b,s(j),j)
        if (j.ge.K) then
          call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
          if (abs(dss).le.EPS*abs(ss)) return
        endif
        s(j+1)=s(j)
        h(j+1)=0.25*h(j)
11    continue
      pause 'too many steps in qromb'
      END
```

Integrales impropias

- Una integral es impropia si
  - → La función no puede evaluarse en alguno de los extremos
  - → Alguno de los extremos es infinito
  - → Presenta alguna singularidad en un extremo
  - → Presenta alguna singularidad en algún punto conocido entre los extremos
  - → Presenta alguna singularidad en algún punto desconocido entre los extremos
- Resolución de integrales impropias
  - → Resolveremos los cuatro primeros casos
  - → Se basa en el uso de fórmulas abiertas y en cambios de variable
- Regla del punto medio extendida

$$\int_{x_1}^{x_n} f(x)dx = h\left[f_{\frac{3}{2}} + f_{\frac{5}{2}} + f_{\frac{7}{2}} + \text{✯} + f_{\frac{n-3}{2}} + f_{\frac{n-1}{2}} \right] + O\left(\frac{1}{n^2}\right)$$

- En función de la regla del trapezoide

$$\int_{x_1}^{x_n} f(x)dx = \frac{9}{8}S_{2n} - \frac{1}{8}S_n$$

- Para resolver el primer caso podemos sustituir trapzd por midpnt en qtrap o qsimp

Integrales impropias

- **N-ésimo paso de refinamiento para la integración mediante la regla del punto medio extendido**

```
SUBROUTINE midpnt(func,a,b,s,n)
      INTEGER n
      REAL a,b,s,func
      EXTERNAL func
      INTEGER it,j
      REAL ddel,del,sum,tnm,x
      if (n.eq.1) then
        s=(b-a)*func(0.5*(a+b))
      else
        it=3**(n-2)
        tnm=it
        del=(b-a)/(3.*tnm)
        ddel=del+del
        x=a+0.5*del
        sum=0.
        do 11 j=1,it
          sum=sum+func(x)
          x=x+ddel
          sum=sum+func(x)
          x=x+del
11      continue
        s=(s+(b-a)*sum/tnm)/3.
      endif
      return
      END
```

Integrales impropias

- Los cambios de variable nos permiten solucionar los cuatro primeros casos
- Generalizamos el método de integración de Romberg para incorporar el cambio de variable
- **Caso 1**: La función no se puede evaluar en los extremos ➪ Fórmula del punto medio
- **Caso 2**: Alguno de los extremos es infinito (válido si b→∞ y a>0, o a →-∞ y b<0)

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right)dt; \quad ab > 0$$

- **Caso 3**: Existe alguna singularidad de integración en alguno de los extremos debido a una potencia fraccionaria

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f\left(t^{\frac{1}{1-\gamma}} + a\right)dt; \quad (b > a)$$

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f\left(b - t^{\frac{1}{1-\gamma}}\right)dt; \quad (b > a)$$

- **Caso 4**: Integral con el límite superior infinito y cuyo integrando decrece exponencialmente

$$\int_a^\infty f(x)dx = \int_0^{e^{-a}} f(-\log t)\frac{dt}{t}$$

Integrales impropias

- **N-ésimo paso de refinamiento para la integración con algún límite infinito**

```
SUBROUTINE midinf(funk,aa,bb,s,n)
      INTEGER n
      REAL aa,bb,s,funk
      EXTERNAL funk
      INTEGER it,j
      REAL a,b,ddel,del,sum,tnm,func,x
      func(x)=funk(1./x)/x**2
      b=1./aa
      a=1./bb
      if (n.eq.1) then
        s=(b-a)*func(0.5*(a+b))
      else
        it=3**(n-2)
        tnm=it
        del=(b-a)/(3.*tnm)
        ddel=del+del
        x=a+0.5*del
        sum=0.
        do 11 j=1,it
          sum=sum+func(x)
          x=x+ddel
          sum=sum+func(x)
          x=x+del
11      continue
        s=(s+(b-a)*sum/tnm)/3.
      endif
      return
      END
```

Integrales impropias

- **Generalización de la integración de Romberg, utilizando fórmulas abiertas**

```
 SUBROUTINE qromo(func,a,b,ss,choose)
       INTEGER JMAX,JMAXP,K,KM
       REAL a,b,func,ss,EPS
       EXTERNAL func,choose
       PARAMETER (EPS=1.e-6, JMAX=14,
     * JMAXP=JMAX+1, K=5, KM=K-1)
CU     USES polint
       INTEGER j
       REAL dss,h(JMAXP),s(JMAXP)
       h(1)=1.
       do 11 j=1,JMAX
         call choose(func,a,b,s(j),j)
         if (j.ge.K) then
           call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
           if (abs(dss).le.EPS*abs(ss)) return
         endif
         s(j+1)=s(j)
         h(j+1)=h(j)/9.
11     continue
       pause 'too many steps in qromo'
       END
```


Cuadratura Gaussiana

- Permite utilizar un espaciado no constante ➪ incrementa los grados de libertad
- Permite calcular integrales exactas no sólo para polinomios sino también para productos de polinomios por una función conocida W(x)
- Sea una función conocida *W(x)* y un entero *n*, podemos encontrar un conjunto de pesos w_j y de abcisas x_j tal que la siguiente aproximación es exacta si *f(x)* es un polinomio

$$\int_a^b W(x) f(x) dx \approx \sum_{j=1}^{n} w_j f(x_j)$$

- El cálculo de pesos y abcisas no es trivial
- Variantes
  - → Gauss-Legendre (W(x) = 1; -1<x<1): gauleg
  - → Gauss-Laguerre (W(x) = $x^{\alpha}e^{-x}$;0<x<∞): gaulag
  - → Gauss-Hermite (W(x) = e^{-x^2}; -∞<x<∞): gauher
  - → Gauss-Jacobi (W(x) = $(1-x)^{\alpha}(1+x)^{\beta}$; -1<x<1): gaujac

Cuadratura Gaussiana

Calculo de la cuadratura Gaussiana

Entrada: func es la función cuya integral estamos calculando
a es el límite inferior de la integral
b es el límite superior de la integral

Salida: El valor devuelto por la función es la integral aproximada utilizando la cuadratura Gaussiana

```
FUNCTION cuadrGauss (func, a, b)
REAL cuadrGauss, func, a, b
INTEGER j, N
C   N : número de abcisas y pesos que vamos a utilizar para calcular la cuadratura
N = 10
REAL result, x(N), w(N)
result = 0.
...
C   Llamada a alguna de las funciones que calcula las abcisas y los pesos
C   correspondientes. La rutina adecuada dependerá de la función W(x) elegida
CALL gauXXX (..., x, w, ...);

C   Suma de los pesos multiplicados por el valor de la función en las abcisas
DO j=1, N
      result = result + w(j)*func(x(j))
END DO
cuadrGauss = result
RETURN
```