```
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
! para verificar ram dinamica y tiempos
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
module m_temps
  use  m_ctes
  implicit none
  !
  private           ! por omision todo es asi
  public timer, nerror, ier
  !
  character (10), parameter :: c = "m_temps > "
  !
  ! para verificar alocamiento de memoria RAM
  integer (iin), parameter :: nerror = 64
  integer (iin), dimension (nerror) :: ier
  !
  contains
  !
  !-----------------------------------------------------------------
  subroutine timer (time, n, m, raiz)
    real    (idp), dimension (:), intent (inout) :: time
    integer (iin), intent (in)   :: n     ! dimension lineal tipica
    integer (iin), intent (in)   :: m     ! exponente tipico
    character (*), intent (in)   :: raiz  ! raiz archivo
    character (len(raiz)+4) :: arch       ! extension prefijada = 4
    character (4) :: exte = ".tim"        ! aqui la exten. es cte
    character (66):: s
    integer (iin) :: l, p
    integer (iin) :: ntime, h3
    real    (idp) :: total
    !
    !nro de procesadores
    !p = 1                                ! secuencial
    p = number_of_processors ()           ! solo en HPF
    !
    !control
    ntime = size (time)
    if (ntime < 2) stop "error (timer): size (time) < 2 "
    !
    !pasa a segundos
    call system_clock (count_rate = h3)  ! cte para pasar a seg
    if (h3 < 1) h3 = 1                   ! control
    time = time / dble (h3)              ! pasa de ciclos a seg
    !
    !tiempo total en ciclos del procesador
    total = sum (time)
    !
    !define archivo
    l = len_trim (raiz)        ! longitud omitiendo blancos
    arch = raiz (1:l) // exte  ! apendiza en ese orden sin blancos
    !
    !resumen a disco
    write (*,*)
    write (*,100) " archivo timer: " // arch
    write (*,*)
    s ="  p           n  k    time columns ... "
    write (*,100) s
    !
    open (1, file = arch         , &
             status = "unknown"  , &
             position = "append")
    write (*,110)  p, n, m, time, total
    write (1,110)  p, n, m, time, total
    close (1, status = 'keep')
    write (*,*)
    !
    ! formatos
    100 format (a)
    110 format (1x, i2, 1x, i12, 1x, i2, 8 (1x, e16.7))
```



```
  end subroutine
  !
end module
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
```

```
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
! Herramientas varias
! precision doble
! precision cuadruple
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
module m_tools0
 use   m_ctes
 use   m_temps
 implicit none
 !
 interface errata
   module procedure errata0, errata1, erratai, erratal
 end interface
 interface siza
   module procedure siza2, siza3, siza4, sizav
 end interface
 interface prog_arit
   module procedure prog_arit_z, prog_arit_c,           &
                    prog_arit_d, prog_arit_r,           &
                    prog_arit_i
 end interface
 interface outer_prod
   module procedure outer_prod_z, outer_prod_c,         &
                    outer_prod_d, outer_prod_r,         &
                    outer_prod_i
 end interface
 interface scatter_add
   module procedure scatter_add_z, scatter_add_c,       &
                    scatter_add_d, scatter_add_r,       &
                    scatter_add_i
 end interface
 interface copia_arreglo
   module procedure copia_arreglo_z, copia_arreglo_c,   &
                    copia_arreglo_d, copia_arreglo_r,   &
                    copia_arreglo_i
 end interface
 interface maxvalue
   module procedure wmaxval, zmaxval, cmaxval, qmaxloc, dmaxval, rmaxval, imaxval
 end interface
 interface maxlocat
   module procedure wmaxloc, zmaxloc, cmaxloc, qmaxloc, dmaxloc, rmaxloc, imaxloc
 end interface
 interface swap
   module procedure wswap, zswap, cswap, qswap, dswap, rswap, iswap
 end interface
 !
 public  ! todo es asi excepto lo indicado en contrario
 character(10), private, parameter :: f = 'm_tools:> '
 !
 contains
 !
 subroutine  errata0 (ley1)
  character (*), intent (in) :: ley1
  write (*,100) f
  write (*,100) f // "error " // ley1
  100 format (a)
  stop
 end subroutine
 subroutine errata1 (jer2,ley2)
  integer (iin), intent (in) :: jer2
  character(*) , intent (in) :: ley2
  write (*,100) f
  write (*,100) f // "error " // ley2
  write (*,*) jer2
  stop
  100 format (a)
 end subroutine
 subroutine  erratai (jer3,ley3)
  integer (iin), dimension (:), intent (in) :: jer3
```



```
  character (*)               , intent (in) :: ley3
  write (*,100) f
  write (*,100) f // "error " // ley3
  write (*,*) jer3
  stop
  100 format (a)
 end subroutine
 subroutine  erratal (ber4,ley4)
  logical, dimension (:), intent (in) :: ber4
  character (*)         , intent (in) :: ley4
  write (*,100) f
  write (*,100) f // "error " // ley4
  write (*,*) ber4
  stop
  100 format (a)
 end subroutine
 !
 ! genera numero aleatorio entero "i" tal que a <= i < b
 ! En este esquema se genera primero un numero aleatorio
 ! real "r" en [0,1), luego se lo mapea al intervalo [a,b)
 ! y, finalmente, se toma i = piso (mapeo_ab(r))
 !
 function  irandom_ab (a, b, n)
  integer (iin), intent (in) :: a, b, n
  integer (iin), dimension (n) :: irandom_ab
  real    (idp), dimension (n) :: r, z
  integer (iin), dimension (n) :: i
  irandom_ab = 0
  call  random_number (r)
  z = a + abs (b - a) * r
  i = floor (z)
  where (i > z)  i = i - 1
  irandom_ab = i
 end function
 !
 ! devuelve dos numeros aleatorios distintos en [a,b)
 function  irandom_ne (a, b)
  integer (iin), intent (in) :: a, b
  integer (iin), dimension (2) :: irandom_ne
  real    (idp), dimension (2) :: r, z
  integer (iin), dimension (2) :: i
  irandom_ne = 0
  do
   call random_number (r)
   z = a + abs (b - a) * r
   i = floor (z)
   where (i > z)  i = i - 1
   if ( i (1) .ne. i (2) ) exit
  end do
  irandom_ne = i
 end function
 !
 ! desordena aleatoriamente la sucesion [a:b]
 function  barajar (a, b)
  integer (iin), intent (in) :: a, b
  integer (iin), dimension (b-a+1) :: barajar
  integer (iin), dimension (b-a+1) :: u, v
  integer (iin) :: h, i, j, k, n
  real    (idp) :: r, z
  u (1) = a
  n = (b - a) + 1
  do i = 2, n
   u (i) = u (i-1) + 1
  end do
  h = n
  do  k = 1, n
   call random_number (r)
   z = 1 + h * r
   i = floor (z)
```

```
   if (i > z)  i = i - 1
   v (k) = u (i)
   do  j = i, (n - 1)
    u (j) = u (j + 1)
   end do
   h = h - 1
 end do
 barajar = v
end function
!
! informa y fin si los enteros no son todos iguales
function  siza2 (n1, n2, s)
 integer (iin), intent (in) :: n1, n2
 character(*) , intent (in) :: s
 integer (iin) :: siza2
 if (n1 .eq. n2) then
  siza2 = n1
 else
  write (*,*) "n1 n2 ", n1, n2
  write (*,100) " error (siza2): fallo en " // s
  stop
 end if
 if (n1 .lt. 0)  stop" error (siza2): n < 0 "
 100 format (a)
end function
!
! informa y fin si los enteros no son todos iguales
function  siza3 (n1, n2, n3, s)
 integer (iin), intent (in) :: n1, n2, n3
 character(*) , intent (in) :: s
 integer (iin) :: siza3
 if (n1 .eq. n2 .and. n2 .eq. n3) then
  siza3 = n1
 else
  write (*,*)" n1 n2 n3 ", n1, n2, n3
  write (*,100) " error (siza3): fallo en " // s
  stop
 end if
 if (n1 .lt. 0)  stop" error (siza3): n < 0 "
 100 format (a)
end function
!
! informa y fin si los enteros no son todos iguales
function  siza4 (n1, n2, n3, n4, s)
 integer (iin), intent (in) :: n1, n2, n3, n4
 character(*) , intent (in) :: s
 integer (iin) :: siza4
 if (n1 .eq. n2 .and. n2 .eq. n3 .and. n3 .eq. n4) then
  siza4 = n1
 else
  write (*,*)" n1 n2 n3 n4 ", n1, n2, n3, n4
  write (*,100) " error (siza4): fallo en " // s
  stop
 end if
 if (n1 .lt. 0)  stop" error (siza4): n < 0 "
 100 format (a)
end function
!
! informa y fin si los enteros no son todos iguales
function  sizav (ii, s)
 integer (iin), dimension (:), intent (in) :: ii
 character(*) , intent (in) :: s
 integer (iin) :: sizav
 if ( all ( ii .eq. ii (1) ) ) then
  sizav = ii (1)
 else
  write (*,*)" ii ", ii
  write (*,100) " error (sizav): fallo en " // s
  stop
```



```
 end if
 if (ii (1) .lt. 0)  stop" error (sizav): n < 0 "
 100 format (a)
end function
!
! devuelve un vector de "n" elementos obtenidos como una progresion
! aritmetica de valor inicial "v_ini" e incremento "dta_v"
!
function  prog_arit_z (zv_ini, zdta_v, n)
 complex (idp), intent (in) :: zv_ini, zdta_v
 integer (iin), intent (in) :: n
 complex (idp), dimension (n) :: prog_arit_z
 integer (iin) :: k
 if (n > 0) prog_arit_z (1) = zv_ini
 forall (k=2:n) prog_arit_z (k) = zv_ini + (k -1) * zdta_v
end function
!
function  prog_arit_c (cv_ini, cdta_v, n)
 complex (isp), intent (in) :: cv_ini, cdta_v
 integer (iin), intent (in) :: n
 complex (isp), dimension (n) :: prog_arit_c
 integer (iin) :: k
 if (n > 0) prog_arit_c (1) = cv_ini
 forall (k=2:n) prog_arit_c (k) = cv_ini + (k -1) * cdta_v
end function
!
function  prog_arit_d (dv_ini, ddta_v, n)
 real    (idp), intent (in) :: dv_ini, ddta_v
 integer (iin), intent (in) :: n
 real    (idp), dimension (n) :: prog_arit_d
 integer (iin) :: k
 if (n > 0) prog_arit_d (1) = dv_ini
 forall (k=2:n) prog_arit_d (k) = dv_ini + (k -1) * ddta_v
end function
!
function  prog_arit_r (rv_ini, rdta_v, n)
 real    (isp), intent (in) :: rv_ini, rdta_v
 integer (iin), intent (in) :: n
 real    (isp), dimension (n) :: prog_arit_r
 integer (iin) :: k
 if (n > 0) prog_arit_r (1) = rv_ini
 forall (k=2:n) prog_arit_r (k) = rv_ini + (k -1) * rdta_v
end function
!
function  prog_arit_i (iv_ini, idta_v, n)
 integer (iin), intent (in) :: iv_ini, idta_v
 integer (iin), intent (in) :: n
 integer (iin), dimension (n) :: prog_arit_i
 integer (iin) :: k
 if (n > 0) prog_arit_i (1) = iv_ini
 forall (k=2:n) prog_arit_i (k) = iv_ini + (k -1) * idta_v
end function
!
! producto exterior
function  outer_prod_z (za,zb)
  complex (idp), dimension (:), intent (in) :: za, zb
  complex (idp), dimension (size(za),size(zb)) :: outer_prod_z
  outer_prod_z = spread (za, dim = 2, ncopies = size (zb) ) * &
                 spread (zb, dim = 1, ncopies = size (za) )
end function
!
function  outer_prod_c (ca,cb)
  complex (isp), dimension (:), intent (in) :: ca, cb
  complex (isp), dimension (size(ca),size(cb)) :: outer_prod_c
  outer_prod_c = spread (ca, dim = 2, ncopies = size (cb) ) * &
                 spread (cb, dim = 1, ncopies = size (ca) )
end function
!
function outer_prod_d (da,db)
```

```
   real (idp), dimension (:), intent (in) :: da, db
   real (idp), dimension (size(da),size(db)) :: outer_prod_d
   outer_prod_d = spread (da, dim = 2, ncopies = size (db) ) * &
                  spread (db, dim = 1, ncopies = size (da) )
 end function
 !
 function outer_prod_r (ra,rb)
   real (isp), dimension (:), intent (in) :: ra, rb
   real (isp), dimension (size(ra),size(rb)) :: outer_prod_r
   outer_prod_r = spread (ra, dim = 2, ncopies = size (rb) ) * &
                  spread (rb, dim = 1, ncopies = size (ra) )
 end function
 !
 function outer_prod_i (ia,ib)
   integer (iin), dimension (:), intent (in) :: ia, ib
   integer (iin), dimension (size(ia),size(ib)) :: outer_prod_i
   outer_prod_i = spread (ia, dim = 2, ncopies = size (ib) ) * &
                  spread (ib, dim = 1, ncopies = size (ia) )
 end function
 !
 ! suma expandida: desde "a" hacia "b" en las posiciones destino "ib"
 ! en HPF esta "scatter_add" se la puede reemplazar por la funcion
 ! de libreria "sum_scatter".
 !
 subroutine  scatter_add_z (zb, za, bindex)
  complex (idp), dimension (:), intent (out) :: zb
  complex (idp), dimension (:), intent (in)  :: za
  integer (iin), dimension (:), intent (in)  :: bindex
  integer (iin) :: m, n, j, i
  n = siza (size (za), size (bindex), "scatter_add")
  m = size (zb)
  zb = cmplx (0.0,0.0)
  do  j = 1, n
    i = bindex (j)               ! indice destino
    if (i < 1 .or. i > m) cycle  ! esta fuera de rango
    zb (i) = zb (i) + za (j)
  end do
 end subroutine
 !
 subroutine  scatter_add_c (cb, ca, bindex)
  complex (isp), dimension (:), intent (out) :: cb
  complex (isp), dimension (:), intent (in)  :: ca
  integer (iin), dimension (:), intent (in)  :: bindex
  integer (iin) :: m, n, j, i
  n = siza (size (ca), size (bindex), "scatter_add")
  m = size (cb)
  cb = cmplx (0.0,0.0)
  do  j = 1, n
    i = bindex (j)               ! indice destino
    if (i < 1 .or. i > m) cycle  ! esta fuera de rango
    cb (i) = cb (i) + ca (j)
  end do
 end subroutine
 !
 subroutine  scatter_add_d (db, da, bindex)
  real    (idp), dimension (:), intent (out) :: db
  real    (idp), dimension (:), intent (in)  :: da
  integer (iin), dimension (:), intent (in)  :: bindex
  integer (iin) :: m, n, j, i
  n = siza (size (da), size (bindex), "scatter_add")
  m = size (db)
  db = 0.0
  do  j = 1, n
    i = bindex (j)               ! indice destino
    if (i < 1 .or. i > m) cycle  ! esta fuera de rango
    db (i) = db (i) + da (j)
  end do
 end subroutine
 !
```



```
 subroutine  scatter_add_r (rb, ra, bindex)
  real    (isp), dimension (:), intent (out) :: rb
  real    (isp), dimension (:), intent (in)  :: ra
  integer (iin), dimension (:), intent (in)  :: bindex
  integer (iin) :: m, n, j, i
  n = siza (size (ra), size (bindex), "scatter_add")
  m = size (rb)
  rb = 0.0
  do  j = 1, n
    i = bindex (j)               ! indice destino
    if (i < 1 .or. i > m) cycle  ! esta fuera de rango
    rb (i) = rb (i) + ra (j)
  end do
 end subroutine
 !
 subroutine  scatter_add_i (ib, ia, bindex)
  integer (iin), dimension (:), intent (out) :: ib
  integer (iin), dimension (:), intent (in)  :: ia
  integer (iin), dimension (:), intent (in)  :: bindex
  integer (iin) :: m, n, j, i
  n = siza (size (ia), size (bindex), "scatter_add")
  m = size (ib)
  ib = 0
  do  j = 1, n
    i = bindex (j)               ! indice destino
    if (i < 1 .or. i > m) cycle  ! esta fuera de rango
    ib (i) = ib (i) + ia (j)
  end do
 end subroutine
 !
 ! copia como salida "b" un arreglo dato "a", donde |b| <= |a|
 !
 subroutine  copia_arreglo_z (za, zb, n_copiados, n_omitidos)
  complex (idp), dimension (:), intent (in)    :: za
  complex (idp), dimension (:), intent (inout) :: zb
  integer (iin)               , intent (out)   :: n_copiados
  integer (iin)               , intent (out)   :: n_omitidos
  integer (iin) :: m, n, j, i
  n_copiados = min (size (za), size (zb))
  n_omitidos = size (za) - n_copiados
  zb (1:n_copiados) = za (1:n_copiados)
 end subroutine
 !
 subroutine  copia_arreglo_c (ca, cb, n_copiados, n_omitidos)
  complex (isp), dimension (:), intent (in)    :: ca
  complex (isp), dimension (:), intent (inout) :: cb
  integer (iin)               , intent (out)   :: n_copiados
  integer (iin)               , intent (out)   :: n_omitidos
  integer (iin) :: m, n, j, i
  n_copiados = min (size (ca), size (cb))
  n_omitidos = size (ca) - n_copiados
  cb (1:n_copiados) = ca (1:n_copiados)
 end subroutine
 !
 subroutine  copia_arreglo_d (da, db, n_copiados, n_omitidos)
  real    (idp), dimension (:), intent (in)    :: da
  real    (idp), dimension (:), intent (inout) :: db
  integer (iin)               , intent (out)   :: n_copiados
  integer (iin)               , intent (out)   :: n_omitidos
  integer (iin) :: m, n, j, i
  n_copiados = min (size (da), size (db))
  n_omitidos = size (da) - n_copiados
  db (1:n_copiados) = da (1:n_copiados)
 end subroutine
 !
 subroutine  copia_arreglo_r (ra, rb, n_copiados, n_omitidos)
  real    (isp), dimension (:), intent (in)    :: ra
  real    (isp), dimension (:), intent (inout) :: rb
  integer (iin)               , intent (out)   :: n_copiados
```

```
  integer (iin)               , intent (out)   :: n_omitidos
  integer (iin) :: m, n, j, i
  n_copiados = min (size (ra), size (rb))
  n_omitidos = size (ra) - n_copiados
  rb (1:n_copiados) = ra (1:n_copiados)
 end subroutine
 !
 subroutine  copia_arreglo_i (ia, ib, n_copiados, n_omitidos)
  integer (iin), dimension (:), intent (in)    :: ia
  integer (iin), dimension (:), intent (inout) :: ib
  integer (iin)               , intent (out)   :: n_copiados
  integer (iin)               , intent (out)   :: n_omitidos
  integer (iin) :: m, n, j, i
  n_copiados = min (size (ia), size (ib))
  n_omitidos = size (ia) - n_copiados
  ib (1:n_copiados) = ia (1:n_copiados)
 end subroutine
 !
 !
 function wmaxval (wa)
   complex (iqp), dimension (:), intent (in) :: wa
   real    (iqp), dimension (1) :: wmax
   real    (iqp) :: wmaxval
   wmax = maxval (abs(wa(:)))
   wmaxval = wmax (1)
 end function
 !
 function zmaxval (za)
   complex (idp), dimension (:), intent (in) :: za
   real    (idp), dimension (1) :: zmax
   real    (idp) :: zmaxval
   zmax = maxval (abs(za(:)))
   zmaxval = zmax (1)
 end function
 !
 function cmaxval (ca)
   complex (isp), dimension (:), intent (in) :: ca
   real    (isp), dimension (1) :: cmax
   real    (isp) :: cmaxval
   cmax = maxval (abs(ca(:)))
   cmaxval = cmax (1)
 end function
 !
 function qmaxval (qa)
   real    (iqp), dimension (:), intent (in) :: qa
   real    (iqp), dimension (1) :: qmax
   real    (iqp) :: qmaxval
   qmax = maxval (qa(:))
   qmaxval = qmax (1)
 end function
 !
 function dmaxval (da)
   real    (idp), dimension (:), intent (in) :: da
   real    (idp), dimension (1) :: dmax
   real    (idp) :: dmaxval
   dmax = maxval (da(:))
   dmaxval = dmax (1)
 end function
 !
 function rmaxval (ra)
   real    (isp), dimension (:), intent (in) :: ra
   real    (isp), dimension (1) :: rmax
   real    (isp) :: rmaxval
   rmax = maxval (ra(:))
   rmaxval = rmax (1)
 end function
 !
 function  imaxval (ia)
   integer (iin), dimension (:), intent (in) :: ia
```



```
   integer (iin), dimension (1) :: imax
   integer (iin) :: imaxval
   imax = maxval (ia(:))
   imaxval = imax (1)
 end function
 !
 function  wmaxloc (wa)
   complex (iqp), dimension (:), intent (in) :: wa
   integer (iin), dimension (1) :: wmax
   integer (iin) :: wmaxloc
   wmax = maxloc (abs(wa(:)))
   wmaxloc = wmax (1)
 end function
 !
 function  zmaxloc (za)
   complex (idp), dimension (:), intent (in) :: za
   integer (iin), dimension (1) :: zmax
   integer (iin) :: zmaxloc
   zmax = maxloc (abs(za(:)))
   zmaxloc = zmax (1)
 end function
 !
 function  cmaxloc (ca)
   complex (isp), dimension (:), intent (in) :: ca
   integer (iin), dimension (1) :: cmax
   integer (iin) :: cmaxloc
   cmax = maxloc (abs(ca(:)))
   cmaxloc = cmax (1)
 end function
 !
 function  qmaxloc (qa)
   real    (iqp), dimension (:), intent (in) :: qa
   integer (iin), dimension (1) :: qmax
   integer (iin) :: qmaxloc
   qmax = maxloc (qa(:))
   qmaxloc = qmax (1)
 end function
 !
 function  dmaxloc (da)
   real    (idp), dimension (:), intent (in) :: da
   integer (iin), dimension (1) :: dmax
   integer (iin) :: dmaxloc
   dmax = maxloc (da(:))
   dmaxloc = dmax (1)
 end function
 !
 function  rmaxloc (ra)
   real    (isp), dimension (:), intent (in) :: ra
   integer (iin), dimension (1) :: rmax
   integer (iin) :: rmaxloc
   rmax = maxloc (ra(:))
   rmaxloc = rmax (1)
 end function
 !
 function  imaxloc (ia)
   integer (iin), dimension (:), intent (in) :: ia
   integer (iin), dimension (1) :: imax
   integer (iin) :: imaxloc
   imax = maxloc (ia(:))
   imaxloc = imax (1)
 end function
 !
 !
 subroutine  wswap (wa,wb)
   complex (iqp), dimension (:), intent (inout) :: wa, wb
   complex (iqp), dimension (size(wa)) :: wt
   wt = wa
   wa = wb
   wb = wt
```

```
 end subroutine
 !
 subroutine  zswap (za,zb)
   complex (idp), dimension (:), intent (inout) :: za, zb
   complex (idp), dimension (size(za)) :: zt
   zt = za
   za = zb
   zb = zt
 end subroutine
 !
 subroutine  cswap (ca,cb)
   complex (isp), dimension (:), intent (inout) :: ca, cb
   complex (isp), dimension (size(ca)) :: ct
   ct = ca
   ca = cb
   cb = ct
 end subroutine
 !
 subroutine  qswap (qa,qb)
   real (iqp), dimension (:), intent (inout) :: qa, qb
   real (iqp), dimension (size(qa)) :: qt
   qt = qa
   qa = qb
   qb = qt
 end subroutine
 !
 subroutine  dswap (da,db)
   real (idp), dimension (:), intent (inout) :: da, db
   real (idp), dimension (size(da)) :: dt
   dt = da
   da = db
   db = dt
 end subroutine
 !
 subroutine  rswap (ra,rb)
   real (isp), dimension (:), intent (inout) :: ra, rb
   real (isp), dimension (size(ra)) :: rt
   rt = ra
   ra = rb
   rb = rt
 end subroutine
 !
 subroutine  iswap (ia,ib)
   integer (iin), dimension (:), intent (inout) :: ia, ib
   integer (iin), dimension (size(ia)) :: it
   it = ia
   ia = ib
   ib = it
 end subroutine
 !
end module
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
```

```
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
! Herramientas varias con:
! precision doble
! precision cuadruple ! it's not possible due to ADAPTOR restrictions
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
module m_tools1
 use   m_ctes
 use   m_temps
 implicit none
 !
 interface errata
   module procedure errata0, errata1, erratai, erratal
 end interface
 interface siza
   module procedure siza2, siza3, siza4, sizav
 end interface
 interface prog_arit
   module procedure prog_arit_z, prog_arit_c,           &
                    prog_arit_d, prog_arit_r,           &
                    prog_arit_i
 end interface
 interface outer_prod
   module procedure outer_prod_z, outer_prod_c,         &
                    outer_prod_d, outer_prod_r,         &
                    outer_prod_i
 end interface
 interface scatter_add
   module procedure scatter_add_z, scatter_add_c,       &
                    scatter_add_d, scatter_add_r,       &
                    scatter_add_i
 end interface
 interface copia_arreglo
   module procedure copia_arreglo_z, copia_arreglo_c,   &
                    copia_arreglo_d, copia_arreglo_r,   &
                    copia_arreglo_i
 end interface
 interface maxvalue
   module procedure zmaxval, cmaxval, dmaxval, rmaxval, imaxval
 end interface
 interface maxlocat
   module procedure zmaxloc, cmaxloc, dmaxloc, rmaxloc, imaxloc
 end interface
 interface swap
   module procedure zswap, cswap, dswap, rswap, iswap
 end interface
 !
 public  ! todo es asi excepto lo indicado en contrario
 character(10), private, parameter :: f = 'm_tools:> '
 !
 contains
 !
 subroutine  errata0 (ley1)
  character (*), intent (in) :: ley1
  write (*,100) f
  write (*,100) f // "error " // ley1
  100 format (a)
  stop
 end subroutine
 subroutine errata1 (jer2,ley2)
  integer (iin), intent (in) :: jer2
  character(*) , intent (in) :: ley2
  write (*,100) f
  write (*,100) f // "error " // ley2
  write (*,*) jer2
  stop
  100 format (a)
 end subroutine
 subroutine  erratai (jer3,ley3)
  integer (iin), dimension (:), intent (in) :: jer3
```



```
  character (*)               , intent (in) :: ley3
  write (*,100) f
  write (*,100) f // "error " // ley3
  write (*,*) jer3
  stop
  100 format (a)
 end subroutine
 subroutine  erratal (ber4,ley4)
  logical, dimension (:), intent (in) :: ber4
  character (*)         , intent (in) :: ley4
  write (*,100) f
  write (*,100) f // "error " // ley4
  write (*,*) ber4
  stop
  100 format (a)
 end subroutine
 !
 ! genera numero aleatorio entero "i" tal que a <= i < b
 ! En este esquema se genera primero un numero aleatorio
 ! real "r" en [0,1), luego se lo mapea al intervalo [a,b)
 ! y, finalmente, se toma i = piso (mapeo_ab(r))
 !
 function  irandom_ab (a, b, n)
  integer (iin), intent (in) :: a, b, n
  integer (iin), dimension (n) :: irandom_ab
  real    (idp), dimension (n) :: r, z
  integer (iin), dimension (n) :: i
  irandom_ab = 0
  call  random_number (r)
  z = a + abs (b - a) * r
  i = floor (z)
  where (i > z)  i = i - 1
  irandom_ab = i
 end function
 !
 ! devuelve dos numeros aleatorios distintos en [a,b)
 function  irandom_ne (a, b)
  integer (iin), intent (in) :: a, b
  integer (iin), dimension (2) :: irandom_ne
  real    (idp), dimension (2) :: r, z
  integer (iin), dimension (2) :: i
  irandom_ne = 0
  do
   call random_number (r)
   z = a + abs (b - a) * r
   i = floor (z)
   where (i > z)  i = i - 1
   if ( i (1) .ne. i (2) ) exit
  end do
  irandom_ne = i
 end function
 !
 ! desordena aleatoriamente la sucesion [a:b]
 function  barajar (a, b)
  integer (iin), intent (in) :: a, b
  integer (iin), dimension (b-a+1) :: barajar
  integer (iin), dimension (b-a+1) :: u, v
  integer (iin) :: h, i, j, k, n
  real    (idp) :: r, z
  u (1) = a
  n = (b - a) + 1
  do i = 2, n
   u (i) = u (i-1) + 1
  end do
  h = n
  do  k = 1, n
   call random_number (r)
   z = 1 + h * r
   i = floor (z)
```

```
   if (i > z)  i = i - 1
   v (k) = u (i)
   do  j = i, (n - 1)
    u (j) = u (j + 1)
   end do
   h = h - 1
 end do
 barajar = v
end function
!
! informa y fin si los enteros no son todos iguales
function  siza2 (n1, n2, s)
 integer (iin), intent (in) :: n1, n2
 character(*) , intent (in) :: s
 integer (iin) :: siza2
 if (n1 .eq. n2) then
  siza2 = n1
 else
  write (*,*) "n1 n2 ", n1, n2
  write (*,100) " error (siza2): fallo en " // s
  stop
 end if
 if (n1 .lt. 0)  stop" error (siza2): n < 0 "
 100 format (a)
end function
!
! informa y fin si los enteros no son todos iguales
function  siza3 (n1, n2, n3, s)
 integer (iin), intent (in) :: n1, n2, n3
 character(*) , intent (in) :: s
 integer (iin) :: siza3
 if (n1 .eq. n2 .and. n2 .eq. n3) then
  siza3 = n1
 else
  write (*,*)" n1 n2 n3 ", n1, n2, n3
  write (*,100) " error (siza3): fallo en " // s
  stop
 end if
 if (n1 .lt. 0)  stop" error (siza3): n < 0 "
 100 format (a)
end function
!
! informa y fin si los enteros no son todos iguales
function  siza4 (n1, n2, n3, n4, s)
 integer (iin), intent (in) :: n1, n2, n3, n4
 character(*) , intent (in) :: s
 integer (iin) :: siza4
 if (n1 .eq. n2 .and. n2 .eq. n3 .and. n3 .eq. n4) then
  siza4 = n1
 else
  write (*,*)" n1 n2 n3 n4 ", n1, n2, n3, n4
  write (*,100) " error (siza4): fallo en " // s
  stop
 end if
 if (n1 .lt. 0)  stop" error (siza4): n < 0 "
 100 format (a)
end function
!
! informa y fin si los enteros no son todos iguales
function  sizav (ii, s)
 integer (iin), dimension (:), intent (in) :: ii
 character(*) , intent (in) :: s
 integer (iin) :: sizav
 if ( all ( ii .eq. ii (1) ) ) then
  sizav = ii (1)
 else
  write (*,*)" ii ", ii
  write (*,100) " error (sizav): fallo en " // s
  stop
```



```
 end if
 if (ii (1) .lt. 0)  stop" error (sizav): n < 0 "
 100 format (a)
end function
!
! devuelve un vector de "n" elementos obtenidos como una progresion
! aritmetica de valor inicial "v_ini" e incremento "dta_v"
!
function  prog_arit_z (zv_ini, zdta_v, n)
 complex (idp), intent (in) :: zv_ini, zdta_v
 integer (iin), intent (in) :: n
 complex (idp), dimension (n) :: prog_arit_z
 integer (iin) :: k
 if (n > 0) prog_arit_z (1) = zv_ini
 forall (k=2:n) prog_arit_z (k) = zv_ini + (k -1) * zdta_v
end function
!
function  prog_arit_c (cv_ini, cdta_v, n)
 complex (isp), intent (in) :: cv_ini, cdta_v
 integer (iin), intent (in) :: n
 complex (isp), dimension (n) :: prog_arit_c
 integer (iin) :: k
 if (n > 0) prog_arit_c (1) = cv_ini
 forall (k=2:n) prog_arit_c (k) = cv_ini + (k -1) * cdta_v
end function
!
function  prog_arit_d (dv_ini, ddta_v, n)
 real    (idp), intent (in) :: dv_ini, ddta_v
 integer (iin), intent (in) :: n
 real    (idp), dimension (n) :: prog_arit_d
 integer (iin) :: k
 if (n > 0) prog_arit_d (1) = dv_ini
 forall (k=2:n) prog_arit_d (k) = dv_ini + (k -1) * ddta_v
end function
!
function  prog_arit_r (rv_ini, rdta_v, n)
 real    (isp), intent (in) :: rv_ini, rdta_v
 integer (iin), intent (in) :: n
 real    (isp), dimension (n) :: prog_arit_r
 integer (iin) :: k
 if (n > 0) prog_arit_r (1) = rv_ini
 forall (k=2:n) prog_arit_r (k) = rv_ini + (k -1) * rdta_v
end function
!
function  prog_arit_i (iv_ini, idta_v, n)
 integer (iin), intent (in) :: iv_ini, idta_v
 integer (iin), intent (in) :: n
 integer (iin), dimension (n) :: prog_arit_i
 integer (iin) :: k
 if (n > 0) prog_arit_i (1) = iv_ini
 forall (k=2:n) prog_arit_i (k) = iv_ini + (k -1) * idta_v
end function
!
! producto exterior
function  outer_prod_z (za,zb)
  complex (idp), dimension (:), intent (in) :: za, zb
  complex (idp), dimension (size(za),size(zb)) :: outer_prod_z
  outer_prod_z = spread (za, dim = 2, ncopies = size (zb) ) * &
                 spread (zb, dim = 1, ncopies = size (za) )
end function
!
function  outer_prod_c (ca,cb)
  complex (isp), dimension (:), intent (in) :: ca, cb
  complex (isp), dimension (size(ca),size(cb)) :: outer_prod_c
  outer_prod_c = spread (ca, dim = 2, ncopies = size (cb) ) * &
                 spread (cb, dim = 1, ncopies = size (ca) )
end function
!
function outer_prod_d (da,db)
```

```
   real (idp), dimension (:), intent (in) :: da, db
   real (idp), dimension (size(da),size(db)) :: outer_prod_d
   outer_prod_d = spread (da, dim = 2, ncopies = size (db) ) * &
                  spread (db, dim = 1, ncopies = size (da) )
end function
!
function outer_prod_r (ra,rb)
   real (isp), dimension (:), intent (in) :: ra, rb
   real (isp), dimension (size(ra),size(rb)) :: outer_prod_r
   outer_prod_r = spread (ra, dim = 2, ncopies = size (rb) ) * &
                  spread (rb, dim = 1, ncopies = size (ra) )
end function
!
function outer_prod_i (ia,ib)
   integer (iin), dimension (:), intent (in) :: ia, ib
   integer (iin), dimension (size(ia),size(ib)) :: outer_prod_i
   outer_prod_i = spread (ia, dim = 2, ncopies = size (ib) ) * &
                  spread (ib, dim = 1, ncopies = size (ia) )
end function
!
! suma expandida: desde "a" hacia "b" en las posiciones destino "ib"
! en HPF esta "scatter_add" se la puede reemplazar por la funcion
! de libreria "sum_scatter".
!
subroutine  scatter_add_z (zb, za, bindex)
 complex (idp), dimension (:), intent (out) :: zb
 complex (idp), dimension (:), intent (in)  :: za
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (za), size (bindex), "scatter_add")
 m = size (zb)
 zb = cmplx (0.0,0.0)
 do  j = 1, n
   i = bindex (j)               ! indice destino
   if (i < 1 .or. i > m) cycle  ! esta fuera de rango
   zb (i) = zb (i) + za (j)
 end do
end subroutine
!
subroutine  scatter_add_c (cb, ca, bindex)
 complex (isp), dimension (:), intent (out) :: cb
 complex (isp), dimension (:), intent (in)  :: ca
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (ca), size (bindex), "scatter_add")
 m = size (cb)
 cb = cmplx (0.0,0.0)
 do  j = 1, n
   i = bindex (j)               ! indice destino
   if (i < 1 .or. i > m) cycle  ! esta fuera de rango
   cb (i) = cb (i) + ca (j)
 end do
end subroutine
!
subroutine  scatter_add_d (db, da, bindex)
 real    (idp), dimension (:), intent (out) :: db
 real    (idp), dimension (:), intent (in)  :: da
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (da), size (bindex), "scatter_add")
 m = size (db)
 db = 0.0
 do  j = 1, n
   i = bindex (j)               ! indice destino
   if (i < 1 .or. i > m) cycle  ! esta fuera de rango
   db (i) = db (i) + da (j)
 end do
end subroutine
!
```



```
subroutine  scatter_add_r (rb, ra, bindex)
 real    (isp), dimension (:), intent (out) :: rb
 real    (isp), dimension (:), intent (in)  :: ra
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (ra), size (bindex), "scatter_add")
 m = size (rb)
 rb = 0.0
 do  j = 1, n
   i = bindex (j)               ! indice destino
   if (i < 1 .or. i > m) cycle  ! esta fuera de rango
   rb (i) = rb (i) + ra (j)
 end do
end subroutine
!
subroutine  scatter_add_i (ib, ia, bindex)
 integer (iin), dimension (:), intent (out) :: ib
 integer (iin), dimension (:), intent (in)  :: ia
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (ia), size (bindex), "scatter_add")
 m = size (ib)
 ib = 0
 do  j = 1, n
   i = bindex (j)               ! indice destino
   if (i < 1 .or. i > m) cycle  ! esta fuera de rango
   ib (i) = ib (i) + ia (j)
 end do
end subroutine
!
! copia como salida "b" un arreglo dato "a", donde |b| <= |a|
!
subroutine  copia_arreglo_z (za, zb, n_copiados, n_omitidos)
 complex (idp), dimension (:), intent (in)    :: za
 complex (idp), dimension (:), intent (inout) :: zb
 integer (iin)               , intent (out)   :: n_copiados
 integer (iin)               , intent (out)   :: n_omitidos
 integer (iin) :: m, n, j, i
 n_copiados = min (size (za), size (zb))
 n_omitidos = size (za) - n_copiados
 zb (1:n_copiados) = za (1:n_copiados)
end subroutine
!
subroutine  copia_arreglo_c (ca, cb, n_copiados, n_omitidos)
 complex (isp), dimension (:), intent (in)    :: ca
 complex (isp), dimension (:), intent (inout) :: cb
 integer (iin)               , intent (out)   :: n_copiados
 integer (iin)               , intent (out)   :: n_omitidos
 integer (iin) :: m, n, j, i
 n_copiados = min (size (ca), size (cb))
 n_omitidos = size (ca) - n_copiados
 cb (1:n_copiados) = ca (1:n_copiados)
end subroutine
!
subroutine  copia_arreglo_d (da, db, n_copiados, n_omitidos)
 real    (idp), dimension (:), intent (in)    :: da
 real    (idp), dimension (:), intent (inout) :: db
 integer (iin)               , intent (out)   :: n_copiados
 integer (iin)               , intent (out)   :: n_omitidos
 integer (iin) :: m, n, j, i
 n_copiados = min (size (da), size (db))
 n_omitidos = size (da) - n_copiados
 db (1:n_copiados) = da (1:n_copiados)
end subroutine
!
subroutine  copia_arreglo_r (ra, rb, n_copiados, n_omitidos)
 real    (isp), dimension (:), intent (in)    :: ra
 real    (isp), dimension (:), intent (inout) :: rb
 integer (iin)               , intent (out)   :: n_copiados
```

```
   integer (iin)               , intent (out)   :: n_omitidos
   integer (iin) :: m, n, j, i
   n_copiados = min (size (ra), size (rb))
   n_omitidos = size (ra) - n_copiados
   rb (1:n_copiados) = ra (1:n_copiados)
 end subroutine
 !
 subroutine  copia_arreglo_i (ia, ib, n_copiados, n_omitidos)
   integer (iin), dimension (:), intent (in)    :: ia
   integer (iin), dimension (:), intent (inout) :: ib
   integer (iin)               , intent (out)   :: n_copiados
   integer (iin)               , intent (out)   :: n_omitidos
   integer (iin) :: m, n, j, i
   n_copiados = min (size (ia), size (ib))
   n_omitidos = size (ia) - n_copiados
   ib (1:n_copiados) = ia (1:n_copiados)
 end subroutine
 !
 !
!function wmaxval (wa)
!  complex (iqp), dimension (:), intent (in) :: wa
!  real    (iqp), dimension (1) :: wmax
!  real    (iqp) :: wmaxval
!  wmax = maxval (abs(wa(:)))
!  wmaxval = wmax (1)
!end function
 !
 function zmaxval (za)
   complex (idp), dimension (:), intent (in) :: za
   real    (idp), dimension (1) :: zmax
   real    (idp) :: zmaxval
   zmax = maxval (abs(za(:)))
   zmaxval = zmax (1)
 end function
 !
 function cmaxval (ca)
   complex (isp), dimension (:), intent (in) :: ca
   real    (isp), dimension (1) :: cmax
   real    (isp) :: cmaxval
   cmax = maxval (abs(ca(:)))
   cmaxval = cmax (1)
 end function
 !
!function qmaxval (qa)
!  real    (iqp), dimension (:), intent (in) :: qa
!  real    (iqp), dimension (1) :: qmax
!  real    (iqp) :: qmaxval
!  qmax = maxval (qa(:))
!  qmaxval = qmax (1)
!end function
 !
 function dmaxval (da)
   real    (idp), dimension (:), intent (in) :: da
   real    (idp), dimension (1) :: dmax
   real    (idp) :: dmaxval
   dmax = maxval (da(:))
   dmaxval = dmax (1)
 end function
 !
 function rmaxval (ra)
   real    (isp), dimension (:), intent (in) :: ra
   real    (isp), dimension (1) :: rmax
   real    (isp) :: rmaxval
   rmax = maxval (ra(:))
   rmaxval = rmax (1)
 end function
 !
 function  imaxval (ia)
   integer (iin), dimension (:), intent (in) :: ia
```



```
   integer (iin), dimension (1) :: imax
   integer (iin) :: imaxval
   imax = maxval (ia(:))
   imaxval = imax (1)
 end function
 !
!function  wmaxloc (wa)
!  complex (iqp), dimension (:), intent (in) :: wa
!  integer (iin), dimension (1) :: wmax
!  integer (iin) :: wmaxloc
!  wmax = maxloc (abs(wa(:)))
!  wmaxloc = wmax (1)
!end function
 !
 function  zmaxloc (za)
   complex (idp), dimension (:), intent (in) :: za
   integer (iin), dimension (1) :: zmax
   integer (iin) :: zmaxloc
   zmax = maxloc (abs(za(:)))
   zmaxloc = zmax (1)
 end function
 !
 function  cmaxloc (ca)
   complex (isp), dimension (:), intent (in) :: ca
   integer (iin), dimension (1) :: cmax
   integer (iin) :: cmaxloc
   cmax = maxloc (abs(ca(:)))
   cmaxloc = cmax (1)
 end function
 !
!function  qmaxloc (qa)
!  real    (iqp), dimension (:), intent (in) :: qa
!  integer (iin), dimension (1) :: qmax
!  integer (iin) :: qmaxloc
!  qmax = maxloc (qa(:))
!  qmaxloc = qmax (1)
!end function
 !
 function  dmaxloc (da)
   real    (idp), dimension (:), intent (in) :: da
   integer (iin), dimension (1) :: dmax
   integer (iin) :: dmaxloc
   dmax = maxloc (da(:))
   dmaxloc = dmax (1)
 end function
 !
 function  rmaxloc (ra)
   real    (isp), dimension (:), intent (in) :: ra
   integer (iin), dimension (1) :: rmax
   integer (iin) :: rmaxloc
   rmax = maxloc (ra(:))
   rmaxloc = rmax (1)
 end function
 !
 function  imaxloc (ia)
   integer (iin), dimension (:), intent (in) :: ia
   integer (iin), dimension (1) :: imax
   integer (iin) :: imaxloc
   imax = maxloc (ia(:))
   imaxloc = imax (1)
 end function
 !
 !
!subroutine  wswap (wa,wb)
!  complex (iqp), dimension (:), intent (inout) :: wa, wb
!  complex (iqp), dimension (size(wa)) :: wt
!  wt = wa
!  wa = wb
!  wb = wt
```

```
!end subroutine
 !
 subroutine  zswap (za,zb)
   complex (idp), dimension (:), intent (inout) :: za, zb
   complex (idp), dimension (size(za)) :: zt
   zt = za
   za = zb
   zb = zt
 end subroutine
 !
 subroutine  cswap (ca,cb)
   complex (isp), dimension (:), intent (inout) :: ca, cb
   complex (isp), dimension (size(ca)) :: ct
   ct = ca
   ca = cb
   cb = ct
 end subroutine
 !
!subroutine  qswap (qa,qb)
!  real (iqp), dimension (:), intent (inout) :: qa, qb
!  real (iqp), dimension (size(qa)) :: qt
!  qt = qa
!  qa = qb
!  qb = qt
!end subroutine
 !
 subroutine  dswap (da,db)
   real (idp), dimension (:), intent (inout) :: da, db
   real (idp), dimension (size(da)) :: dt
   dt = da
   da = db
   db = dt
 end subroutine
 !
 subroutine  rswap (ra,rb)
   real (isp), dimension (:), intent (inout) :: ra, rb
   real (isp), dimension (size(ra)) :: rt
   rt = ra
   ra = rb
   rb = rt
 end subroutine
 !
 subroutine  iswap (ia,ib)
   integer (iin), dimension (:), intent (inout) :: ia, ib
   integer (iin), dimension (size(ia)) :: it
   it = ia
   ia = ib
   ib = it
 end subroutine
 !
end module
! -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
```