# Lección 14
# Caso de estudio: cálculo disperso

ÍNDICE: 14.1. Mejoras en la representación de listas e implementación de operaciones. 14.2. Vectores cuasi-vacíos. 14.3. Matrices cuasi-vacías.

PALABRAS CLAVE: Representación de listas con observación secuencial eficiente. Implementación de las operaciones con listas. Vectores cuasi-vacíos y algunas operaciones vectoriales. Matrices cuasi-vacías y algunas operaciones matriciales.

## 14.1. Mejoras en la representación de listas e implementación de operaciones

El objetivo de esta lección es abordar el problema del *cálculo vectorial y matricial* en aquellos casos en los que los vectores y matrices que intervienen tienen una *gran dimensión* pero un *gran número de componentes nulas*. En tales casos, una solución adecuada para la representación de los vectores y matrices (denominados *cuasi-vacíos*) consiste en almacenar exclusivamente las componentes no nulas.

Se abordará el problema en dos fases. En primer lugar, se utilizarán las listas homogéneas con acceso por posición como soporte fundamental de los vectores cuasi-vacíos, de forma que cada elemento de una lista almacenará el valor de una componente no nula del vector y su índice correspondiente. A continuación, se construirán las matrices cuasi-vacías como vectores de listas homogéneas con acceso por posición.

Las operaciones que van a considerarse son: la suma de vectores y de matrices, el producto de un escalar por un vector, el producto escalar de dos vectores, la trasposición de matrices, el producto de matrices, y la lectura desde teclado y escritura en pantalla de vectores y matrices. La implementación de esas operaciones sólo precisa, en cuanto a la capacidad de acceso por posición de las listas, el *recorrido secuencial de los elementos para observar su contenido*; por tanto, se va a considerar la siguiente *mejora de la representación de listas* desarrollada en la lección anterior:

```
tipos componente = registro
                     ind: 1..maxEntero;
```

```
                val: real
               freg;
ptUnDato = ↑unDato;
listaDeComp = registro
                prim,ult,ptActual: ptUnDato;
                n,indActual: 0..maxEntero
               freg;
unDato = registro
           dato: componente;
           sig: ptUnDato
         freg
```

El índice `indActual` guarda el número de orden en la lista del elemento más recientemente accedido y el puntero `ptActual` guarda la dirección de la celda correspondiente a dicho elemento. En el caso de la lista vacía, supondremos que `indActual = 0` y `ptActual = nil`.

Se incluye a continuación una implementación de las operaciones que se utilizarán posteriormente (el resto se plantean como ejercicio):

```
módulo listasDeComponentes
exporta
  tipos componente = registro
                       ind: 1..maxEntero;
                       val: real
                      freg;
        listaDeComp

  algoritmo creaVacía(referencia l:listaDeComp)
  {1}
  algoritmo añadeIzq(valor c:componente;
                     referencia l:listaDeComp)
  algoritmo concatena(referencia l1:listaDeComp;
                      valor l2:listaDeComp)
  algoritmo creaUnitaria(valor c:componente;
                         referencia l:listaDeComp)
  algoritmo añadeDch(referencia l:listaDeComp;
                     valor c:componente)
  algoritmo eliminaIzq(referencia l:listaDeComp)
  algoritmo eliminaDch(referencia l:listaDeComp)
  algoritmo observaIzq(valor l:listaDeComp) devuelve componente
  algoritmo observaDch(valor l:listaDeComp) devuelve componente
  algoritmo long(valor l:listaDeComp) devuelve 0..maxEntero
```

1 La especificación de todos los algoritmos es análoga a la de los correspondientes a las listas genéricas con acceso por posición, desarrollada en la lección anterior.

```
  algoritmo está(valor c:componente; valor l:listaDeComp)
              devuelve booleano
  algoritmo esVacía(valor l:listaDeComp) devuelve booleano
  algoritmo inser(referencia l:listaDeComp;
                  valor i:1..maxEntero; valor c:componente)
  algoritmo borra(referencia l:listaDeComp;
                  valor i:1..maxEntero)
  algoritmo modif(referencia l:listaDeComp;
                  valor i:1..maxEntero; valor c:componente)
  algoritmo observa(valor l:listaDeComp; valor i:1..maxEntero)
             devuelve componente
  algoritmo pos(valor c:componente; valor l:listaDeComp)
             devuelve 1..maxEntero

  algoritmo asignar(referencia nueva:listaDeComp;
                    valor vieja:listaDeComp)
  algoritmo liberar(referencia l:listaDeComp)
implementación
  tipo ptUnDato = ↑unDato;
       listaDeComp = registro
                       prim,ult,ptActual: ptUnDato;
                       n,indActual: 0..maxEntero
                     freg;
       unDato = registro
                  dato: componente;
                  sig: ptUnDato
                freg

  algoritmo creaVacía(referencia l:listaDeComp)
  principio
    l.prim:=nil;
    l.ult:=nil;
    l.n:=0;
    l.ptActual:=nil;
    l.indActual:=0
  fin

  algoritmo añadeDch(referencia l:listaDeComp;
                     valor c:componente)
  principio
    si l.ult=nil
    entonces
      nuevoDato(l.ult);
      l.prim:=l.ult
    sino
      nuevoDato(l.ult↑.sig);
      l.ult:=l.ult↑.sig
```

```
      fsi;
      l.ult↑.dato:=c;
      l.ult↑.sig:=nil;
      l.n:=l.n+1
      l.ptActual:=l.ult;
      l.indActual:=l.n
    fin

    algoritmo long(valor l:listaDeComp) devuelve 0..maxEntero
    principio
      devuelve(l.n)
    fin

    algoritmo observa(valor l:listaDeComp; valor i:1..maxEntero)
              devuelve componente
    principio
      si i<l.indActual
      entonces
        l.ptActual:=l.prim;
        l.indActual:=1
      fsi;
      mientrasQue l.indActual<i hacer
        l.ptActual:=l.ptActual↑.sig;
        l.indActual:=l.indActual+1
      fmq;
      devuelve(l.ptActual↑.dato)
    fin
  fin
```

## 14.2. Vectores cuasi-vacíos

A continuación se va a implementar el *TAD de los vectores de números reales con las operaciones de lectura, escritura, suma, producto por un escalar y producto escalar* de dos vectores. No se detallará la especificación del TAD, pues es sobradamente conocida en el ámbito del álgebra lineal.

El *soporte* de los valores del tipo se realizará sobre una *lista homogénea con acceso por posición*, de forma que cada elemento de la lista almacenará una componente no nula del vector (índice y valor) y los elementos de la lista se guardarán *ordenados por valores crecientes del índice*.

```
módulo vectoresCVs
importa listasDeComponentes
exporta
```

```
  tipo vectorCV

  algoritmo leeVectorCV(referencia v:vectorCV)
  { Lee desde teclado un vector y su dimensión y lo almacena en
    v. }

  algoritmo escribeVectorCV(valor v:vectorCV)
  { Escribe en pantalla el valor del vector v. }

  algoritmo dimensión(valor v:vectorCV) devuelve 1..maxEntero
  { Devuelve la dimensión del vector v. }

  algoritmo observaComp(valor v:vectorCV; valor i:1..maxEntero)
            devuelve real
  { Devuelve el valor de la componente i-ésima de v. }

  algoritmo sumaVectoresCVs(valor v1,v2:vectorCV;
                            referencia vs:vectorCV;
                            referencia error:booleano)
  { Si v1 y v2 tienen igual dimensión, guarda su suma en vs y el
    valor falso en error; si tienen distinta dimensión, guarda
    verdad en error. }

  algoritmo escalarPorVectorCV(valor x:real; valor v:vectorCV;
                               referencia p:vectorCV)
  { Guarda en p el producto de x por v. }

  algoritmo productoEscalarCV(valor v1,v2:vectorCV;
                              referencia p:real;
                              referencia error:booleano)
  { Si v1 y v2 tienen igual dimensión, guarda su producto escalar
    en p y el valor falso en error; si tienen distinta dimensión,
    guarda verdad en error.}

  algoritmo asignarVectorCV(referencia nuevo:vectorCV;
                            valor viejo:vectorCV)
  { Duplica la representación del vector viejo guardándolo en
    nuevo.}

  algoritmo liberarVectorCV(referencia v:vectorCV)
  { Libera la memoria dinámica accesible desde v, quedando v con
    valor indefinido. }
implementación
  tipo vectorCV = registro
                    dim:1..maxEntero;
                    comp:listaDeComp
                  freg
```

```
algoritmo leeVectorCV(referencia v:vectorCV)
variables c:componente; índice,indAnterior:0..maxEntero
principio
  escribir('Dimensión del vector: ');
  leerLínea(v.dim);
  escribirLínea('Componentes no nulas y en orden creciente
                 de índice (índice 0 para terminar)');
  escribir('Indice: ');
  leerLínea(índice);
  mientrasQue índice>v.dim hacer
    escribirLínea('El índice no puede ser mayor que ', v.dim);
    escribir('Indice: ');
    leerLínea(índice)
  fmq;
  creaVacía(v.comp);
  mientrasQue índice≠0 hacer
    c.ind:=índice;
    escribir('Valor: ');
    leerLínea(c.val);
    mientrasQue c.val=0.0 hacer
      escribirLínea('El valor debe ser no nulo');
      escribir('Valor: ');
      leerLínea(c.val)
    fmq;
    añadeDch(v.comp,c);
    indAnterior:=c.ind;
    escribir('Indice: ');
    leerLínea(índice);
    mientrasQue (índice≤indAnterior) and (índice≠0) or
                  (índice>v.dim) hacer
      escribirLínea('El índice debe ser mayor que ',
                    indAnterior, y menor o igual que ',
                    v.dim,' (0 para terminar)');
      escribir('Indice: ');
      leerLínea(índice)
    fmq
  fmq
fin

algoritmo escribeVectorCV(valor v:vectorCV)
constante compPorLínea = 5;
variables c:componente; i:1..maxEntero
principio
  escribir('Dimensión del vector: ',v.dim);
  si long(v.comp)=0
  entonces
    escribirLínea('Todas las componentes son nulas')
```

```
  sino
    escribirLínea('Componentes no nulas ("índice -> valor"):');
    para i:=1 hasta long(v.comp) hacer
      c:=observa(v.comp,i);
      escribir(c.ind,' -> ',c.val,' |');
      si (i mod compPorLínea = 0)
      entonces
        escribirLínea
      fsi
    fpara
  fsi
fin

algoritmo dimensión(valor v:vectorCV) devuelve 1..maxEntero
principio
  devuelve(v.dim)
fin

algoritmo observaComp(valor v:vectorCV; valor i:1..maxEntero)
          devuelve real
variables j,l:0..maxEntero; encontrada:booleano
principio
  j:=1;
  encontrada:=falso;
  l:=long(v.comp);
  mientrasQue (j≤l) and not encontrada hacer
    c:=observa(v.comp,j);
    j:=j+1;
    encontrada:=c.ind≥i
  fmq;
  si encontrada
  entonces
    si c.ind=i
    entonces
      devuelve(c.val)
    sino
      devuelve(0)
    fsi
  sino
    devuelve(0)
  fsi
fin

algoritmo sumaVectoresCVs(valor v1,v2:vectorCV;
                          referencia vs:vectorCV;
                          referencia error:booleano)
variables c,c1,c2:componente; i1,i2:1..maxEntero
```

```
  algoritmo copiarResto(valor v:vectorCV; valor i:1..maxEntero;
                        referencia vs:vectorCV)
  { Añade al vector vs las componentes de v cuyo número de
    orden en v.comp es mayor o igual que i. }
  variable j:1..maxEntero
  principio {de copiarResto}
    para j:=i hasta long(v.comp) hacer
      añadeDch(vs.comp,observa(v.comp,j))
    fpara
  fin
principio {de sumaVectoresCVs}
  si v1.dim≠v2.dim
  entonces
    error:=verdad
  sino
    error:=falso;
    vs.dim:=v1.dim;
    creaVacía(vs.comp);
    i1:=1;
    i2:=1;
    mientrasQue (i1≤long(v1.comp)) and (i2≤long(v2.comp)) hacer
      c1:=observa(v1.comp,i1);
      c2:=observa(v2.comp,i2);
      selección
        c1.ind<c2.ind: añadeDch(vs.comp,c1);
                       i1:=i1+1;
        c1.ind>c2.ind: añadeDch(vs.comp,c2);
                       i2:=i2+1;
        c1.ind=c2.ind: si c1.val+c2.val≠0
                       entonces
                         c.val:=c1.val+c2.val;
                         c.ind:=c1.ind;
                         añadeDch(vs.comp,c)
                       fsi;
                       i1:=i1+1;
                       i2:=i2+1
      fselección
    fmq;
    si i1≤long(v1.comp)
    entonces
      copiarResto(v1,i1,vs)
    sino
      copiarResto(v2,i2,vs)
    fsi
  fsi
fin
```

```
algoritmo escalarPorVectorCV(valor x:real; valor v:vectorCV;
                             referencia p:vectorCV)
variables i:1..maxEntero; c:componente
principio
  p.dim:=v.dim;
  creaVacía(p.comp);
  si x≠0
  entonces
    para i:=1 hasta long(v.comp) hacer
      c:=observa(v.comp,i);
      c.val:=x*c.val;
      añadeDch(p.comp,c)
    fpara
  fsi
fin

algoritmo productoEscalarCV(valor v1,v2:vectorCV;
                            referencia p:real;
                            referencia error:booleano)
variables i1,i2:1..maxEntero; c1,c2:componente
principio
  si v1.dim≠v2.dim
  entonces
    error:=verdad
  sino
    error:=falso;
    p:=0.0;
    i1:=1;
    i2:=1;
    mientrasQue (i1≤long(v1.comp)) and (i2≤long(v2.comp)) hacer
      c1:=observa(v1.comp,i1);
      c2:=observa(v2.comp,i2);
      selección
        c1.ind<c2.ind: i1:=i1+1;
        c1.ind>c2.ind: i2:=i2+1;
        c1.ind=c2.ind: p:=p+c1.val*c2.val;
                       i1:=i1+1;
                       i2:=i2+1
      fselección
    fmq
  fsi
fin

algoritmo asignarVectorCV(referencia nuevo:vectorCV;
                          valor viejo:vectorCV)
{ Se propone como ejercicio. }
```

```
  algoritmo liberarVectorCV(referencia v:vectorCV)
  { Se propone como ejercicio. }
fin
```

## 14.3. Matrices cuasi-vacías

Finalmente, una matriz de números reales puede representarse como un vector (de los predefinidos en nuestro lenguaje algorítmico) de listas homogéneas con acceso por posición. Limitaremos el número de filas de la matriz con una constante `maxNumFil`. La implementación es la siguiente:

```
módulo matricesCVs
importa listasDeComponentes
exporta
  constante maxNumFil {una constante entera positiva}
  tipo matrizCV {matrices de números reales con un n° de
                 filas menor o igual que maxNumFil}

  algoritmo leeMatrizCV(referencia m:matrizCV)
  {Lee desde teclado una matriz y la almacena en m.}

  algoritmo escribeMatrizCV(referencia[2] m:matrizCV)
  {Escribe en pantalla el valor de la matriz m.}

  algoritmo dimensiones(referencia m:matrizCV;
                        referencia fil,col:1..maxEntero)
  { Devuelve las dimensiones de m en fil y col. }

  algoritmo observaComp(referencia m:matrizCV;
                        valor i,j:1..maxEntero) devuelve real
  { Devuelve el valor de la componente (i,j) de m. }

  algoritmo sumaMatricesCVs(referencia m1,m2,ms:matrizCV;
                            referencia error:booleano)
  {Si m1 y m2 tiene iguales dimensiones, su suma se guarda en
   ms y el valor falso en error; si tienen distintas
   dimensiones, se guarda verdad en error.}

  algoritmo trasponeMatrizCV(referencia m,mt:matrizCV;
                             referencia error:booleano)
```

2 Por razones de eficiencia.

```
  {Guarda en mt la traspuesta de m y en error el valor falso
   si el número de columnas de m es menor o igual que
   maxNumFil. En caso contrario, error toma el valor verdad.}

  algoritmo multiplicaMatricesCVs(referencia m1,m2,mp:matrizCV;
                                  referencia error:booleano)
  {Guarda en mp la matriz producto de m1 y m2 y en error el valor
   falso si las dimensiones de m1 y m2 permiten realizar el pro-
   ducto. En caso contrario, error toma el valor verdad.}

  algoritmo asignarMatricesCV(referencia nueva,vieja:matrizCV)
  { Duplica la representación de la matriz vieja guardándola en
    nueva. }

  algoritmo liberarMatrizCV(referencia m:matrizCV)
  { Libera la memoria dinámica accesible desde m, quedando m
    con valor indefinido. }
implementación
  constante maxNumFil = 1000;
  tipo matrizCV = registro
                    numFil:1..maxNumFil;
                    numCol:1..maxEntero;
                    fila: vector[1..maxNumFil] de listaDeComp
                  freg

  algoritmo leeMatrizCV(referencia m:matrizCV)
  variable fil:1..maxNumFil; c:componente;
           índice,indAnterior:0..maxEntero
  principio
    escribir('N.° de filas de la matriz (no mayor que ',
             maxNumFil,'): ');
    leerLínea(m.numFil);
    escribir('N.° de columnas de la matriz: ');
    leerLínea(m.numCol);
    para fil:=1 hasta m.numFil hacer
      escribirLínea('Introducción del vector fila ',fil,':');
      escribirLínea('Componentes no nulas y en orden creciente de
                     índice de columna (índice 0 para
                     terminar)');
      escribir('Indice de columna: ');
      leerLínea(índice);
      mientrasQue índice>m.numCol hacer
        escribirLínea('El índice no puede ser mayor que ',
                      m.numCol);
        escribir('Indice de columna: ');
        leerLínea(índice)
      fmq;
```

```
    creaVacía(m.fila[fil]);
    mientrasQue índice≠0 hacer
      c.ind:=índice;
      escribir('Valor: ');
      leerLínea(c.val);
      mientrasQue c.val=0.0 hacer
        escribirLínea('El valor debe ser no nulo');
        escribir('Valor: ');
        leerLínea(c.val)
      fmq;
      añadeDch(m.fila[fil],c);
      indAnterior:=c.ind;
      escribir('Indice de columna: ');
      leerLínea(índice);
      mientrasQue (índice≤indAnterior) and (índice≠0) or
                   (índice>m.numCol) hacer
        escribirLínea('El índice debe ser mayor que ',
                      indAnterior,
                     ' y menor o igual que ',m.numCol,
                     '(0 para terminar)');
        escribir('Indice de columna: ');
        leerLínea(índice)
      fmq
    fmq
  fpara
fin

algoritmo escribeMatrizCV(referencia m:matrizCV)
constante compPorLínea = 5;
variable fil:1..maxNumFil; i:1..maxEntero; c:componente
principio
  escribirLínea('N° de filas de la matriz: ',m.numFil);
  escribirLínea('N° de columnas de la matriz: ',m.numCol);
  para fil:=1 hasta m.numFil hacer
    escribirLínea('Componentes no nulas del vector fila ',
                  fil, ("índice de columna -> valor"):');
    si long(m.fila[fil])=0
    entonces
      escribirLínea('Todas las componentes son nulas')
    sino
      para i:=1 hasta long(m.fila[fil]) hacer
        c:=observa(m.fila[fil],i);
        escribir(c.ind,' -> ',c.val,' |');
        si (i mod compPorLínea = 0)
        entonces
          escribirLínea
        fsi
```

```
      fpara
    fsi
  fpara
fin

algoritmo dimensiones(referencia m:matrizCV;
                      referencia fil,col:1..maxEntero)
{ Se propone como ejercicio. }

algoritmo observaComp(referencia m:matrizCV;
                      valor i,j:1..maxEntero) devuelve real
{ Se propone como ejercicio. }

algoritmo sumaMatricesCVs(referencia m1,m2,ms:matrizCV;
                          referencia error:booleano)
variables fil:1..maxNumFil; c,c1,c2:componente;
          i1,i2:1..maxEntero
  algoritmo copiarResto(valor l:listaDeComp;
                        valor i:1..maxEntero;
                        referencia ls:listaDeComp)
  {Añade a la lista ls las componentes de l cuyo número de
   orden es mayor o igual que i.}
  variable j:1..maxEntero
  principio {de copiarResto}
    para j:=i hasta long(l) hacer
      añadeDch(ls,observa(l,j))
    fpara
  fin
principio
  si (m1.numFil≠m2.numFil) or (m1.numCol≠m2.numCol)
  entonces
    error:=verdad
  sino
    error:=falso;
    ms.numFil:=m1.numFil;
    ms.numCol:=m1.numCol;
    para fil:=1 hasta ms.numFil hacer
      creaVacía(ms.fila[fil]);
      i1:=1;
      i2:=1;
      mientrasQue (i1≤long(m1.fila[fil]))
                  and (i2≤long(m2.fila[fil])) hacer
        c1:=observa(m1.fila[fil],i1);
        c2:=observa(m2.fila[fil],i2);
        selección
          c1.ind<c2.ind: añadeDch(ms.fila[fil],c1);
                         i1:=i1+1;
```

```
          c1.ind>c2.ind: añadeDch(ms.fila[fil],c2);
                         i2:=i2+1;
          c1.ind=c2.ind: si c1.val+c2.val≠0
                         entonces
                           c.val:=c1.val+c2.val;
                           c.ind:=c1.ind;
                           añadeDch(ms.fila[fil],c)
                         fsi;
                         i1:=i1+1;
                         i2:=i2+1
        fselección
      fmq;
      si i1≤long(m1.fila[fil])
      entonces
        copiarResto(m1.fila[fil],i1,ms.fila[fil])
      sino
        copiarResto(m2.fila[fil],i2,ms.fila[fil])
      fsi
    fpara
  fsi
fin

algoritmo trasponeMatrizCV(referencia m,mt:matrizCV;
                           referencia error:booleano)
variables fil,col,i:1..maxNumFil; c:componente
principio
  si m.numCol>maxNumFil
  entonces
    error:=verdad
  sino
    error:=falso;
    mt.numFil:=m.numCol;
    mt.numCol:=m.numFil;
    para fil:=1 hasta mt.numFil hacer
      creaVacía(mt.fila[fil])
    fpara;
    para fil:=1 hasta m.numFil hacer
      para i:=1 hasta long(m.fila[fil]) hacer
        c:=observa(m.fila[fil],i);
        col:=c.ind;
        c.ind:=fil;
        añadeDch(mt.fila[col],c)
      fpara
    fpara
  fsi
fin
```

```
  algoritmo multiplicaMatricesCVs(referencia m1,m2,mp:matrizCV;
                                  referencia error:booleano)
  variables m2t:matrizCV; nada:booleano; c,c1,c2:componente;
            fil,col,i1,i2:1..maxNumFil; p:real
  principio
    si (m1.numCol≠m2.numFil) or (m2.numCol>maxNumFil)
    entonces
      error:=verdad
    sino
      error:=falso;
      mp.numFil:=m1.numFil;
      mp.numCol:=m2.numCol;
      trasponeMatrizCV(m2,m2t,nada);
      para fil:=1 hasta mp.numFil hacer
        creaVacía(mp.fila[fil]);
        para col:=1 hasta mp.numCol hacer
          p:=0.0;
          i1:=1;
          i2:=1;
          mientrasQue (i1≤long(m1.fila[fil])) and
                      (i2≤long(m2.fila[col])) hacer
            c1:=observa(v1.comp,i1);
            c2:=observa(v2.comp,i2);
            selección
              c1.ind<c2.ind: i1:=i1+1;
              c1.ind>c2.ind: i2:=i2+1;
              c1.ind=c2.ind: p:=p+c1.val*c2.val;
                             i1:=i1+1;
                             i2:=i2+1
            fselección
          fmq;
          si p≠0
          entonces
            c.ind:=col;
            c.val:=p;
            añadeDch(mp.fila[fil],c)
          fsi
        fpara
      fpara
    fsi

  fin
  algoritmo asignarMatricesCV(referencia nueva,vieja:matrizCV)
  { Se propone como ejercicio. }

  algoritmo liberarMatrizCV(referencia m:matrizCV)
  { Se propone como ejercicio. }
fin
```