PRÁCTICA 1: LISP

1. Vector más cercano a un vector dado:

1.1. Norma p de un vector:

PSEUDOCÓDIGO

Entrada: x (vector del que se calcula la norma)
p (orden de la norma)
Salida: n (norma p de x)

Procesamiento:
Si x es vacio,
evalúa a 0
en caso contrario
evalúa a (|primer-elemento(x)|^p + normap(siguientes-elementos(x))^(1/p)

A) Usando recursión

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;FUNCION AUXLIAR
;;;lp-rec-aux
;;;Funcion que realiza el sumatorio de los elementos del vector
;;;elevados al exponente p
;;;
;;;INPUT : x: vector, en forma de lista
;;;p: orden de la norma que se quiere calcular
;;;
;;;OUTPUT:sumatorio interior de la norma
(defun lp-rec-aux(x p)
  (if (null x)
      0
    (+ (expt (abs (first x)) p) (lp-rec-aux (rest x) p))))

;;;EJEMPLOS
;;;(lp-rec-aux nil 2); -> 0 caso especial
;;;(lp-rec-aux '(3 4) 2);-> 25 caso general

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lp-rec (x p)
;;; Calcula la norma Lp de un vector de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; p: orden de la norma que se quiere calcular
;;;
;;; OUTPUT: norma Lp de x
;;;
(defun lp-rec (x p)
  (expt (lp-rec-aux x p) (/ 1 p)))

;;;EJEMPLOS
;;;(lp-rec nil 2); -> 0 caso especial
;;;(lp-rec '(3 4) 2);-> 5.0 caso general
```

B) Usando mapcar

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lp-mapcar (x p)
;;; Calcula la norma Lp de un vector usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; p: orden de la norma que se quiere calcular
;;;
;;; OUTPUT: norma Lp de x
;;;
(defun lp-mapcar (x p)
  (expt (apply #'+ (mapcar #'(lambda (n) (expt (abs n) p)) x)) (/ 1 p)))

;;;EJEMPLOS
;;;(lp-mapcar nil 2); -> 0 caso especial
;;;(lp-mapcar '(3 4) 2);-> 5.0 caso general
;;;
;;;COMENTARIOS
;;;al hacer apply #'+ con un vector es nil da 0
```

COMENTARIOS

El código con mapcar es mucho más claro y sencillo que el de recursión, aunque el último es más intuitivo puesto que se trata de un sumatorio. Asimismo, cuando usamos mapcar, Lisp se encarga de comprobar los casos en los que la lista sea vacía y por tanto ahorramos comprobaciones.
En la función recursiva hemos necesitado una función auxiliar para calcular el sumatorio. En la función principal ya podemos calcular la raíz una vez hemos llamado a la auxiliar.

1.2. Norma euclídea y norma 1

Una vez definidas las dos funciones anteriores, tan solo tenemos que llamarlas utilizando el valor de p que corresponda.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l2-rec (x)
;;; Calcula la norma L2 de un vector de forma recursiva
;;;
;;; INPUT: x: vector
;;;
;;; OUTPUT: norma L2 de x
;;;
(defun l2-rec (x)
  (lp-rec x 2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l2-mapcar (x)
;;; Calcula la norma L2 de un vector usando mapcar
;;;
;;; INPUT: x: vector
;;;
;;; OUTPUT: norma L2 de x
;;;
(defun l2-mapcar (x)
  (lp-mapcar x 2))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l1-rec (x)
;;; Calcula la norma L1 de un vector de forma recursiva
;;;
;;; INPUT: x: vector
;;;
;;; OUTPUT: norma L1 de x
;;;
(defun l1-rec (x)
  (lp-rec x 1))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l1-mapcar (x)
;;; Calcula la norma L1 de un vector usando mapcar
;;;
;;; INPUT: x: vector
;;;
;;; OUTPUT: norma L1 de x
;;;
(defun l1-mapcar (x)
  (lp-mapcar x 1))
```

COMENTARIOS

Para probar la norma infinito podemos introducir vectores con valores negativos y ver que el valor absoluto se aplica correctamente. También podemos introducir vectores distintos cuya norma euclídea es igual pero la norma infinito cambia, por ejemplo, (1 1) tiene norma euclídea $\sqrt{2}$ y (1 0), 1, pero ambos tienen norma infinito 1.

1.3. Función nearest

PSEUDOCÓDIGO

Entrada: lst-vectors(lista de vectores a comparar)
vector (vector referencia contra el que se compara)
fn-dist: referencia a función para medir distancias
Salida: vector de entre los de lst-vectors más cercano al de referencia

Procesamiento:
si no(vacío(vector) o vacío(lst-vectors))
    lista-distancias= aplicar (calcular-distancia(lst-vectores))
    n=posición(minimo(lista-distancias))
    devolver (lst-vectors(n))

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;FUNCION AUXILIAR
;;;eval-dist
;;;evalua la distancia entre el vector ref y el vectro vec mediente
;;;la funcion fn-dist
;;;INPUT : ref ; primer vector
;;;vec: segundo vector
;;;fn-dist: funcion con la que se consigue la distancia entre los vectores
;;;
;;;OUTPUT: distancia entre los vectores
(defun eval-dist(ref vec fn-dist)
  (funcall fn-dist (mapcar #'- ref vec)))

;;;EJEMPLOS
;;;(eval-dist '()'(1 0) #'l2-mapcar)-> 0 caso particular
;;;(eval-dist '(1 0)'(1 0) #'l2-mapcar)-> 0 caso general
;;;(eval-dist '(0 0)'(1 0) #'l2-mapcar)-> 1 caso general

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; nearest (lst-vectors vector fn-dist)
;;; Calcula de una lista de vectores el vector más cercano a uno dado,
;;; usando la función de distancia especificada
;;;
;;; INPUT: lst-vectors: lista de vectores para los que calcular la distancia
;;; vector: vector referencia, representado como una lista
;;; fn-dist: referencia a función para medir distancias
;;;
;;; OUTPUT: vector de entre los de lst-vectors más cercano al de referencia
;;;
(defun nearest (lst-vectors vector fn-dist)
  (unless (null vector)
  (let ((distancias (mapcar #'(lambda(x) (eval-dist vector x fn-dist))
                      lst-vectors)))
    (first (nthcdr (position (apply #'min distancias)
                             distancias) lst-vectors)))))

;;;EJEMPLOS
;;;(nearest '((2 3) (1 2) ()) '(0 0) #'l2-mapcar)-> NIL caso particular
;;;(nearest '(()) '(0 0) #'l2-mapcar)-> NIL caso particular
;;;(nearest '((2 3)(1 2) (5 7)) '() #'l2-mapcar)-> NIL caso particular
;;;(nearest '((2 3) (1 2) (0 0)) '(0 0) #'l2-mapcar)-> (0 0) caso general
;;;
;;;COMENTARIOS
;;;la funcion espera una lista de listas y esto no se comprueba,
;;;se sigue el principio RTFM
```

COMENTARIOS

Se ha utilizado mapcar para la implementación debido a que nos parecía más sencillo que una opción recursiva y por posibles ahorros de memoria (como al final se confirma con las pruebas realizadas a continuación)

Hemos necesitado una función auxiliar que compara un único vector contra el vector referencia.

1.4. Pruebas función nearest con diferentes normas

```
CG-USER(114): (time (nearest list-vectors v #'l2-rec))
; cpu time (non-gc) 16 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  16 msec user, 0 msec system
; real time  3 msec
; space allocation:
;  9,970 cons cells, 354,416 other bytes, 0 static bytes
(1 0 0 0 0 0 0)
CG-USER(115): (time (nearest list-vectors v #'l2-mapcar))
; cpu time (non-gc) 15 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  15 msec user, 0 msec system
; real time  2 msec
; space allocation:
;  5,715 cons cells, 139,000 other bytes, 0 static bytes
(1 0 0 0 0 0 0)
CG-USER(116): (time (nearest list-vectors v #'l1-rec))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  2 msec
; space allocation:
;  9,970 cons cells, 350,248 other bytes, 0 static bytes
(1 0 0 0 0 0 0)
CG-USER(117): (time (nearest list-vectors v #'l1-mapcar))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  1 msec
; space allocation:
;  5,715 cons cells, 134,832 other bytes, 0 static bytes
(1 0 0 0 0 0 0)
CG-USER(118): (time (nearest list-vectors v #'linf-rec))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  2 msec
; space allocation:
;  9,992 cons cells, 311,768 other bytes, 0 static bytes
(1 1 1 1 1 1 1 1)
CG-USER(119): (time (nearest list-vectors v #'linf-mapcar))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  1 msec
; space allocation:
;  2,352 cons cells, 62,680 other bytes, 0 static bytes
(1 1 1 1 1 1 1 1)
```

COMENTARIOS

Hemos medido los tiempos con veinticinco vectores de dimensión nueve y vemos la distancia al origen. Observamos que la norma euclídea es la más costosa, ya que se ha

obtenido un tiempo de 3 ms. Asimismo, en general el mapcar es más eficiente que la recursión, tanto temporal como espacialmente (podemos ver como el número de bloques cons utilizados en todas las funciones recursivas es casi 10.000 mientras que en el mapcar se reduce aproximadamente a la mitad).

2. Ceros de una función

2.1. Función : secante

PSEUDOCÓDIGO

Siguiendo el pseudocódigo del enunciado se ha implementado el código que se presenta más abajo.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; secante (f tol-abs max-iter par-semillas)
;;; Estima el cero de una función mediante el método de la secante
;;;
;;; INPUT: f: función cuyo cero se desea encontrar
;;; tol-abs: tolerancia para convergencia
;;; max-iter: máximo número de iteraciones
;;; par-semillas: estimaciones iniciales del cero (x0 x1)
;;;
;;; OUTPUT: estimación del cero de f, o NIL si no converge
;;;
(defun  secante (f tol-abs max-iter par-semillas)
  (unless (or (< max-iter 0) (null par-semillas))
    (let* ((x0 (first par-semillas)) (x1 (second par-semillas))
           (fx0 (funcall f x0)) (fx1 (funcall f x1)))
      (if (< (abs (apply #'- par-semillas)) tol-abs)
          x1
        (secante f tol-abs (- max-iter 1)
                 (list x1 (- x1 (* fx1 (/ (- x1 x0)
                                          (- fx1 fx0)))))))))))
;;;EJEMPLOS
;;;caso de no convergencia
;;;(setf funcion (lambda (x) (+ (sin x) 1.5)))
;;;(secante funcion tol iters semillas)->NIL
;;;
;;;En el caso general nos remitimos al ejemplo proporcionado en el enunciado
```

COMENTARIOS

Sobre la implementación elegida destacamos que en cada llamada recursiva a la función creamos una nueva lista donde X1 pasa a ser X0 y el nuevo X1 se calcula mediante la fórmula que se nos ha proporcionado.

Hemos probado funciones que no tienen ceros para ver que ocurría. Una de ellas es sin(x)+1.5 que cumple el máximo de iteraciones y acaba con salida NIL. Otra de ellas es x^2+1 que no llega al máximo de iteraciones puesto que las semillas crecen rápidamente y se desborda la memoria.

2.2. Función : un-cero-secante

PSEUDOCÓDIGO

Entrada: f: función
Tol-abs: tolerancia
max-iter: máximo iteraciones
pares-semillas: lista-pares-semillas
Salida: El primer cero encontrado

Procesamiento:
si no (secante (primer-elemento(pares-semillas)))
entonces un-cero-secante(siguientes-elementos(pares-semillas))

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; un-cero-secante (f tol-abs max-iter pares-semillas)
;;; Prueba con distintos pares de semillas iniciales hasta que
;;; la secante converge
;;;
;;; INPUT: f: función de la que se desea encontrar un cero
;;; tol-abs: tolerancia para convergencia
;;; max-iter: máximo número de iteraciones
;;; pares-semillas: pares de semillas con las que invocar a secante
;;;
;;; OUTPUT: el primer cero de f que se encuentre, o NIL si se diverge
;;; para todos los pares de semillas
;;;
(defun un-cero-secante (f tol-abs max-iter pares-semillas)
  (unless (null pares-semillas)
    (let ((par1 (secante f tol-abs max-iter (first pares-semillas))))
      (if par1
          par1
        (un-cero-secante f tol-abs max-iter (rest pares-semillas))))))

;;;EJEMPLOS
;;;caso de no convergencia
;;;(setf funcion (lambda (x) (+ (sin x) 1.5)))
;;;(un-cero-secante funcion tol iters semillas)->NIL
;;;
;;;En el caso general nos remitimos al ejemplo proporcionado en el enunciado
;;;
;;;COMENTARIOS
;;;
;;;la funcion espera una lista de listas y esto no se comprueba,
;;;se sigue el principio RTFM
```

COMENTARIOS

La función solo admite una lista de listas, en caso contrario se produce un error.
De nuevo hemos probado el caso de no convergencia con sin(x)+1.5 cumple el máximo de iteraciones con cada par de semillas.

2.3. Función: todos-ceros-secante

PSEUDOCÓDIGO

Entrada: f: función de la que se desea encontrar un cero
tol-abs: tolerancia para convergencia
max-iter: máximo número de iteraciones
pares-semillas: pares de semillas con las que invocar a secante
Salida: Lista con todos los ceros encontrados a partir de las semillas

Procesamiento:
si no (vacío (pares-semillas)
secante(primer-elemento(pares-semillas))
todos-ceros-secante(siguientes-elementos(pares-semillas))

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; todos-ceros-secante (f tol-abs max-iter pares-semillas)
;;; Prueba con distintas pares de semillas iniciales y devuelve
;;; las raíces encontradas por la secante para dichos pares
;;;
;;; INPUT: f: función de la que se desea encontrar un cero
;;; tol-abs: tolerancia para convergencia
;;; max-iter: máximo número de iteraciones
;;; pares-semillas: pares de semillas con las que invocar a secante
;;;
;;; OUTPUT: todas las raíces que se encuentren, o NIL si se diverge
;;; para todos los pares de semillas
;;;
(defun todos-ceros-secante (f tol-abs max-iter pares-semillas)
  (unless (null pares-semillas)
    (cons (secante f tol-abs max-iter (first pares-semillas))
          (todos-ceros-secante f tol-abs max-iter
                               (rest pares-semillas)))))

;;;EJEMPLOS
;;;caso de no convergencia
;;;(setf funcion (lambda (x) (+ (sin x) 1.5)))
;;;(todos-ceros-secante funcion tol iters semillas)->(NIL)
;;;
;;;En el caso general nos remitimos al ejemplo proporcionado en el enunciado
```

COMENTARIOS

Se espera que se conozcan las semillas previamente, ya que la función no calcula las semillas ni los intervalos donde se encuentran las raíces, simplemente las evalúa y devuelve la lista de resultados.

3. Combinación de listas

3.1. Combinación elemento-lista

PSEUDOCÓDIGO

Entrada: elt: elemento
lst: lista
Salida: lista que combina el elemento con la lista introducida

Procesamiento:
Si no (null(lst))
par(lista(elt, primer-elemento(lst)), combine-elt-lst(siguientes-elementos(lst))

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;; Combina un elemento con una lista devolviendo una lista que
;;; contiene listas con el formato ((elt first)(elt second)...)
;;; donde first y second son respectivamente los 2 primeros elementos
;;; de la lista
;;;
;;; INPUT: elt: elemento
;;; lst: lista
;;;
;;; OUTPUT: lista que combina el elemto con la lista introducida
(defun combine-elt-lst (elt lst)
  (unless (null lst)
    (cons ( list elt (first lst)) (combine-elt-lst elt (rest lst)))))

;;;EJEMPLOS
;;;(combine-elt-lst 'a '())-> NIL caso particular
;;;(combine-elt-lst 5 '(a b))-> ((5 A)(5 B)) caso general
```

COMENTARIOS

Debido a que no se especifica en el enunciado se admite el caso en el que el elemento es la lista vacía.

3.2. Combinación lista-lista

PSEUDOCÓDIGO

Entrada: lst1: lista1
lst2: lista2
Salida: lista que realiza el producto vectorial de las dos listas entrantes

Procesamiento:
sino(null(lst1) o null(lst2))
une((combine-elt-lst(primer-elemento(lst1), lst2),
combine-elt-lst(siguientes-elementos(lst1),lst2))

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;; Realiza el producto cartesiano de las dos listas
;;;
;;; INPUT: lst1: lista1
;;; lst2: lista2
;;; OUTPUT: lista con las tuplas con las combinaciones producidas por el producto cartesiano
(defun combine-lst-lst (lst1 lst2)
  (unless (or(null lst1) (null lst2))
    (append (combine-elt-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2))))

;;;EJEMPLOS
;;;(combine-lst-lst '(1 2)'())-> NIL caso particular
;;;(combine-lst-lst '()'(1 2))-> NIL caso particular
;;;(combine-lst-lst '(1 2)'( a b))->((1 A) (1 B) (2 A) (2 B)) caso general
```

3.3. Combinación lista de listas

PSEUDOCÓDIGO

Entrada: lstolsts: lista de listas con las que realizaremos las disposiciones

Salida: lista con las diferentes disposiciones producidas por la función

Procesamiento:
(si formato-okp(lstolsts))
l1=primer-elemento(lstolsts)
lr= siguientes-elementos(lstolsts)
primero= primer-elemento(l1)
combinar-elemento-lista(primero, lr)
combinar-lst-of-lsts(lista(siguientes-elementos(l1),lr))

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;FUNCION AUXLIAR
;;;check-lists-okp(listoflsts)
;;;Funcion que comprueba que la list-of-lsts no contiene listas vacias
;;;
;;;INPUT : listoflsts: lista de listas
;;;
;;;OUTPUT:T si ok, NIL si alguna lista vacia
(defun check-lists-okp(listoflsts)
  (if (null listoflsts)
      T
    (unless(null (first listoflsts))
      (check-lists-okp (rest listoflsts)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts (lstolsts)
;;; Representa todas las posibles disposiciones de elementos
;;; pertenecientes a
;;; N listas de forma que en cada disposición aparezca únicamente
;;; un elemento de cada lista
;;;
;;; INPUT: lstolsts: lista de listas con las que realizaremos las disposiciones
;;;
;;; OUTPUT: lista con las diferentes disposiciones producidas por la funcion
(defun combine-list-of-lsts (lstolsts)
  (unless (null (check-lists-okp lstolsts))
    (let ((first-lst (first lstolsts)) (other-lsts (rest lstolsts)))
      (if (null first-lst)
          '(NIL)
        (mapcan #'(lambda (x)
                    (mapcar #'(lambda (y) (cons x y))
                      (combine-list-of-lsts other-lsts)))
          first-lst)))))

;;;EJEMPLOS
;;;(combine-list-of-lsts '((1 2)(+)(A B C)))-> NIL caso particular
;;;(combine-list-of-lsts '(())-> (NIL) caso particular
;;;(combine-list-of-lsts '((1 2))-> ((1) (2)) caso particular
;;;(combine-list-of-lsts '((1 2)(+)(A B C)))->
;;;                        ((1 + A) (1 + B) (1 + C) (2 + A) (2 + B) (2 + C)) caso general
```