# Tema 9

# Exámenes resueltos de años anteriores

## Criterios de evaluación

Cada uno de los ejercicios que componen esta Prueba de Evaluación Continua (PEC) de la parte de Maxima se evaluará de 0 a 10 puntos de acuerdo a los siguientes criterios de evaluación, aplicables a todas las convocatorias:

- El código aportado realiza correctamente las tareas que se pedían en el enunciado, cálculos simbólicos y/o numéricos, representaciones gráficas, etc., (sin errores sintácticos): **5 puntos**
- El código está bien estructurado, se entiende claramente lo que se hace en cada parte del mismo, la estructura es lógica y está ordenada: **2 puntos**
- El código realiza las tareas que se piden de manera eficiente: **1.5 puntos**
- El código está documentado con comentarios que facilitan entender qué es lo que se está haciendo en cada parte del mismo, incluyendo descripción del *input* y *output* y la finalidad del código: **1.5 puntos**

La calificación final de esta parte será la media aritmética de las calificaciones obtenidas en todos los ejercicios que forman la PEC, siempre y cuando se haya obtenido una calificación mínima de 5 puntos en todos ellos. Si uno (o más) de los ejercicios propuestos no alcanzan la calificación mínima de 5 puntos la calificación global de la PEC será *suspenso*, y no se calculará la media.

**Muy Importante:**

- Es muy importante darse cuenta de que lo que se pide en cada uno de estos ejercicios es la programación de una *función*, no la resolución de un problema concreto.

- La solución a estas PECs que debe presentar se reduce a un archivo de texto plano que contenga el código en Maxima de las funciones pedidas en la PEC, de tal manera que podamos cargar dicho código desde una sesión de Maxima para verificar su funcionamiento.
- En la medida de lo posible ajústese al input y output especificado en cada ejercicio. Si lo considera preciso puede introducir pequeñas modificaciones, en ese caso introduzca una breve frase explicando las modificaciones introducidas y su justificación.
- Muy importante, evite la definición de *variables globales* fuera de estas funciones.
- También le recomendamos que, antes de realizar estos ejercicios, revise la colección de problemas resueltos que puede encontrar en la página de la asignatura, así como las soluciones de las pruebas evaluables anteriores.

## 9.1. Curso 2010-2011, convocatoria de junio

### Ejercicio 1

Escriba una función en Maxima que resuelva numéricamente sistemas de $n$ ecuaciones diferenciales ordinarias de orden 1 por medio de la función `rk` y que posteriormente haga un ajuste por medio de splines cúbicos de la solución obtenida:

*input*:
- Lista de ecuaciones de orden 1 a resolver
- Lista de variables dependientes
- Lista de condiciones iniciales
- Variable independiente y valores mínimo y máximo de la variable independiente
- Parámetro de *paso* a emplear en el algoritmo de Runge-Kutta.

*output*:
- Matriz con los resultados numéricos obtenidos por medio del comando `rk`.
- Lista de funciones con los splines cúbicos que ajustan la solución.

### Ejercicio 2

Escriba una función que realice la representación gráfica de los resultados obtenidos por medio de la función definida en el primer ejercicio, incluyendo la solu-

ción numérica de la ecuación (por medio de puntos) y el ajuste por splines cúbicos (líneas contínuas).

## Ejercicio 3

Dado el problema de condiciones iniciales de orden 1

$$\frac{dx}{dt} = f(x,t), \qquad x(t=0) = x_0$$

escriba una función en Maxima que proporcione una aproximación a la solución de esta ecuación por medio del desarrollo en serie de Taylor de $x(t)$, centrado en $t = 0$, hasta orden $n = 2$, dado por:

$$x(t) \simeq \sum_{j=0}^{n} \frac{1}{j!} \left.\frac{d^j x}{dt^j}\right|_{t=0} t^j$$

Datos: El valor de $x(t)$ en $t = 0$ está dado por la condición inicial del problema, el valor de la primera derivada se obtiene sustituyendo en la ecuación

$$\left.\frac{dx}{dt}\right|_{t=0} = f(x_0, 0)$$

el de la segunda derivada se obtiene derivando en la ecuación de partida una vez

$$\frac{d^2x}{dt^2} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial t}$$

y posteriormente sustituyendo los valores $t = 0$, $x = x_0$, y el valor calculado previamente para $dx/dt$ en $t = 0$.

*input*: $f(x,t)$, $x_0$.

*output*: La aproximación a $x(t)$ mencionada.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
    junio de 2011
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2010-2011
```

```
        En este archivo de texto plano se incluye el código en Maxima de
           las funciones
        pedidas en la prueba evaluable.

        Para escribir este archivo se ha empleado el editor de textos "
           KWrite" en un PC
        con Linux Fedora 14

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2011-J-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2011-J-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";
*/

/*      EJERCICIO 1

        Escriba una función en Maxima que resuelva numéricamente sistemas
           de n ecuaciones
        diferenciales ordinarias de orden 1 por medio de la función rk y
           que posteriormente
        haga un ajuste por medio de splines cúbicos de la solución
           obtenida:

        input:
                Lista de ecuaciones de orden 1 a resolver
                Lista de variables dependientes
                Lista de condiciones iniciales
                Variable independiente y valores mínimo y máximo de la
                   variable independiente
                Parámetro de paso a emplear en el algoritmo de Runge-Kutta.
        output:
                Matriz con los resultados numéricos obtenidos por medio del
                    comando rk
                Lista de funciones con los splines cúbicos que ajustan la
                   solución
*/

/*      SOLUCION 1

        La solución al ejercicio 1 está implementada en la "funcion1",
           cuyo código se incluye
        a continuación.
```

```
El input y output de esta función es el especificado en el
   enunciado del ejercicio

Notación empleada en el input:

       ecuaciones     =      Lista de ecuaciones de orden 1 a
          resolver
       vardep         =      Lista de variables dependientes
       conini         =      Lista de condiciones iniciales
       dominiovarindep =     Variable independiente y valores mí
          nimo y máximo de
                              la variable independiente
       paso           =      Parámetro de paso a emplear en el
          algoritmo de Runge-Kutta.

Notación empleada en el output:

       solnum         =      Matriz con los resultados numéricos
          obtenidos por medio
                              del comando rk
       solspline      =      Lista de funciones con los splines cú
          bicos que ajustan
                              la solución

Observaciones:

A.
       El enunciado no indica nada acerca de la forma en la que
          están escritas las n EDOs
       de orden 1 que debemos resolver. En principio éstas podrían
            estar escritas de muchas
       formas totalmente equivalentes pero ligeramente distintas,
          p. ej. la forma estándar sería

              diff(x1(t), t) = f1(...)
              diff(x2(t), t) = f2(...)
              ...
              diff(xn(t), t) = fn(...)

       pero otra forma igualmente valida podría ser

              diff(x1(t), t) - f1(...) = 0
              diff(x2(t), t) - f2(...) = 0
              ...
              diff(xn(t), t) - fn(...) = 0

       sin embargo lo que tenemos que pasarle al rk es sólamente
          el conjunto de funciones
```

```
                f1, ..., fn, es decir, el lado derecho de las EDOs escritas
                     en la forma que hemos
                denominado como "estándar" más arriba (ver soluciones tema
                   2.10).

                Dado que el enunciado no dice nada a este respecto vamos a
                   suponer que el input
                suministrado es justamente lo que la función rk del Maxima
                   nos pide, el conjunto
                de funciones f1, ..., fn que nos dan las derivadas de las
                   variables dependientes
                x_i en función de las x_i y la variable independiente.

        B.
                Para definir esta función vamos a emplear el comando "block
                   ", cuya sintaxis y
                funcionamiento se ha visto en el tema 2.5 y siguientes.
*/

funcion1(ecuaciones, vardep, conini, dominiovarindep, paso) := block( [
   aux, solnum, solspline],

        /*
                Definimos la variable local auxiliar "aux" como la lista:

                        [variable independiente, valor inicial, valor final,
                            paso]

                para ello empleamos la función append (ver soluciones tema
                   2.7) para añadir
                a dominiovarindep el paso de integración, que es el
                   elemento que faltaba

                Por cierto, si no se conoce la función append, otra forma
                   de obtener este
                mismo resultado podría haber sido directamente:

                aux : [dominiovarindep[1], dominiovarindep[2],
                   dominiovarindep[3], paso]

        */

        aux : append(dominiovarindep, [paso]),

        /*      Resolvemos el sistema con la función rk    */

        solnum : rk(ecuaciones, vardep, conini, aux ),
```

```
/*      La solución numérica que nos proporciona rk es una lista de
    listas de valores.
       El enunciado nos pide el output de rk con formato matricial
          , para ello convertimos
       esta lista en una matriz (ver soluciones tema 2.10)
*/

solnum : apply(matrix, solnum),

/*      Cargamos el paquete de interpolacines "interpol" para
   realizar los ajustes
       por splines cúbicos pedidos (ver soluciones tema 2.9)
*/

load(interpol),

/*
       Realizamos el ajuste por splines cúbicos de la primera
          variable dependiente
       para ello asignamos a la variable axiliar "aux" el conjunto
           de valores de la
       variable independiente (primera columna de solnum) y de la
          primera de las
       variables dependientes (segunda columna de solnum)

       Concretamente lo que hace la siguiente instruccción es lo
          siguiente

       transpose(solnum)[1] es la lista de valores de la var.
          independiente = t

       transpose(solnum)[2] es la lista de valores de la primera
          var. dependiente = x1

       [transpose(solnum)[1], transpose(solnum)[2]] es una lista
          cuyo primer elemento

       es [t_1, t_2, ..., t_k] y cuyo segundo elemento es [x1_1,
          x1_2, ..., x1_k]

       siendo k el número de puntos generados por rk en la solució
          n numérica del sistema

       convertimos esto en una matrix por medio de apply(matrix,
          ...)

       finalmente al cacular la traspuesta obtenemos una matriz
          cuyos elementos son
```

```
        ( (t_1, x1_1), (t_2, x1_2), (t_3, x1_3), ... , (t_k, x1_k)
           )

        que es lo que tenemos que pasarle a la función cspline para
            que calcule la aproximación
        por splines cúbicos de x como función de t
*/

aux : transpose( apply(matrix, [transpose(solnum)[1], transpose(
   solnum)[2]] ) ),

/*      a continuación realizamos el ajuste y lo guardamos como el
   primer elemento
        de la lista solspline
*/

solspline : [ cspline( aux ) ],

/*      La función pedida debe funcionar para un número arbitrario
   "n" de ecuaciones
        por tanto repetimos esta operación por medio de un bucle
           para todas las variables
        dependientes restantes, desde x_2 (cuyos valores están en
           la columna 3 de solnum)
        hasta x_n (con valores en la columna n+1 de solnum), donde
           n está dado por el
        número de ecuaciones, es decir, el número de elementos del
           input "ecuaciones"
        (o alternativamente el número de elementos en vardep o
           conini, que obviamente
        debe coincidir con el número de ecuaciones).
        A medida que vamos calculando los ajustes para las
           restantes variables dependientes
        los vamos añadiendo a la lista solspline mediante la funció
           n append.
        Esta misma estrategia se ha empleado en la función para
           aplicar cambios de base
        (ver soluciones tema 2.5)
*/

for i : 3 thru ( 1 + length(vardep) ) step 1 do (

        aux : transpose( apply(matrix, [transpose(solnum)[1],
           transpose(solnum)[i]] ) ),

        solspline : append( solspline, [ cspline(aux) ] )
```

```
        ),

        /*      finalmente la función devuelve una lista cuyo primer
           elemento es la solución
                numérica solnum y cuyo segundo elemento es la lista de
                   aproximaciones por splines
                cúbicos solspline
         */

        [solnum, solspline]

)$


/*      EJERCICIO 2

        Escriba una función que realice la representación gráfica de los
           resultados obtenidos por medio
        de la función definida en el primer ejercicio, incluyendo la
           solución numérica de la ecuación
        (por medio de puntos) y el ajuste por splines cúbicos (líneas cont
           ínuas).
*/

/*      SOLUCION 2

        La solución al ejercicio 2 está implementada en la "funcion2",
           cuyo código se incluye
        a continuación.

        Notación empleada en el input:

                resultadosanteriores = output de la función del ejercicio
                   anterior

        Observaciones:

        A.
                El enunciado no indica nada sobre el rango de valores en
                   que se debe mostrar la gráfica
                de modo que tomaremos para el rango de valores de la
                   valores de la variable independiente
                todo el dominio en donde se ha realizado la integración num
                   érica del sistema, y para las
                variables dependientes, dado que no se especifica nada,
                   dejaremos que el wxMaxima determine
                el rango de valores de manera automática
        B.
```

```
            Igual que antes, para definir esta función vamos a emplear
               el comando "block", cuya
            sintaxis y funcionamiento se ha visto en el tema 2.5 y
               siguientes.
*/

funcion2(resultadosanteriores) := block( [puntos, splines, format, aux],

       /*     Definimos la variable local "puntos" como la traspuesta de
              la matriz de puntos generada por
              rk al resolver el sistema del ejercicio anterior
       */

       puntos : transpose(resultadosanteriores[1]),

       /*     El motivo de definir puntos como la traspuesta de
              resultadosanteriores[1] es que de esta
              forma puntos[1] nos da la lista de valores de la variable
                 independiente,
               puntos[2] la lista de valores de la primera variable
                  dependiente
               puntos[3] la lista de valores de la segunda variable
                  dependiente y así sucesivamente
               hasta completar las n variables dependientes
       */

       /*     Definimos la variable local "splines" como la lista de
              splines cúbicos generada en el
              ejercicio anterior
       */

       splines : resultadosanteriores[2],

       /*     Para la función que se pide en este problema tenemos que
              emplear la función plot2d del
              wxMaxima, de acuerdo a la sintaxis de esta función (ver
                 soluciones a los ejercicios propuestos
              en el curso o ayuda del wxMaxima) tenemos que hacer los
                 siguiente

              plot2d( lista de funciones a representar, dominio, formato
                 )

              Veamos por partes cada uno de estos ingredientes:

              La "lista de funciones a representar" debe tener todos los
                 conjuntos puntos discretos que
              queremos representar de acuerdo a la siguiente sintaxis
```

```
                [discrete, puntos[1], puntos[2]]
                [discrete, puntos[1], puntos[3]]
                [discrete, puntos[1], puntos[4]]
                ...
                [discrete, puntos[1], puntos[1+n]]

                (recordar que puntos[1] es la lista de valores de la
                    variable independiente,
                mientras que puntos[i] con i entre 2 y 1+n es lista
                   de valores de cada
                variable dependiente)

        y todas las funciones splines[1], splines[2], ..., splines[
           n].

        Por otra parte en formato tenemos que introducir la lista
           de especificaciones de formato
        que queremos que emplee el plot2d para haer las gráficas.
           El enunciado pide que se representen
        los puntos discretos generados por rk como puntos, y las
           aproximaciones por splines cúbicos
        como líneas, por tanto en formato tenemos que introducir la
            lista de especificaciones
        siguiente

                [style, points, ..., points, lines, ..., lines]

        donde las instrucciones "points" y "lines" se repiten "n"
           veces, siendo n el número
        de variables dependientes (dado por length(splines)).

        Esta función debe ser válida para un número arbitrario de
           variables, que en principio no
        conocemos. Por tanto, para definir estas listas de
           instruciones vamos a emplear la
        estrategia que ya hemos empleado antes:
                i. Definimos estas listas asignaándoles el valor de
                   su primer elemento
                ii. Vamos añadiendo elementos hasta completar todas
                   las variables por medio de
                   un bucle
*/

/*      asignamos a la variable local "format" el primer elemento
   de la lista de intrucciones
        de formato
*/
```

```
format : [style, points],

/*     asignamos a la variable local aux el primer elemento de la
   lista de puntos a representar */

aux : [ [discrete, puntos[1], puntos[2]] ],

/*     en este bucle vamos añadiendo a "aux" los puntos
   correspondientes a las variables
       dependientes restantes, y añadimos a la lista "format" una
          instrucción points por cada
       nueva variable
*/

for i : 3 thru length(puntos) step 1 do (

       aux : append(aux, [ [discrete, puntos[1], puntos[i]] ]),

       format : append(format, [points])

),

/*     Ya tenemos todos los puntos discretos que queremos
   representar en la gráfica, ahora,
       mediante un bucle similar al anterior vamos añadiendo todos
           los splines, y al mismo
       tiempo vamos añadiendo en la variable local "format" una
          instrucción "lines" por cada
       spline
*/

for i : 1 thru length(splines) step 1 do (

       aux : append(aux, [ splines[i] ]),

       format : append(format, [lines])

),

/*     Con esto la variable aux contiene la lista de objetos a
   representar y la variable format
       contiene la lista de instrucciones de formato con que deben
           representarse estos objetos
       (points para los puntos discretos y lines para las
          funciones).
       Lo único que nos falta es la instruccción referente al
          dominio en el que queremos representar
```

```
        las funciones, la sintaxis del dominio de valores de la var
            . independiente para plot2d es
        la siguiente (ver soluciones de ejercicios propuestos y
            ayuda del wxMaxima)

                dominio = [x, valor mínimo de la var. indep, valor m
                    áximo de la var indep.]

        Una forma posible de obtener estos valores es aplicando las
             funciones min y max del
        wxMaxima a la colección de valores de la variable
            independiente (ver soluciones tema 2.7), de
        modo que para indicar el dominio de valores de la var.
            indep. podemos hacer

                [x, apply(min, puntos[1]), apply(max, puntos[1])]

        Con esto ya tenemos todos los ingredientes necesarios, sólo
             queda llamar a plot2d para
        generar la gráfica pedida.
    */

    plot2d(aux, [x, apply(min, puntos[1]), apply(max, puntos[1])],
        format)

)$

/*      EJERCICIO 3

    Dado el problema de condiciones iniciales de orden 1

            dx/dt = f(x, t),      x(t=0) = x_0

    escriba una función en Maxima que proporcione una aproximación a
        la solución de esta ecuación
    por medio del desarrollo en serie de Taylor de x(t), centrado en t
        =0, hasta orden n = 2,
    dado por:

    x(t) = suma_{desde j=0}{hasta n} 1/j! d^j x/dt^j_{en t = 0} t^j

    Datos:

    El valor de x(t) en t=0 está dado por la condición inicial del
        problema, el valor de la
    primera derivada se obtiene sustituyendo en la ecuación

            dx/dt_{en t=0} = f(x_0, 0)
```

```
        el de la segunda derivada se obtiene derivando en la ecuación de
           partida una vez

                d^2 x/dt^2 = {parcial f}/{parcial x} dx/dt + {parcial f}{
                   parcial t}

        y posteriormente sustituyendo los valores t=0, x=x_0, y el valor
           calculado previamente
        para dx/dt en t=0.

        input:
                f(x, t), x_0

        output:
                La aproximación a x(t) mencionada.
*/


/*      SOLUCION 3

        La solución al ejercicio 3 está implementada en la "funcion3",
           cuyo código se incluye
        a continuación.

        Observaciones:

        A.
                De acuerdo a los datos del enunciado, el outpput de la funci
                   ón que se pide es
                sencillamente

                        x_0 + f(x_0, 0) * t + (1/2) derivada2 * t^2

                donde derivada2 es la derivada segunda de x respecto a t
                   evaluada en t=0, cuyo valor
                indica el propio enunciado.
        B.
                Igual que antes, para definir esta función vamos a emplear
                   el comando "block", cuya
                sintaxis y funcionamiento se ha visto en el tema 2.5 y
                   siguientes.
*/

funcion3(f, x0) := block( [derivada1, derivada2],

        /*      Asignamos a la variable local derivada1 el valor de la
           primera derivada de x
```

```
            respecto a t en t = 0, calculada de acuerdo a las
               instrucciones del enunciado.
            La función del wxMaxima que vamos a emplear para las
               sustituciones es subst
            (ver soluciones tema 2.8)
        */

        derivada1 : subst(0, t, subst(x0, x, f) ),

        /*      Tal y como está escrito el enunciado, el input f(x, t) es
           una función de x y de t,
            de modo que antes de derivar para calcular derivada2
               debemos sustituir x por x(t),
            por tanto sustituimos x -> x(t), a continuación derivamos
               respecto a t y por último
            sustituimos. El diagrama de flujo es el siguiente:

                    1. sustituir x -> x(t) en f

                    2. derivar respecto a t

                    3. sustituir dx/dt -> derivada1

                    4. sustituir x(t) -> x_0

                    5. sustituir t -> 0
        */

        derivada2 : (

/* 5 */         subst(0, t,

/* 4 */                 subst( x0, x(t),

/* 3 */                         subst( derivada1, diff(x(t), t, 1),

/* 2 */                                 diff(

/* 1 */                                         subst(x(t), x, f)

/* 2 fin */                             , t, 1)

/* 3 fin */                     )

/* 4 fin */             )

/* 5 fin */    )
```

```
        ),

        /*      Finalmente la función devuelve el desarrollo pedido */

        x0 + derivada1 * t + (1/2) * derivada2 * t^2

)$

/*      FIN de la prueba evaluable de programación en wxMaxima, junio de
    2011 */
```

## 9.2. Curso 2010-2011, convocatoria de septiembre

### Ejercicio 1

Escriba una función en Maxima que genere la aproximación por interpolación Lagrangiana con nodos de Chebyshev de una función $f(x)$ dada, definida sobre un intervalo $[a, b]$ dado, empleando $n$ nodos de interpolación.

*input*:
- Función $f(x)$ cuya aproximación por interpolación va a calcularse.
- Intervalo $[a, b]$ (suponemos que $a < b$, $|a| < \infty$ y $|b| < \infty$).
- Orden $n$ de la aproximación.

*output*:
- Aproximación por interpolación Lagrangiana con nodos de Chebyshev de $f(x)$ sobre el intervalo $[a, b]$ hasta orden $n$.

*Diagrama de flujo*. Para programar esta función necesitarán (al menos) realizar estas operaciones:

1. Generar los $n$ nodos de Chebyshev $x_i$ (con $i = 1, 2, \ldots, n$) correspondientes al intervalo $[a, b]$, que están dados por

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{2i-1}{2n}\pi\right), \qquad i = 1, 2, \ldots, n.$$

2. Generar los valores de $f(x)$ correspondientes a la lista de nodos de interpolación

$$y_i = f(x_i), \qquad i = 1, 2, \ldots, n.$$

3. Emplear el comando `lagrange` del wxMaxima para generar el polinomio de interpolación de Lagrange basado en el conjunto de datos $(x_i, y_i)$ (previamente es necesario cargar el paquete de interpolaciones `interpol`).

## Ejercicio 2

Con el fin de visualizar el comportamiento de la aproximación anterior, escriba una función que genere las gráficas de $f(x)$, $f'(x)$ y $f''(x)$, junto con las aproximaciones dadas por el anterior polinomio de interpolación y su primera y segunda derivada.

*input*:
- Función $f(x)$.
- Intervalo $[a, b]$.
- Función $f_n(x)$, definida como la aproximación por interpolación Lagrangiana con nodos de Chebyshev de $f(x)$ sobre el intervalo $[a, b]$ hasta orden $n$ (calculada previamente).

*output*:
- Gráfica de $f(x)$ y $f_n(x)$ con $x \in [a, b]$.
- Gráfica de $f'(x)$ y $f'_n(x)$ con $x \in [a, b]$.
- Gráfica de $f''(x)$ y $f''_n(x)$ con $x \in [a, b]$.

## Ejercicio 3

Con el fin de visualizar la velocidad de convergencia de la aproximación anterior, escriba una función que genere la gráfica del *error* ($\varepsilon$) de la aproximación como función del orden ($n$) de la aproximación, para $n$ entre 1 y un orden máximo $N$. Para ello definimos el error de la aproximación como

$$\varepsilon(n) \equiv \max_{x \in [a,b]} |f(x) - f_n(x)|$$

es decir, $\varepsilon(n)$ es el valor máximo del módulo de $f(x) - f_n(x)$ con $x$ en $[a, b]$.

*input*:
- Función $f(x)$.
- Intervalo $[a, b]$.
- Orden máximo ($N$) a considerar.

*output*:
- Gráfica de $\varepsilon(n)$ para $n = 1, 2, \ldots, N$.
- Gráfica de $\varepsilon(n)$ para $n = 1, 2, \ldots, N$ en escala logarítmica (tanto para $\varepsilon(n)$ como para $n$).

**Nota:**
Para el cálculo de $\varepsilon(n)$ hay que resolver

$$\varepsilon(n) \equiv \max_{x \in [a,b]} |f(x) - f_n(x)|$$

para ello una posibilidad muy sencilla es introducir una discretización *muy fina* de este intervalo, por ejemplo podemos definir el conjunto de valores

$$X_i = a + (b-a)\,\frac{i-1}{m-1}, \qquad i = 1, 2, \ldots, m$$

con $m \gg n$ (p. ej. se podría tomar $m = 10^3 n$).

Una vez hecho esto calculamos los valores de $|f(x) - f_n(x)|$ con $x = X_i$ ($i = 1, 2, \ldots, m$) y aproximamos $\varepsilon(n)$ como el máximo de estos valores. Si $m$ es suficientemente grande la anterior estrategia nos dará una excelente aproximación al error $\varepsilon(n)$.

Si se prefiere hacer el cálculo de $\varepsilon(n)$ de una manera más rigurosa hay que tener en cuenta que el máximo de $|f(x) - f_n(x)|$ con $x$ en $[a, b]$ estará, o bien en alguno de los puntos donde se anula la derivada de $f(x) - f_n(x)$, o bien en alguno de los extremos del intervalo ($x = a$ o $x = b$). También es importante darse cuenta de que, por definición, $f(x_i) - f_n(x_i) = 0$ (para $i = 1, 2, \ldots, n$), ya que el polinomio de interpolación de Lagrange basado en los puntos $(x_i, y_i = f(x_i))$ pasa por esos puntos. Por tanto todos los extremos relativos de $f(x) - f_n(x)$ con $x \in [a, b]$ están intercalados entre los nodos de interpolación $x_i$. Teniendo esto en cuenta podemos idear la siguiente estrategia para calcular $\varepsilon(n)$:

1. Definimos la función auxiliar $g(x) = f'(x) - f_n'(x)$.
2. Calculamos los $n-1$ valores de $x$ donde se anula $g(x)$, para ello resolvemos por medio del método de Newton la ecuación $g(x) = 0$ tomando como valor inicial de $x$ para el método de Newton a $x = (x_i + x_{i+1})/2$ con $i = 1, 2, \ldots, n-1$ sucesivamente.

   Esto nos proporciona la lista de los $n-1$ valores de $x$ en los que se anula la derivada de $f(x) - f_n(x)$, definimos esta lista de valores como $x_i'$.
3. Calculamos la lista de valores de $|f(x) - f_n(x)|$ con $x = x_i'$, $x = a$ y $x = b$.
4. El máximo de la anterior lista es el valor de $\varepsilon(n)$.

Para este ejercicio daremos por válidas cualquiera de estas 2 estrategias para el cálculo de $\varepsilon(n)$ (o cualquiera otra, siempre y cuando funcione correctamente).

# Ejercicio 4

Emplee la función definida en el ejercicio 1 para programar una función que genere la aproximación por interpolación Lagrangiana con nodos de Chebyshev de la función $y = f(x)$, definida de manera implícita por la ecuación $F(x, y) = 0$, con $x$ tomando valores en un determinado intervalo finito $[a, b]$.

*input*:
- Función $F(x, y)$.
- Intervalo $[a, b]$.

- Orden $n$ de la aproximación.
- $y_0$ valor inicial de $y$ para resolver $F(x, y) = 0$ por medio del método de Newton (ver ejercicio 2 del tema 2.8).

*output*:
- Aproximación por interpolación Lagrangiana con $n$ nodos de Chebyshev de la función $y = f(x)$, definida de manera implícita por la ecuación $F(x, y) = 0$, con $x$ tomando valores en el intervalo $[a, b]$.

**Nota:**

Como ven, todos los ejercicios de esta convocatoria versan sobre el tema de interpolación polinomial con nodos de Chebyshev. Para realizar estos ejercicios necesitarán probar sus programas sobre alguna función $f(x)$ dada, definida en algún intervalo $[a, b]$ dado, y con $n$ desde 1 hasta algún valor concreto del orden máximo $N$. ¿Qué casos concretos se pueden tomar para hacer las pruebas?

- Función $f(x)$: mi recomendación es que para sus pruebas tomen como ejemplo una función $f(x)$ que sea continua, y que tenga infinitas derivadas continuas en $a \leq x \leq b$. En ese caso la aproximación por interpolación Lagrangiana con nodos de Chebyshev converge muy rápidamente. Si toman una función que tenga singularidades en $[a, b]$ la aproximación por interpolación no funcionará bien.

  Por otra parte, en sus pruebas procuren que $f(x)$ no sea un polinomio de $x$, ya que en ese caso tan pronto como el orden $n$ de la aproximación por interpolación supere al grado de $f(x)$, tendremos que el polinomio de interpolación es idéntico a $f(x)$, resultando en un caso trivial.

- Intervalo $[a, b]$: mientras $a$ y $b$ sean finitos pueden tomar cualquier valor. Por ejemplo, pueden hacer todas sus pruebas basándose en el intervalo estándar $[-1, 1]$, o en el $[0, 1]$. De todas formas, es **muy importante** recordar que las funciones que programen deben estar preparadas para operar sobre cualquier intervalo $[a, b]$, cuyos valores se suministran como input a la función.

- Orden máximo $N$: si toman una función $f(x)$ continua con infinitas derivadas continuas en $[a, b]$, y que no tenga variaciones demasiado bruscas en ese intervalo, observarán que $f_n(x)$ se empieza a parecer a $f(x)$ para $n \sim 3 \sim 5$. En general, para obtener una buena aproximación, tanto para $f(x)$ como para sus 2 primeras derivadas, es necesario considerar valores de $n$ entre $10$ y $20$ (si la función $f(x)$ considerada tiene variaciones bruscas en $[a, b]$ es posible que sea necesario considerar valores de $n$ mayores). Por tanto, en sus pruebas empiecen tomando $N = 10$ y si la potencia de su ordenador lo permite aumenten este valor hasta $20$. De todas formas, es **muy importante** recordar que la función del ejercicio 3 debe estar preparada para operar sobre cualquier valor de $N$, suministrado como input a la función.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
   septiembre de 2011
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2010-2011

        Este archivo contiene el código en Maxima de las funciones pedidas
             en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 14.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2011-S-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2011-S-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";
*/

/*      EJERCICIO 1

        Escriba una función en Maxima que genere la aproximación por
           interpolación Lagrangiana
        con nodos de Chebyshev de una función f(x) dada, definida sobre un
             intervalo [a, b]
        dado, empleando n nodos de interpolación. (Diagrama de flujo y
           datos adicionales en enunciado).

        input:
                f: Función f(x) cuya aproximación por interpolación va a
                   calcularse.
                intervalo: Intervalo [a, b] (suponemos que a < b, |a| <
                   infinito y |b| < infinito).
                n: Orden n de la aproximación.
        output:
                Aproximación por interpolación Lagrangiana con nodos de
                   Chebyshev de f(x) sobre
                el intervalo [a, b] hasta orden n.

        SOLUCION 1

        La solución al ejercicio 1 está implementada en la "funcion1",
           cuyo código se incluye
        a continuación.
        El input y output de esta función es el especificado en el
```

```
           enunciado del ejercicio
*/

funcion1(f, intervalo, n) := block( [aux1, aux2],

       /* Definimos la variable local aux1 como la lista de abscisas de
          interpolación,
          calculada aplicando la fórmula dada en el enunciado */

       aux1 : makelist(

              ((intervalo[1] + intervalo[2])/2) + ((intervalo[2] -
                 intervalo[1])/2) * cos( (2*i - 1)*%pi/(2*n) )

              , i, 1, n, 1),

       /* Realmente sólo necesitamos los valores numéricos de estas
          abscisas */

       aux1 : float(aux1),

       /* Aunque no es imprescindible, las ordenamos de menor a mayor */

       aux1 : sort(aux1),

       /* Definimos la variable local aux2 como la lista de ordenadas (
          valores y = f(x))
          correspondiente estas abscisas */

       aux2 : makelist( f(aux1[i]), i, 1, n, 1),

       /* Cargamos el paquete de funciones de interpolación */

       load(interpol),

       /* Finalmente generamos la interpolación pedida aplicando la
          función lagrange sobre
          el conjunto de datos que hemos calculado. Para ello
             previamente combinamos las listas
          aux1 y aux2 en una matriz, con forma: [[x1, y1], [x2, y2],
             ..., [xn, yn]]
        */

       lagrange( transpose( apply( matrix, [aux1, aux2] ) ) )

)$

/*     EJERCICIO 2
```

```
        Con el fin de visualizar el comportamiento de la aproximación
           anterior, escriba una función
        que genere las gráficas de f(x), f'(x) y f''(x), junto con las
           aproximaciones dadas por el
        anterior polinomio de interpolación y su primera y segunda
           derivada.

        input:
                f: Función f(x).
                intervalo: Intervalo [a, b].
                fn: aproximación a f(x) generada con la función del
                   ejercicio 1.
        output:
                gráficas de f(x) y fn(x), f'(x) y fn'(x), f''(x) y fn''(x)

        SOLUCION 2

        La solución al ejercicio 2 está implementada en la "funcion2",
           cuyo código se incluye
        a continuación.
*/

funcion2(f, intervalo, fn) := block( [aux],

        /*      En principio no tiene ninguna dificultad generar cada una
           de las gráficas pedidas.
                Para poder ver todas estas gráficas simultáneamente, sin
                   necesidad de representarlas
                todas en la misma ventana, una posible solución es usar la
                   función wxplot2d del
                wxmaxima. Si no se conoce esta función, otra posibilidad es
                    guardar cada gráfica en
                un archivo (ver tema 2.7). Cualquiera de estas dos
                   posibilidades es válida, en este
                ejercicio haremos ambas cosas.
        */

        /*      Generamos la gráfica de f(x) junto con fn(x) y la guardamos
             en el archivo "f-fn.eps" */

                aux : [ f(x), fn(x) ],

                plot2d( aux, [ x, intervalo[1], intervalo[2] ], [psfile, "f
                   -fn.eps"] ),

                wxplot2d( aux, [ x, intervalo[1], intervalo[2] ] ),
```

```
        /*      Generamos la gráfica de f'(x) junto con fn'(x) y la
           guardamos en el archivo "f-fn-dif1.eps"*/

                aux : [ diff( f(x), x, 1 ), diff( fn(x), x, 1 ) ],

                plot2d( aux, [ x, intervalo[1], intervalo[2] ], [psfile, "f
                   -fn-dif1.eps"] ),

                wxplot2d( aux, [ x, intervalo[1], intervalo[2] ] ),

        /*      Generamos la gráfica de f''(x) junto con fn''(x) y la
           guardamos en el archivo "f-fn-dif1.eps"*/

                aux : [ diff( f(x), x, 2 ), diff( fn(x), x, 2 ) ],

                plot2d( aux, [ x, intervalo[1], intervalo[2] ], [psfile, "f
                   -fn-dif2.eps"] ),

                wxplot2d( aux, [ x, intervalo[1], intervalo[2] ] )

)$

/*      EJERCICIO 3

        Con el fin de visualizar la velocidad de convergencia de la
           aproximación anterior, escriba
        una función que genere la gráfica del error (epsilon) de la
           aproximación como función del
        orden (n) de la aproximación, para n entre 2 y un orden máximo N.
           Para ello definimos el
        error de la aproximación como

                epsilon = max |f(x) - fn(x)| con x en [a,b]

        es decir, epsilon(n) es el valor máximo del módulo de f(x) - fn(x)
            con x en [a, b].

        input:
                f: Función f(x).
                intervalo: Intervalo [a, b].
                N: Orden máximo a considerar.
        output:
                Gráfica de epsilon(n) para n = 2, ..., N.
                Gráfica de epsilon(n) para n = 2, ..., N en escala logarí
                   tmica (tanto para
                epsilon(n) como para n).
```

```
        SOLUCION 3

        La solución al ejercicio 3, siguiendo la primera estrategia para
            el cálculo del error epsilon
        (ver enunciado), está implementada en la "funcion3A", cuyo código
            se incluye a continuación.
        La solución con el error epsilon calculado siguiendo la segunda
            estrategia mencionada en el
        enunciado está programada en la función "funcion3B".
*/

funcion3A(f, intervalo, N) := block( [n, fn, m, X, epsilon],

        /*      Definimos la variable local n como el índice que va a
            recorrer todos los valores
                posibles del orden de la aproximación, desde n = 2 hasta n
                    = N, por medio de un bucle.

                Para cada valor de n la variable local fn contendrá la
                    correspondiente aproximación
                a f, calculada con la función programada en el ejercicio 1,
                     y epsilon contendrá el
                valor del error correspondiente.

                Definimos la variable local X como la lista de valores de x
                     que emplearemos para
                calcular el error, de acuerdo con el método explicado en el
                     enunciado
                (tomando m = N * a * 10^2, siendo "a" un número de orden
                    unidad).
        */

        m : N * 5 * 10^2,

        X : makelist( intervalo[1] + (intervalo[2] - intervalo[1]) * (i -
            1) / (m - 1), i, 1, m, 1),

        X : float(X),

        /*      Inicializamos la variable local epsilon como una lista vací
            a, en la que iremos
                añadiendo los resultados obtenidos en cada nivel de
                    aproximación.
        */

        epsilon : [],

        for n : 2 thru N step 1 do (
```

```
            /* Como las operaciones realizadas en este bucle llevan
               bastante tiempo vamos sacar
               por pantalla el valor de n, para saber por dónde vamos
                  */

            print("vamos por n = ", n),

            fn : funcion1(f, intervalo, n),

            epsilon : append( epsilon, [[n, apply( max,

                   makelist( abs( float( f(X[i]) - subst(X[i], x, fn) )
                       ), i, 1, m, 1)

            ) ]] )

      ),    /* fin del bucle */

      /*
            Como f(x) es una función, para evaluar f(x) en X[i] en el
               bucle anterior sencillamente
            hacemos f(X[i]), pero como la variable local fn no está
               definida como una función
            para evaluar fn en X[i] tenemos que emplear subst para
               sustituir x por X[i] en fn.

            Una vez tenemos calculada la lista de valores [[n2, error2
               ], ..., [nN, errorN]]
            generamos las gráficas pedidas igual que en el ejercicio 2,
                las mostramos en pantalla
            con wxplot2d y las guardamos en dos archivos con formato .
               eps
      */

      plot2d([discrete, epsilon], [psfile, "error-vs-n-lineal-A.eps"]),

      wxplot2d([discrete, epsilon]),

      plot2d([discrete, epsilon], [logx], [logy], [psfile, "error-vs-n-
         log-A.eps"]),

      wxplot2d([discrete, epsilon], [logx], [logy])

)$

funcion3B(f, intervalo, N) := block( [tolnewton, epsilon, n, fn, g, aux,
   xdifmax, thisepsilon],
```

```
/*     En este caso vamos a recorrer el mismo bucle que antes pero
    calcularemos el error
       correspondiente a cada valor del orden n de una forma más
          precisa que como se
       hizo en la función anterior. Ahora, en lugar de
          sencillamente muestrear la diferencia
       entre f(x) y fn(x) en todo el intervalo y tomar el máximo
          de esas diferencias, lo que
       haremos será calcular el máximo real de dicha diferencia,
          resolviendo por el método
       de Newton la correspondiente ecuación, tal y como se indica
           en el enunciado del
       examen.
*/

/* Cargamos el paquete de solución numérica de ecuaciones por el
   método de Newton */

load(newton1),

/* Definimos la variable local tolnewton como el error máximo con
    el que queremos aplicar
   el método de Newton. */

tolnewton : 10^(-4),

/* Definimos la variable local epsilon como una lista vacía en la
    que iremos añadiendo los
   sucesivos valores del error correspondientes a cada valor de n
       */

epsilon : [],

for n : 2 thru N step 1 do (

       /* Como las operaciones realizadas en este bucle llevan
          bastante tiempo vamos sacar
          por pantalla el valor de n, para saber por dónde vamos
             */

       print("vamos por n = ", n),

       /* aproximación fn correspondiente a este valor de n */

       fn : funcion1(f, intervalo, n),

       /* fn es igual a f(x) en cada uno de los nodos empleados
```

```
    para la interpolación.
    La diferencia f(x) - fn alcanza un máximo (o mínimo)
        local para un valor de
    x situado entre cada dos nodos de interpolación
        consecutivos. Para calcular
    las posiciones donde se alcanzan estos máximos (o mí
        nimos) locales tenemos
    que calcular dónde se anula la derivada de la
        diferencia f(x) - fn.
    Definimos g como la derivada de la diferencia entre la
        aproximación y f(x): */

g : diff( f(x) - fn, x, 1 ),

/* Calculamos los nodos de interpolación correspondientes
    a este valor de n */

aux : sort( float( makelist(

((intervalo[1] + intervalo[2])/2) + ((intervalo[2] -
    intervalo[1])/2) * cos( (2*i - 1)*%pi/(2*n) )

, i, 1, n, 1 ) ) ),

/* Calculamos los puntos medios entre nodos de interpolaci
    ón consecutivos */

aux : makelist( (aux[i] + aux[i+1]) / 2, i, 1, n-1, 1 ),

/* Resolvemos la ecuación g(x) = 0 tomando como punto de
    partida cada uno de estos
    puntos intermedios entre los nodos de interpolación,
        esto nos da la colección
    de valores de x donde la diferencia alcanza un máximo o
         mínimo local */

xdifmax : [],

for j : 1 thru n-1 step 1 do (

        xdifmax : append(xdifmax, [newton( g, x, aux[j],
            tolnewton )])

),

/* El error que estamos buscando es la máximo de los
    valores absolutos de las
    diferencias f - fn. Este valor máximo estará, o bien en
```

```
                uno de los puntos
               "xdifmax" anteriores, o bien en uno de los extremos del
                intervalo.
               Evaluamos el valor absoluto de la diferencia f - fn en
                todos los puntos donde
               se alcanzan extremos locales y lo guardamos en la
                variable local thisepsilon. */

              thisepsilon : makelist( abs( float( subst(xdifmax[i], x, f(
                x) - fn) ) ), i, 1, n-1, 1 ),

              /* Evaluamos el valor absoluto de la diferencia en los
                extremos del intervalo y
                lo añadimos a la lista de valores thisepsilon. */

              thisepsilon : append( thisepsilon, abs( float( [subst(
                intervalo[1], x, f(x) - fn)] ) ) ),

              thisepsilon : append( thisepsilon, abs( float( [subst(
                intervalo[2], x, f(x) - fn)] ) ) ),

              /* Finalmente nos quedamos con el máximo de todos esos
                valores: */

              epsilon : append( epsilon, [[n, apply( max, thisepsilon )
                ]] )

      ),      /* fin del bucle */

              /* Esta función nos proporciona una estimación del error
                muy precisa, mientras que
                la realizada en la funcion3A sólo nos daba una
                aproximación (que además siempre
                es menor que el error máximo real). El cálculo del
                error con funcion3B lleva
                mucho más tiempo que con funcion3A si n es grande, a la
                 vista de los resultados
                proporcionados por ambas funciones ¿merece la pena el
                tiempo adicional de cálculo?

                Una vez calculada la lista de valores [[n2, error2],
                ..., [nN, errorN]] generamos
                las gráficas pedidas igual que en el ejercicio 2, las
                mostramos en pantalla con
                wxplot2d y las guardamos en dos archivos con formato .
                eps, igual que antes */

    plot2d([discrete, epsilon], [psfile, "error-vs-n-lineal-B.eps"]),
```

```
        wxplot2d([discrete, epsilon]),

        plot2d([discrete, epsilon], [logx], [logy], [psfile, "error-vs-n-
           log-B.eps"]),

        wxplot2d([discrete, epsilon], [logx], [logy])

)$

/*      EJERCICIO 4

        Emplee la función definida en el ejercicio 1 para programar una
           función que genere la
        aproximación por interpolación Lagrangiana con nodos de Chebyshev
           de la función y = f(x),
        definida de manera implícita por la ecuación F(x, y) = 0, con x
           tomando valores en un
        determinado intervalo finito [a, b].

        input:
                F: función F(x, y).
                intervalo: intervalo [a, b].
                n: orden n de la aproximación.
                y0: valor inicial de y para resolver F(x, y) = 0 por medio
                   del método de Newton
                       (ver ejercicio 2 del tema 2.8).
        output:
                Aproximación por interpolación Lagrangiana con n nodos de
                   Chebyshev de la función
                y = f(x), definida de manera implícita por la ecuación F(x,
                    y) = 0, con x tomando
                valores en el intervalo [a, b].

        SOLUCION 4

        La solución al ejercicio 4 está implementada en la "funcion4",
           cuyo código se incluye
        a continuación.
*/

        /* Este ejercicio es prácticamente inmediato si tomamos como
           punto de partida las funciones
           que tenemos ya programadas. Todo lo relativo a definir una
              función f(x) por medio de la
           solución numérica de la ecuación F(x, y) = 0 ya lo tenemos
              programado en la función "solvenewton"
           que programamos para el ejercicio 2 del tema 2.8 (tal y como
```

```
            se indicaba en el enunciado).
           De modo que lo único que necesitamos hacer es llamar a la
            función "solvenewton" y a la
           función "funcion1" definida en el primer ejercicio de este
            examen. */

       /* Comenzamos cargando la función "solvenewton" proporcionada en
           la soluciones a los
           ejercicios del tema 2.8 */

/* FUNCION "solvenewton"
   Resuelve numéricamente la ecuación F(x, y) = 0 para la variable y por
      medio del método de Newton
   Input:
     F = función que define la ecuación F = 0
     y = variable que queremos despejar
     y0 = punto de partida para el método de Newton
   Output:
     valor de y encontrado para la solución de F = 0 */

solvenewton(F, y, y0) := block( [epsilon],

  /* cargamos el paquete newton1 */

  load(newton1),

  /* epsilon = tolerancia (suponemos que hemos encontrado la solución
     con un grado de aproximación aceptable cuando F(x, y) < epsilon) */

  epsilon : 10^(-6),

  newton(F, y, y0, epsilon)

)$

/* fin */

       /* Una vez cargada la función "solvenewton" definimos la función
           "funcion4" */

funcion4(F, intervalo, n, y0) := block( [solucion],

       /* Definimos la función auxiliar "fauxiliar" como la solución num
           érica de F(x, y) = 0,
           resolviendo para y en función de x por medio de "solvenewton"
            */

       fauxiliar(x) := solvenewton( F(x, y), y, y0 ),
```

```
        /* Para poder emplear la función anterior como un argumento para
           la función "funcion1",
           necesitamos que esta funcion auxiliar sea una función de x (y
              no una mera expresión),
           y por ese motivo no podemos definirla como una variable local.
               Por ese motivo hemos
           definido "fauxiliar" como una variable global en la instrucció
              n anterior.
           Ahora lo único que tenemos que hacer es llamar a la función "
              funcion1" operando sobre
           esta "fauxiliar" */

        solucion : funcion1(fauxiliar, intervalo, n),

        /* Una vez tenemos calculado el polinomio interpolador que pedía
           el enunciado eliminamos
           la función "fauxiliar" que ya no necesitamos */

        kill(fauxiliar),

        /* Finalmente retornamos el polinonio interpolador pedido */

        solucion

)$

/*      FIN de la prueba evaluable de programación en wxMaxima, septiembre
     de 2011 */


/*      C O M E N T A R I O S  A D I C I O N A L E S, S O L O  P O R  C U R
    I O S I D A D

        Todos los comentarios incluidos a continuación no forman parte del
             temario de la asignatura, los incluimos sólo por curiosidad,
           para aquellos alumnos interesados en profundizar un poco más en
             estas cuestiones.

        Una vez programadas las funciones de los ejercicios 1 y 2 es
           interesante emplearlas para ver cómo funciona este tipo de
           interpolaciones. Obviamente, la calidad de la aproximación es
           peor para la primera derivada que para la propia función, y es
           peor para la segunda derivada que para la primera, y así
           sucesivamente si calculamos aproximaciones a las derivadas de
           orden superior por medio de derivación del polinomio
           interpolador (en física es poco habitual trabajar con derivadas
             de orden mayor que 2). De todas formas, si la función f(x) es
```

*suficientemente suave, este desarrollo converge rápidamente alcanzándose una precisión excelente, tanto para la aproximación de f(x) como de sus dos primeras derivadas, para valores de n no demasiado grandes (del orden de unas pocas decenas). Por supuesto existe una teoría matemática que explica cuándo estos desarrollos son convergentes y cuando no, y qué velocidad de convergencia tienen, pero este no es el lugar para exponer dicha teoría (los alumnos interesados pueden consultar el tema de "Interpolación Lagrangiana con nodos de Chebyshev" en cualquier libro de cálculo numérico).*

*Para explorar el funcionamiento de estos desarrollos se puede ejecutar este código:*

```
int : [-2, 4]$
orden : 8$
A(x) := exp(x);
B(x) := funcion1(A, int, orden);
funcion2(A, int, B);
```

*variando el orden de la aproximación, también se recomienda probar con otras funciones y otros intervalos. Otro ejemplo interesante es la función sen(w x) en el intervalo [-pi, pi] para distintos valores de la frecuencia w. A medida que aumentamos la frecuencia es necesario aumentar el orden de la aproximación para tener una aproximación buena. Un ejercicio interesante es, entonces, hacer una representación gráfica del orden mínimo necesario para una buena aproximación en función de w. (Por cierto, para funciones que cumplen condiciones de contorno periódicas, como sen(w x) en [-pi, pi], tiene más sentido hacer desarrollos basados en las funciones sen y cos que desarrollos polinomiales, pero de todas formas es un ejemplo interesante).*

*La función del ejercicio 2 nos permite apreciar visualmente la calidad de la aproximación realizada (lo cual es muy importante para entender cómo funcionan estas aproximaciones) pero no nos proporciona una medida cuantitativa realmente objetiva de esa calidad. Esa medida es precisamente lo que se calcula en el ejercicio 3. De las dos gráficas del error frente al orden de la aproximación proporcionadas por la función del ejercicio 3 la más interesante es la representación en escala logarítmica. Supongamos que el error disminuye con el orden n de forma proporcional a n^(-p) para un cierto número p > 0. En ese caso veríamos que, para n suficientemente grande, la gráfica logarítmica del error se aproxima a una recta con pendiente -p. Cuando sucede esto se dice que la velocidad de convergencia es algebráica. Sin embargo, si la función f(x) que estamos*

```
    aproximando es suficientemente suave, en la gráfica logarítmica
     del error veremos que éste no se aproxima a una recta, sino
    que la velocidad de disminución de Ln(epsilon) frente a Ln(n)
    es cada vez mayor. Cuando sucede esto se dice que la velocidad
    de convergencia del desarrollo es exponencial. La velocidad de
    convergencia exponencial es la propiedad más notable de los
    desarrollos basados en "Interpolación Lagrangiana con nodos de
    Chebyshev". Esta velocidad de convergencia sólo se alcanza si
    la función que queremos aproximar es suficientemente suave, en
    caso contrario sólo tendremos velocidad de convergencia algebrá
    ica, con un exponente p determinado por el comportamiento de la
     función que queremos aproximar y sus derivadas en el intervalo
     considerado. Por supuesto, este no es el tema de esta
    asignatura, de modo que no profundizaremos más en esto.

Teóricamente uno podría seguir aumentando el orden de estos
    desarrollos indefinidamente, y hacer que el error de la
    aproximación tendiese a cero. Sin embargo, como estamos
    haciendo todas las operaciones numéricas con precisión finita,
    lo que conlleva un inevitable error de redondeo (del orden de
    10^(-16) en los PCs habituales), lo que se observará en la grá
    fica del error es que, una vez que éste ha disminuido hasta un
    número del orden de 10^(-16), a partir de ahí ya no disminuye m
    ás por mucho que aumentemos el orden del desarrollo, sino que
    se mantiene estable.
*/
```

## 9.3. Curso 2011-2012, convocatoria de junio

### Ejercicio 1

Dada una función de densidad de probabilidad dependiente de $n$ variables ($f(x_1, x_2, \ldots, x_n)$) escriba una función que calcule el momento $\mu_{i_1,i_2,\ldots,i_n}$, definido como

$$\mu_{i_1,i_2,\ldots,i_n} = \int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} x_1^{i_1} \cdot x_2^{i_2} \cdot \cdots \cdot x_n^{i_n} \cdot f(x_1, x_2, \ldots, x_n)\, dx_1 dx_2 \ldots dx_n$$

siendo $[a_i, b_i]$ el intervalo donde puede tomar valores la variable $x_i$.

*input*:
- Función de densidad de probabilidad $f(x_1, x_2, \ldots, x_n)$.
- Intervalos de definición $[a_i, b_i]$ de todas estas variables.
- Multi-índice $i_1, i_2, \ldots, i_n$ correspondiente al momento $\mu_{i_1,i_2,\ldots,i_n}$ que va a calcularse.

*output*:
- Momento $\mu_{i_1,i_2,\ldots,i_n}$.

## Ejercicio 2

Emplee la función programada en el ejercicio anterior para programar una nueva función que calcule la matriz de covariancia $\Sigma_{ij}$ de $f(x_1, x_2, \ldots, x_n)$.

*input*:
- Función de densidad de probabilidad $f(x_1, x_2, \ldots, x_n)$.
- Intervalos de definición $[a_i, b_i]$ de todas estas variables.

*output*:
- Matriz de covariancia $\Sigma_{ij}$ de $f(x_1, x_2, \ldots, x_n)$.

DATOS:

La matriz de covariancia $\Sigma_{ij}$ es la generalización al caso de $n$ variables de la variancia, se define como

$$\Sigma_{ij} = \langle (x_i - \langle x_i \rangle) \cdot (x_j - \langle x_j \rangle) \rangle \,, \qquad i, j, = 1, \ldots, n$$

donde la operación $\langle x \rangle$ denota el "valor promedio de x". Recordando que la operación "tomar el promedio" es lineal, es inmediato comprobar que la matriz de covariancia puede calcularse como

$$\Sigma_{ij} = \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \,, \qquad i, j, = 1, \ldots, n$$

## Ejercicio 3

Por definición es evidente que la matriz de covariancia es simétrica, por tanto existe una base ortonormal de autovectores en la que esta matriz es diagonal.

Programe una función que calcule los autovectores de la matriz de covariancia de $f(x_1, \ldots, x_n)$ y que exprese la densidad de probabilidad $f(x_1, \ldots, x_n)$ en términos de las coordenadas ($y_j$), definidas como las coordenadas respecto de la base de autovectores de $\Sigma_{ij}$.

*input*:
- Función de densidad de probabilidad en términos de las variables de partida $f(x_1, x_2, \ldots, x_n)$.

*output*:
- Función de densidad de probabilidad $g(y_1, y_2, \ldots, y_n)$ en términos de las coordenadas $y_i$ respecto a la base de autovectores de $\Sigma_{ij}$.

**Nota:**

Para simplificar este problema suponga que el dominio de definición de todas las coordenadas $x_i$ es $\mathbb{R}$, por tanto también será este el dominio de definición de las coordenadas $y_i$.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
   junio de 2012
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2011-2012

        Este archivo contiene el código en Maxima de las funciones pedidas
            en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 16.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2012-J-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2012-J-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";
*/


/*
        EJERCICIO 1

        Func. momentocruzado: Calcula el momento cruzado de orden mi de la
            variable continua x
        definida sobre el intervalo int, con densidad de probabilidad f (
           ver enunciado).
*/

momentocruzado(f, x, int, mi) := block( [i, aux],
        /*
        INPUT:
                f: función de densidad de probabilidad.
                   Suponemos que f est\'a debidamente normalizada, en caso
                      contrario basta con
                   redefinir f como f : f / mu0.
                x: lista de las "n" variables (x[1], ..., x[n]) de las que
                   depende f.
                int: lista de los "n" intervalos de definici\'on
```

```
            correspondientes a cada una de las
             variables contenidas en "x", suponemos que int[i][1] <
                 int[i][2].
        mi: multi-\'{\i}ndice correspondiente al momento cruzado
            que va a calcularse.
OUTPUT: momento cruzado de orden mi[1], mi[2], ..., mi[n] de f(x
    [1], ..., x[n])

OBS: Este programa realiza todas las integrales de forma anal\'{\i
    }tica (no num\'erica),
     suponemos por tanto que todas las integrales necesarias para
         el c\'alculo del momento
     cruzado pedido pueden hacerde de forma anal\'{\i}tica sin
         problemas.
*/

/* Comenzamos realizando la integral correspondiente a la primera
    variable */

       aux : integrate(x[1]^mi[1] * f, x[1], int[1][1], int[1][2])
           ,

/* Seguidamente vamos realizando de manera recursiva todas las
   integrales restates */

       for i : 2 thru length(x) step 1 do (

              aux : integrate(x[i]^mi[i] * aux, x[i], int[i][1],
                 int[i][2])

       ),

/* Finalemente retornamos el valor de la integral pedida */

       aux

)$

/*
       EJERCICIO 2

       Func. matrizcovariancia: Calcula la matriz de covariancia de la
           func. de densidad de probabilidad
       f, dependiente de las variables continuas x = {x1, ..., x_n},
           definidas sobre los intervalos int =
       {int_1, ..., int_n} (ver enunciado).
*/
```

```
matrizcovariancia(f, x, int) := block( [i, j, zero, promedios, diagonal,
   superdiagonal],
      /*
      INPUT:
            f: función de densidad de probabilidad.
               Suponemos que f est\'a debidamente normalizada, en caso
                  contrario basta con
               redefinir f como f : f / mu0.
            x: lista de las "n" variables (x[1], ..., x[n]) de las que
                depende f.
            int: lista de los "n" intervalos de definici\'on
                correspondientes a cada una de las
                 variables contenidas en "x", suponemos que int[i][1] <
                     int[i][2].
      OUTPUT: matriz de covariancia de f(x[1], ..., x[n])

      OBS: Este programa emplea la funci\'on "momentocruzado" definida m
         \'as arriba.
      */

      /*
      OBS:
            El enunciado indica que la matriz de covariancia Sigma est|'a
                dada por <x[i]*x[j]> - <x[i]><x[j]>,
            por tanto, una forma directa de calcular esta matriz es:

                  makelist( makelist( <x[i]*x[j]> - <x[i]><x[j]>, j, 1,
                      length(x)), i, 1, length(x))

            donde los promedios <...> pueden calcularse empleando la
               funci\'on que acabamos de definir.
            Aunque esta forma de calcular la matriz de covariancia es
               correcta, es poco eficiente ya
            que no tiene en cuenta la simetría de la matriz de
               covariancia. Seg\'un indica el enunciado
            la matriz de covariancia es sim\'etrica, es decir, Sigma_{ij}
                = Sigma_{ji}, por tanto, nos
            podemos ahorrar el c\'alculo de los elementos que caen por
               debajo de la diagonal, ya que son
            iguales a los correspondientes elementos de por encima de la
               diagonal.

            En este examen damos por v\'alido el c\'lculo directo (poco
               eficiente) de la matriz de
            covariancia aplicando directamente la definici\'on. De todas
               formas, a continuaci\'on se
            proporciona una estrategia de c\'alculo m\'as eficiente.
      */
```

```
/* Comenzamos calculando la lista de valores promedio de las
   variables x[i] (i = 1, ..., n).

   Suponiendo que f est\'a normalizada, cada uno de estos
      promedios est\'a dado por la
   integral m\'ultiple de x[i] * f. Una manera muy sencilla de
      programar este c\'alculo es
   empleando la funci\'on momentocruzado(f, x, int, mi) que
      acabamos de definir.

   Para ello nos fijamos que si definimos el multi-\'{\i}ndice
      que aparece en la func. momentocruzado
   como una lista de "n" ceros, entonces "momentocruzado( X * f,
      x, int, [0, 0, ..., 0])" realiza
   precisamente la integral m\'ultiple de X * f, que es lo que
      necesitamos.

   Definimos entonces zero como una lista de "n" ceros: */

       zero : makelist(0, i, 1, length(x), 1),

/* Los promedios de las n variables contenidas en x pueden
   calcularse sencillamente como: */

       promedios : makelist(momentocruzado( x[i] * f, x, int, zero
          ), i, 1, length(x), 1),

/* Una vez hemos calculado los promedios, calculamos los
   elementos diagonales de la matriz de
   covariancia empleando una estrategia similar a la que acabamos
       de usar */

       diagonal : makelist(

              momentocruzado( x[i]^2 * f, x, int, zero) -
                 promedios[i]^2

              , i, 1, length(x), 1),

/* Definimos ahora una matriz cuyos elementos diagonales sean
   iguales a los elementos diagonales
   de Sigma (que acabamos de calcular), y cuyos elementos no
      diagonales sean cero.

   Una forma sencilla de hacer esto es multiplicando componente a
       componente los elementos de
   la matriz identidad (de n = length(x) dimensiones) por los
```

```
            elementos diagonales de Sigma. */

            diagonal : ident( length(x) ) * diagonal,

        /* Definimos ahora una matriz cuyos elementos de por encima de la
            diagonal [super-diagonales]
           sean los correspondientes elementos de Sigma, con todos los
              restantes elementos nulos */

            superdiagonal : makelist( makelist(

                   if (i>j) then (

                          momentocruzado( x[i] * x[j] * f, x, int,
                             zero) - promedios[i] * promedios[j]

                   ) else 0

                   , i, 1, length(x), 1), j, 1, length(x), 1),

            superdiagonal : apply(matrix, superdiagonal),

        /* Finalmente la matriz Sigma est\'a dada por la suma de 3
           matrices:

            diagonal: matriz con los elementos diagonales de Sigma (
               elementos restantes = 0)

            superdiagonal: matriz triangular-superior con los elementos
                super-diagonales de Sigma
                          (elementos restantes = 0)

            traspuesta de superdiagonal: matriz triangular-inferior con
                los elementos de por
                          debajo de la diagonal de Sigma (elementos
                             restantes = 0)
        */

            diagonal + superdiagonal + transpose(superdiagonal)

        )$

/*
        EJERCICIO 3

        Func. PDF2componentesprincipales: Calcula la expresi\'on de la
           func. de densidad de probabilidad
        f, dependiente de las variables continuas x = {x1, ..., x_n}, en t
```

```
        \'erminos de sus "componentes
      principales", definidas por las coordenadas respecto a la base de
        autovectores de la matriz de
      covariancia de f (ver enunciado).
*/

PDF2componentesprincipales(f, x, y) := block( [aux, MC, vectores, g],
      /*
      INPUT:
            f: función de densidad de probabilidad.
              Suponemos que f est\'a debidamente normalizada, en caso
                 contrario basta con
              redefinir f como f : f / mu0.
            x: lista de las "n" variables (x[1], ..., x[n]) de las que
               depende f.
            y: lista de nombres de las "n" variables (y[1], ..., y[n])
               de las que depende g.
      OUTPUT: g(y): expresión de la función de densidad de probabilidad
        f en términos de las coordenadas
            respecto a la base de autovectores de la matriz de
               covariancia de f.

      OBS:   Para que sea posible aplicar este cambio de variable es
        necesario que el determinante de
            la matriz de covariancia de f sea distinto de cero. Tomamos
                esto como hipótesis.

      OBS:
            Este programa emplea las funcs. "matrizcovariancia"
              definida m\'as arriba y "eigenvectorlist"
            definida en el tema de aplicaciones de Maxima en \'algebra.
      */

            /* Tal y como se indica en el enunciado, suponemos que cada
                una de las variables aleatorias
              continuas contenidas en x puede tomar valores en toda la
                  recta real, es decir

                        -infinito < x_i < +infinito.

              Partiendo de esta suposici\'on generamos la lista de
                 intervalos de defici\'on de
              las variables contenidas en x, guardamos esta lista en
                 la variable local aux.
            */

            aux : makelist( [-inf, +inf], i, 1, length(x), 1),
```

```
/* Calculamos la matriz de covariancia de f por medio de
   la func. "matrizcovariancia" */

MC : matrizcovariancia(f, x, aux),

/* Cargamos la func. "eigenvectorlist" y calculamos la
   lista de autovectores de MC */

/* OBS:

       El path del archivo "eigenvector_list.mc" que figura
           en la instrucci\'on siguiente
       debe modificarse de acuerdo a la direcci\'on de este
           archivo en el PC donde estemos
       ejecutando este programa.
*/

batchload("path/eigenvector_list.mc"),

vectores : eigenvectorlist(MC),

/* normalizamos esta lista de autovectores dividiendo cada
    uno de ellos por su norma */

for i : 1 thru length(vectores) step 1 do (

       vectores[i] : vectores[i] / sqrt( vectores[i].
          vectores[i] )

),

/* Definimos la matriz del cambio de base como la matriz
   formada por los autovectores
   tomados como vectores columna.
   Guardamos la matriz del cambio de base en la variable
      local MC */

MC : apply(matrix, vectores),

/* La relaci\'on entre las coordenadas respecto a esta
   nueva base (y) y las coordenadas
   respecto a la base de partida (x) es [y] = [MC]^(-1).[x
      ], es decir, [x] = [MC].[y].

   El volumen del elemento diferencial de volumen dvol =
      dx_1 * ... * dx_n en t\'erminos
   de las coordenadas respecto a la base nueva queda como
      dvol = |J| * dy_1 * ... * dy_n
```

```
        donde |J| es el valor absoluto del determinante de la
           matriz jacobiana, que en este
        caso coincide con la matriz del cambio de base (que
           acabamos de definir como MC).
    */

    /* Escribimos las x en t\'erminos de las y, guardamos el
       resultado en la var. local aux */

    aux : MC . y,

    /* Aplicamos el cambio x -> x(y) en f */

    g : subst( aux[1], x[1], f )[1],

    for i : 2 thru length(x) step 1 do (

           g : subst(aux[i], x[i], g)[1]

    ),

    /* finalmente multiplicamos este resultado por el valor
       absoluto del determinante jacobiano */

    g * abs( determinant( MC ) )

)$
```

## 9.4. Curso 2011-2012, convocatoria de septiembre

### Ejercicio 1

Dado el problema de condiciones iniciales de orden 1

$$\frac{dx}{dt} = f(x,t), \qquad x(t=0) = x_0$$

escriba una función en Maxima que proporcione una aproximación a la solución de esta ecuación por medio del desarrollo en serie de Taylor de $x(t)$, centrado en $t = 0$, hasta orden $n$.

Datos: La solución pedida está dada por:

$$x(t) \simeq \sum_{j=0}^{n} \frac{1}{j!} \left.\frac{d^j x}{dt^j}\right|_{t=0} t^j$$

El valor de $x(t)$ en $t = 0$ está dado por la condición inicial del problema, el valor de la primera derivada se obtiene sustituyendo en la ecuación

$$\left.\frac{dx}{dt}\right|_{t=0} = f(x_0, 0)$$

el de la segunda derivada se obtiene derivando en la ecuación de partida una vez

$$\frac{d^2x}{dt^2} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial t}$$

y posteriormente sustituyendo los valores $t = 0$, $x = x_0$, y el valor calculado previamente para $dx/dt$ en $t = 0$. Derivando una vez más la ecuación de partida, y sustituyendo los valores encontrados para $x(0)$, $x'(0)$ y $x''(0)$, encontramos el valor de $x'''(0)$.

Este proceso se repite de forma iterativa ($n$ veces) para calcular los valores de las derivadas de orden superior (desde $1$ hasta $n$) que aparecen en el desarrollo en serie de Taylor.

*input*: $f(x,t)$, $x_0$, $n$.

*output*: La aproximación a $x(t)$ mencionada.

## Ejercicio 2

Generalice la función programada en el ejercicio anterior para el caso en que la condición inicial del problema no esté dada en el punto $t = 0$, sino en un valor arbitrario ($t = t_0$) de la variable independiente.

Datos: En este caso la solución pedida está dada por:

$$x(t) \simeq \sum_{j=0}^{n} \frac{1}{j!} \left.\frac{d^j x}{dt^j}\right|_{t=t_0} (t - t_0)^j$$

El valor de $x(t)$ en $t = t_0$ está dado por la condición inicial del problema ($x(t = t_0) = x_0$). Los valores de $\left.\frac{d^j x}{dt^j}\right|_{t=t_0}$ se calculan de manera análoga a como se ha indicado en el ejercicio anterior.

*input*: $f(x,t)$, $x_0$, $t_0$, $n$.

*output*: La aproximación a $x(t)$ mencionada.

## Ejercicio 3

Escriba una función en Maxima que resuelva numéricamente la ecuación diferencial ordinaria de orden 1

$$\frac{dx}{dt} = f(x,t), \qquad x(t=t_0) = x_0$$

por medio de la función `rk`:

*input*:
- Función $f(x,t)$ que define la EDO de orden 1: $x' = f(x,t)$.
- Variable dependiente $x$.
- Condición inicial $x_0$.
- Variable independiente y valores mínimo y máximo de la variable independiente a considerar en la solución numérica.
- Parámetro de *paso* a emplear en el algoritmo de Runge-Kutta.

*output*:
- Matriz con los resultados numéricos obtenidos por medio del comando `rk`.

## Ejercicio 4

Escriba una función que realice la representación gráfica de los resultados obtenidos por medio de la función generalizada programada en el ejercicio 2, para un grado de aproximación $n$ entre $1$ y un cierto valor máximo $n_{\max}$, y para valores de la variable independiente entre $t_0$ y $t_0+T$, así como la solución numérica calculada en el ejercicio anterior. El objetivo de esta función es que podamos visualizar cómo mejora el resultado de la aproximación al aumentar el orden del desarrollo en serie de Taylor.

*input*: $f(x,t)$, $x_0$, $t_0$, $n_{\max}$, $T$.

*output*: Gráfica con las aproximaciones a $x(t)$ por serie de Taylor correspondientes a los valores de $n$ entre $1$ y $n_{\max}$, junto con la aproximación proporcionada por la solución numérica calculada por medio del método de Runge-Kutta.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
   septiembre de 2012
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2011-2012

        Este archivo contiene el código en Maxima de las funciones pedidas
            en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 17.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2012-S-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2012-S-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";
*/


/*      EJERCICIOS 1 y 2

        El ejercicio 1 es un caso particular del ejercicio 2, lo que nos
           permite resolver ambos
        ejercicios con una única función, que llamamos serieEDO1.
        Este ejercicio es una generalización del problema propuesto en el
           examen de junio de 2011. */

serieEDO1(f, x, t, t0, x0, n) := block( [i, aux, coefs, ecuaciones],
        /*
        INPUT:
                f: función de "x" y "t" que define el lado derecho de la
                   EDO de 1er orden x' = f(x, t)
                x: variable dependiente
                t: variable independiente
                t0, valor de t donde aplicamos la condición inicial
                x0, condición inicial dada por el valor de x en t = t0
                n: orden del desarrollo en serie de Taylor de x(t) en torno
                    a t = t0.
        OUTPUT:
                desarrollo en serie de Taylor de orden n y centrado en t0
                   de la solución x(t).
        */

        /* Alojamos en la variable local "coefs" la lista de derivadas de
```

```
        x(t) necesaria para
        el desarrollo en serie de Taylor pedido. */

            coefs : makelist( diff( x(t), t, i ), i, 0, n, 1 ),

        /* Calculamos los valores de estas derivadas como se indica en el
            enunciado, para ello
        derivamos f(x(t), t) respecto a t "i" veces, con "i" entre 0 y
            "n-1".
        De esta forma obtenemos la lista de ecuaciones que determina
            los valores de las derivadas
        que aparecen en coefs. Alojamos esta lista de ecuaciones en la
            variable local "ecuaciones". */

            ecuaciones : makelist( coefs[i+2] = diff( subst(x(t), x, f)
                , t, i ), i, 0, n-1, 1 ),

        /* Añadimos a esta lista de ecuaciones la condición inicial del
            problema. */

            ecuaciones : append( [x(t) = x0], ecuaciones ),

        /* Invertimos el orden de la lista, ya que para calcular los
            valores de las derivadas es
        mejor ir sustituyendo desde la derivada de mayor grado hasta
            la de menor grado. */

            ecuaciones : reverse(ecuaciones),

        /* Sustituimos en la lista coefs cada derivada por su valor. */

            for i : 1 thru length(ecuaciones) step 1 do coefs : subst(
                ecuaciones[i], coefs),

        /* Finalmente sustituimos t por el valor en torno al que se
            realiza el desarrollo. */

            coefs : subst(t0, t, coefs),

        /* Retornamos el desarrollo en serie de Taylor de acuerdo a la fó
            rmula indicada en el enunciado. */

            coefs . makelist( (1/i!) * (t - t0)^i, i, 0, n, 1 )

)$

/*      EJERCICIO 3
```

```
        La solución al ejercicio 3 está implementada en la función
           solaproxRK. Esta solución se basa en
        parte en los ejercicios resueltos del examen de junio de 2011 */

solaproxRK(f, x, t, t0, x0, tmax, paso) := block( [aux],
        /*
        INPUT:
                f: función de "x" y "t" que define el lado derecho de la
                   EDO de 1er orden x' = f(x, t)
                x: variable dependiente
                t: variable independiente
                t0, valor de t donde aplicamos la condición inicial
                x0, condición inicial dada por el valor de x en t = t0
                tmax: valor máximo de t considerado en la solución numérica
                paso: parámetro de paso a emplear en el algoritmo de Runge-
                   Kutta.

        OUTPUT:
                Solución numérica calculada por medio del algoritmo de
                   Runge-Kutta válida en [t0, tmax].
        */

        /*  Definimos la variable local auxiliar "aux" como la lista:

                [variable independiente, valor inicial, valor final, paso]
                   */

                aux : [t, t0, tmax, paso],

        /*      Resolvemos el EDO con la función rk, guardamos el resultado
            en aux */

                load("dynamics"),

                aux : rk(f, x, x0, aux ),

        /*      La solución numérica que nos proporciona rk es una lista de
            listas de valores.
                El enunciado nos pide el output de rk con formato matricial
                   , para ello convertimos
                esta lista en una matriz (ver soluciones tema 2.10)
        */

                apply(matrix, aux)
)$

/*      EJERCICIO 4
```

```
        La solución al ejercicio 4 está implementada en la función
           grafRKTaylor. Esta solución se basa
        en parte en los ejercicios resueltos del examen de junio de 2011.
           */

grafRKTaylor(f, x, t, t0, x0, T, nmax, paso) := block( [i, solRK, solT,
   format, aux, s],

        /*
        INPUT:
                f: función de "x" y "t" que define el lado derecho de la
                   EDO de 1er orden x' = f(x, t)
                x: variable dependiente
                t: variable independiente
                t0, valor de t donde aplicamos la condición inicial
                x0, condición inicial dada por el valor de x en t = t0
                T: longitud del intervalo [t0, t0 + T] donde se van a
                   representar los resultados
                nmax: grado máximo del desarrollo en serie de Taylor
                   considerado.

                Esta función emplea las anteriores funciones "solaproxRK" y
                    "serieEDO1".
        OUTPUT:
                Gráfica de la solución numérica calculada por medio del
                   algoritmo de Runge-Kutta válida
                en [t0, t0 + T], junto con los correspondientes desarrollos
                    en serie de Taylor con orden
                entre 1 y nmax.

        EJEMPLO:
                grafRKTaylor(-z + sin(r), z, r, 0, 1, 10, 11, 0.01);
        */

        /* Definimos las variables locales solRK y solT como las
           soluciones numérica y la obternida
           por medio del desarrollo en serie de Taylor a orden máximo */

                solRK : transpose( solaproxRK(f, x, t, t0, x0, t0 + T, paso
                   ) ),

                solT : serieEDO1(f, x, t, t0, x0, nmax),

                print("Desarrollo en seria de Taylor de grado máximo = ",
                   solT),

        /* Asignamos a la variable local aux los puntos obtenidos por
           Runge-Kutta a representar */
```

```
        aux : [ [discrete, solRK[1], solRK[2]] ],

/*  Asignamos a la variable local "format" el primer elemento de
    la lista de intrucciones
    de formato para la representación gráfica: "points" */

        format : [style, points],

/*  La asignación realizada para solT contiene directamente el
    desarrollo en serie de Taylor de
    orden máximo, de esta forma evitamos repetir el cálculo de
       desarrollos de Taylor para n < nmax.
    Sin embargo el enunciado pide representar todos los
       desarrollos desde orden n=1 hasta el orden
    máximo nmax. Para generar esta lista de desarrollos basta con
       sustituir en solT de manera
    secuencial (t-t0)^i -> 0, para i mayor que un orden dado "n",
       que vaya tomando valores desde
    n=1 hasta nmax. Para ello definimos la lista de reglas de
       sustitución "s" */

        s : makelist( (t - t0)^i = 0, i, 2, nmax, 1 ),

/*  Para obtener la lista de desarrollos pedida basta con aplicar
    secuencialmente las sustituciones
    "s" en solT, eliminando de forma consecutiva la primera de las
        sustituciones incluida en "s" por
    medio de la función "rest".

        Inicialmente, al aplicar las sustituciones "s" en solT
           sustituimos por 0 todos los
        términos de solT de orden mayor que 2, obteniendo el
           desarrollo de orden 1.
        A continuación, para obterner el desarrollo de orden 2
           eliminamos la primera de las
        sustituciones de "s" (por medio de "rest") y aplicamos las
           restantes.
        Seguidamente, eliminando la siguiente sustitución de "s" y
           aplicando las restantes
        obtenemos el desarrollo de orden 3.
        Repetimos esta operación tantas veces como sea necesario
           para generar todos los
        desarrollos desde orden 1 hasta nmax.

    En el siguiente bucle implementamos esta estrategia para
       generar la lista de desarrollos pedida
    y añadimos esta lista de desarrollos a la variable local "aux
```

```
            ".
           Finalmente, en este bucle añadimos una instrucción de formato
              "lines" por cada uno de los
           desarrollos considerados, de modo que en la gráfica pedida se
              representen por medio de líneas. */

              for i : 1 thru nmax-1 step 1 do (

                     aux : append(aux, [subst(s, solT)] ),

                     s : rest(s),

                     format : append(format, [lines])

              ),

       /* Por último añadimos el desarrollo completo y la
          correspondiente instrucción de formato */

                     aux : append(aux, [solT] ),

                     format : append(format, [lines]),

       /* Finalmente generamos la gráfica, empleando los datos numéricos
           obtenido con RK para definir el
          tamaño de la ventana en la que se visualizarán los resultados
              */

              plot2d(       aux,

                            [t, apply(min, solRK[1]), apply(max, solRK
                               [1])],

                            [y, apply(min, solRK[2]), apply(max, solRK
                               [2])],

                            format

              )

)$
```

## 9.5. Curso 2012-2013, convocatoria de junio

## Ejercicio 1

Una *curva paramétrica* $\boldsymbol{x} = \boldsymbol{x}(\lambda)$ en $\mathbb{R}^3$ es un conjunto de 3 funciones

$$\boldsymbol{x}(\lambda) = \{x(\lambda), y(\lambda), z(\lambda)\}$$

que representan las cooordenadas cartesianas de la posición en $\mathbb{R}^3$ de un punto en función del parámetro $\lambda$, de tal forma que cuando $\lambda$ toma valores en un cierto intervalo (p. ej. $\lambda \in [0, 1]$) el punto $\boldsymbol{x}(\lambda)$ describe una curva contenida en $\mathbb{R}^3$.

En el primer ejercicio del examen de Maxima de este curso se pide programar una función que calcule el vector tangente unitario a una curva cualquiera $\boldsymbol{x}(\lambda)$, denotado por $\boldsymbol{t}$.

El input y el output de la función pedida son los siguientes:

*input*: Curva paramétrica $\boldsymbol{x}$, parámetro $\lambda$.

*output*: Vector tangente unitario $\boldsymbol{t}$.

**Obs.:** Para calcular $\boldsymbol{t}$ calculamos en primer lugar el vector tangente

$$\boldsymbol{x}' = \frac{d\boldsymbol{x}}{d\lambda}$$

y posteriormente calculamos el vector tangente unitario $\boldsymbol{t}$ por medio de

$$\boldsymbol{t} = \frac{\boldsymbol{x}'}{\|\boldsymbol{x}'\|}$$

Como puede verse, si interpretamos $\boldsymbol{x}$ como el vector de posición de un móvil y el parámetro $\lambda$ como el tiempo, $\boldsymbol{x}'$ es simplemente el vector velocidad y $\boldsymbol{t}$ es el vector unitario en la dirección definida por el vector velocidad.

## Ejercicio 2

Dada la curva paramétrica del ejercicio anterior, se pide programar una función que calcule el vector normal a la curva $\boldsymbol{x}(\lambda)$, que denotaremos por $\boldsymbol{n}$.

*input*: Curva paramétrica $\boldsymbol{x}$, parámetro $\lambda$.

*output*: Vector normal $\boldsymbol{n}$.

**Obs.:** Para calcular el vector normal $\boldsymbol{n}$ calculamos en primer lugar el vector de curvatura $\boldsymbol{k}$, dado por

$$\boldsymbol{k} = \frac{\boldsymbol{t}'}{\|\boldsymbol{x}'\|}$$

donde hemos empleado la notación definida en el ejercicio anterior (es decir, $\boldsymbol{t}' = d\boldsymbol{t}/d\lambda$).

Una vez calculado el vector $\boldsymbol{k}$, el vector normal es el vector unitario en la dirección de $\boldsymbol{k}$, es decir

$$\boldsymbol{n} = \frac{\boldsymbol{k}}{\|\boldsymbol{k}\|}$$

## Ejercicio 3

Dada la curva paramétrica del ejercicio primero, se pide programar una función que calcule el vector binormal $\boldsymbol{b}$, definido como

$$\boldsymbol{b} = \boldsymbol{t} \times \boldsymbol{n}$$

donde $\boldsymbol{t}$ es el vector tangente unitario, $\boldsymbol{n}$ es el vector normal, y $\times$ denota el "producto vectorial".

*input*: Curva paramétrica $\boldsymbol{x}$, parámetro $\lambda$.

*output*: Vector binormal $\boldsymbol{b}$.

## Ejercicio 4

En la teoría de curvas paramétricas las definiciones anteriores (y en general todas las expresiones) son considerablemente más sencillas cuando se escriben en términos del *parámetro longitud de arco*. Dicho parámetro suele denotarse por $s$ y está definido por la relación

$$\left\| \frac{d\boldsymbol{x}}{ds} \right\| = 1$$

de donde se deduce que $\int_a^b ds = \int_{\boldsymbol{x}(a)}^{\boldsymbol{x}(b)} \|d\boldsymbol{x}\|$ es igual a la longitud del arco de la curva $\boldsymbol{x}(\lambda)$ comprendido entre los puntos $\boldsymbol{x}(a)$ y $\boldsymbol{x}(b)$, lo que explica el nombre de $s$. En los ejercicios de 1 a 3 hemos supuesto que el parámetro $\lambda$ es un parámetro arbitrario, no necesariamente igual a $s$ (es decir, en general tendremos $\|\boldsymbol{x}'\| \neq 1$).

Un factor que aparece con frecuencia cuando se trabaja con curvas en términos de un parámetro arbitrario (como $\lambda$) es la derivada $d\lambda/ds$, dada por

$$\frac{d\lambda}{ds} = \frac{1}{\|\boldsymbol{x}'\|}$$

En este ejercicio se pide programar una función que proporcione el valor de esta derivada, lo cual le puede resultar útil para la programación de las funciones pedidas en los ejercicios 1 y 2, ya que este factor aparece en las definiciones de $\boldsymbol{t}$ y $\boldsymbol{k}$.

*input*: Curva paramétrica $\boldsymbol{x}$, parámetro arbitrario $\lambda$.

*output*: Derivada del parámetro arbitrario $\lambda$ respecto del parámetro longitud de arco: $1/\|\boldsymbol{x}'\|$.

## Ejercicio 5

Emplee las funciones `draw3d` y `parametric` (definidas en el paquete de funciones `draw` incluido en el wxMaxima) para programar una función que genere la gráfica en 3D de la curva paramétrica $\boldsymbol{x}(\lambda) = \{x(\lambda), y(\lambda), z(\lambda)\}$ con $\lambda \in [a, b]$.

*input*:
- Curva paramétrica $\boldsymbol{x}$.
- Nombre del parámetro $\lambda$.
- Intervalo $[a, b]$ de definición del parámetro $\lambda$.

*output*: Gráfica de $\boldsymbol{x}(\lambda)$ con $\lambda \in [a, b]$.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
    junio de 2013
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2012-2013

        Este archivo contiene el código en Maxima de las funciones pedidas
            en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 17.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

               batchload( concat( path, "2013-J-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2013-J-R.mc en el disco, p. ej. path podría ser algo parecido a

               path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                  /";
```

```
        El input/output de todas las funciones siguientes es el indicado
           en el enunciado del examen.
*/

/*      EJERCICIO 4

        Por conveniencia realizamos primero el ejercicio cuarto, de modo
           que podamos usar esta
        función en posteriormente. El input/output de esta función es el
           indicado en el enunciado. */

        dlambda_ds( curva, param ) := block( [vel],

                /* Calculamos el vector velocidad y lo guardamos en la
                   variable local vel */

                vel : diff( curva, param ),

                /* Retornamos 1 / ||vel|| */

                1 / sqrt( vel[1]^2 + vel[2]^2 + vel[3]^2 )

        )$

/*      EJERCICIO 1

        El ejercicio primero es tan sencillo que realmente no necesita
           ningún comentario.
        Por simplicidad, repetimos aquí el código para el cálculo de 1 /
           ||vel||. */

        vect_tangente( curva, param ) := block( [vel],

                /* Calculamos el vector velocidad y lo guardamos en la
                   variable local vel */

                vel : diff( curva, param ),

                /* Retornamos vel / ||vel|| */

                vel / sqrt( vel[1]^2 + vel[2]^2 + vel[3]^2 )

        )$

/*      EJERCICIO 2

        Calculamos el vector unitario normal tal y como se indica en el
```

```
        enunciado.
    Por simplicidad, repetimos aquí el código para el cálculo de 1 /
        ||vel||. */

    vect_normal( curva, param ) := block( [k],

            /* Calculamos el vector de curvatura y lo guardamos en la
               variable local k */

            k : diff( vect_tangente( curva, param ), param ),

            k : k * dlambda_ds( curva, param ),

            /* Retornamos k / ||k|| */

            k / sqrt( k[1]^2 + k[2]^2 + k[3]^2 )

    )$

/*  EJERCICIO 3

    Calculamos el vector binormal por medio del producto vectorial b =
        t x n.

    Para calcular el producto vectorial se puede emplear el comando
        "~" incluido en el paquete
    "vect". De todas formas, dado que para resolver este ejercicio só
        lo necesitamos emplear el
    producto vectorial, cuya definición es muy sencilla, y que el
        paquete "vect" incluye multitud
    de funciones que no necesitamos (al menos de momento), en lugar de
         cargar el paquete "vect"
    optamos por programar el cálculo del producto vectorial.
    (Por supuesto, la opción de cargar el paquete "vect" y usar "~" es
         válida). */

    vect_binormal( curva, param ) := block( [t, n],

            /* Calculamos los vectores t y n */

            t : vect_tangente( curva, param ),

            n : vect_normal( curva, param ),

            /* Retornamos b */

            [
                    + ( t[2]*n[3] - t[3]*n[2] ),
```

```
                - ( t[1]*n[3] - t[3]*n[1] ),

                + ( t[1]*n[2] - t[2]*n[1] )
        ]

)$


/*      EJERCICIO 5

        Incluimos la orden de cargar el paquete "draw" fuera de la función
            para evitar volver a cargarlo
        cada vez que se ejecute la función */

        load(draw);

        plot_curva( curva, param, intervalo ) := block( [aux],

                /* Guardamos en aux en número de puntos que emplearemos
                   para representar la gráfica.
                    El valor por defecto (29) es muy bajo, lo que produce
                       gráficas con poca calidad. */

                aux : 200,

                /* Dependiendo de la curva a representar y del intervalo
                   de valores considerado, el valor
                    anterior puede no ser una buena elección para producir
                       una buena representación, en cuyo
                    caso habrá que cambiarlo. */

                draw3d( nticks = aux, parametric(curva[1], curva[2], curva
                   [3], param, intervalo[1], intervalo[2]) )

        )$
```

## 9.6. Curso 2012-2013, convocatoria de septiembre

# Ejercicio 1

Una *superficie paramétrica* $\boldsymbol{r} = \boldsymbol{r}(u, v)$ en $\mathbb{R}^3$ es un conjunto de 3 funciones (dependientes de 2 parámetros)

$$\boldsymbol{r}(u, v) = \{x(u, v), y(u, v), z(u, v)\}$$

que representan las cooordenadas cartesianas $x$, $y$ y $z$ de la posición en $\mathbb{R}^3$ de un punto en función de los parámetros $u$ y $v$, de tal forma que cuando $u$ y $v$ toman valores en un cierto intervalo (p. ej. $u \in [0, 1]$, $v \in [0, 1]$) el punto $\boldsymbol{r}(u, v)$ describe una superficie contenida en $\mathbb{R}^3$.

En este ejercicio se pide programar una función que calcule la matriz de la *primera forma fundamental* de una superficie paramétrica como la anterior.

El input y el output de la función pedida son los siguientes:

*input*: Lista de las 3 funciones ($\{x(u, v), y(u, v), z(u, v)\}$) que definen la superficie paramétrica, nombres de los 2 parámetros de los que dependen estas funciones.

*output*: Matriz de la primera forma fundamental.

**Obs.:** Dada una superficie paramétrica con la forma indicada más arriba, la matriz de la primera forma fundamental es la matriz simétrica $2 \times 2$ dada por los productos escalares siguientes

$$\begin{pmatrix} E & F \\ F & G \end{pmatrix} = \begin{pmatrix} \boldsymbol{r}_u \cdot \boldsymbol{r}_u & \boldsymbol{r}_u \cdot \boldsymbol{r}_v \\ \boldsymbol{r}_u \cdot \boldsymbol{r}_v & \boldsymbol{r}_v \cdot \boldsymbol{r}_v \end{pmatrix}$$

donde los vectores $\boldsymbol{r}_u$ y $\boldsymbol{r}_v$ están dados por

$$\boldsymbol{r}_u = \frac{\partial \boldsymbol{r}}{\partial u}, \qquad \boldsymbol{r}_v = \frac{\partial \boldsymbol{r}}{\partial v}$$

respectivamente. Como puede verse, cada uno de estos vectores (que no son necesariamente unitarios) es el vector tangente a la superficie según el correspondiente parámetro.

# Ejercicio 2

Dada la superficie paramétrica del ejercicio anterior, se pide programar una función que calcule el vector normal unitario a la superficie, que denotaremos por $\boldsymbol{n}$.

*input*: Como en el ejercicio anterior.

*output*: Vector normal unitario $\boldsymbol{n}$.

**Obs.:** El vector normal unitario $\boldsymbol{n}$ está dado por

$$\boldsymbol{n} = \frac{\boldsymbol{r}_u \times \boldsymbol{r}_v}{\|\boldsymbol{r}_u \times \boldsymbol{r}_v\|}$$

donde hemos empleado la notación definida en el ejercicio anterior y donde $\times$ representa el producto vectorial.

## Ejercicio 3

A partir de las definiciones de los ejercicios anteriores, se pide programar una función que calcule la matriz de la *segunda forma fundamental* de la superficie, que está dada por

$$\begin{pmatrix} L & M \\ M & N \end{pmatrix} = \begin{pmatrix} \boldsymbol{r}_{uu} \cdot \boldsymbol{n} & \boldsymbol{r}_{uv} \cdot \boldsymbol{n} \\ \boldsymbol{r}_{uv} \cdot \boldsymbol{n} & \boldsymbol{r}_{vv} \cdot \boldsymbol{n} \end{pmatrix}$$

donde los vectores $\boldsymbol{r}_{uu}$, $\boldsymbol{r}_{uv}$ y $\boldsymbol{r}_{vv}$ están dados por

$$\boldsymbol{r}_{uu} = \frac{\partial \boldsymbol{r}_u}{\partial u}, \qquad \boldsymbol{r}_{uv} = \frac{\partial \boldsymbol{r}_u}{\partial v} = \frac{\partial \boldsymbol{r}_v}{\partial u}, \qquad \boldsymbol{r}_{vv} = \frac{\partial \boldsymbol{r}_v}{\partial v}$$

respectivamente.

*input*: Como en el ejercicio 1.

*output*: Matriz de la segunda forma fundamental.

## Ejercicio 4

La primera forma fundamental permite calcular longitudes de curvas definidas sobre la superficie. Dada la superficie paramétrica del ejercicio 1, una curva paramétrica definida sobre esta superficie puede definirse por medio de 2 funciones

$$u = u(t), \qquad v = v(t)$$

que nos definen los parámetros $u$ y $v$ de la superficie como funciones de un nuevo parámetro $t$. Cuando $t$ toma valores en un cierto intervalo (p. ej. $t \in [0, 1]$), el punto $\boldsymbol{r}(t) = \{x[u(t), v(t)], y[u(t), v(t)], z[u(t), v(t)]\}$ describe una curva contenida en la superficie.

La longitud de esta curva desde el punto $\boldsymbol{r}(t_1)$ hasta el punto $\boldsymbol{r}(t_2)$ está dada por la correspondiente integral del elemento diferencial de longitud:

$$s = \int_{t_1}^{t_2} \left\| \frac{d\boldsymbol{r}}{dt} \right\| dt$$

Aplicando la regla de la cadena es inmediato verificar que $s$ está dada por

$$s = \int_{t_1}^{t_2} \left[ E \left( \frac{du}{dt} \right)^2 + 2F \left( \frac{du}{dt} \right) \left( \frac{dv}{dt} \right) + G \left( \frac{dv}{dt} \right)^2 \right]^{1/2} dt$$

de forma que para el cálculo de la longitud de esta curva hacen falta los coeficientes ($E$, $F$ y $G$) de la primera forma fundamental, junto con las derivadas de $u$ y $v$ respecto del nuevo parámetro $t$ ($du/dt$ y $dv/dt$).

Dada la superficie paramétrica del ejercicio 1 y la curva paramétrica contenida en dicha superficie definida por las funciones $u = u(t)$, $v = v(t)$, programe una función que calcule la longitud de dicha curva cuando $t$ varía en un cierto intervalo $[t_1, t_2]$.

*input*: Superficie paramétrica como en el ejercicio 1, funciones $u = u(t)$, $v = v(t)$ que definen la curva, nombre del parámetro $t$ que define la curva paramétrica, intervalo $[t_1, t_2]$ considerado para el parámetro $t$.

*output*: Longitud de la curva de acuerdo a

$$s = \int_{t_1}^{t_2} \left[ E \left( \frac{du}{dt} \right)^2 + 2F \left( \frac{du}{dt} \right) \left( \frac{dv}{dt} \right) + G \left( \frac{dv}{dt} \right)^2 \right]^{1/2} dt$$

# Ejercicio 5

Emplee las funciones `draw3d` y `parametric_surface` (definidas en el paquete de funciones `draw` incluido en el wxMaxima) para programar una función que genere la gráfica en 3D de la superficie paramétrica $\boldsymbol{r}(u, v) = \{x(u, v), y(u, v), z(u, v)\}$ con $u \in [a, b]$ y $v \in [c, d]$.

*input*:

- Superficie paramétrica $\boldsymbol{r}(u, v)$.
- Nombre de los parámetros $u$ y $v$ empleados en la definición de la superficie.
- Intervalos $[a, b]$ y $[c, d]$ de definición de estos parámetros.

*output*: Gráfica de $\boldsymbol{r}(u, v)$ con $u \in [a, b]$ y $v \in [c, d]$.

## Ejercicio 6

Apliquen los resultados de los ejercicios anteriores a la superficie

$$\boldsymbol{r}(u,v) = \left\{u\cos v, u\sin v, u^2\right\}$$

con $u \in [0,2]$, $v \in [0,2\pi]$. Calculen la longitud de la curva definida sobre esta superficie por medio de las funciones

$$u(t) = 2, \qquad v(t) = t$$

con $t \in [0,\pi]$.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
   septiembre de 2013
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2012-2013

        Este archivo contiene el código en Maxima de las funciones pedidas
            en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 17.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2013-S-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2013-S-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";

        El input/output de todas las funciones siguientes es el indicado
           en el enunciado del examen.
*/

/*      EJERCICIO 1

        El ejercicio primero es tan sencillo que realmente no necesita
           ningún comentario. */
```

```
primeraFF( superficie, param ) := block( [ru, rv, E, F, G],

        /* Calculamos las derivadas parciales de la superficie */

        ru : diff( superficie, param[1] ),

        rv : diff( superficie, param[2] ),

        /* Calculamos los valores de E, F, G según definición en
           enunciado */

        E : ru . ru,

        F : ru . rv,

        G : rv . rv,

        /* Retornamos la matriz de la primera forma fundamental de
            la superficie */

        apply( matrix, [ [E, F], [F, G] ] )

)$

/*      EJERCICIO 2

        El ejercicio segundo también es muy sencillo, repetimos aquí el có
           digo para el cálculo de r_u y r_v. */

        vect_normal( superficie, param ) := block( [ru, rv, n],

                /* Calculamos las derivadas parciales de la superficie */

                ru : diff( superficie, param[1] ),

                rv : diff( superficie, param[2] ),

                /* Calculamos el producto vectorial y lo guardamos en la
                   variable local n */

                n : [ ru[2] * rv[3] - ru[3] * rv[2], -(ru[1] * rv[3] - ru
                   [3] * rv[1]), ru[1] * rv[2] - ru[2] * rv[1] ],

                /* Retornamos n / ||n|| */

                n / sqrt( n . n )
```

```
)$

/*      EJERCICIO 3

        En este caso programamos el cálculo de las derivadas parciales
           segundas r_uu, r_uv, r_vv y llamamos a
        la función del ejercicio anterior para el cálculo del vector
           normal. */

        segundaFF( superficie, param ) := block( [ruu, ruv, rvv, n, L, M,
           N],

                /* Calculamos las derivadas segundas */

                ruu : diff( superficie, param[1], 2 ),

                ruv : diff( superficie, param[1], 1, param[2], 1 ),

                rvv : diff( superficie, param[2], 2 ),

                /* Calculamos el vector normal unitario */

                n : vect_normal( superficie, param ),

                /* Calculamos los elementos L, M, N según definición en
                   enunciado */

                L : ruu . n,

                M : ruv . n,

                N : rvv . n,

                /* Retornamos la matriz de la segunda forma fundamental de
                    la superficie */

                apply( matrix, [ [L, M], [M, N] ] )

        )$

/*      EJERCICIO 4

        En este caso el input de la función tiene 2 partes, la
           correspondiente a la superficie y
        la correspondiente a la curva. Definimos por tanto el input de
           esta función como:

                        ( superficie, paramsup, curva, paramcurva, limitest
```

```
                )

donde

                superficie = [x(u, v), y(u, v), z(u, v)]

                paramsup = [u, v]

                curva = [u(t), v(t)]

                paramcurva = t

También necesitamos como input los valores de los límites t_1 y
   t_2 entre los que se
calcula la longitud de la curva, definimos estos valores como
   limitest = [t_1, t_2].

Aparte de esto, lo único que necesitamos es llamar a la función
   del primer ejercicio para
calcular la primera forma fundamental, y calcular las derivadas de
    u(t), v(t) respecto a t. */

longitud_curva_en_sup( superficie, paramsup, curva, paramcurva,
   limitest ) := block( [I, uvdot, aux],

        /* Calculamos la primera forma fundamental y la guardamos
           en la variable local I */

        I : primeraFF( superficie, paramsup ),

        /* Calculamos las derivadas de u y v respecto a t */

        uvdot : diff( curva, paramcurva ),

        /* Calculamos la norma de dr/dt según enunciado */

        aux : sqrt( transpose( I . uvdot ) . uvdot ),

        /*
                Sustituimos en la expresión anterior los parámetros
                   de la superficie "u" y "v"
                por sus expresiones en términos del parámetro de la
                   curva "t": "u(t)" y "v(t)":
        */

        aux : sublis( [ paramsup[1] = curva[1], paramsup[2] = curva
           [2] ], aux ),
```

```
            /* simplificamos la expresión obtenida */

            aux : factor( ratsimp( trigsimp( aux ) ) ),

            /* Retornamos la integral de ||dr/dt||dt entre t_1 y t_2
               */

            integrate(aux, paramcurva, limitest[1], limitest[2])

        )$


/*      EJERCICIO 5

        Incluimos la orden de cargar el paquete "draw" fuera de la función
            para evitar volver a cargarlo
        cada vez que se ejecute la función */

        load(draw);

        plot_superficie( superficie, param, intervalo1, intervalo2 ) :=
           block( [aux],

                /* OBS, INPUT:

                        superficie = [x(u, v), y(u, v), z(u, v)]

                        param = [u, v]

                        intervalo1 = límites para u, intervalo2 = límites
                           para v
                 */

                /* Guardamos en aux en número de puntos que emplearemos
                   para representar la gráfica.
                    El valor por defecto (29) es muy bajo, lo que produce
                       gráficas con poca calidad. */

                aux : 200,

                /* Dependiendo de la superficie a representar y del
                   intervalo de valores considerado, el valor
                    anterior puede no ser una buena elección para producir
                       una buena representación, en cuyo
                    caso habrá que cambiarlo. */

                draw3d( nticks = aux, parametric_surface(
```

```
                        superficie[1], superficie[2], superficie[3],

                        param[1], intervalo1[1], intervalo1[2],

                        param[2], intervalo2[1], intervalo2[2]

            ) )

        )$

/*      EJERCICIO 6

        Una vez programadas las funciones anteriores, el ejercicio 6 es
           inmediato. */

        /*
            En primer lugar definimos la superficie a considerar, obsé
               rvese que ahora definimos la
            superficie y los parámetros a emplear como variables
               globales, ya que en este caso no
            estamos programando una función que vayamos a emplear en el
                futuro, sino que estamos
            aplicando a "este" caso particular las funciones
               programadas en los ejercicios 1 a 5.

            Aunque ahora estamos suministrando las soluciones de los
               ejercicios 1 a 6 en un único
            archivo (para simplificar), lo lógico sería almacenar por
               separado los ejercicios 1 a 5
            en una biblioteca de funciones, ya que se trata de
               funciones generales que nos pueden
            venir bien en el futuro, y almacenar el ejercicio 6 como
               una sesión de wx maxima, ya que
            se trata de la aplicación de estas funciones generales a un
                caso particular.
        */

            estasup : [u*cos(v),u*sin(v),u^2];

        /*
            A continuación definimos las variables que contienen los
               nombres de los parámetros
            empleados y sus límites.
        */

            estospar : [u, v];

            estoslimites1 : [0, 2];
```

```
        estoslimites2 : [0, 2*%pi];

/*
        GRAFICA de esta superficie:
*/

        plot_superficie( estasup, estospar, estoslimites1,
           estoslimites2 );

/*
        PRIMERA FORMA FUNDAMENTAL de esta superficie:
*/

        estaPFF : primeraFF( estasup, estospar );

        /*  en este caso el resultado se puede simplificar */

        estaPFF : trigsimp(estaPFF);

/*
        VECTOR NORMAL de esta superficie:
*/

        esten : vect_normal( estasup, estospar );

        /*  en este caso el resultado se puede simplificar */

        esten : factor(trigsimp(esten));

/*
        SEGUNDA FORMA FUNDAMENTAL de esta superficie:
*/

        estaSFF : segundaFF( estasup, estospar );

        /*  en este caso el resultado se puede simplificar */

        estaSFF : factor(trigsimp(estaSFF));

/*
        FINALMENTE introducimos la información relativa a la curva
           mencionada en el enunciado:
*/

        estacurva : [2, t];

        esteparametro : t;
```

```
        estoslimitest : [0, %pi];
/*
        LA LONGITUD de esta curva, contenida en la anterior
           superficie, entre los valores de t especificados es:
*/

        estalong : longitud_curva_en_sup( estasup, estospar,
           estacurva, esteparametro, estoslimitest );

/*
        Si se emplean estas soluciones como elementos de una
           biblioteca de funciones es MUY IMPORTANTE
        recordar que debemos separar la solución del ejercicio 6,
           ya que contiene diversas variables
        globales.
*/
```

## 9.7. Curso 2013-2014, convocatoria de junio

### Ejercicio 1

Consideremos el sistema dinámico definido por una masa puntual $m$ en una dimensión, cuya posición $x(t)$ y momento lineal $p(t)$ varían bajo la acción de la fuerza conservativa

$$F = -\frac{dV}{dx}$$

de acuerdo a la ecuación de Newton

$$\frac{dx}{dt} = \frac{p}{m}, \qquad \frac{dp}{dt} = -V'$$

donde las funciones del tiempo $x(t)$ y $p(t)$ son las variables del problema, $m$ es un parámetro constante y la función $V(x)$ es una determinada función potencial ($V' = dV/dx$).

Programe una función en Maxima que, dado el potencial de fuerzas $V$ dependiente la variable $x$, y dado el intervalo de valores $[x_1, x_2]$, represente gráficamente la función $V(x)$ con $x$ en dicho intervalo y guarde la gráfica correspondiente en un archivo con formato EPS, de modo que pueda usarse posteriormente en un documento.

*input*: Función $V(x)$, variable $x$, intervalo $[x_1, x_2]$ de variación de la variable $x$.

*output*: Archivo con formato EPS que contenga la gráfica de $V(x)$ con $x \in [x_1, x_2]$.

## Ejercicio 2

Programe una función en Maxima que, dado el potencial de fuerzas $V$ dependiente de la variable $x$, y dado el intervalo de valores $[x_1, x_2]$ calcule todos los puntos de equilibrio del sistema, resolviendo para ello la ecuación

$$V'(x) = 0$$

de manera analítica si es posible, y de forma numérica mediante el método de Newton si lo anterior no es posible. En este segundo caso no calcularemos todos los puntos de equilibrio de $V(x)$, sino sólamente aquellos que se encuentren dentro del intervalo acotado $[x_1, x_2]$ (por supuesto, en todos estos ejercicios supondremos que $x_2 > x_1$ y que ambos son finitos).

*input*: Función $V(x)$, variable $x$, intervalo $[x_1, x_2]$ de variación de la variable $x$.

*output*: Lista de soluciones analíticas de la ecuación $V'(x) = 0$.
Si lo anterior no es posible, lista de soluciones numéricas de dicha ecuación dentro del intervalo de trabajo $[x_1, x_2]$.

## Ejercicio 3

Basándonos en los ejercicios anteriores programe una función que devuelva las frecuencias propias ($\omega_i$) de oscilaciones de baja amplitud en torno a los puntos de equilibrio encontrados. Estas frecuencias se obtienen aproximando el potencial $V(x)$ en torno a cada uno de sus puntos de equilibrio por medio de su desarrollo en serie de Taylor centrado en dicho punto

$$V(x) \simeq \frac{1}{2} \left.\frac{d^2V}{dx^2}\right|_{x_i} (x - x_i)^2$$

donde se ha omitido una constante que es irrelevante y se ha tenido en cuenta que $V'(x_i) = 0$. Por tanto, para oscilaciones de amplitud suficientemente pequeña en torno a cada punto de equilibrio de $V(x)$ encontramos un oscilador armónico con constante recuperadora

$$V''(x_i) = \left.\frac{d^2V}{dx^2}\right|_{x_i}$$

y por tanto

$$\omega_i = \sqrt{V''(x_i)/m}, \qquad i = 1, 2, ...$$

donde $x_i$ ($i = 1, 2, ...$) es el conjunto de puntos de equilibrio de $V(x)$.

*input*: Masa $m$ de la partícula, función $V(x)$, variable $x$, intervalo $[x_1, x_2]$ de variación de la variable $x$.

*output*: Lista de frecuencias $\omega_i$.

## Ejercicio 4

Para el mismo sistema considerado en los ejercicios precedentes, dadas las condiciones iniciales $x(0) = x_0$, $p(0) = p_0$, y el intervalo finito de tiempo $t \in [0, t_f]$, calcule la evolución temporal del sistema en dicho intervalo, resolviendo para ello el sistema de ecuaciones diferenciales ordinarias

$$\frac{dx}{dt} = \frac{p}{m}, \qquad \frac{dp}{dt} = -V', \qquad x(0) = x_0, \qquad p(0) = p_0$$

mediante el método de Runge-Kutta (empleando el comando `rk` del Maxima).

A partir de esta solución numérica represente gráficamente los resultados en el espacio de configuraciones (gráficas de $x(t)$ y $p(t)$ frente a $t$) y en el espacio de fases (gráfica de $p(t)$ frente a $x(t)$) del sistema, guardando estas gráficas en archivos EPS que podamos emplear posteriormente.

*input*: Masa $m$ de la partícula, función $V(x)$, variable $x$, valores iniciales para la posición ($x_0$) y el momento ($p_0$) en $t = 0$, valor final del tiempo ($t_f$) para la integración numérica.

*output*: Solución numérica $x(t)$ y $p(t)$ con $t \in [0, t_f]$.
Archivo EPS con la gráfica de $x(t)$ frente a $t$ con $t \in [0, t_f]$.
Archivo EPS con la gráfica de $p(t)$ frente a $t$ con $t \in [0, t_f]$.
Archivo EPS con la gráfica de $p(t)$ frente a $x(t)$ con $t \in [0, t_f]$.

## Ejercicio 5

El sistema dinámico con el que estamos trabajando en estas PECs es conservativo, es decir, existe una función *energía total* que se mantiene constante a lo largo de la evolución temporal. En este caso esta función energía total está dada por la suma de las energías cinética y potencial de la partícula

$$E = \frac{p^2}{2m} + V(x)$$

y es muy sencillo comprobar que se mantiene constante.

Dado el potencial $V(x)$ y los intervalos de valores $[x_1, x_2]$ y $[p_1, p_2]$ para posiciones y momentos, escriba una función que genera por un lado la gráfica de la energía total del sistema $E$ frente a $x$ y $p$, y por otro el mapa de fases del sistema, representando para ello las curvas de nivel de la función energía total.

Para la primera de las gráficas que se pide en este ejercicio puede usar, p. ej., la función `plot3d`, o si lo prefiere `wxplot3d`, para la segunda se puede usar la función `contour_plot`, o análogamente `wxcontour_plot`.

Para la segunda gráfica aparece la cuestión de ¿cuántas curvas de nivel de $E$ consideramos? Para resolver esta cuestión hay varias posibilidades: la más sencilla es incluir un parámetro adicional en la función mediante el cual especifiquemos el número de curvas que queremos ver.

Escriba la función de modo que ambas gráficas queden almacenadas en archivos con formato EPS, de modo que podamos emplearlas posteriormente.

*input*: Nombres de las variables de posición y momento y sus respectivos intervalos de variación: $x$, $[x_1, x_2]$, $p$, $[p_1, p_2]$.
Masa $m$ de la partícula, función $V(x)$.
Número $n$ de curvas $E =$ cte a representar en la segunda gráfica.

*output*: Archivo EPS con la gráfica de $E(x, p)$ frente a $x \in [x_1, x_2]$ y $p \in [p_1, p_2]$.
Archivo EPS con la gráfica de las $n$ curvas de nivel $E(x, p)$ con $x \in [x_1, x_2]$ y $p \in [p_1, p_2]$.

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
   junio de 2014
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2013-2014

        Este archivo contiene el código en Maxima de las funciones pedidas
            en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 17.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2014-J-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2014-J-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";
```

```
        El input/output de todas las funciones siguientes es el indicado
           en el enunciado del examen.
*/

/*      EJERCICIO 1    */

funcion1(V, x, intervalo) := block( [nombrearchivo],

        /*
                Realizamos la gráfica de la función potencial "V"
                   dependiente de la variable "x" que toma valores en "
                   intervalo". Emplearemos la variable local "nombrearchivo
                   " para definir el nombre del archivo donde vamos a
                   guardar la gráfica de V, tal y como se indica a
                   continuación.
                El "path" empleado por maxima por defecto para alojar
                   archivos es el "home" del usuario. El valor de dicho
                   path está indicado en la variable maxima_tempdir. Vamos
                   a emplear la variable local "nombrearchivo" para indicar
                    el nombre completo del archivo eps en el que queremos
                   que se aloje la gráfica de V(x), que llamaremos "
                   potencial.eps". En este caso, en lugar del "home" del
                   usuario vamos a pedirle al maxima que guarde dicho
                   archivo en el escritorio, el cual en este ordenador es
                   el directorio "Desktop" (en otro ordenador posiblemente
                   habrá que modificar este nombre).
        */

        nombrearchivo : concat( maxima_tempdir, "/Desktop/potencial.eps" )
           ,

        plot2d( V, [ x, intervalo[1], intervalo[2] ], [psfile,
           nombrearchivo] )

)$

/*      EJERCICIO 2    */

funcion2(V, x, intervalo) := block(

                        [Vp, soluciones, i, epsilon, numvaloresini, mallado,
                            solucionesnuevas, contador, inc],

        /*
                Calculamos la derivada del potencial y la guardamos en la
                   valiable local "Vp"
        */
```

```
Vp : diff(V, x),

/*
       Por medio de solve intentamos resolver analíticamente Vp =
          0, con esto obtenemos la lista de expresiones x = sol_i,
           donde las sol_i serán soluciones en aquellos casos en
          que Vp = 0 pueda resolverse de forma analítica, mientras
           que serán expresiones de x en caso contrario.
       Almacenamos esta lista de expresiones en la variable local
          "soluciones"
*/

soluciones : solve(Vp = 0, x),

/*
       a continuación sustituyendo la variable por cada una de las
           soluciones encontradas previamente calculamos la lista
          de soluciones pedida.
*/

soluciones : makelist( sublis( [soluciones[i]], x ), i, 1, length(
   soluciones) ),

/*
       Verificamos si las soluciones calculadas son verdaderas
          soluciones o expresiones dependientes de x. En el primer
           caso el problema está resuelto, en el segundo hay que
          resolverlo de manera numérica. Para verificar si las
          soluciones obtenidas son verdaderas soluciones basta con
           verificar si lo es la primera de ellas.
*/

if ( diff(soluciones[1], x) = 0 ) then (

       print("las soluciones analíticas son verdaderas soluciones,
           problema 2 resuelto")

) else (

       print("las soluciones analíticas son funciones de la
          variable, lanzamos cálculo numérico"),

       load ( newton1 ),

       /*
              El método de Newton va ha encontrar una solución
                 aproximada, no exacta, por eso necesitamos
                 definir una "tolerancia" que llamaremos "epsilon
```

```
            " (ver tema 6). Suponemos que hemos encontrado
            una solución con un grado de aproximación
            aceptable cuando Vp(x) < epsilon.
        Tomamos 10^(-8) como un valor aceptable para la
            tolerancia.
*/

epsilon : 10^(-9),

/*
        Un valor de epsilon pequeño conduce a resultados num
            éricos muy precisos, pero con un tiempo de cá
            lculo que puede ser elevado, y viceversa.
*/

/*
        Para lanzar el método de Newton vamos a aplicar la
            estrategia sugerida en el foro de la asignatura.
             Inicialmente tomamos como valores de partida
            para lanzar el método de Newton a 3 valores de
            la variable "x", dados por los límites del
            intervalo de trabajo junto con el valor medio de
             dicho intervalo. Para ello empleamos las
            siguientes variables locales:

                numvaloresini = número de valores de x
                    empleados para lanzar el método de Newton

                mallado = lista valores de x empleados para
                    lanzar el método de Newton
*/

numvaloresini : 3,

mallado : makelist(

        intervalo[1] + (intervalo[2] - intervalo[1]) * (i -
            1) / (numvaloresini - 1)

        , i, 1, numvaloresini, 1 ),

/*
        Calculamos la lista de soluciones que obtenemos al
            lanzar el método de Newton a partir de ese
            conjunto de valores iniciales, guardamos el
            resultado en la variable local "soluciones"
*/
```

```
soluciones : makelist( newton(Vp, x, mallado[i], epsilon),
   i, 1, numvaloresini ),

/*
       Con el procedimiento anterior es posible que el mé
          todo de Newton haya encontrado la misma solución
           más de una vez, de esta lista de soluciones só
          lo nos interesan las que sean distintas. Para
          eliminar las soluciones repetidas en esta lista
          hacemos lo siguiente:

       paso 1.
              Ordenamos la lista de soluciones de menor a
                 mayor con el comando "sort".
*/

soluciones : sort (soluciones),

/*     paso 2.
              Asumimos que dos soluciones son en realidad
                 la misma cuando difieran en una cantidad
                 inferior a la tolerancia "epsilon" con
                 que han sido calculadas. Para implementar
                  esto en Maxima recorremos la lista una
                 vez ordenada, comparando cada elemento
                 con el anterior, de forma que cuando
                 estos elementos difieran en menos de "
                 epsilon" definimos que dichos elementos
                 son iguales.
*/

for i : 2 thru length(soluciones) do if (

       abs( soluciones[i] - soluciones[i - 1] ) < epsilon

       ) then soluciones[i] : soluciones[i - 1],

/*     paso 3.
              Una vez hemos definido como iguales a los
                 elementos que difieren en menos de
                 epsilon, empleamos el comando "unique"
                 para quedarnos sólo con los elementos
                 distintos, eliminando repeticiones.
*/

soluciones : unique( soluciones),

/*
```

```
        Hasta aquí hemos calculado una lista de 3 soluciones
            numéricas, de la que posteriormente hemos
           eliminado las soluciones repetidas, pero
           ¿tenemos ya todas las soluciones de Vp = 0 o só
           lo unas pocas? Para verlo sub-dividimos el
           intervalo de trabajo y volvemos a lanzar el mé
           todo de Newton a partir de una rejilla de
           valores iniciales de x más fina que la empleada
           más arriba. Si al hacer eso volvemos a encontrar
            las soluciones de antes asumimos que ya tenemos
            todas las soluciones de Vp = 0, en caso
           contrario continuamos sub-dividiendo el
           intervalo y re-calculando soluciones hasta que
           las soluciones encontradas en dos iteraciones
           sucesivas coincidan, en cuyo caso asumimos que
           ya hemos encontrado todas las soluciones que
           buscábamos.

        A la hora de comparar 2 soluciones para ver si son o
            no iguales hay que tener en cuenta que todas
           las soluciones que estamos encontrando son
           aproximadas, no exactas, de modo que cuando 2
           soluciones difieran en una cantidad menor que la
            tolerancia "epsilon" definida más arriba
           asumiremos que, en realidad, son la misma soluci
           ón.

        Este detalle ya ha sido tenido en cuenta más arriba
           para eliminar soluciones repetidas, y ahora
           tendremos que volver a tenerlo en cuenta para
           verificar si las soluciones encontradas en una
           iteración coinciden con las encontradas en la
           iteración siguiente.

        Vamos a guardar las soluciones encontradas en la "
           iteración siguiente" en la variable local "
           solucionesnuevas", cuyo valor fijamos
           inicialmente en "soluciones"
*/

solucionesnuevas : soluciones,

/*      La implementación del método se resume en estos
    pasos:
        paso1.
                Sub-dividimos de manera iterativa la lista
                   de valores iniciales empleada para lanzar
                    el método de Newton incrementando en "
```

```
                    inc" la variable "numvaloresini".
            paso 2.
                    Asignamos a "soluciones" el valor de "
                       solucionesnuevas" del paso anterior.
                    Calculamos la nueva lista de soluciones,
                       ordenada y sin repeticiones igual que
                       hemos hecho más arriba.
                    Asignamos a "solucionesnuevas" la nueva
                       lista de soluciones.
            paso 3.
                    Repetimos el proceso hasta que el número de
                       elementos de "solucionesnuevas" coincida
                       con el de "soluciones" y además se cumpla
                        que la suma de las diferencias en valor
                       absoluto entre las soluciones de una
                       iteración y la iteración siguiente

                           sum( abs( solucionesnuevas[i] -
                              soluciones[i] ), i, ... )

                    sea menor que epsilon. Para hacer esta suma
                       de diferencias hay que tener en cuenta
                       que las listas "solucionesnuevas" y "
                       soluciones" pueden no tener la misma
                       longitud, de modo que sumamos para "i"
                       entre 1 y el menor de las longitudes de
                       ambas listas.
                    Para lanzar el proceso iterativo
                       condicionado anterior empleamos la
                       instrucción "for ... while". Para llevar
                       la cuenta del número de iteraciones que
                       realizamos definimos un "contador", cuyo
                       valor for incrementa en una unidad en
                       cada iteración.
*/

for contador : 1 step 1 while (

        ( contador = 1 ) or

        ( length(solucionesnuevas) # length(soluciones) ) or

        sum( abs( solucionesnuevas[i] - soluciones[i] )

                , i, 1, min(length(solucionesnuevas), length
                   (soluciones)) ) > epsilon

) do (
```

```
                print("iteración = ", contador),

                /* paso 1. */

                inc : 5,

                numvaloresini : numvaloresini + inc,

                mallado : makelist(

                       intervalo[1] + (intervalo[2] - intervalo[1])
                           * (i - 1) / (numvaloresini - 1)

                       , i, 1, numvaloresini, 1 ),

                /* paso 2. */

                soluciones : solucionesnuevas,

                print("soluciones = ", soluciones),

                /* calculamos la nueva lista de soluciones */

                solucionesnuevas : makelist( newton(Vp, x, mallado[i
                   ], epsilon), i, 1, numvaloresini ),

                /* ordenamos y eliminamos repeticiones, igual que
                   antes */

                solucionesnuevas : sort (solucionesnuevas),

                for i : 2 thru length(solucionesnuevas) do if (

                       abs( solucionesnuevas[i] - solucionesnuevas[
                          i - 1] ) < epsilon

                       ) then solucionesnuevas[i] :
                          solucionesnuevas[i - 1],

                solucionesnuevas : unique( solucionesnuevas)

        ),

        soluciones : solucionesnuevas

),
```

```
        /*      Finalmente retornamos el valor de la lista de soluciones

                OBSERVACIÓN: Tanto el procedimiento numérico que hemos
                   empleado como el analítico pueden generar soluciones que
                    estén fuera del intervalo de trabajo indicado.
         */

        soluciones

)$

/*      EJERCICIO 3    */

funcion3(V, x, intervalo, masa) := block( [Vpp, soluciones, i, wf],

        /*
                Calculamos la lista de puntos críticos de V(x) por medio de
                    la función del ejercicio anterior
        */

        soluciones : funcion2(V, x, intervalo),

        /*
                De la anterior lista de soluciones sólo nos interesan las
                   que correspondan a puntos de equilibrio, es decir, los m
                   ínimos locales de V, que serán aquéllas para las que la
                   segunda derivada del potencial sea positiva.
                Las soluciones con V'' negativo corresponden a puntos de
                   equilibrio inestable, y las soluciones con V'' igual a
                   cero serán generalmente puntos de inflexión, aunque
                   también podrían ser máximos o mínimos locales si V''' es
                    también nula (ver p. ej. el caso y = x^4 en x=0), esos
                   casos no serán tenidos en cuenta en este ejercicio ya
                   que en ellos la aproximación para las oscilaciones de
                   pequeña amplitud dada en el enunciado no es válida.
        */

        /*      Calculamos la segunda derivada del potencial y la guaramos
                en la variable local Vpp: */

        Vpp : diff(V, x, 2),

        /*
                A continuación recorremos la lista de soluciones y sólo en
                   aquellos casos en que Vpp sea positivo calculamos la
                   frecuencia correspondiente, añadiendo los valores del mí
                   nimo local y la frecuencia a la lista de valores "xf",
                   que inicializamos como una lista vacía.
```

```
        */

        xf : [],

        for i : 1 thru length(soluciones) do if ( sublis( [x=soluciones[i
           ]], Vpp ) > 0

            ) then xf : append( xf, [[soluciones[i], sqrt( sublis( [x=
               soluciones[i]], Vpp ) / masa )]] ),

        /*
               Retornamos la lista de parejas de valores [mínimo_local_i,
                  frecuencia_i]
        */

        xf

)$

/*      EJERCICIO 4    */

funcion4(V, x, masa, xini, pini, tfin) := block( [ecuaciones, p, t, paso,
    data, tlist, xlist, plist, nombrearchivo],

        /*
               Para escribir esta función se han tenido en cuenta los
                  comentarios sobre el uso de "rk" que figuran en el
                  ejercicio 1 de la PEC de junio de 2011, junto con el
                  tema 8 de los apuntes.
               En primer lugar definimos "ecuaciones" como la lista de
                  ecuaciones que vamos a pasarle a rk
        */

        ecuaciones : [p/masa, -diff(V, x)],

        /*      definimos el valor del paso inicial de integración para el
           método de Runge-Kutta */

        paso : 0.01,

        /*      Resolvemos el sistema numéricamente */

        load("dynamics"),

        data : rk( ecuaciones, [x, p], [xini, pini], [t, 0, tfin, paso] ),

        /*      Para las representaciones gráficas aplicamos el
           procedimiento del tema 8 de los apuntes  */
```

```
data : apply( matrix, data ),

tlist : transpose ( data )[1],
xlist : transpose ( data )[2],
plist : transpose ( data )[3],

/*      Generamos los nombres de archivo para las gráficas según el
    procedimiento del 1er ejercicio */

/*      Gráfica de posición frente al tiempo */

nombrearchivo : concat( maxima_tempdir, "/Desktop/posicion-vs-t.
   eps" ),

plot2d( [ [ discrete, tlist, xlist ] ], [psfile, nombrearchivo] ),

/*      Gráfica de momento frente al tiempo */

nombrearchivo : concat( maxima_tempdir, "/Desktop/momento-vs-t.eps
   " ),

plot2d( [ [ discrete, tlist, plist ] ], [psfile, nombrearchivo] ),

/*      Gráfica de momento frente a posición */

nombrearchivo : concat( maxima_tempdir, "/Desktop/momento-vs-
   posicion.eps" ),

plot2d( [ [ discrete, xlist, plist ] ], [psfile, nombrearchivo] ),

/*
       Finalmente, aunque el enunciado no lo pedía, retornamos el
          resultado de la integración numérica del sistema,
          contenido en la variable local "data".
 */

[tlist, xlist, plist]

/*
       Esto último aconseja ejecutar esta función con una
          instrucción "$" al final
       (por ejemplo resultados : funcion4(estaV, x, 99, 1, 20,
          100)$)
       ya que en general [tlist, xlist, plist] tendrá una extensió
          n demasiado grande como para visualizarse por pantalla.
*/
```

```
)$

/*     EJERCICIO 5    */

funcion5(V, x, intervalo_x, p, intervalo_p, masa, numcurvas) := block( [E
   , nombrearchivo, opcionnumcurvas],

       /*     Cambiamos el valor por defecto del parámetro "grid", para
          obtener gráficas más precisas  */

       set_plot_option([grid, 75, 75]),

       /*     Definimos la función de energía total */

       E : ( p^2 / (2*masa) ) + V,

       print("energía = ", E),

       /*     Generamos el nombre del archivo como en el ejercicio
          anterior.   */

       nombrearchivo : concat( maxima_tempdir, "/Desktop/energia-vs-
          posicion-momento.eps" ),

       plot3d( E,
              [ x, intervalo_x[1], intervalo_x[2] ],
              [ p, intervalo_p[1], intervalo_p[2] ],
              [ psfile, nombrearchivo ]
        ),

       /*     Gráfica de curvas de nivel de E según procedimiento
          explicado en foro de la asignatura. */

       nombrearchivo : concat( maxima_tempdir, "/Desktop/energia-curvas-
          de-nivel-vs-posicion-momento.eps" ),

       opcionnumcurvas : concat( "set cntrparam levels ", string(
          numcurvas) ),

       contour_plot( E,
              [ x, intervalo_x[1], intervalo_x[2] ],
              [ p, intervalo_p[1], intervalo_p[2] ],
              [ gnuplot_preamble, opcionnumcurvas ],
              [ psfile, nombrearchivo ]
       ),

       /*
              Finalmente, aunque el enunciado no lo pedía, retornamos la
```

```
                función energía total, contenida en la variable local "E
                ".
        */

       E

)$
```

## 9.8. Curso 2013-2014, convocatoria de septiembre

### Ejercicio 1

En este ejercicio vamos a analizar uno de los sistemas de ecuaciones diferenciales ordinarias (EDOs) más famosos que dan lugar a *dinámica caótica*: el **sistema de Lorenz**. Este sistema está definido por las siguientes EDOs de primer orden

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= \sigma(y-x)\\ \frac{\mathrm{d}y}{\mathrm{d}t} &= x(\rho-z)-y\\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy-\beta z\end{aligned}$$

junto con las condiciones iniciales

$$x(t=0)=x_0, \qquad y(t=0)=y_0, \qquad z(t=0)=z_0$$

donde los parámetros $\sigma$, $\rho$ y $\beta$ son números reales.

Para determinados valores de los parámetros $\sigma$, $\rho$ y $\beta$ este sencillo conjunto de 3 EDOs no lineales acopladas tiene soluciones *caóticas*. Esto significa que el comportamiento de algunas soluciones depende fuertemente de la condición inicial de donde parten, de tal forma que 2 soluciones inicialmente muy próximas se alejan entre sí rápidamente en su evolución temporal.

Para el primer ejercicio de esta PEC escriba una función que resuelva el anterior sistema de EDOs numéricamente por medio del algoritmo de Runge-Kutta, empleando para ello la función `rk` del maxima, a partir de unas condiciones iniciales dadas y de unos valores concretos para los parámetros del sistema $\sigma$, $\rho$ y $\beta$.

*input*: Valores de los parámetros del sistema $\sigma$, $\rho$ y $\beta$.
Condiciones iniciales $x_0 = x(t=0)$, $y_0 = y(t=0)$, $z_0 = z(t=0)$.
Valor final del tiempo ($t_f$) considerado en la solución numérica.

*output*: Solución numérica para $x(t)$, $y(t)$ y $z(t)$ con $t \in [0, t_f]$ calculada por medio del método de Runge-Kutta.

## Ejercicio 2

La mejor forma de ver lo que significa la frase *dependencia fuerte respecto de condiciones iniciales* es visualizando la evolución temporal de algunas soluciones correspondientes a condiciones iniciales próximas. Para ello, programe una función en Maxima que genere las gráficas de $x(t)$, $y(t)$ y $z(t)$ con $t \in [0, t_f]$ obtenidas a partir de un cierto conjunto de $N$ valores iniciales

$$\left\{x_0^i, y_0^i, z_0^i\right\}_{i=1}^N$$

y las guarde en archivos EPS.

*input*: Valores de los parámetros del sistema $\sigma$, $\rho$ y $\beta$ (los mismos para todas las condiciones iniciales).
Conjunto de $N$ condiciones iniciales $\{x_0^i, y_0^i, z_0^i\}_{i=1}^N$.
Valor final del tiempo ($t_f$) considerado en la solución numérica (el mismo para todas las condiciones iniciales).

*output*: Archivo EPS con todas las gráficas de las soluciones $x_i(t)$ ($t \in [0, t_f]$) para $i = 1, \ldots, N$, obtenidas por medio de la función del Ejercicio 1.
Archivo EPS con todas las gráficas de las soluciones $y_i(t)$ ($t \in [0, t_f]$) para $i = 1, \ldots, N$, obtenidas por medio de la función del Ejercicio 1.
Archivo EPS con todas las gráficas de las soluciones $z_i(t)$ ($t \in [0, t_f]$) para $i = 1, \ldots, N$, obtenidas por medio de la función del Ejercicio 1.

## Ejercicio 3

Aparte de visualizar las soluciones como funciones del tiempo también es interesante ver el comportamiento de estas soluciones en el espacio de configuraciones, es decir, en el espacio $(x, y, z)$. Para ello, programe una función que realice la gráfica paramétrica de una solución $(x(t), y(t), z(t))$ (con $t \in [0, t_f]$) obtenida en el Ejercicio 1.

*input*: Como en el Ejercicio 1.

*output*: Archivo EPS con la gráfica paramétrica de $(x(t), y(t), z(t))$ (con $t \in [0, t_f]$).

Esta función le servirá para generar la gráfica del conocido *atractor de Lorenz*.

# Ejercicio 4

Uno de los indicadores de comportamiento caótico en sistemas dinámicos es la función de autocorrelación. En el caso de una función del tiempo $f(t) \in \mathbb{R}$ con media nula, la función de autocorrelación se suele definir como

$$A(t) = \int_{-\infty}^{\infty} f(\tau) f(\tau - t)\, d\tau$$

$\star$ ¿Cómo nos indica esta función si $f(t)$ es o no caótica?

En primer lugar es importante darse cuenta de que la gráfica de la función $f(\tau - t)$ (considerada como función de la variable $\tau$, con $t$ fijo) es igual a la gráfica de de la función $f(\tau)$ pero desplazada en una magnitud igual a $t$ (hacia la derecha si $t > 0$, hacia la izquierda si $t < 0$). Si la función $f(t)$ *no* es caótica, entonces será periódica o tendrá algún patrón reconocible que se repita en el tiempo, al menos de manera aproximada. En ese caso habrá algún (o algunos) valores de $t$ para los que $f(\tau - t)$ coincida al menos aproximadamente con $f(\tau)$, de modo que su producto será positivo y por tanto dará una contribución positiva a la integral. Para los valores de $t$ en los que la función desplazada $f(\tau - t)$ sea muy distinta de $f(\tau)$, es de esperar que las contribuciones positivas no sean significativamente mayores que las negativas, cancelándose ambas mutuamente al realizar la integral. Por tanto, para funciones *caóticas* es de esperar que la función de autocorrelación decaiga a cero rápidamente (exponencialmente) con $t$, mientras que para señales *no caóticas* $A(t)$ será periódica. En la práctica es habitual que tengamos señales periódicas contaminadas con *ruido*, en ese caso la función de autocorrelación decrecerá a cero con una velocidad que será tanto mayor cuanto mayor sea la intensidad del ruido.

En esta PEC vamos a calcular la autocorrelación de las funciones $x(t)$, $y(t)$ y $z(t)$ dadas por la solución del sistema de Lorenz. Estas funciones están definidas en el intervalo finito $t \in [0, t_f]$, que es el intervalo en el que hemos realizado la integración numérica. En ese caso es conveniente modificar la definición anterior de la siguiente forma

$$A[x](t) = \frac{1}{t_f} \int_0^{t_f/2} [x(\tau) - \overline{x}]\, [x(\tau + t) - \overline{x}]\; d\tau, \qquad t \in [0, \frac{t_f}{2}]$$

donde

$$\overline{x} = \frac{1}{t_f} \int_0^{t_f} x(t)\, dt$$

es el valor promedio de $x(t)$ en el intervalo considerado, con definiciones análogas para la autocorrelación de $y(t)$ o de $z(t)$. Observe que, de acuerdo a esta definición, para calcular la autocorrelación $A[x](t)$ en el intervalo $t \in [0, \frac{t_f}{2}]$ es necesario conocer la función $x(t)$ en el intervalo de duración *doble* $t \in [0, t_f]$.

Programe una función en Maxima que calcule la autocorrelación de una función $x(t)$ con $t \in [0, t_f]$ según esta definición (calculando la integral de manera numérica), y que genere la correspondiente gráfica en un archivo EPS.

*input*: Función $x(t)$, valor final de $t$ ($t_f$).

*output*: Gráfica de $A[x](t)$ con $t \in [0, \frac{t_f}{2}]$, calculada según la definición anterior.

## Ejercicio 5

La función programada en el ejercicio anterior es aplicable para calcular la autocorrelación de *funciones continuas*. Sin embargo lo que obtenemos al integrar numéricamente un sistema de ODEs no es una función continua, sino un conjunto de $M$ valores discretos ($x_i \equiv x(t_i)$, $i = 1, \ldots, M$), correspondientes al conjunto de valores discretos de la variable independiente considerados por el algoritmo numérico (en nuestro caso dados por $t_i = (i-1)\,h$, con $i = 1, \ldots, M$, donde $h = t_f/\,(M-1)$ es la distancia entre dos nodos consecutivos del mallado).

Una posibilidad para calcular la autocorrelación de señales discretas es generar la correspondiente función continua $x(t)$ por medio de un algoritmo de interpolación (p. ej. con splines cúbicos) y posteriormente operar con la función interpolada. Otra posibilidad mucho más sencilla (y también más frecuente) es definir una función de autocorrelación aplicable a señales discretas. En ese caso, en lugar de calcular la autocorrelación $A(t)$ como una función continua de la variable $t$, calculamos un conjunto de valores discretos $A_i$ por medio de

$$A_i = \frac{1}{M} \sum_{j=1}^{M/2} \left[x_j - \overline{x}\right] \left[x_{i+j} - \overline{x}\right], \qquad i = 1, \ldots, M/2$$

siendo $M$ el número de valores discretos $x_i$ y $\overline{x}$ su valor promedio $\overline{x} = \frac{1}{M} \sum_{j=1}^{M} x_j$.

Programe una función en Maxima para calcular la autocorrelación de una función $x(t)$ arbitraria, con $t \in [0, t_f]$, aproximada por medio del conjunto de valores discretos $x_i = x(t_i)$ ($i = 1, \ldots, M$) según la definición anterior, y genere la correspondiente gráfica en un archivo EPS.

*input*: Conjunto de valores $x_i = x(t_i)$ ($i = 1, \ldots, M$).

*output*: Gráfica de $A_i$ con $i = 1, \ldots, M/2$, calculada según la definición anterior.

## Observación

Como ya hemos comentado, no todos los valores de los parámetros $\sigma$, $\rho$ y $\beta$ llevan a un comportamiento caótico en el sistema de Lorenz. Para ver un caso concreto interesante que *sí* es caótico puede considerar los valores

$$\sigma = 10, \qquad \beta = 8/3, \qquad \rho = 28$$

que son los considerados inicialmente por Lorenz.

Para visualizar el fenómeno de la dependencia fuerte respeto de condiciones iniciales estudie cómo se comportan las soluciones que parten de puntos próximos a

$$x_0 = -8, \qquad y_0 = 8, \qquad z_0 = 27$$

Por ejemplo, considere un número de unas 5 trayectorias, con condiciones iniciales dadas por los valores anteriores más una pequeña perturbación y vea cómo acaban separándose unas de otras. De todas formas, recuerde que el objetivo de las funciones que se piden en los ejercicios no es calcular una trayectoria concreta, sino programar una serie de funciones que sean útiles para estudiar muchos casos. De hecho, con un mínimo de trabajo adicional los ejercicios que se piden en estas PECs le resultarán útiles para estudiar otros sistemas caóticos, como por ejemplo el sistema de Hénon-Heiles o tantos otros.

Para más información sobre el sistema de Lorenz puede consultar:
http://en.wikipedia.org/wiki/Lorenz_system

## Solución

```
/*      PRUEBA EVALUABLE DE PROGRAMACION EN WXMAXIMA, convocatoria de
   septiembre de 2014
        Física Computacional I, 1er curso del grado en física
        Dept. de Física Matemática y de Fluidos, UNED, curso 2013-2014

        Este archivo contiene el código en Maxima de las funciones pedidas
            en la prueba evaluable.
        Para escribirlo se ha empleado el editor de textos "KWrite" en un
           PC con Linux Fedora 17.

        Para cargar el archivo desde wxMaxima se puede ejecutar:

                batchload( concat( path, "2014-S-R.mc" ) );

        donde la variable path es una cadena que indica la localización
           del archivo
        2014-S-R.mc en el disco, p. ej. path podría ser algo parecido a

                path : "/home/usuario/Fisica-Computacional-1/Maxima/examen
                   /";

        El input/output de todas las funciones siguientes es el indicado
           en el enunciado del examen.
*/
```

```
/*      EJERCICIO 1    */

/*      Escribimos una función que resuelve el sistema de 3 EDOs del
   modelo de Lorenz.
        Input:
                sigma           =       Parámetro sigma del sistema de Lorenz
                rho             =       Parámetro rho del sistema de Lorenz
                beta            =       Parámetro beta del sistema de Lorenz
                conini          =       Lista de condiciones iniciales: x(0),
                    y(0), z(0)
                tfin            =       Valor máximo de la variable
                   independiente (tfin)
                paso            =       Parámetro de paso a emplear en el
                   algoritmo de Runge-Kutta.

        Para un ejemplo de funcionamiento de esta función hacer:

                ejemplo : [10, 28, 8/3];
                conini0 : [-8, 8, 27];
                sol : funcion1(ejemplo[1], ejemplo[2], ejemplo[3], conini0,
                    10)$
                plot2d([discrete, sol[1], sol[2]]);
*/

funcion1(sigma, rho, beta, conini, tfin) := block( [x, y, z, t, aux, p],

        /*      Definimos las variables del problema [x, y, z, t] como
           variables locales,
                para evitar posibles errores debidos al uso de variables
                   con esos nombres
                fuera de esta función.
                Guardamos las lista de EDOs (lado derecho) a resolver en la
                    variable local aux
                (las EDOs son d[x, y, z]/dt = aux).
        */

                aux : [ sigma * (y - x), x * (rho - z) - y, x * y - beta *
                   z ],

        /*      Resolvemos el sistema por medio de rk, con paso dado por la
            variable local p,
                guardamos la solución en la variable local aux.
        */

                p: 1/100,

                aux : rk(aux, [x, y, z], conini, [t, 0, tfin, p]),
```

```
        /*      La solución numérica que nos proporciona rk es una lista de
            listas de valores.
                El conveniente convertir el output de rk a formato
                    matricial, para ello convertimos
                esta lista en una matriz (ver soluciones tema 2.10 y examen
                    2011)
        */

                aux : transpose( apply(matrix, aux) )
)$


/*      EJERCICIO 2    */

/*      Escribimos una función que resuelve el sistema de 3 EDOs del
    modelo de Lorenz para una colección de datos iniciales.
        Para ello llamamos a la funcion1 con cada una de las condiciones
            iniciales de la lista "coninilist"
        Input:
                sigma           =       Parámetro sigma del sistema de Lorenz
                rho             =       Parámetro rho del sistema de Lorenz
                beta            =       Parámetro beta del sistema de Lorenz
                coninilist      =       Lista de listas de condiciones
                    iniciales: [x(0), y(0), z(0)]_i
                tfin            =       Valor máximo de la variable
                    independiente (tfin)
                paso            =       Parámetro de paso a emplear en el
                    algoritmo de Runge-Kutta.

        Para un ejemplo de funcionamiento de esta función hacer:

                ejemplo : [10, 28, 8/3];      ---> estos son los valores de
                    los parámetros
                conini0 : [-8, 8, 27];        ---> esta es la condición
                    inical básica
                incremento : 0.001;           ---> este es el incremento
                    relativo aplicado a cada variable
                conini1 : [
                                conini0 * [1-incremento, 1-incremento, 1-
                                    incremento],
                                conini0 * [1-incremento, 1-incremento, 1+
                                    incremento],
                                conini0 * [1-incremento, 1+incremento, 1-
                                    incremento],
                                conini0 * [1-incremento, 1+incremento, 1+
                                    incremento],
                                conini0 * [1, 1, 1],
                                conini0 * [1+incremento, 1-incremento, 1-
```

```
                                incremento],
                            conini0 * [1+incremento, 1-incremento, 1+
                                incremento],
                            conini0 * [1+incremento, 1+incremento, 1-
                                incremento],
                            conini0 * [1+incremento, 1+incremento, 1+
                                incremento]
                    ];                      ---> hacemos una lista de
                        condiciones iniciales dada por un cubo
                                            con centro en conini0 y con vé
                                                rtices obtenidos aplicando
                                                un
                                            desplazamientondo relativo de
                                                1 (+/-)incremento a cada
                                                varible

            funcion2(ejemplo[1], ejemplo[2], ejemplo[3], conini1, 10)$
*/

funcion2(sigma, rho, beta, coninilist, tfin) := block( [aux, soluciones,
   nombrearchivo],

      /*     Asignamos a la variable local conini cada una de las
         condiciones iniciales incluidas en la lista
             coninilist de manera secuencial.
             Llamamos a la función del ejercicio 1 para resolver el
                correspondiente sistema.
             Almacenamos el resultado en la variable local soluciones.
             Repetimos el proceso hasta agotar la lista de condiciones
                iniciales.
             Una vez tenemos la lista de soluciones correspondiente a la
                 lista de condiciones iniciales suministrada hacemos las
                 gráficas
      */

      soluciones : [],

      for i : 1 thru length(coninilist) step 1 do (

             soluciones : append( soluciones, [ funcion1(sigma, rho,
                beta, coninilist[i], tfin) ] )

      ),

      /*     Gráfica de x_i vs t ---> guardada en el archivo tipo "EPS"
         x-vs-t.eps, localizado en "Desktop"
      */
```

```
        nombrearchivo : concat( maxima_tempdir, "/Desktop/x-vs-t.eps" ),

        /*      Almacenamos en la variable local aux la lista de conkuntos
           de puntos a
                representar, empleamos wxplot2d para visualizar la gráfica
                   dentro de la
                sesión de wxmaxima y plot2d para generar el archivo eps con
                    la gráfica.
                Se opera análogamente para y(t) vs t, z(t) vs t.
        */

        aux : makelist( [discrete, soluciones[i][1], soluciones[i][1+1]],
           i, 1, length(coninilist), 1),

        wxplot2d( aux ),
        plot2d( aux, [psfile, nombrearchivo] ),

        /*      Gráfica de y_i vs t ---> guardada en el archivo tipo "EPS"
           y-vs-t.eps, localizado en "Desktop"
        */

        nombrearchivo : concat( maxima_tempdir, "/Desktop/y-vs-t.eps" ),

        aux : makelist( [discrete, soluciones[i][1], soluciones[i][1+2]],
           i, 1, length(coninilist), 1),

        wxplot2d( aux ),
        plot2d( aux, [psfile, nombrearchivo] ),

        /*      Gráfica de z_i vs t ---> guardada en el archivo tipo "EPS"
           z-vs-t.eps, localizado en "Desktop"
        */

        nombrearchivo : concat( maxima_tempdir, "/Desktop/z-vs-t.eps" ),

        aux : makelist( [discrete, soluciones[i][1], soluciones[i][1+3]],
           i, 1, length(coninilist), 1),

        wxplot2d( aux ),
        plot2d( aux, [psfile, nombrearchivo] )

)$

/*      EJERCICIO 3    */

funcion3(sigma, rho, beta, conini, tfin) := block( [aux, nombrearchivo],

        /* Para un ejemplo de funcionamiento de esta función ejecutar este
```

```
    código

       ejemplo : [10, 28, 8/3];
       conini0 : [-8, 8, 27];
       funcion3(ejemplo[1], ejemplo[2], ejemplo[3], conini0, 50)$
*/

/*     Primero resolvemos el sistema con los datos suministrados
*/

aux : funcion1(sigma, rho, beta, conini, tfin),

/*     Generamos el conjunto de puntos [x(t), y(t), z(t)] con t en
    [0, tfin] a representar
*/

aux : transpose( matrix( aux[1+1], aux[1+2], aux[1+3] ) ),

/*     Generamos la gráfica paramétrica de [x, y, z] y la
   guardamos en un archivo EPS
       para ello hay que emplear la función wxdraw3d del paquete
          draw, ya que plot3d
       no representa conjuntos discretos.
*/

load(draw),

nombrearchivo : concat( maxima_tempdir, "/Desktop/atractor-Lorenz"
    ),

draw3d(
              file_name = nombrearchivo,

              terminal = eps,

              point_type = none,

              points_joined = true,

              color = blue,

              xlabel = "x", ylabel = "y", zlabel = "z",

              xtics = 10, ytics = 10, ztics = 10,

              points(aux)

)
```

```
)$

/*      EJERCICIO 4    */

        /*

        Para la evaluación de las integrales pedidas en este ejercicio
           vamos a emplear la función:

                quad_qags( integrando, variable de integración, límite
                   inferior, límite superior )

        perteneciente al paquete QUADPACK, que se carga automáticamente al
            iniciar wxMaxima.

        La función quad_qags( f(x), x, a, b ) calcula una aproximación num
           érica a la integral definida

                integrate( f(x), x, a, b )

        El valor de la integral se encuentra en el primero de los
           argumentos que devuelve quad_qags( f(x), x, a, b ).

        En la ayuda del wxMaxima pueden leer los detalles sobre los mé
           todos numéricos empleados en QUADPACK
        así como las diversas opciones de la función quad_qags y el
           significado de los demás elementos de la
        lista que genera.

        En este ejercicio nos limitaremos a emplear quad_qags( f(t), t, a,
            b ) sin especificar más opciones
        de modo que los valores de los parámetros que controlan la precisi
           ón de la integración numérica son
        los definidos "por defecto". Esta elección puede modificarse fá
           cilmente en caso de ser necesario.
        */

funcion4(expresion, t, tfin) := block( [media, numpuntos, i, aux, tau,
   nombrearchivo],

        /*      Calculamos el valor medio de "expresión" en el intervalo
           [0, tfin]
        */

                media : (1/tfin) * quad_qags( expresion, t, 0, tfin )[1],

        /*      En el problema se pide evaluar las integrales de manera num
```

```
            érica, esto implica que previamente
                hemos asignado algún valor numérico a la variable t.
                En esta función vamos a calcular los valores de la función
                    de autocorrelación para un conjunto
                de "1 + numpuntos" valores de "t" equiespaciados en [0,
                    tfin/2].
                A continuación definimos el número de valores de "t"
                    considerados.
         */

                numpuntos : 200,

        /*
                Calculamos las "1 + numpuntos" integrales correspondientes
                    a los valores de "t" que vamos a considerar.
                Para ello calculamos en primer lugar la lista de
                    integrandos y a continuación la lista de integrales.
         */

                aux : [],

                for i : 0 thru numpuntos step 1 do (

                        aux : endcons(

                        (subst(tau, t, expresion) - media) * (subst(tau + i
                            *(tfin/2)/numpuntos, t, expresion) - media),

                        aux)

                ),

                /*      Evaluamos las integrales y retornamos la lista [t, A
                    [x](t)]
                */

                aux : makelist( [i*(tfin/2)/numpuntos, (1/tfin) * quad_qags
                    ( aux[1+i], tau, 0, tfin/2 )[1]], i, 0, numpuntos, 1),

                nombrearchivo : concat( maxima_tempdir, "/Desktop/
                    autocorrelacion-continua.eps" ),

                wxplot2d( [discrete, aux] ),

                plot2d( [discrete, aux], [psfile, nombrearchivo] )
)$

/*      EJERCICIO 5    */
```

```
funcion5(puntos) := block( [numpuntos, media, i, j, aux, nombrearchivo],

                /*      La versión discreta de la función de autocorrelación
                     es considerablemente más sencilla
                        de programar que la anterior, y en la práctica es la
                            única que se emplea.
                        El código que sigue a continuación implementa esta
                           función tal y como se define en el
                        enunciado.
                        Para un ejemplo de uso de esta función ver:

                                ejemplo : [10, 28, 8/3];
                                coninit0 : [-8, 8, 27];
                                sol : funcion1(ejemplo[1], ejemplo[2],
                                   ejemplo[3], coninit0, 50)$
                                puntos : sol[2]$
                                funcion5(puntos);
                */

                numpuntos : length(puntos),

                media : sum( puntos[i], i, 1, numpuntos ) / numpuntos,

                aux : [],

                for i : 1 thru floor(numpuntos/2) step 1 do (

                        aux : endcons(

                        [i, sum( (puntos[j] - media) * (puntos[i+j] - media)
                           , j, 1, floor(numpuntos/2) ) / numpuntos],

                        aux)

                ),

                nombrearchivo : concat( maxima_tempdir, "/Desktop/
                   autocorrelacion-discreta.eps" ),

                wxplot2d( [discrete, aux] ),

                plot2d( [discrete, aux], [psfile, nombrearchivo] )

)$
```