ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación :

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

Título del proyecto:

LIBRERÍA GPL C++ PARA LA MANIPULACIÓN DE EXPRESIONES SIMBÓLICAS EN LA DINÁMICA DE SISTEMAS MULTICUERPO

Luis Miguel Arrondo Martínez

Javier Ros Ganuza

Pamplona, Noviembre de 2007

Índice

1 - Introducción

El objetivo de este proyecto, es el desarrollo de una librería GPL C++ para la manipulación de expresiones simbólicas de la dinámica de sistemas multicuerpo, que pueda ser un sustituto del kernel de la aplicación *3D_mec*.

Se trata de una librería informática escrita en el lenguaje *C++*, capaz de definir y trabajar con rapidez y robustez sobre expresiones simbólicas en el campo de la mecánica como pueden ser vectores, tensores, puntos, bases, matrices, expresiones vectoriales y matriciales, además de poder calcular vectores de posición, matrices de cambio de base, velocidades y aceleraciones angulares, etc.

Además la librería tiene la posibilidad de automatizar la exportación de funciones y datos generados en la ejecución de aplicaciones que usen la librería, para así poder hacer uso de estos datos en aplicaciones externas como puede ser, entre otros, aplicaciones de generación de gráficas.

La decisión de escoger el lenguaje C++ para su implementación, ha sido tomada por las siguientes razones que se van a redactar a continuación. Por la perfecta integración que este lenguaje tiene con la librería GiNaC, la cual es base fundamental en el desarrollo de este proyecto. La base de esta perfecta integración es debida a que GiNaC también esta implementada en lenguaje C++. Además, las características propias de este lenguaje orientado a objetos también lo hacen propicio, ya que adicionalmente posee algunas características de la programación que satisfacen el paradigma iterativo, como la posibilidad de la declaración de variables globales a toda la aplicación o posibilidad de uso de punteros para la optimización del uso de los recursos de nuestro ordenador. También posee la posibilidad de la creación de plantillas, lo que ha supuesto una reducción considerable en el número de líneas de código, y alguna característica más que mas adelante podremos ver. Y finalmente la mas importante de las razones es porque este lenguaje de programación orientado a objetos, permite la sobrecarga de operadores, es decir, permite usar y configurar entre otros, los operadores “+”, “-”, “*”, “^”; con lo que podremos crear instrucciones en el programa cliente de forma natural, como si las estuviésemos creando de forma manual con un bolígrafo sobre un papel, con la consiguiente rapidez de asimilación de la sintaxis por parte de los usuarios de la librería.

Como se ha comentado anteriormente, la librería GiNaC, es parte fundamental del proyecto que ahora se presenta, y se trata de una librería bajo licencia GPL de calidad contrastada en el ámbito del algebra computacional y que haciendo uso de sus objetos y funciones, ha hecho que la librería sea mucho mas potente y fiable además del ahorro de esfuerzo que hubiese supuesto crear una librería de similares características.

Una vez introducidos algunos conceptos sobre la librería, vamos a describir con detenimiento que es lo que la librería va a ser capaz haciendo una etapa de análisis detallada.

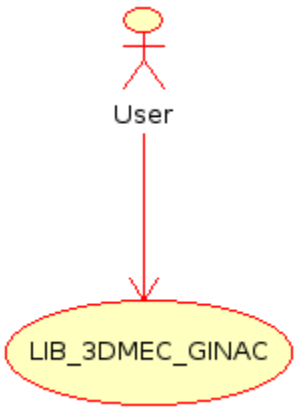
2 - Análisis

Lib3D_mec_GiNaC, que es el nombre que va a tomar la librería, debe ser capaz de realizar múltiples operaciones entre diferentes objetos y expresiones para que podamos definir de la forma más ágil y flexible las ecuaciones de la dinámica de sistemas multicuerpo.

A continuación, mediante una serie de diagramas de casos de uso, que son los diagramas que se utilizan en el UML para representar en un primer momento los requisitos y funcionalidades que debe tener la aplicación final, se va a proceder a mostrar, de una forma de fácil comprensión las operaciones que un usuario va a ser capaz de realizar con la librería desarrollada. Estos diagramas se van a ir desarrollando de forma descendente empezando por el nivel cero o contextual y siguiendo por el nivel uno y sucesivos, mostrando en cada nivel un nivel mas alto de detalle en los objetivos finales de funcionalidad de la librería.

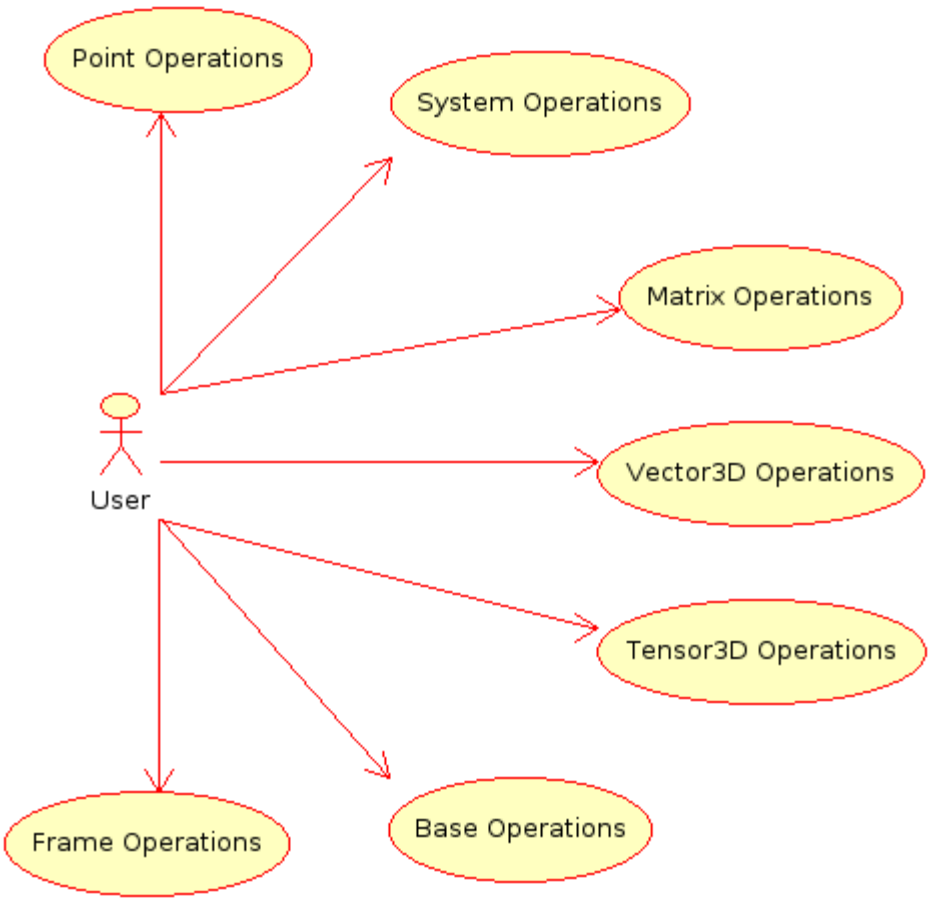
2.1 - Nivel Contextual

En el nivel más alto de abstracción, el usuario va a encontrarse con la librería llamada lib3d_mec_GiNaC, que es la que contiene todos los objetos y funciones que más adelante vamos a poder utilizar. Esta es la librería que deberá ser invocada en nuestro programa principal que contenga el método *main*.



2.2 - Nivel 1

En un nivel de abstracción mas bajo, podemos ver de forma básica, las funcionalidades que deseamos que contenga la librería. En este caso la librería debe de ser capaz de realizar operaciones o simplemente trabajar con objetos de tipo Point (punto), System (sistema), Matrix (matriz), Vector3D (vector 3d), Tensor3D (tensor 3d), Base (base de proyección) y Frame (marco de referencias).



2.3 - Nivel 2

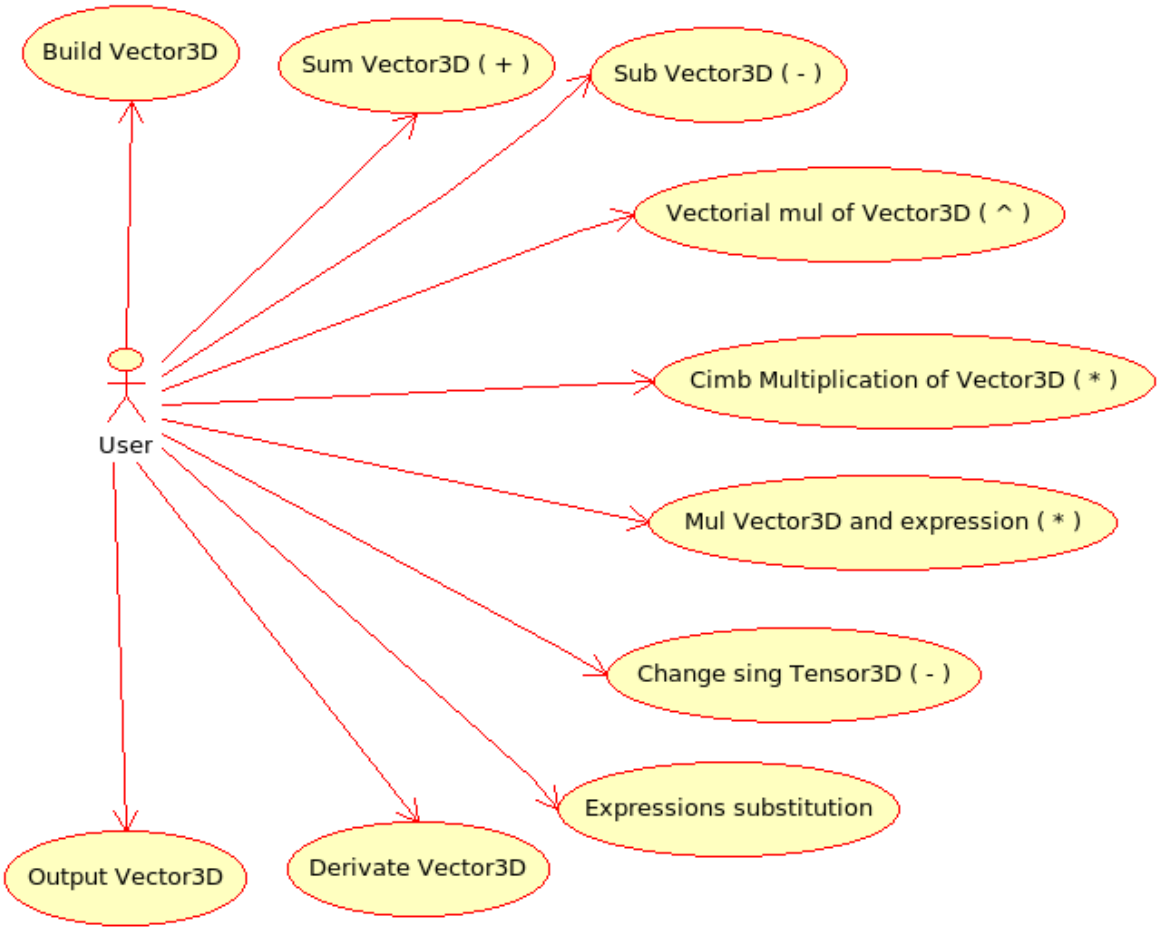
A continuación vamos a poder ver mas concretamente que es lo que queremos que haga la librería; es decir, las funciones que va a tener cada tipo de objeto que hemos podido ver en el nivel anterior.

Nivel 2 – 1 Operaciones con Matrices

En el apartado de operaciones con matrices, un usuario debe poder construir matrices de diferentes formas, sumar, restar, y multiplicar matrices, poder cambiar el signo a una matriz, multiplicar matrices con expresiones, hacer la derivada de todos sus elementos, obtener su transpuesta, substituir las expresiones que contengan las celdas por otras expresiones y finalmente debe poder visualizar textualmente la matriz.

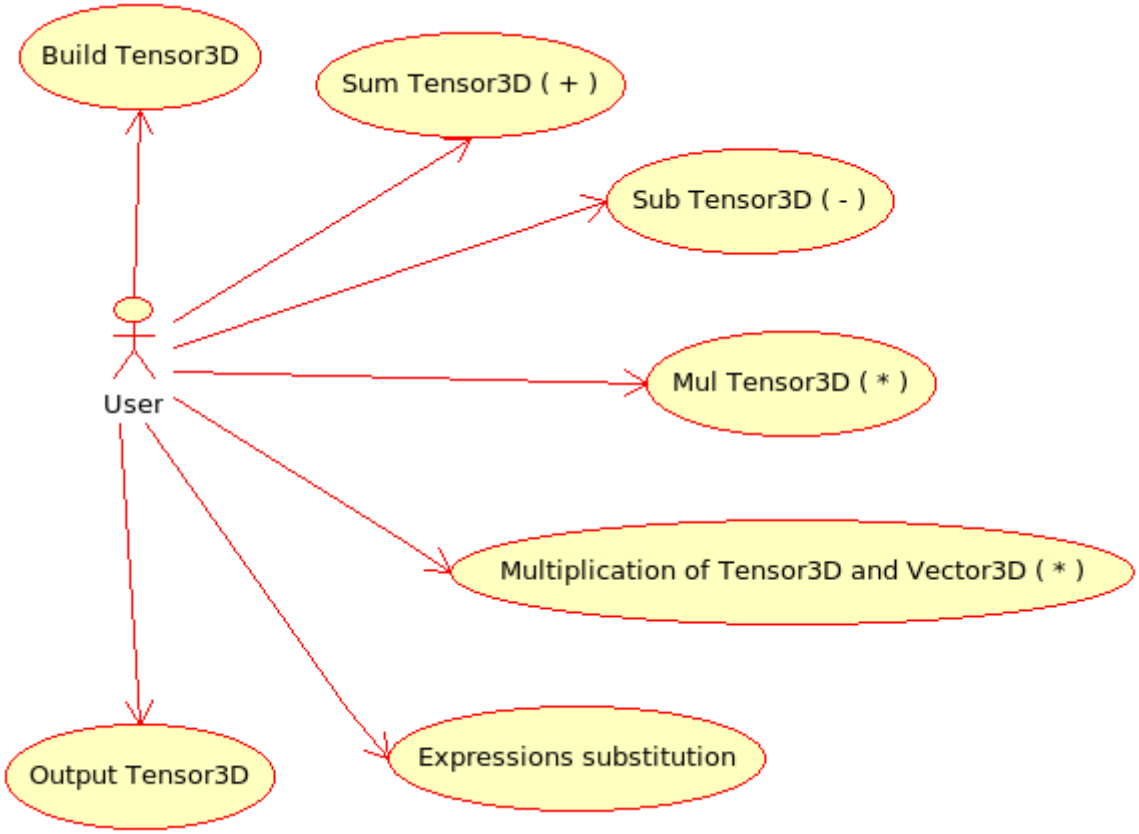
Nivel 2 – 2 Operaciones con Vectores 3D

En el apartado de operaciones con vectores 3D, un usuario debe poder construir vectores 3D de diferentes formas. Sumar, restar, y multiplicar escalar y vectorialmente objetos vector 3D, poder cambiar su signo, multiplicarlos con expresiones simbólicas escalares, hacer la derivada de todos sus elementos, poder cambiar las expresiones de sus celdas por otras expresiones y finalmente debe poder visualizar textualmente el vector 3D.



Nivel 2 – 3 Operaciones con Tensores 3D

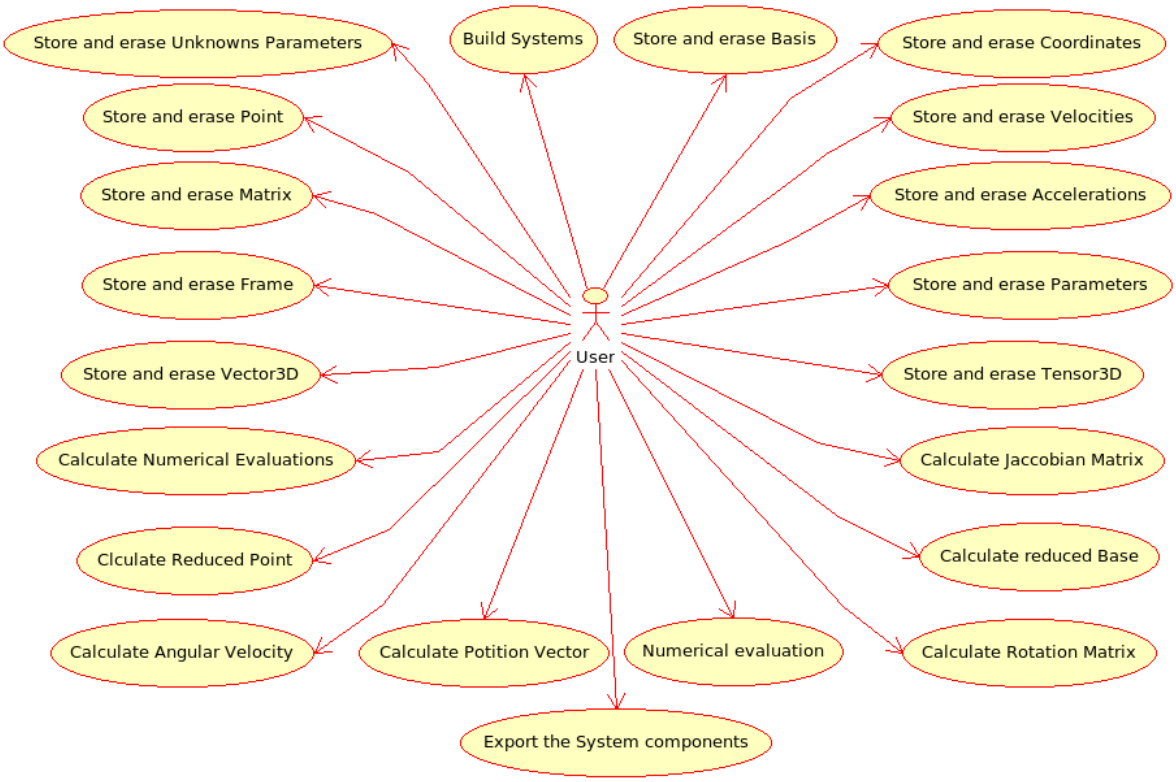
En el apartado de operaciones con objetos tensor 3D, un usuario debe poder construir tensores 3D de diferentes formas, sumar, restar, y multiplicar tensores 3D, multiplicarlos con vectores 3D, poder cambiar las expresiones de sus celdas por otras expresiones y finalmente debe poder visualizar por pantalla textualmente el tensor 3D.



Nivel 2 – 4 Operaciones con Sistemas

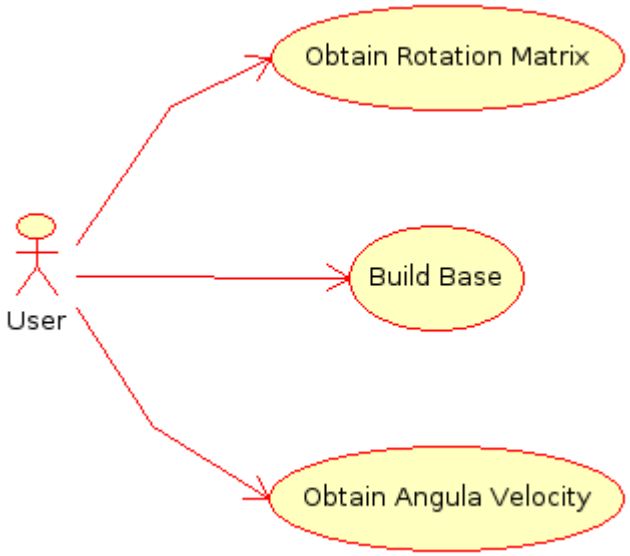
En el apartado de las operaciones del sistema un usuario debe poder crear sistemas, tendrá que poder almacenar y eliminar del sistema matrices, coordenadas, aceleraciones, velocidades, parámetros, parámetros desconocidos, vectores 3D, tensores 3D, bases, marcos de referencia y puntos. También podrá evaluar numéricamente matrices, vectores 3D y tensores 3D. Finalmente tendrá que poder calcular la velocidad angular entre dos bases, el vector de posición entre dos puntos, la matriz de rotación entre dos bases, el jacobino de un vector respecto de otro, calcular cual es la base reducida entre dos bases y cual es el punto reducido entre dos puntos y poder exportar a diferentes ficheros externos escritos en lenguaje

C todos los objetos que se han creado e introducido en el sistema para así poder ser utilizados por otras aplicaciones.



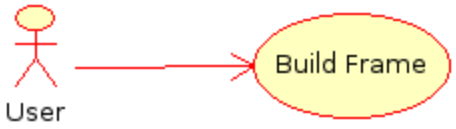
Nivel 2 – 5 Operaciones con Bases

Un usuario tiene que poder crear bases, y obtener la matriz de rotación y la velocidad angular de la misma.



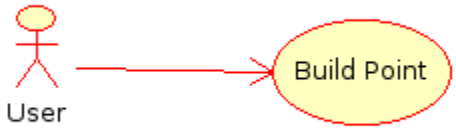
Nivel 2 – 6 Operaciones con Referencias

El usuario tiene que poder crear referencias.



Nivel 2 – 7 Operaciones con Puntos

El usuario tiene que poder crear puntos.



Una vez conocemos todas las funcionalidades que debe poseer la librería lib3d_mec_GiNaC vamos a proceder al diseño de las clases que podrán ser utilizadas directamente por el usuario final.

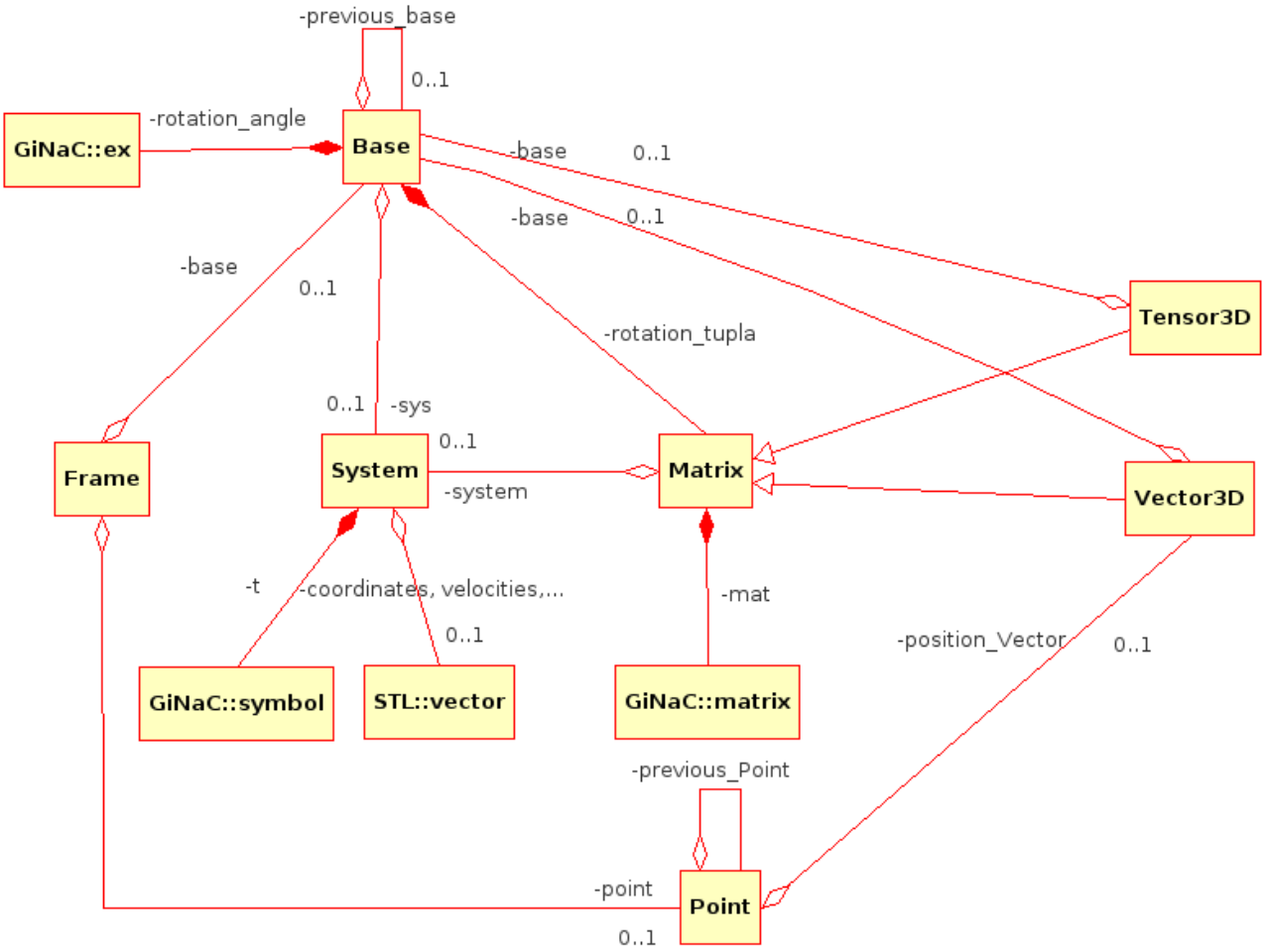
3 - Diseño

A continuación vamos a proceder a realizar la etapa de diseño, en esta etapa mediante los diagramas de clase, se muestra cuales son los atributos y métodos que van a tener finalmente en la implementación. Esta visión de la aplicación es de gran ayuda para los desarrolladores que una vez comenzado el proyecto tengan que integrarse en el, de forma que la comprensión del mismo sea mucho más rápida.

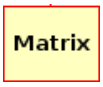
En este apartado también se muestran las relaciones que existen entre las clases, así como los diferentes tipos de empaquetamiento que pueden tener sus atributos y métodos.

Además se podrán ver ejemplos de uso algunos de los constructores y métodos, así como la explicación detallada de algunos de ellos debido a su especial complejidad o características singulares.

Comenzaremos con un diagrama de clases que nos va a mostrar la totalidad de las clases y las relaciones existentes entre los mismos. Para a continuación profundizar en cada una de las clases creadas.



Para poder entender correctamente el diagrama anteriormente expuesto vamos a definir los siguientes elementos



Rectángulo: clase con identidad y significado propio y diferenciable. En ellos se indica el nombre de la clase y si procede, el nombre de la librería a la que pertenecen de la siguiente forma

nombre_librería::nombre_clase

Mas adelante, cuando se haga una explicación detallada de cada clase, se detallará el contenido de estas clases, pudiendo ver todas sus propiedades.

Rombo coloreado: relación de composición. En la clase en la que se encuentra el rombo, existe un atributo y solo uno, cuyo de tipo de objeto es la clase que se encuentra al otro extremo de la relación.

Rombo sin colorear: relación de agregación. En la clase en la que se encuentra el rombo, existe un atributo que puede estar enlazado o inicializado, o no, cuyo de tipo de objeto es la clase que se encuentra al otro extremo de la relación Estos atributos son de tipo puntero, y mas adelante se verá lo que son los atributos de tipo puntero. También podemos observar que en estas relaciones se indica la cardinalidad de *"0..1"* indicando que no es necesario que un objeto tenga asociado un atributo de ese tipo para poder crearse, pero en muchos casos para su correcto funcionamiento si que será necesario.

Flecha sin colorear: relación que indica herencia entre dos clases. La clase en la que se encuentra la punta de la flecha se denomina "clase padre" y la clase que se encuentra en el otro extremo, se denomina "clase hija" o heredera, la cual heredará todos los atributos y métodos públicos y protegidos.

Los nombres que aparecen en las relaciones de composición y agregación, indican los nombres de los atributos con dichas relaciones.

3.1 - Constantes, variables y funciones globales.

A diferencia de otros lenguajes orientados a objetos como puede ser Java, C++ puede beneficiarse de algunas de las características de los lenguajes estructurados como C. En este caso vamos a hacer uso de variables, constantes, plantillas y funciones globales, que podrán ser usados desde cualquier clase, sin necesidad de instanciar clase alguna previamente.

3.1.1 - Constantes globales

Las constantes que se muestran a continuación nos permiten codificar un código mucho más legible y fácil de seguir. De esta manera utilizaremos *"DOWN"* y *"UP"* para indicar que la variable global *"gravity"* contiene los valores 0 o 1 respectivamente. Así mismo las constates *"YES"* o *"NO"* las utilizaremos para indicar que la variable global *"atomization"* contiene los valores 0 o 1 respectivamente. En el apartado de variables globales, vamos a poder ver para que se utilizan las variables *"gravity"* y *"atomization"*

```
#define DOWN 0
#define UP 1
#define NO 0
#define YES 1
```

3.1.2 - Variables globales

```
extern int atomization;
```

Esta variable, indica si queremos trabajar con atomización o sin ella. Si queremos trabajar sin atomización, tendremos que asignar a esta variable el valor *"NO"*, si por contra queremos trabajar con atomización le asignaremos el valor *"YES"*. Por defecto se encuentra con el valor *"NO"*. La función de la atomización consiste en hacer que trabajar con expresiones muy largas sea más eficiente y cómodo. A lo largo de este documento así como en los ejemplos finales podremos ver ejemplos de atomización de expresiones.

```
extern int gravity;
```

Esta variable indica si queremos trabajar con una gravedad hacia arriba o hacia abajo. Si queremos trabajar con una gravedad hacia arriba, tendremos que asignar a esta variable el valor *"UP"*, y si por el contrario, queremos trabajar con una gravedad hacia abajo, tendremos que asignar a esta variable el valor *"DOWN"*. Por defecto, se encuentra con el valor *"DOWN"*. Podremos ver la utilidad de esta variable en los métodos *Reduced_Base* y *Reduced_Point* de la clase System.

```
extern void ( * outError ) ( char * );
```

Variable global que almacena un puntero a una función. Esta función es la que contiene las instrucciones correspondientes para el tratamiento de los errores que surgen durante el programa.

```
void printError( char * args );
```

Función por defecto para el tratamiento de errores, la cual muestra por la salida estándar (línea de comandos) el mensaje del error que se ha producido. La función de tratamiento de errores se puede redefinir para dar un tratamiento a los errores acorde con las necesidades en cada momento.

```
extern vector < symbol > atoms;
```

Variable del tipo *stl::vector* que almacena los átomos que se han ido creando durante el proceso de atomización de expresiones, en el caso de que la variable global *"atomization"*, contenga el valor *"YES"*.

```
extern vector < ex > atom_expresions;
```

Variable del tipo *stl::vector* que almacena las expresiones asociadas a los átomos que se han ido creando durante el proceso de atomización de expresiones, en el caso de que la variable global *"atomization"*, contenga el valor *"YES"*.

```
extern vector < symbol > exclude_atoms;
```

Variable del tipo *stl::vector* que almacena los átomos excluidos de la atomización. Estos deben ser especificados directamente por el usuario.

3.1.3 - Funciones globales

```
template < class T > T atomize ( T m );
```

Función de tipo plantilla que atomiza objetos del tipo *Matrix*, *Vector3D* o *Tensor3D*.

```
template < class T > T unatomize ( T m );
```

Función de tipo plantilla que desatomiza objetos del tipo *Matrix*, *Vector3D* o *Tensor3D*.

```
template < class T > lst atom_list( T m , lst & list ) ;
```

Función de tipo plantilla que devuelve los átomos contenidos en el parámetro *"m"* y que no están contenidos en el objeto *GiNaC::lst* llamado *"list"* . El parámetro *"m"* puede ser del tipo *Matrix*, *Vector3D* o *Tensor3D*. Esta función es útil en la exportación de expresiones o matrices atomizadas que podremos ver en la clase *System*.

```
ex atomize_ex ( ex e );
```

Función que atomiza un objeto del tipo *GiNaC::ex*.

Al tratarse de la función más importante en el apartado de atomización, vamos a verlo y a explicarlo un poco más en detalle a continuación.

Explicación detallada:

```
/*
This function atomize an expression
*/
ex atomize_ex( ex expression ){
      if ( atomization == NO )
            return expression;
      if ( is_a < symbol > ( expression ) )
            return expression;
      if ( is_a < numeric > ( expression ) )
            return expression;

      expression = unatomize_ex( expression );

      if ( is_a < symbol > ( expression ) )
            return expression;
      if ( is_a < numeric > ( expression ) )
            return expression;

      for ( int i = 0 ; i < atom_expressions.size() ; i++ ){
            expression = expression.subs
                        ( atom_expressions[i] + wild( 1 ) == atoms[i] + wild( 1 )
                        , subs_options::algebraic );
            expression = expression.subs
                        ( atom_expressions[i] == atoms[i]
                        ,subs_options::algebraic );
            expression = expression.subs
                        ( -atom_expressions[i] + wild( 1 ) == -atoms[i] + wild( 1 )
                        , subs_options::algebraic );
            expression = expression.subs
                        ( -atom_expressions[i] == -atoms[i]
                        , subs_options::algebraic );
      }

      bool excluding_atoms = false;
      ex sub_expression = expression;
      for ( int i = 0 ; i < exclude_atoms .size() ; i++ )
            if ( sub_expression.has( exclude_atoms[i] ) == 1 ) {
                  sub_expression = sub_expression.subs
                                    ( exclude_atoms[i] == 0
                                    , subs_options::algebraic ) ;
                  excluding_atoms = true ;
            }

      if ( excluding_atoms ) {
            ex sub_expression2 = expression;
            expression = atomize_ex( sub_expression );
            for ( int i = 0 ; i<exclude_atoms.size() ; i++ )
                  if ( sub_expression2.has( exclude_atoms[i] ) == 1 ) {
                        sub_expression = diff
                                          ( sub_expression2
                                          , exclude_atoms[i] );
                        sub_expression = atomize_ex( sub_expression );
                        expression = expression +
                                          sub_expression  *
                                          exclude_atoms[i];
                  }
      return expression;
      }

      symbol atom ;
      if ( expression != 0 ) {
            if ( ( is_a < symbol > ( expression ) == 0 ) and
                  ( is_a < numeric > ( expression ) == 0 ) ){

                  if (  not(  ( is_a < mul > ( expression ) == 1 ) and
                        ( expression.nops() == 2 ) and
                        ( is_a < symbol > ( expression.op( 0 ) ) == 1 ) and
```

```
                        ( is_a < numeric > ( expression.op( 1 ) ) == 1 )  )  ) {

                              atom = symbol ( "atom" +
                                          itoa ( atom_expressions.size() ) ) ;
                              ex atom_expression = expression ;

                              atoms.push_back ( atom ) ;
                              atom_expressions.push_back ( atom_expression ) ;

                              expression = expression.subs
                                                ( atom_expression == atom ,
                                                subs_options::algebraic ) ;
                        expression = expression.subs
                                                ( -atom_expression == atom ,
                                                subs_options::algebraic ) ;
                  }
            }
      }
      return expression ;
}
```

El objetivo de esta función es la de atomizar expresiones GiNaC::ex. Lo que conseguimos con ello es "sustituir" los objetos de tipo *GiNaC::ex* por símbolos *GiNaC:symbol* del tipo *"atomXX"*, de forma que es más cómodo trabajar con átomos que con expresiones muy largas y complejas.

Para la realización de esta atomización, primero tenemos que comprobar que esta esté activada, ya que si no es así se devolverá la expresión que se pasó como parámetro tal cual, al igual que si esta expresión se trata de un objeto de tipo *GiNaC::numeric*, o un objeto de tipo *GiNaC::symbol*. Después se realiza una desatomización completa de la expresión para eliminar de esta de forma automática posibles simplificaciones. A continuación, mediante el método *GiNaC::subs*, se vuelve a hacer una atomización de la expresión substituyendo las subexpresiones de la expresión desatomizada, por las expresiones de los átomos anteriormente creados, de forma que como resultado obtendremos una expresión compuesta por átomos ya conocidos, por expresiones numéricas, símbolos básicos o por una combinación de ellos El siguiente paso es observar si en la expresión hay átomos excluidos en cuyo caso estos no se atomizarán. En el último paso, lo que hacemos es crear una expresión del tipo *"atomXX"* cuyo valor será el obtenido anteriormente.

`ex unatomize_ex ( ex e );`

Función que desatomiza completamente un objeto del tipo *GiNaC::ex*.

`void exclude_atom ( symbol s );`

Función que excluye un objeto *GiNaC::symbol* de la atomización. La funcionalidad de excluir átomos solo funciona si la expresión es lineal con respecto a un átomo. Esta funcionalidad en la actualidad tiene poca utilidad.

`void exclude_Coordinates_atoms ( System * system );`

Función que excluye de la atomización a todas las coordenadas del objeto de tipo *System* al que apunta el puntero que se pasa como parámetro.

```
void exclude_Velocities_atoms( System * system );
```

Función que excluye de la atomización a todas las velocidades del objeto de tipo *System* al que apunta el puntero que se pasa como parámetro.

```
void exclude_Accelerations_atoms ( System * system );
```

Función que excluye de la atomización a todas las aceleraciones del objeto de tipo *System* al que apunta el puntero que se pasa como parámetro.

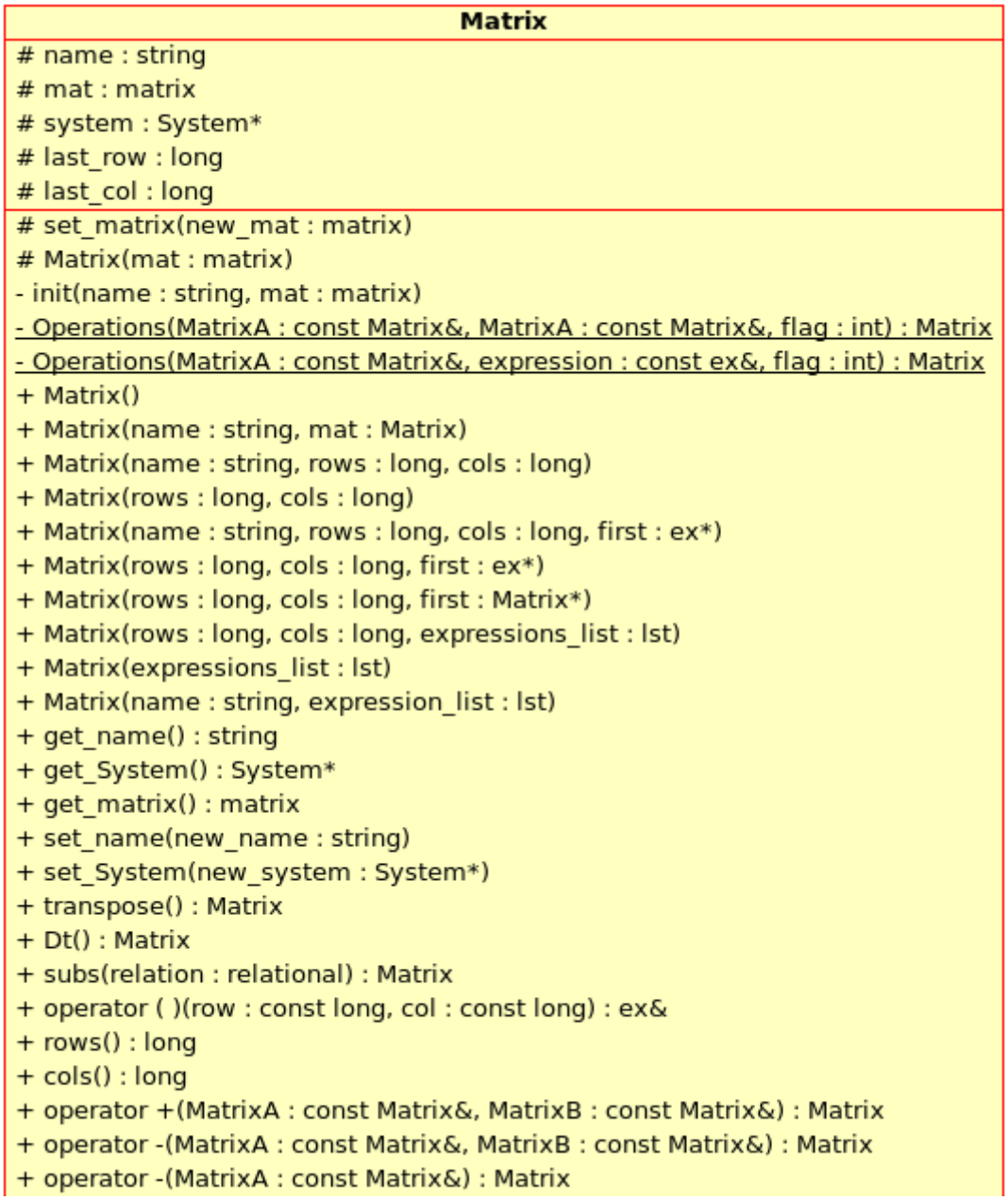
```
void exclude_Unknowns_atoms ( System * system );
```

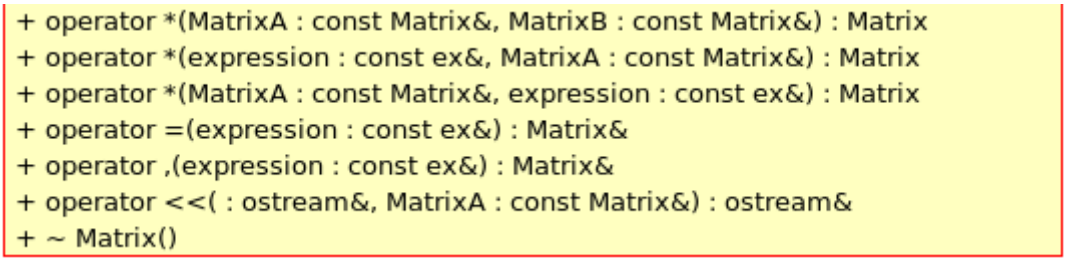
Función que excluye de la atomización a todos los parámetros desconocidos del objeto de tipo *System* al que apunta el puntero que se pasa como parámetro.

3.2 - Clase Matrix

Como podemos ver en el diagrama de clases general de la librería mostrado al inicio del capitulo de diseño, la clase *Matrix* es una de las más importantes ya que de ella dependen otras clases como son la clase *Vector3D* y la clase *Tensor3D*, las cuales por la herencia, que es una de las características de los lenguajes orientados a objetos como C++, van a poder hacer un uso directo de los métodos y atributos protegidos y públicos de esta clase.

3.2.1 - Diagrama de clase





Para entender correctamente los diagramas de clases podemos hacer las siguientes definiciones, las cuales son válidas tanto para la clase Matrix como para las demás clases que podremos ver mas adelante.

Las clases están divididas en 3 partes por líneas horizontales: una que indica el nombre de la misma, otra en el que se encuentran todos sus atributos y otra en la que se encuentran todos sus constructores, métodos y destructores.

: Indica que un atributo o método es protegido. Se dice que un atributo o método es protegido, cuando no va a ser accesible directamente por un usuario de la clase, pero si va a poder ser accedido por las clases herederas de la misma

- : Indica que un atributo o método es privado. Se dice que un atributo o método es privado, cuando solo la clase a la que pertenecen dichos atributos o métodos puede tener acceso a ellos.

+: Indica que un atributo o método es público. Se dice que un atributo o método es público, cuando todas las clases y usuarios de la clase, pueden tener acceso a los mismos.

(Subrayado): indica que un atributo o método es estático. Se dice que un atributo o método es estático cuando no es necesario instanciar una clase para poder ser usado.

Una vez hemos observado el diagrama de clase de la clase *Matrix*, vamos a proceder a la explicación de todos sus atributos y métodos, profundizando en aquellos que así lo requieran.

3.2.2 - Elementos protegidos

- *Atributos*

```
string name;
```

Almacena el nombre del objeto *Matrix*. El atributo es del tipo *stl::string*.

matrix mat;

Es la matriz propiamente dicha. La librería *GiNaC* posee una clase llamada *matrix* que ha sido encapsulada en esta clase y que de esta forma dotamos de una funcionalidad adicional para crear el objeto de tipo *Matrix*.

System * system;

Almacena un puntero a un objeto de tipo *System* al que pertenece el objeto *Matrix* y que podremos ver en detalle mas adelante. Este atributo es imprescindible que se inicialice si se desean realizar derivadas respecto al tiempo mediante el método "*Dt*".

long last_row;

Atributo que se utiliza a modo de contador para indicar la fila del objeto *Matrix* en la que se va a introducir el próximo valor. Este atributo es de utilidad en la modificación de los valores del objeto Matrix y es usado por los operadores "*=*" y "*,*" que podremos ver en el apartado de operadores.

long last_col;

Atributo que se utiliza a modo de contador para indicar la columna del objeto *Matrix* en la que se va a introducir el próximo valor. Este atributo es de utilidad en la modificación de los valores del objeto Matrix y es usado por los operadores "*=*" y "*,*" que podremos ver en el apartado de operadores.

- *Constructor protegido*

Matrix (matrix mat);

Constructor que genera un objeto de tipo *Matrix* a partir de un objeto de tipo *GiNaC::matrix* y cuyo nombre es vacío. Este constructor por su condición de protegido no podrá ser utilizado por los usuarios, pero es de gran ayuda en la implementación de la librería.

Ejemplo:

```
matrix a ( 3 , 3 );

a =      1 , 2 , 3 ,
         4 , 5 , 6 ,
         7 , 8 , 9;

Matrix A ( a );

//Visualizamos el resultado
cout << A << endl;
```

```
[1,2,3;
4,5,6;
7,8,9]
```

- *Método protegido*

```
void set_matrix ( matrix mat );
```

Método que asigna una nueva *GiNaC::matrix* al objeto *Matrix*. Este método por su condición de protegido no podrá ser utilizado por los usuarios, pero es de gran ayuda en la implementación de la librería.

Ejemplo:

```
Matrix A ( "A" ,
           2 , 1 ,
           0 , 0 );

matrix a ( 2 , 2 );
a = 1 , 1 ,
    1 , 2;

A.set_matrix ( a );

//Visualizamos el resultado
cout << A << endl;
```

```
A
[1,1;
1,2]
```

3.2.3 - Elementos privados

- *Métodos privados*

```
void init (  string name ,
             matrix mat );
```

Método de generalización de los constructores, que da valores a los atributos de un objeto *Matrix*. Estos valores son el nombre de tipo *stl::string* (cadena de caracteres) y la matriz de tipo *GiNaC::matrix*.

```
static Matrix Operations (     const Matrix & MatrixA ,
                               const Matrix & MatrixB ,
                               const int  flag );
```

Método estático en el que se realizan los cálculos de las operaciones de suma, resta, cambio de signo y multiplicación entre objetos de tipo *Matrix*. Que se realice una operación u otra, depende del valor del parámetro "*flag*". Si "*flag*" es igual a uno, se realizará la suma, si es igual a 2 se realizará la multiplicación, si es igual a 3 se realizará la resta y si es igual a 4, se realizará el cambio de signo unitario del primer objeto *Matrix* que se pasa como parámetro. La razón por la que este método sea estático es que va a se usado por los métodos "*operator*" los cuales necesitan de un método estático para acceder de forma sencilla a los elementos del objeto *Matrix*.

```
static Matrix Operations (     const Matrix & MatrixA ,
                               const ex & expression ,
                               const int flag );
```

Método estático en el que se realizan los cálculos de las operaciones de multiplicación entre objetos de tipo *Matrix* y objetos de tipo *GiNaC::ex*. Si el parámetro *"flag"* posee el valor 1, se realizará el producto entre un objeto de tipo *Matrix* y uno de tipo *GiNaC::ex* y en caso de poseer el valor 2 se realizará entre un objeto *GiNaC::ex* y uno del tipo *Matrix*.

3.2.4 - Elementos públicos

- *Constructores*

```
Matrix( void );
```

Constructor sin parámetros que construye un objeto de tipo *Matrix* sin inicializar sus atributos.

```
Matrix ( long rows , long cols );
```

Constructor que genera un objeto de tipo *Matrix* sin nombre y unas dimensiones (fila y columna) dadas. Los valores de las celdas del objeto *Matrix* serán 0.

Ejemplo:

```
Matrix A ( 5 , 4 );

//Visualizamos el resultado
cout << A << endl;
```

```
[0,0,0,0;
0,0,0,0;
0,0,0,0;
0,0,0,0;
0,0,0,0]
```

```
Matrix ( string name ,
         Matrix mat );
```

Constructor que genera un objeto de tipo *Matrix* a partir de un objeto de tipo *Matrix* con su nombre correspondiente.

Ejemplo:

```
Matrix a ( 3 , 3 );

//Inicialización de valores que podremos ver en el apartado
Operadores
a =      1 , 2 , 3 ,
         4 , 5 , 6 ,
         7 , 8 , 9;

Matrix A ( "A" , a );

//Visualizamos el resultado
cout << A << endl;
```

```
A
[1,2,3;
4,5,6;
7,8,9]
```

```
Matrix ( string name ,
         long rows , long cols );
```

Constructor que genera un objeto de tipo *Matrix* con un nombre y unas dimensiones (fila y columna) dadas. Los valores de las celdas del objeto *Matrix* serán 0.

Ejemplo:

```
Matrix A ( "A" , 3 , 2 );

//Visualizamos el resultado
cout << A << endl;
```

```
A
[0,0;
0,0;
0,0]
```

```
Matrix ( long rows , long cols ,
         ex * first , ... );
```

Constructor que genera un objeto de tipo *Matrix* con un nombre vacío y unas dimensiones (fila y columna) dadas. Los valores de las celdas del objeto *Matrix* serán los valores que se introduzcan después del numero de filas y columnas, empezando por los

valores de la fila uno, siguiendo por los valores de la fila dos y así sucesivamente hasta completar el numero de filas introducidas.

Explication detallada:

```
/*
Constructor with name, number of rows and cols and the values of this
*/
Matrix::Matrix ( string name , long rows , long cols , ex * first , ... ) {
      try{
            int j = 1;

            this-> name = name;
            matrix aux ( rows , cols );
            aux ( 0 , 0 ) = * first;
            /* Classic form of catch a elements from a function with a undefined
            number of parameters  */
            va_list marcador;
            va_start ( marcador , first );
            for ( int i = 0 ; i < aux.rows () ; i++ ) {
                  while ( j < aux.cols () ) {
                        ex * exClase = va_arg ( marcador , ex * );
                        aux ( i , j ) = * exClase;
                        j++;
                  }
                  j = 0;
            }
            va_end ( marcador );
            mat = aux;
            system = NULL;
            * this = atomize ( * this );
      }catch ( exception & p ) {
            outError ( "ERR - Unspected error building Matrix" );
      }
}
```

En este constructor, utilizamos una función con un número indeterminado de parámetros, para dotar a la misma de la característica especial de poder introducir en la llamada al constructor un número indeterminado de expresiones después de indicar el número de filas y de columnas que va a tener el mismo. Para ello , escribimos "..." como último parámetro del método indicando que el numero de argumentos con los que se va a llamar al método no esta definido de antemano así de esta forma podemos observar el funcionamiento del siguiente ejemplo.

Ejemplo:

```
symbol x ( "x" );
symbol y ( "y" );

ex expresión = 2 * x + 1;
ex expresion2 = x + y;

Matrix A (     3 , 1 ,
               & expresion ,
```

```
                    & expresion ,
                    & expresion2 );

    //Visualizamos el resultado
    cout << A << endl;
```

```
    [1+2*x;
    1+2*x;
    x+y]
```

Matrix (string name ,
long rows , long cols ,
ex * first , ...);

Constructor que genera un objeto de tipo *Matrix* con un nombre y unas dimensiones (fila y columna) dadas. Los valores de las celdas de la *Matrix* serán los valores que se introduzcan después del número de filas y columnas, empezando por los valores de la fila uno, siguiendo por los valores de la fila dos y así sucesivamente hasta completar el número de filas introducidas.

Ejemplo:

```
    symbol x ( "x" );
    symbol y ( "y" );

    ex expresion = 2 * x + 1;
    ex expresion2 = x + y;

    Matrix A ( "A" ,
               3 , 1 ,
               & expresión ,
               & expresión ,
               & expresion2 );

    //Visualizamos el resultado
    cout << A << endl;
```

```
    A
    [1+2*x;
    1+2*x;
    x+y]
```

Matrix (long rows ,
long cols ,
Matrix * first , ...);

Constructor que genera un objeto de tipo *Matrix* con unas dimensiones (fila y columna) dadas, pero en este caso las celdas no son un objeto de tipo *GiNaC::ex* sino de tipo *Matrix* creando una matriz por bloques. Los valores de las celdas de la *Matrix* serán los valores que se introduzcan después del numero de filas y columnas, empezando por los valores de la fila uno, siguiendo por los valores de la fila dos y así sucesivamente hasta completar el numero de filas introducidas. Podremos ver más detalladamente el

funcionamiento de este constructor mas adelante.

Explication detallada:

```
/*
Constructor with number of Matrix in rows and cols for compose a new Matrix with
this Matrix
*/
Matrix::Matrix ( long rows , long cols , Matrix * first , ... ) {
      try{
            int j = 1;
            name = "";
            vector < Matrix * > auxMatrixes;
            /* Introduce all the Matrix in the auxMatrixes vector */
            /* Classic form of catch a elements from a function with a undefined
            number of parameters  */
            auxMatrixes.push_back ( first );
            va_list marcador;
            va_start ( marcador , first );
            for ( int i = 0 ; i < rows ; i++ ) {
                  while ( j < cols ) {
                        auxMatrixes.push_back ( va_arg ( marcador , Matrix * ));
                        j++;
                  }
                  j = 0;
            }
            va_end ( marcador );

            //Calculate the number of cols of the final Matrix using auxMatrix
            int i = 0;
            long numCols = 0;

            while ( i < cols ) {
                  numCols = numCols + auxMatrixes[i]-> cols () ;
                  i++;
            }

            //Calculate the number of rows of the final Matrix using auxMatrix
            long numRows = 0;
            i = 0;
            while ( i < auxMatrixes.size () ) {
                  numRows = numRows + auxMatrixes[i]->rows () ;
                  i += cols;
            }

            /*
            Comprove that all the Matrix are compatible in number of rows and
            cols.
            The sum of the cols of the Matrixes in the same row should be have
            the same num of cols.
            This num of cols is numCols
            */
            long compNumCols = 0;
            int k = 0;
            int r = cols - 1;
            while ( k < auxMatrixes.size () ) {
                  compNumCols = compNumCols + auxMatrixes[k]-> cols () ;
```

```
                if ( compNumCols > numCols ) throw 1;
                if ( k == ( r ) ) {
                        if ( compNumCols != numCols ) throw 1;
                        compNumCols = 0;
                        r += cols;
                }
                k++;
        }

        //Also should be perform that all the Matrix of the same row , have
        the same number of rows;
        long compNumRows = 0;
        long compNumRows2 = 0;
        k = 0;
        while ( k < auxMatrixes.size () ) {
                compNumRows = auxMatrixes[k]-> rows () ;
                //Perform that the number of rows is the same of before
                if (  ( compNumRows != compNumRows2 ) and
                        ( compNumRows2 != 0 ) ) throw 1;
                compNumRows2 = compNumRows;
                if ( k == cols-1 ) {
                        compNumRows2 = 0;
                }
                k += cols;
        }


        //Create the final matrix with the adecuate number of rows and cols
        matrix aux ( numRows , numCols );

        int n = 0;
        int m = 0;
        int r2 = 0;
        //Run auxMatrixes and to put this values in the final matrix
        for ( int k = 0 ; k < auxMatrixes.size () ; k++ ) {
                for ( int i = 0 ; i < auxMatrixes[k]-> rows () ; i++ ) {
                        for ( int j = 0 ; j < auxMatrixes[k]-> cols () ; j++ ) {
                                //Write the values
                                aux ( i + n , j + m ) =
                                        auxMatrixes[k]-> get_matrix () ( i , j );
                        }
                }

                //Row change
                if ( k == ( cols-1 ) + r2 ) {
                        n = n + auxMatrixes[k]-> rows ( void );
                        m = 0;
                        r2 += cols;
                }else
                        m = m + auxMatrixes[k]-> cols ( void );

        }

        mat = aux;
        system = NULL;
        * this = atomize ( * this );
}catch ( exception &  p ) {
        outError ( "ERR - Unspected error building Matrix" );
}catch ( int i ) {
        outError ( "ERR - Incompatible dimensions in input Matrixes" );
}
```

```
}
```

Ejemplo:

Primero vamos a crear 4 objetos de tipo Matrix, de la forma que consideremos más conveniente:

```
Matrix A ( lst (     lst ( 1 , 2 ) ,
                     lst ( 3 , 4 ) ,
                     lst ( 5 , 6 ) ));

//Inicialización de valores que podremos ver en el apartado
Operadores
Matrix B ( 1 , 2 );
B = 7 , 8;

Matrix C ( 1 , 2 , lst ( 9 , 10 ) );
Matrix D ( 1 , 2 , lst ( 11 , 12 ) );
```

Ahora creamos el objeto de tipo *Matrix* por bloques, en este caso con 2 filas y 2 columnas de objetos *Matrix*

```
Matrix E (     2 , 2 ,
               & A , & B ,
               & C , & D );


//Visualizamos el resultado
cout << E << endl;
```

```
[1,2,3,7;
4,5,6,8;
9,10,11,12]
```

En el caso de que las dimensiones de la composición no sean compatibles, esta no llegaría a crearse, por ejemplo si la composición anterior la hubiéramos creado con 4 filas y una columna, esta no llegaría a crearse como veremos a continuación

```
Matrix E (     4 , 1 ,
               & A ,
               & B ,
               & C ,
               & D );
```

```
[1,2,3;
4,5,6;
7;
8;
9,10;
11,12]
```

Vemos como se genera un objeto *Matrix* incompatible con la definición de matrices tradicional, con lo que no llegará a crearse visualizándose el siguiente mensaje por pantalla. También decir que en todos los métodos de esta clase y de las siguientes, el uso erróneo de los mismos puede dar lugar a este tipo de mensajes que indican errores controlados.

```
ERR - Incompatible dimensions in input Matrix
```

`Matrix ( long rows, long cols,`
`lst expressions_list );`

Constructor que genera un objeto de tipo *Matrix* sin nombre y el número de filas y columnas de la misma. Los valores que tendrán dichas celdas serán los que contenga la lista que se pasa como parámetro. Esta lista es del tipo *GiNaC::lst*.

Ejemplo:

```
lst lista ( 1 , 2 , 3 , 4 );
Matrix A ( 2 , 2 , lista );

//Visualizamos el resultado
cout << A << endl;
```

```
A
[1,2;
3,4]
```

`Matrix( lst expressions_list );`

Constructor que genera un objeto de tipo *Matrix* a con un nombre vacío. El número de filas y de columnas, así como el valor de sus celdas, vendrán determinados por los valores del objeto *GiNaC::lst* que pasamos como parámetro. Tenemos que recordar que la lista puede estar compuesta de listas con lo que podremos obtener las diferentes filas y columnas y sus valores.

Expliación detallada:

```
/*
Constructor with one list for compose a new Matrix
*/
Matrix::Matrix ( lst expressions_list ) {
      try{
            //The number of columns of this Matrix we obtain it for the number
            //of elements of the first list
```

```
            matrix aux (      expressions_list.nops () ,
                              expressions_list.op ( 0 ).nops () );

            //One lst is one row of the final Matrix
            for ( int i = 0 ; i< aux.rows () ; i++ )
                  for ( int j = 0 ; j < aux.cols () ; j++ )
                        aux ( i , j ) = expressions_list.op ( i ) .op ( j );
            init ( "" , aux );
      }catch ( int e ) {
            outError ( "ERR - Incompatible dimensions in input Matrixes" );
      }catch ( exception & p ) {
            outError ( "ERR - Unspected error building Matrix" );
      }
}
```

Ejemplo:

```
lst lista (     lst ( 1 , 2 , 3 , 4 ) ,
                lst ( 5 , 6 , 7 , 8 ) ,
                lst ( 9 , 10 , 11 , 12 ) );

Matrix A ( 3 , 4 , lista );

//Visualizamos el resultado
cout << A << endl;
```

```
[1,2,3,4;
5,6,7,8;
9,10,11,12]
```

`Matrix(  string name ,`
`         lst expresión_list );`

Constructor que genera un objeto de tipo *Matrix* con nombre. El número de filas y de columnas, así como el valor de sus celdas, vendrán determinados por los valores de la lista que pasamos como parámetro. Tenemos que recordar que la lista puede estar compuesta de listas con lo que podremos obtener las diferentes filas y columnas y sus valores. Esta lista es del tipo *GiNaC::lst*.

Ejemplo:

```
list lista ( lst (    lst ( 1 , 2 , 3 , 4 ) ,
                      lst ( 5 , 6 , 7 , 8 ) ,
                      lst ( 9 , 10 , 11 , 12 ) ) );

Matrix A ( “A” , lista );

//Visualizamos el resultado
cout << A << endl;
```

```
A
[1,2,3,4;
5,6,7,8;
9,10,11,12]
```

- *Métodos de modificación y acceso*

`string get_name ( void );`

Método que retorna el nombre del objeto *Matrix*.

Ejemplo:

```
Matrix A ( "A" ,
           2 , 1 ,
           1 , 1 );

//Visualizamos el resultado
cout << A.get_name () << endl;
```

```
A
```

`matrix get_matrix ( void );`

Método que retorna el objeto de tipo *GiNaC::matrix* del que está compuesto el objeto de tipo *Matrix*. No es de mucha utilidad a los usuarios de la librería, pero si en la implementación de la misma.

Ejemplo:

```
Matrix A ( "A" ,
          lst ( lst ( 1 , 2 ) ,
                lst ( 3 , 4 ) ) );

//Visualizamos el resultado
cout << A.get_matrix () << endl;
```

```
[[1,2],[3,4]];
```

`System * get_System ( void );`

Método que retorna el sistema al que pertenece el objeto *Matrix*

`void set_name ( string name );`

Método que asigna un nuevo nombre al objeto *Matrix*.

Ejemplo:

```
Matrix A ( “A” ,
           2 , 1 ,
           1 , 1 );

A.set_name( “newA” );

//Visualizamos el resultado
cout << A.get_name () << endl;
```

```
newA
```

`void set_System ( System * system );`

Método que asigna un sistema al objeto *Matrix*.

- *Métodos públicos*

`Matrix transpose ( void );`

Método que retorna la transpuesta de un objeto *Matrix*.

Ejemplo:

```
lst lista (    lst ( 1 , 2 , 3 ) ,
               lst ( 4 , 5 , 6 ) ,
               lst ( 7 , 8 , 9 ) );

Matrix A ( “A” , lista );

//Visualizamos el resultado
cout << A.transpose () << endl;
```

```
A
[1,4,7;
2,5,8;
3,6,9]
```

`Matrix Dt ( void );`

Método que retorna la derivada con respecto al tiempo de un objeto de tipo *Matrix*. Para poder realizar esta operación, es imprescindible que la Matrix pertenezca a algún objeto *System*, que se lo podemos agregar mediante el método `set_System ( System * system )`. En el apartado *“clase System”* podemos ver como se crea un objeto de tipo *System*.

Ejemplo:

```
symbol t;
ex expresión = 2 * t;
```

```
Matrix A (      2 , 1 ,
                7 , expresión );
A.set_System ( & system );

//Visualizamos el resultado
cout << A.Dt () << endl;
```

```
[0;
2]
```

`Matrix subs ( relational relation );`

Método que substituye una expresión por otra en un objeto de tipo *Matrix* según se indique en el parámetro de tipo *GiNaC::relational*, y retorna otro objeto de tipo *Matrix* con la substitución realizada. El formato de los objetos *GiNaC::relational* son de tipo *"expresión1 == expresión2"*, donde se substituirán las apariciones de "*expresion1"* por "*expresion2"*.

Ejemplo:

```
symbol x ( "x" );
symbol y ( "y" );

Matrix A ( "A" ,
           lst (     lst( x + 5 ) ,
                     lst( x + y ) ) );

Matrix B = A.subs ( x == 10 );

//Visualizamos el resultado
cout << B () << endl;
```

```
A
[15;
10+y]
```

`long rows ();`

Método que nos retorna el número de filas del objeto de tipo *Matrix*.

Ejemplo:

```
Matrix A ( 3 , 2 );

//Visualizamos el resultado
cout << A.rows () << endl;
```

```
3
```

long cols ();

Método que nos retorna el número de columnas del objeto de tipo *Matrix*.

Ejemplo:

```
Matrix A ( 3 , 2 );

//Visualizamos el resultado
cout << A.cols () << endl;
```

```
2
```

- *Operadores*

friend Matrix operator + (const Matrix & MatrixA ,
const Matrix & MatrixB);

Sobrecarga del operador "+" que nos permite realizar sumas entre objetos de la clase *Matrix*. Los métodos en cuya declaración aparezca la palabra reservada *friend*, son métodos que no pertenecen a la clase propiamente dicha, sino que se toman de otras clases ajenas a la librería y que se sobrecargan para adaptarlos a nuestras necesidades.

Ejemplo:

```
Matrix A (      2 , 2 ,
                1 , 2 ,
                3 , 4 );

Matrix B (      2 , 2 ,
                4 , 2 ,
                2 , 4 );

Matrix C = A + B;

//Visualizamos el resultado
cout << C << endl;
```

```
[5,4;
5,8]
```

friend Matrix operator - (const Matrix & MatrixA ,
const Matrix & MatrixB);

Sobrecarga del operador "–" que nos permite realizar restas entre objetos de la clase *Matrix*.

Ejemplo:

```
Matrix A (      2 , 2 ,
                1 , 2 ,
                3 , 4 );

Matrix B (      2 , 2 ,
                4 , 2 ,
                2 , 4 );

Matrix C = A – B;

//Visualizamos el resultado
cout << C << endl;
```

```
[-3,0;
1,0]
```

```
friend Matrix operator - ( const Matrix & MatrixA );
```

Sobrecarga del operador unitario "–" que nos permite cambiar el signo de un objeto *Matrix*

Ejemplo:

```
Matrix A (      2 , 3 ,
                1 , 2 , 3 ,
                4 , 5 , 6 );

Matrix B = - A;

//Visualizamos el resultado
cout << C << endl;
```

```
[-1,-2,-3;
-4,-5,-6]
```

```
friend Matrix operator * (      const Matrix & MatrixA ,
                                const Matrix & MatrixB );
```

Sobrecarga del operador "*" nos permite realizar multiplicaciones entre objetos de la clase *Matrix*.

Ejemplo:

```
Matrix A (      2 , 2 ,
                1 , 2 ,
                3 , 4 );

Matrix B (      2 , 2 ,
                4 , 2 ,
                2 , 4 );
```

```
Matrix C = A * B;

//Visualizamos el resultado
cout << C << endl;
```

```
[8,10;
20,22]
```

```
friend Matrix operator * (     const ex & expression ,
                               const Matrix & MatrixA );
```

Sobrecarga del operador "*" nos permite realizar multiplicaciones entre expresiones *GiNaC::ex* y objetos de la clase *Matrix*.

Ejemplo:

```
Matrix A ( lst (      lst ( 1 , 2 ) ,
                      lst ( 3 , 4 ) ) );

symbol x ( "x" );
ex expresión = 2 * x + 1;

Matrix C = expresión * A;

//Visualizamos el resultado
cout << C << endl;
```

```
[1+2*x,2+4*x;
3+6*x,4+8*x]
```

```
friend Matrix operator * (     const Matrix & MatrixA ,
                               const ex & expression );
```

Sobrecarga del operador "*" nos permite realizar multiplicaciones entre objetos de la clase *Matrix* y expresiones *GiNaC::ex*.

Ejemplo:

```
Matrix A (      2 , 2 ,
                1 , 2 ,
                3 , 4 );

symbol x ( "x" );
ex expresión = 2 * x + 1;

Matrix C = A * expresion;

//Visualizamos el resultado
cout << C << endl;
```

```
[1+2*x,2+4*x;
3+6*x,4+8*x]
```

ex & operator( ) (long row, long col);

Sobrecarga del operador *"( )"* que retorna la expresión del tipo *GiNaC::ex* que contiene una determinada celda del objeto *Matrix*. Las coordenadas de esta celda, vienen determinadas por el número de fila y de columna. Además de la misma forma, se pueden asignar nuevos valores a una celda determinada

Ejemplo:

```
lst lista (     lst( 1 , 2 , 3 ) ,
                lst( 4 , 5 , 6 ) ,
                lst( 7 , 8 , 9 ) );

Matrix A( "A" , lista );

//Visualizamos el resultado
cout << A ( 1 , 1 ) << endl;
```

```
5
```

```
A ( 1 , 1 ) = 3;

//Visualizamos el resultado
cout << A ( 2 , 2 ) << endl;
```

```
3
```

Matrix& operator = (const ex & expression);

Sobrecarga del operador "=" que mediante la combinación con el operador"," que veremos a continuación, podremos inicializar o modificar los valores de un objeto de tipo *Matrix*.

Matrix& operator , (const ex & expression);

Sobrecarga del operador "," que mediante la combinación con el operador"=" que hemos podido ver anteriormente, podremos inicializar o modificar los valores de un objeto de tipo *Matrix*.

Ejemplo:

```
//Creación de una matriz de dimensiones 3x3 inicializados a 0
Matrix A (  3 , 3 );

//Modificación
A = 1 , 2 , 3
    4 , 5 , 6
    7 , 8 , 9;
```

```
//Visualizamos el resultado
cout << A << endl;
```

```
[1,2,3;
4,5,6;
7,8,9]
```

```
friend ostream & operator << (      ostream & os ,
                                    const Matrix & MatrixA );
```

Sobrecarga del operador "<<" nos permite formatear la salida por la salida estándar de un objeto de tipo *Matrix*. De este método hemos podemos ver ejemplos en los ejemplos de los métodos anteriores.

- *Destructor*

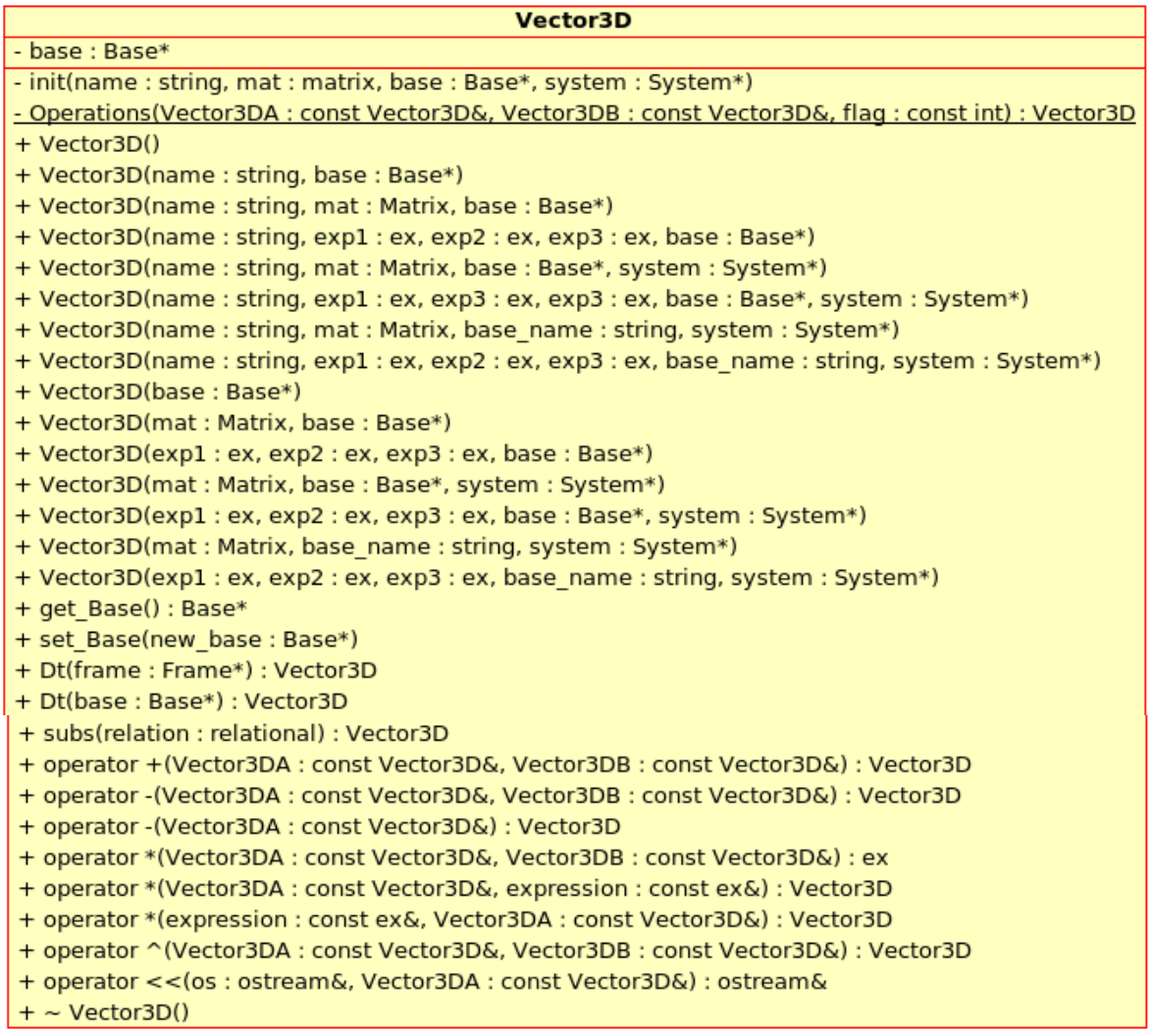
```
~Matrix ( void );
```

Destructor de la clase *Matrix*

3.3 - Clase Vector3D

La clase *Vector3D* es una clase hija de la clase *Matrix*, por lo que hereda directamente todos los métodos públicos y protegidos de esta, pudiéndolos utilizar en cualquier momento. De forma muy abstracta, se podría decir que un Vector 3D es una Matriz de 3 filas y una columna.

3.3.1 - Diagrama de clases



Vector3D es una clase que hereda todos los atributos y métodos públicos y protegidos de la clase *Matrix*. Por ello a continuación solo se van a mostrar los atributos que esta clase posee nuevos y métodos que son nuevos o se han sobrecargado.

3.3.2 - Elementos Privados

- *Atributos*

```
Base * base;
```

Atributo que almacena un puntero a un objeto de tipo *Base* que será su base asociada.

- *Métodos privados*

```
void init ( string name , matrix mat , Base * base , System * system );
```

Método que inicializa todos los atributos de un objeto de tipo *Vector3D*. Estos atributos son el nombre, *GiNaC::matrix*, un puntero a un objeto *Base* y puntero a un objeto *System* que se pasan como parámetros. Este método es utilizado por los constructores de la clase *Vector3D*.

```
static Vector3D Operations (   const Vector3D & Vector3DA ,
                               const Vector3D & Vector3DB ,
                               const int flag );
```

Método de clase o estático que realiza operaciones entre dos objetos de la clase *Vector3D*. El tipo de operación dependerá del valor que le indiquemos al parámetro llamado *"flag"*. Estos valores pueden ser 1 para realizar sumas de objetos *Vector3D*, 2 para realizar restas de objetos *Vector3D* o 3 para realizar la multiplicación vectorial de objetos *Vector3D*. La razón de que este método sea estático es la de que va a ser llamado desde los operadores suma, resta, etc…, que podremos ver mas adelante y que para que funcione correctamente debe poder acceder al método sin instanciar la clase.

A continuación podemos ver la implementación de este código, así como una explicación de su funcionamiento.

Explicación detallada:

```
/*
Method that make the Vector3D operations calculations
 */
Vector3D Vector3D::Operations (    const Vector3D & Vector3DA ,
                                   const Vector3D & Vector3DB ,
                                   const int flag ){
      try{
            if ( ( Vector3DA.system == NULL ) or
                  ( Vector3DB.system == NULL ) ) throw 1;
            System * sys = Vector3DA.system;
```

```
            Base * reducebase = sys-> Reduced_Base ( Vector3DA.base ,
                                                     Vector3DB.base );

            /* Use Matrix object for make these operations */
            Matrix a = sys-> Rotation_Matrix ( reducebase , Vector3DA.base ) *
                                             ( Matrix )Vector3DA;
            Vector3D vectorAux1 ( a.get_matrix () , reducebase );

            a = sys-> Rotation_Matrix ( reducebase , Vector3DB.base ) *
                                      ( Matrix )Vector3DB;
            Vector3D vectorAux2 ( a.get_matrix () , reducebase );

            if ( flag == 1 )// +
                  a = ( Matrix )vectorAux1 + ( Matrix )vectorAux2;
            else if ( flag == 2 )//-
                  a = ( Matrix )vectorAux1 - ( Matrix )vectorAux2;
            else if ( flag == 3 ){//^
                  a ( 0 , 0 ) = atomize_ex ( atomize_ex ( vectorAux1 ( 1 , 0 ) *
                                vectorAux2 ( 2 , 0 ) ) –
                                atomize_ex ( vectorAux1 ( 2 , 0 ) *
                                vectorAux2 ( 1 , 0 ) ) );
                  a ( 1 , 0 ) = atomize_ex ( atomize_ex ( -vectorAux1 ( 0 , 0 ) *
                                vectorAux2 ( 2 , 0 ) ) +
                                atomize_ex ( vectorAux1 ( 2 , 0 ) *
                                vectorAux2 ( 0 , 0 ) ) );
                  a ( 2 , 0 ) = atomize_ex ( atomize_ex ( vectorAux1 ( 0 , 0 ) *
                                 vectorAux2 ( 1 , 0 ) ) –
                                 atomize_ex ( vectorAux1 ( 1 , 0 ) *
                                 vectorAux2 ( 0 , 0 ) ) );
            }

            Vector3D vectorSol ( a.get_matrix () , reducebase );
            vectorSol.set_System ( sys );
            return vectorSol;
      }catch ( exception & p ) {
            outError ( "ERR - Unexpected error building Vector3D" );
      }catch ( int i ){
            outError ( "ERR- Vectors3D with incompatibles Systems" );
      }
      Vector3D vacio;
      return vacio;

}
```

Haciendo uso de la reutilización de código, se ha implementado el método anteriormente escrito llamado “*Operations*”, que es el encargado de hacer las operaciones de suma, resta y producto vectorial de objetos *Vector3D*. Para hacer estas operaciones de forma correcta podemos observar que antes tenemos que calcular la base reducida, *“Reduced_Base”*; método que se localiza en la clase *System*. Una vez obtenida esta base, transformamos los dos objetos *Vector3D* mediante el método *“Rotation_Matrix”* a dos nuevos objetos *Vector3D*, en la nueva base reducida. Ahora ya podemos operar entre los objetos *Vector3D* y realizar la operación pertinente según hayamos indicado mediante el parámetro *“flag”*.

3.3.3 - Elementos públicos

- *Constructores*

```
Vector3D ( void );
```

Constructor sin parámetros que construye un objeto de la clase *Vector3D* sin inicializar sus atributos.

```
Vector3D (   string name ,
             Base * base );
```

Constructor que genera un *Vector3D* con un nombre dado, un puntero a su *Base* asociada y la inicialización de sus celdas a 0. La dimensión de la matrix asociada al objeto *Vector3D* es de 3x1.

Ejemplo:

```
Vector3D V ( "V" , & Base_uno );

//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
0
0
0
}%Base1
```

```
Vector3D (   string name ,
             Matrix mat ,
             Base * base );
```

Constructor que genera un objeto de tipo *Vector3D* con un nombre, un objeto *Matrix* cuyas dimensiones deben ser de 3x1 y un puntero a un objeto *Base*.

Ejemplo:

```
Matrix a ( 3 , 1 );

a =      1 ,
         2 ,
         3;

Vector3D V(     "V" ,
                a ,
```

```
                    & Base_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
1
2
3
}%Base1
```

```
Vector3D (   string name ,
             ex expression1 ,
             ex expression2 ,
             ex expression3 ,
             Base * base );
```

Constructor que genera un objeto de tipo *Vector3D* con un nombre dado, 3 expresiones que corresponderán a los 3 valores de su *GiNaC::matrix* asociada y un puntero a un objeto *Base*.

Ejemplo:

```
Vector3D V (    "V" ,
                3 ,
                4 ,
                5 ,
                & Base_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
3
4
5
}%Base1
```

```
Vector3D (   string name ,
             Matrix * mat ,
             Base * base,
             System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con un nombre dado, un puntero a un objeto *Matrix* cuyas dimensiones deben ser de 3x1, un puntero a su objeto *Base* asociado y un puntero a su objeto *System* asociado.

Ejemplo:

```
Vector3D V (    “V” ,
                & Matrix_uno ,
                & Base_uno ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
1
1
1
}%Base1
```

```
Vector3D (   string name ,
             ex expression1 ,
             ex expression2 ,
             ex expression3 ,
             Base * base ,
             System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con un nombre dado, 3 objetos *GiNaC::ex* que corresponderán a los 3 valores de su objeto *GiNaC::matrix* asociada, un puntero a su objeto *Base* asociado y un puntero a su objeto *System* asociado.

Ejemplo:

```
Vector3D V (    “V” ,
                1 ,
                2 ,
                3 ,
                & Base_uno ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
1
2
3
}%Base1
```

```
Vector3D (   string name ,
             Matrix mat ,
             string base_name ,
             System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con un nombre dado, un objeto *Matrix* cuyas dimensiones deben ser de 3x1, el nombre de un objeto *Base* asociada y un puntero a su objeto *System* asociado. Para que el objeto Vector3D se cree correctamente, la *Base* cuyo nombre se pasa como parámetro debe existir en el objeto *System* que se pasa como parámetro, de lo contrario el *Vector3D* no se creará.

Ejemplo:

```
Vector3D V (    "V" ,
                matrix_uno ,
                "Base1" ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
1
1
1
}%Base1
```

```
Vector3D (   string name ,
             ex expression1 ,
             ex expression2 ,
             ex expression3 ,
             string base_name ,
             System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con un nombre dado, 3 expresiones *GiNaC::ex* que corresponderán a los 3 valores de su objeto *GiNaC::matrix* asociado, el nombre de un objeto *Base* asociada y un puntero a su objeto *System* asociado.

Ejemplo:

```
Vector3D V (    "V" ,
                1 ,
                2 ,
                3 ,
                "Base1" ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
V
%{
1
```

```
2
3
}%Base1
```

Vector3D (Base * base);

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, un puntero a su objeto *Base* asociada y la inicialización de sus celdas con el valor 0. La dimensión de la *GiNaC::matrix* asociada al *Vector3D* debe ser de 3x1.

Ejemplo:

```
Vector3D V ( & Base_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
%{
0
0
0
}%Base1
```

Vector3D (Matrix mat ,
Base * base);

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, un objeto M*atrix* cuyas dimensiones deben ser de 3x1 y un puntero a un objeto *Base*.

Ejemplo:

```
Vector3D V (    Matrix_uno ,
                & Base_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
%{
1
1
1
}%Base1
```

Vector3D (ex expression1 ,
ex expression2 ,
ex expression3 ,
Base * base);

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, 3 objetos *GiNaC::ex* que corresponderán a los 3 valores de su *GiNaC::matrix* asociada y un puntero a un objeto *Base* asociado.

Ejemplo:

```
Vector3D V (    6 ,
                5 ,
                4 ,
                & Base_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
%{
6
5
4
}%Base1
```

```
Vector3D (    Matrix mat ,
              Base * base ,
              System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, un objeto *Matrix* cuyas dimensiones deben ser de 3x1, un puntero a un objeto *Base* y un puntero a un objeto *System* al que esta asociado.

Ejemplo:

```
Vector3D V (    Matrix_uno ,
                & Base_uno ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
%{
1
1
1
}%Base1
```

```
Vector3D (    ex expression1 ,
              ex expression2 ,
              ex expression3 ,
              Base * base ,
              System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, 3 objetos *GiNaC::ex* que corresponderán a los 3 valores de su *GiNaC::matrix* asociada, un puntero al objeto *Base* al que esta asociado y un puntero al objeto *System* al que está asociado.

Ejemplo:

```
Vector3D V (    6 ,
                5 ,
                4 ,
                & Base_uno ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
%{
6
5
4
}%Base1
```

```
Vector3D (    Matrix mat ,
              string base_name ,
              System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, un objeto *Matrix* cuyas dimensiones deben ser de 3x1, el nombre de un objeto *Base* al cual esta asociado y un puntero a un objeto *System*.

Ejemplo:

```
Vector3D V (    matrix_uno ,
                ”Base1” ,
                & System_uno );


//Visualizamos el resultado
cout << V << endl;
```

```
%{
1
1
1
}%Base1
```

```
Vector3D (    ex expression1 ,
              ex expression2 ,
              ex expression3 ,
              string base_name ,
              System * system );
```

Constructor que genera un objeto de tipo *Vector3D* con nombre vacío, 3 expresiones de tipo *GiNaC::ex* que corresponderán a los 3 valores de su matrix asociada, el nombre de un objeto *Base* al cual esta asociado y un puntero a un objeto *System*.

Ejemplo:

```
Vector3D V (    7 ,
                9 ,
                11 ,
                ”Base1” ,
                & System_uno );

//Visualizamos el resultado
cout << V << endl;
```

```
%{
7
9
11
}%Base1
```

- *Métodos de modificación y acceso*

`Base * get_Base ( void );`

Método que retorna un puntero al objeto *Base* asociado al *Vector3D*.

Ejemplo:

```
Vector3D V (    7 ,
                9 ,
                11 ,
                ”Base1” ,
                & System_uno );

//Visualizamos el resultado
cout << V.get_Base ().get_name () << endl;
```

```
Base1
```

`void set_Base ( Base * new_base );`

Método que asigna una nueva base al *Vector3D*. La base es un puntero a un objeto de tipo *Base*.

Ejemplo:

```
Vector3D V (    7 ,
                9 ,
                11 ,
                ”Base1” ,
                & System_uno );

V.set_Base ( & Base_dos );

//Visualizamos el resultado
cout << V << endl;
```

```
%{
7
9
11
}%Base2
```

- *Métodos públicos*

`Vector3D Dt( Frame * frame);`

Método que realiza la derivada temporal del objeto *Vector3D* respecto de un objeto *Frame*.

`Vector3D Dt ( Base * base);`

Método que realiza la derivada temporal del objeto *Vector3D* respecto de un objeto *Base*.

`Vector3D subs ( relational relation );`

Método que substituye una expresión por otra en un objeto de tipo *Vector3D* según se indique en el parámetro de tipo *GiNaC::relational* y retorna otro objeto de tipo *Vector3D* con la substitución realizada. El formato de los objetos *GiNaC::relational* son de tipo “*expresión1 == expresión2*”, donde se substituirán las apariciones de *expresion1* por *expresion2*.

- *Operadores*

```
friend Vector3D operator + (   const Vector3D & Vector3DA ,
                               const Vector3D & Vector3DA );
```

Sobrecarga del operador “+” que nos permite realizar sumas entre objetos de tipo *Vector3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse en una base
común para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Vector3D V (    7 ,
                9 ,
                11 ,
                ”Base1” ,
                & System_uno );

Vector3D W (    3 ,
                2 ,
                1 ,
                ”Base1” ,
                & System_uno );

Matrix X = V + W;

//Visualizamos el resultado
cout << X << endl;
```

```
%{
10
11
12
}%Base1
```

```
friend Vector3D operator - (   const Vector3D & Vector3DA ,
                               const Vector3D & Vector3DB );
```

Sobrecarga del operador “-“ que nos permite realizar restas entre objetos de tipo *Vector3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse en una base
común para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Vector3D V (    7 ,
                9 ,
                11 ,
                ”Base1” ,
                & System_uno );

Vector3D W (    3 ,
                2 ,
                1 ,
                ”Base1” ,
                & System_uno );

Matrix X = V - W;

//Visualizamos el resultado
```

```
cout << X << endl;
```

```
%{
4
7
10
}%Base1
```

friend Vector3D operator - (const Vector3D & Vector3DA);

Sobrecarga del operador unitario "-" que permite cambiar el signo de un objeto *Vector3D*.

Ejemplo:

```
Vector3D V (    7 ,
                9 ,
                11 ,
                ”Base1” ,
                & System_uno );

Vector3D X = - V;

//Visualizamos el resultado
cout << X << endl;
```

```
%{
-7
-9
-11
}%Base1
```

friend ex operator * (const Vector3D & Vector3DA ,
const Vector3D & Vector3DB);

Sobrecarga del operador "*" que permite realizar multiplicaciones escalares entre objetos *Vector3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse en una base
común para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Vector3D V (    1 ,
                2 ,
                3 ,
                ”Base1” ,
                & System_uno );

Vector3D W (    2 ,
```

```
                    2 ,
                    1 ,
                    ”Base1” ,
                    & System_uno );

    ex x = V + W;

    //Visualizamos el resultado
    cout << x << endl;
```

```
    9
```

```
friend Vector3D operator * (   const Vector3D & Vector3DA ,
                               const ex & expresión );
```

Sobrecarga del operador “*” que permite realizar multiplicaciones entre objetos de tipo *Vector3D* y *GiNaC::ex*.

Ejemplo:

```
    Vector3D V (    1 ,
                    2 ,
                    3 ,
                    ”Base1” ,
                    & System_uno );

    ex x = 2 * x;

    Vector3D W = V * x;

    //Visualizamos el resultado
    cout << W << endl;
```

```
    %{
    2*x
    4*x
    6*x
    }%Base1
```

```
friend Vector3D operator * (   const ex & expresion ,
                               const & Vector3D Vector3DA );
```

Sobrecarga del operador “*” que permite realizar multiplicaciones entre objetos de tipo *GiNaC::ex* y *Vector3D*.

Ejemplo:

```
    Vector3D V (    1 ,
                    2 ,
                    3 ,
                    ”Base1” ,
                    & System_uno );
```

```
ex x = 2 * x;

Vector3D W = x * V;

//Visualizamos el resultado
cout << W << endl;
```

```
%{
2*x
4*x
6*x
}%Base1
```

```
friend Vector3D operator ^ (   const Vector3D & Vector3DA ,
                               const Vector3D & Vector3DB );
```

Sobrecarga del operador "^" que permite realizar multiplicaciones vectoriales entre objetos de tipo *Vector3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse en una base
común para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Vector3D V (    1 ,
                2 ,
                3 ,
                ”Base1” ,
                & System_uno );

Vector3D W (    2 ,
                2 ,
                1 ,
                ”Base1” ,
                & System_uno );

//Para asegurar la precedencia del operador “^” se recomienda,
utilizar paréntesis
Vector3D X = ( V ^ W );

//Visualizamos el resultado
cout << x << endl;
```

```
%{
2
4
3
}%Base1
```

```
friend ostream & operator << (      ostream & os ,
                                    const Vector3D & Vector3DA );
```

Sobrecarga del operador “<<” que permite formatear la salida por la salida estándar de un objeto *Vector3D*.

- *Destructor*

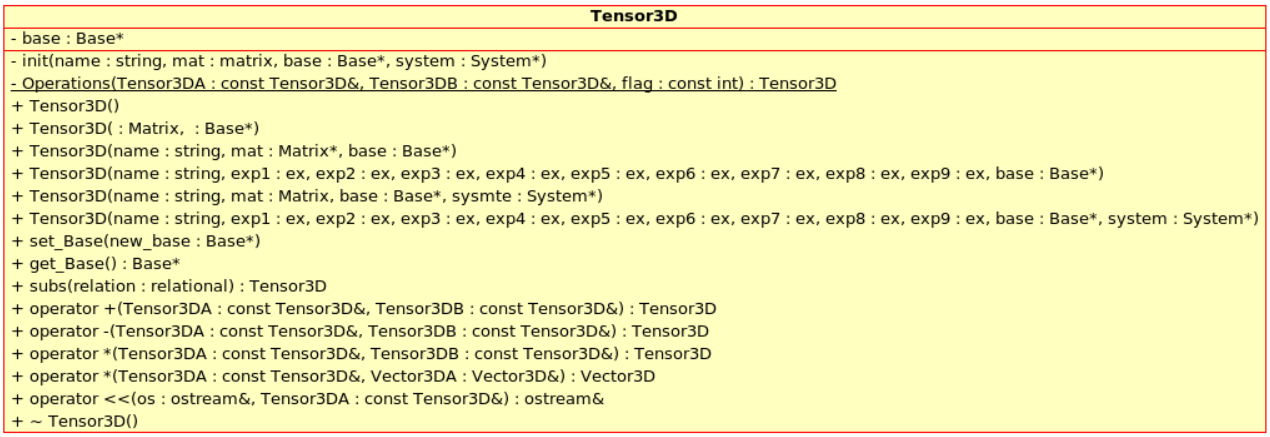
```
~Vector3D ( void );
```

Destructor de la clase Vector3D.

3.4 - Clase Tensor3D

La clase *Tensor3D* es una clase hija de la clase *Matrix*, por lo que hereda directamente todos los métodos públicos y protegidos de esta, pudiéndolos utilizar en cualquier momento. De forma muy abstracta, se podría decir que un Tensor 3D es una Matriz de 3 filas y 3 columnas.

3.4.1 - Diagrama de clase



3.4.2 - Elementos privados

- *Atributos*

```
Base* base;
```

Atributo que almacena un puntero a un objeto de tipo *Base* que será su base asociada.

- *Métodos privadoss*

```
void init(   string name ,
             matrix mat ,
             Base * base ,
             System * system );
```

Método que inicializa todos los atributos de un objeto de tipo *Vector3D*. Estos atributos son nombre, *GiNaC::matrix*, puntero a objeto *Base* y puntero a objeto *System* que se pasan como parámetros. Este método es utilizado por los constructores de la clase *Tensor3D*.

```
static Vector3D Operations (   const Vector3D & Vector3DA ,
                               const Vector3D & Vector3DB ,
                               int flag );
```

Método estático que realiza operaciones entre dos objetos de la clase *Tensor3D*. El tipo de operación dependerá del valor que le indiquemos mediante el parámetro *"flag"*. Estos valores pueden ser 1 para realizar sumas de objetos *Tensor3D* o 2 para realizar restas de objetos *Tensor3D*.

3.4.3 - Elementos públicos

- *Costructores*

```
Tensor3D ( void );
```

Constructor que genera un objeto de tipo *Tensor3D* sin inicializar sus atributos.

```
Tensor3D (   string name ,
             Matrix mat ,
             Base * base );
```

Constructor que genera un objeto de tipo *Tensor3D* con un nombre dado, un objeto de tipo *Matrix* y el puntero a un objeto de tipo *Base* al que está asociado.

Ejemplo:

```
Tensor3D T (    "T" ,
                Matrix_uno ,
                & Base_uno );

//Visualizamos el resultado
cout << T << endl;
```

```
T
%{
1,1,1;
1,1,1;
1,1,1;
}%Base1
```

```
Tensor3D (   Matrix mat ,
             Base * base );
```

Constructor que genera un objeto de tipo *Tensor3D* sin nombre, un objeto de tipo *Matrix* y el puntero a un objeto de tipo *Base* al que está asociado.

Ejemplo:

```
Tensor3D T (    Matrix_uno ,
                & Base_uno );

//Visualizamos el resultado
cout << T << endl;
```

```
%{
1,1,1;
1,1,1;
1,1,1;
}%Base1
```

```
Tensor3D (   string name ,
             ex exp1 , ex exp2 , ex exp3 ,
             ex exp4 , ex exp5 , ex exp6 ,
             ex exp7 , ex exp8 , ex exp9 ,
             Base * base );
```

Constructor que genera un objeto de tipo *Tensor3D* con un nombre, 9 expresiones de tipo *GiNaC::ex* que compondrán la matriz de (3x3) y un puntero a un puntero al objeto *Base* al que esta asociado.

Ejemplo:

```
Tensor3D T (    "T" ,
                1 , 2 , 3 ,
                4 , 5 , 6 ,
                7 , 8 , 9 ,
                & Base_uno );

//Visualizamos el resultado
cout << T << endl;
```

```
T
%{
1,2,3;
4,5,6;
7,8,9;
}%Base1
```

```
Tensor3D (   string name ,
             Matrix mat ,
             Base * base ,
             System * system );
```

Constructor que genera un objeto de tipo *Tensor3D* con un nombre dado, un objeto de tipo *Matrix* de (3x3), un puntero a un objeto Base y un puntero al objeto *System* al que está asociado.

Ejemplo:

```
Tensor3D T (    “T” ,
                Matrix_uno ,
                & Base_uno ,
                & System_uno );

//Visualizamos el resultado
cout << T << endl;
```

```
T
%{
1,1,1;
1,1,1;
1,1,1;
}%Base1
```

```
Tensor3D (   string name ,
             Matrix * mat ,
             Base * base ,
             System * system );
```

Constructor que genera un objeto de tipo *Tensor3D* con un nombre, un puntero a un objeto *Matrix* de (3x3), un puntero a un objeto *Base* y un puntero al objeto *System* al que está asociado.

Ejemplo:

```
Tensor3D T (    “T” ,
                & Matrix_uno ,
                & Base_uno ,
                & System_uno );

//Visualizamos el resultado
cout << T << endl;
```

```
T
%{
1,1,1;
1,1,1;
1,1,1;
}%Base1
```

```
Tensor3D (   string name ,
             ex exp1 , ex exp2 , ex exp3 ,
             ex exp4 , ex exp5 , ex exp6 ,
             ex exp7 , ex exp8 , ex exp9 ,
             Base * base ,
             System * system );
```

Constructor que genera un objeto de tipo *Tensor3D* con un nombre, 9 expresiones de tipo *GiNaC::ex* que compondrán la matrix de (3x3) , un puntero a un objeto *Base* y un puntero al objeto *System* al que está asociado.

Ejemplo:

```
Tensor3D T (    “T” ,
                1 , 2 , 3 ,
                4 , 5 , 6 ,
                7 , 8 , 9 ,
                & Base_uno ,
                & System_uno );

//Visualizamos el resultado
cout << T << endl;
```

```
T
%{
1,2,3;
4,5,6;
7,8,9;
}%Base1
```

- *Métodos de modificación y acceso*

`void set_Base ( Base * new_base );`

Método que asigna una nueva base al objeto *Tensor3D*. Esta base es un puntero a un objeto de tipo *Base*.

Ejemplo:

```
Tensor3D T (    “T” ,
                matrix_uno ,
                & Base_uno ,
                & System_uno );

T.set_Base ( & Base_dos );

//Visualizamos el resultado
cout << T.get_Base()-> get_name() << endl;
```

```
Base2
```

`Base * get_Base ( void );`

Método que retorna la base a la que pertenece el objeto *Tensor3D*.

Ejemplo:

```
Tensor3D T (    “T” ,
```

```
                    matrix_uno ,
                    & Base_uno ,
                    & System_uno );

  //Visualizamos el resultado
  cout << T.get_Base ()-> get_name() << endl;
```

```
  Base1
```

- *Métodos Públicos*

`Tensor3D subs ( relational relation );`

Método que substituye una expresión por otra en un objeto de tipo *Tensor3D* según se indique en el parámetro de tipo *GiNaC::relational*, y retorna otro objeto de tipo *Tensor3D* con la substitución realizada. El formato de los objetos *GiNaC::relational* son de tipo *"expresión1 == expresión2"*, donde se substituirán las apariciones de "*expresion1"* por "*expresion2"*.

Ejemplo:

```
  symbol x ( "x" );
  symbol y ( "y" );

  Tensor3D Tensor_A ( "A" ,
                      x + 5 , 3 , y ,
                      x + y , x , x ,
                      0 , 1 , 2*x,
                      & Base_uno);

  Tensor3D Tensor_B = Tensor_A.subs ( x == 10 );

  //Visualizamos el resultado
  cout << Tensor_B () << endl;
```

```
  Tensor_B
  %{
  15,3,y;
  10+y,10,10;
  0,1,20;
  }%Base1
```

- *Operadores*

```
friend Tensor3D operator + (   const Tensor3D & Tensor3DA ,
                               const Tensor3D & Tensor3DB );
```

Sobrecarga del operador "+" que permite realizar sumas entre objetos *Tensor3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse a la base
reducida para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Tensor3D T1 (   1 , 2 , 3 ,
                4 , 5 , 6 ,
                7 , 8 , 9 ,
                & Base_uno ,
                & System_uno );

Tensor3D T2 (   1 , 1 , 1 ,
                3 , 3 , 3 ,
                4 , 4 , 4 ,
                & Base_uno ,
                & System_uno );

Tensor3D T3 = T1 + T2 ;

//Visualizamos el resultado
cout << T3 << endl;
```

```
%{
2,3,4;
7,8,9;
11,12,13;
}%Base1
```

```
friend Tensor3D operator - (   const Tensor3D & Tensor3DA ,
                               const Tensor3D & Tensor3DB );
```

Sobrecarga del operador "-" que permite realizar restas entre objetos *Tensor3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse a la base
reducida para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Tensor3D T1 (   1 , 2 , 3 ,
                4 , 5 , 6 ,
                7 , 8 , 9 ,
                & Base_uno ,
                & System_uno );

Tensor3D T2 (   1 , 1 , 1 ,
                3 , 3 , 3 ,
                4 , 4 , 4 ,
                & Base_uno ,
                & System_uno );

Tensor3D T3 = T1 - T2 ;
```

```
//Visualizamos el resultado
cout << T3 << endl;
```

```
%{
0,1,2;
1,2,3;
3,4,5;
}%Base1
```

```
friend Tensor3D operator * (   const Tensor3D & Tensor3DA ,
                               const Tensor3D & Tensor3DB );
```

Sobrecarga del operador "*" que permite realizar multiplicaciones entre objetos *Tensor3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse a la base
reducida para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Tensor3D T1 (   1 , 2 , 3 ,
                4 , 5 , 6 ,
                7 , 8 , 9 ,
                & Base_uno ,
                & System_uno );

Tensor3D T2 (   1 , 1 , 1 ,
                3 , 3 , 3 ,
                4 , 4 , 4 ,
                & Base_uno ,
                & System_uno );

Tensor3D T3 = T * T2 ;

//Visualizamos el resultado
cout << T3 << endl;
```

```
%{
0,1,2;
1,2,3;
3,4,5;
}%Base1
```

```
friend Vector3D operator * (   const Tensor3D & Tensor3DA ,
                               const Vector3D & Vector3DA );
```

Sobrecarga del operador "*" que permite realizar multiplicaciones entre objetos *Tensor3D* y *Vector3D*.

Ejemplo:

```
/*Si los vectores entre los que se van a relazar las operaciones
tienen diferentes bases, estos deberán transformarse a la base
reducida para así peder operar. En el ejemplo las bases son las
mismas para que así se vea mas clara la operación*/

Tensor3D T (    1 , 2 , 3 ,
                4 , 5 , 6 ,
                7 , 8 , 9 ,
                & Base_uno ,
                & System_uno );

Vector3D V (    1 ,
                1 ,
                3 ,
                & Base_uno ,
                & System_uno );

Vector3D X = T * V;

//Visualizamos el resultado
cout << X << endl;
```

```
%{
12;
32;
42;
}%Base1
```

```
friend ostream & operator << (      ostream & os ,
                                    const Tensor3D & Tensor3DA );
```

Sobrecarga del operador "<<" que permite formatear la salida por la salida estándar de un objeto *Tensor3D*.

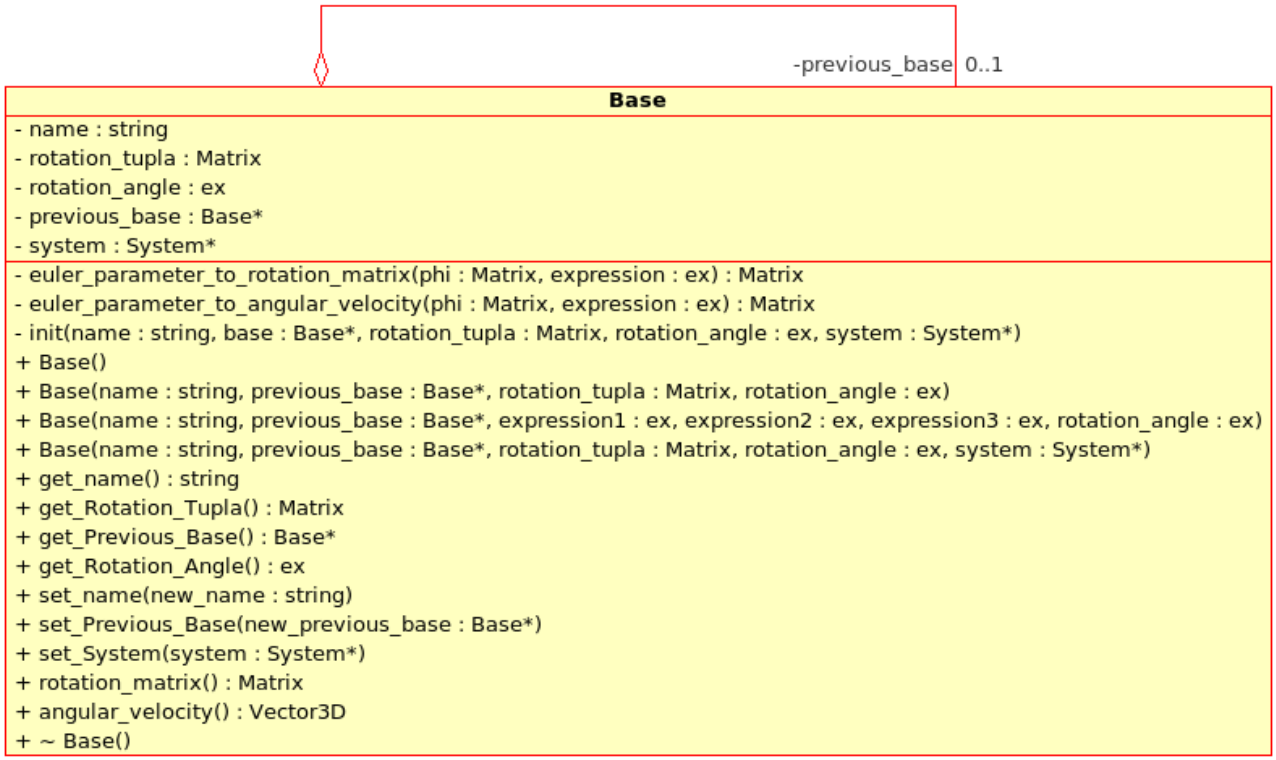
- *Destructor*

```
~Tensor3D ( void );
```

Destructor de la clase *Tensor3D*.

3.5 - Clase Base

3.5.1 - Diagrama de clase



3.5.2 - Elementos privados

- *Atributos*

`string name;`

Atributo que almacena el nombre del objeto *Base*.

`Matrix rotation_tupla;`

Atributo que almacén la tupla de rotación del objeto *Base*.

`ex rotation_angle;`

Atributo que almacena el ángulo de rotación del objeto *Base*.

`Base * previous_base;`

Atributo que almacena la base previa del objeto *Base*. Esta base previa es un puntero a un objeto de la clase *Base*.

`System * sys;`

Atributo que almacena el sistema al que pertenece la *Base*. Este sistema es un puntero a un objeto de la clase *System*

- *Métodos privados*

`matrix euler_parameter_to_rotation_matrix ( Matrix phi , ex expression );`

Método que realiza los cálculos necesarios para obtener la matriz de rotación dado un objeto de tipo *Matrix* y otro de tipo *GiNaC::ex*.

```
/*
Method that transform the euler parameter to rotation matrix
*/
Matrix Base::euler_parameter_to_rotation_matrix ( Matrix phi , ex expression ) {

      Matrix aux = phi.transpose () * phi;

      ex expresion1 = atomize_ex ( cos ( atomize_ex (
                                          expression * aux ( 0 , 0 ) ) ) );
      Matrix A  ( 3 , 3 , lst ( 1 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 1 ) );
      ex expresion2 = atomize_ex ( 1-cos ( atomize_ex (
                                          expression* ( aux ( 0 , 0 ) ) ) ) );
      Matrix B  ( 3 , 3 ,
                  lst ( 0 ,
                        -phi ( 2 , 0 ) ,
                        phi ( 1 , 0 ) ,
                        phi ( 2 , 0 ) ,
                        0 ,
                        -phi ( 0 , 0 ) ,
                        -phi ( 1 , 0 ) ,
                        phi ( 0 , 0 ) ,
                        0 ) );
      ex expresion3 = atomize_ex ( sin ( atomize_ex (
                                          expression* ( aux ( 0 , 0 ) ) ) ) );
      Matrix C = B.transpose ( void );

      return A * expresion1 + expresion2 * phi * phi.transpose () - expresion3 * C;
}
```

`Matrix euler_parameter_to_angular_velocity( Matrix phi , ex expression );`

Método que realiza los cálculos necesarios para obtener la velocidad angular dado un objeto de tipo *Matrix* y otro de tipo *GiNaC::ex*.

```
/*
Method that transform the euler parameter to angular velocity
*/
Matrix Base::euler_parameter_to_angular_velocity ( Matrix phi , ex expression ) {
      return Matrix ( 3 , 1 ,
                      lst ( system-> dt ( expression ) * phi ( 0 , 0 ) ,
                      system-> dt ( expression ) * phi ( 1 , 0 ) ,
                      system-> dt ( expression ) * phi ( 2 , 0 ) ) );
}
```

```
void init(   string name ,
             Base * previous_base ,
             Matrix rotation_tupla ,
             ex rotation_angle ,
             System * system );
```

Método que inicializa todos los elementos del objeto *Base*.

3.5.3 - Elementos públicos

- *Costructores*

```
Base ( void );
```

Constructor que genera un objeto de tipo Base sin inicializar sus atributos.

```
Base ( string name ,
       Base * previous_base ,
       Matrix rotation_tupla ,
       ex totation_amgle );
```

Constructor que genera un objeto de tipo *Base* con su nombre, un puntero a un objeto *Base* que indica su base predecesora, un objeto de tipo *Matrix* que indica la tupla de rotación y un objeto *GiNaC::ex* que indica el ángulo de rotación.

Ejemplo:

```
Base Base_uno (      “Base1” ,
                     ”xyz” ,
                     Matrix_uno ,
                     angulo_uno );
```

```
Base ( string name ,
       Base * previous_base ,
       ex expression1 ,
       ex expression2 ,
       ex expression3 ,
       ex rotation_angle );
```

Constructor que genera un objeto de tipo *Base* con su nombre, su *"Base predecesora"*, las expresiones de tipo *GiNaC::ex* que componen la tupla de rotación (3x1) y un objeto de tipo *GiNaC::ex* su ángulo de rotación.

Ejemplo:

```
Base Base_dos (      "Base2" ,
                     & Base_dos ,
                     7 ,
                     8 ,
                     9 ,
                     angulo_uno );
```

```
Base ( string name ,
       Base * previous_base ,
       Matrix rotation_tupla ,
       ex rotation_angle ,
       System * system );
```

Constructor que genera un objeto de tipo *Base* con su nombre, un puntero a un objeto Base que representa su base predecesora, un objeto de tipo *Matrix* que es la tupla de rotación, un objeto de tipo *GiNaC::ex* para el ángulo de rotación y un puntero a un objeto *System* que representa el sistema que tiene asignado.

Ejemplo:

```
Base Base_uno ( "Base1" ,
                "xyz" ,
                Matrix_uno ,
                angulo_uno ,
                & System_uno );
```

- *Métodos de modificación y acceso*

```
string get_name ( void );
```

Método que retorna el nombre del objeto *Base*.

Ejemplo:

```
Base Base_uno ( "Base1" ,
```

```
                    ”xyz” ,
                    Matrix_uno ,
                    angulo_uno ,
                    & System_uno );

    //Visualizamos el resultado
    cout << Base_uno.get_name () << endl;
```

```
Base1
```

`Matrix get_Rotation_Tupla ( void );`

Método que retorna la tupla de rotación del objeto *Base*.

`Base* get_Previous_Base ( void );`

Método que retorna la base previa del objeto *Base*.

Ejemplo:

```
Base Base_uno ( “Base1” ,
                ”xyz” ,
                Matrix_uno ,
                angulo_uno ,
                & System_uno );

//Visualizamos el resultado
cout << Base_uno.get_Previous_Base()-> get_name () << endl;
```

```
xyz
```

`ex get_Rotation_Angle ( void );`

Método que retorna el ángulo de rotación del objeto *Base*.

`void set_name ( string new_name );`

Método que asigna un nuevo nombre al objeto *Base*.

Ejemplo:

```
Base Base_uno ( “Base1” ,
                ”xyz” ,
                Matrix_uno ,
                angulo_uno ,
                & System_uno );

Base_uno.set_name ( “Base_1_” );
```

```
//Visualizamos el resultado
cout << Base_uno.get_name () << endl;
```

```
Base_1_
```

`void set_System ( System * system );`

Método que asigna un nuevo sistema a la *Base*. Este sistema es un puntero a un objeto de tipo *System*

`void set_Previous_Base ( Base * new_previous_base );`

Método que asigna una nueva base previa al objeto *Base*.

Ejemplo:

```
Base Base_uno( "Base1",
               "xyz",
               Matrix_uno,
               angulo_uno,
               & System_uno );

Base_uno.set_Previous_Base(&Base_dos);

//Visualizamos el resultado
cout << Base_uno.get_Previous_Base()->get_name() << endl;
```

```
Base2
```

- *Métodos públicos*

`Matrix rotation_matrix ( void );`

Método que retorna la matriz de rotación del objeto *Base*. Esta matriz de rotación es de tipo *Matrix*.

`Vector3D angular_velocity ( void );`

Método que retorna la velocidad angular del objeto *Base*. Esta velocidad angular es de tipo *Vector3D*.

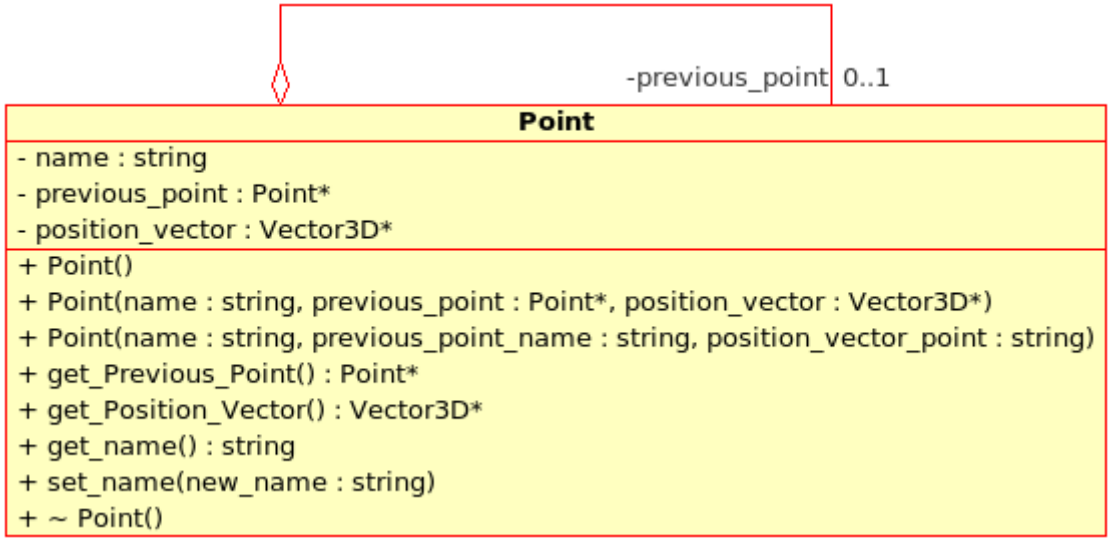
- *Destructor*

~Base (void);

Destructor de la clase *Base*.

3.6 - Clase Point

3.6.1 - Diagrama de clase



3.6.2 - Elementos privados

- *Atributos*

`string name;`

Atributo que almacena el nombre del objeto *Point*.

`Point * previous_Point;`

Atributo que almacena el punto predecesor del objeto *Point*.

`Vector3D * position_vector;`

Atributo que almacena el vector de posición del objeto *Point*.

3.6.3 - Elementos públicos

- *Constructores*

```
Point ( void );
```

Constructor que genera objeto de tipo *Point* sin inicializar sus atributos.

```
Point (  string name ,
         Point * previous_point ,
         Vector3D * position_vector );
```

Constructor que genera objeto de tipo *Point* con su nombre, su punto predecesor de tipo puntero a un objeto *Point* y un vector de posición de tipo puntero a objeto *Vector3D*

Ejemplo:

```
Point P (       "Point1" ,
                & Point_uno ,
                & Vector3D_uno );
```

- *Métodos de modificación y acceso*

```
Point * get_Previous_Point ( void );
```

Método que retorna el punto predecesor que es de tipo puntero a un objeto *Point*.

Ejemplo:

```
Point P ( "Point1" , "O" , "Vector3D1" );

//Visualizamos el resultado
cout << P.get_Previous_Point()-> get_name () << endl;
```

```
O
```

```
Vector3D * get_Position_Vector ( void );
```

Método que retorna el vector de posición que es de tipo puntero a un objeto *Vector3D*

Ejemplo:

```
Point P ( "Point1" , "O" , "Vector3D1" );

//Visualizamos el resultado
cout << P.get_Position_Vector()-> get_name() << endl;
```

```
Vector3D1
```

`string get_name ( void );`

Método que retorna el nombre del objeto *Point*.

Ejemplo:

```
Point P ( “Point1” , ”O” , ”Vector3D1” );

//Visualizamos el resultado
cout << P.get_name () << endl;
```

```
Point1
```

`void set_name ( string new_name );`

Método que asigna un nuevo nombre al objeto *Point*.

Ejemplo:

```
Point P ( “Point1” , ”O” , ”Vector3D1” );

p.set_name (  “Point_1_”  );

//Visualizamos el resultado
cout << P.get_name () << endl;
```

```
Point_1_
```

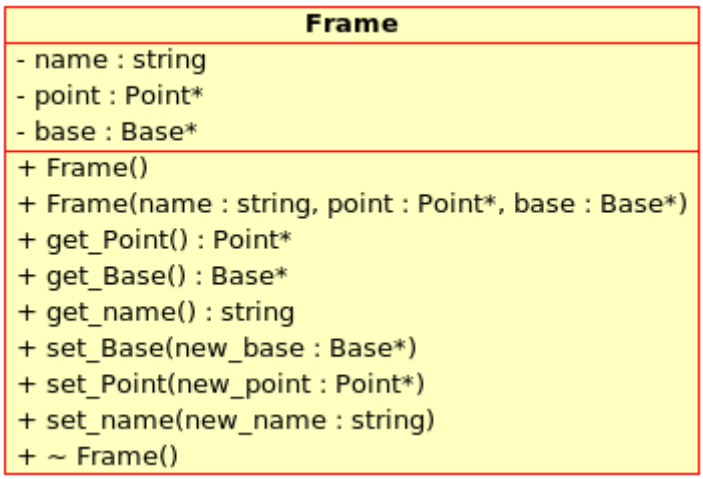
- *Destructor*

`~Point ( void );`

Destructor de la clase *Point*

3.7 - Clase Frame

3.7.1 - Diagrama de clase



3.7.2 - Elementos privados

- *Atributos*

`string name;`

Atributo que almacena el nombre del objeto *Frame*.

`Point * point;`

Atributo que almacena el *Point* del objeto *Frame*.

`Base * base;`

Atributo que almacena la *Base* del objeto *Frame*.

3.7.3 - Elementos públicos

- *Constructores*

`Frame ( void );`

Constructor que genera un objeto de tipo *Frame* sin inicializar sus atributos.

```
Frame (   string name ,
          Point * point ,
          Base * base );
```

Constructor que genera un objeto de tipo *Frame* con su nombre, su punto que es un puntero a un objeto *Point* y su base que es un puntero a un objeto de tipo *Base*.

Ejemplo:

```
Frame F (       "Frame1" ,
                & Point_uno ,
                & Base_uno );
```

- *Métodos de modificación y acceso*

```
Point* get_Point ( void );
```

Método que retorna el objeto *Point* que tiene asignado el objeto F*ra*me.

Ejemplo:

```
Frame F (       "Frame1" ,
                & Point_uno ,
                & Base_uno );

//Visualizamos el resultado
cout << F.get_Point ()-> get_name () << endl;
```

```
Point1
```

```
Base * get_Base ( void );
```

Método que retorna el objeto *Base* que tiene asignado el objeto *Frame*

Ejemplo:

```
Frame F (       "Frame1" ,
                & Point_uno ,
                & Base_uno );

//Visualizamos el resultado
cout << F.get_Base ()-> get_name () << endl;
```

```
Base1
```

`string get_name ( void );`

Método que retorna el nombre que tiene asignado el objeto *Frame*.

Ejemplo:

```
Frame F (       “Frame1” ,
                & Point_uno ,
                & Base_uno );

//Visualizamos el resultado
cout << F.get_name () << endl;
```

```
Frame1
```

`void set_Base ( Base * new_base );`

Método que asigna un nuevo puntero a un objeto *Base* al objeto de tipo Fr*a*me

Ejemplo:

```
Frame F (       “Frame1” ,
                & Point_uno ,
                & Base_uno );

F.set_Base ( & Base_dos )

//Visualizamos el resultado
cout << F.get_Base ()-> get_name << endl;
```

```
Base2
```

`void set_Point ( Point * new_point );`

Método que asigna un nuevo puntero a un objeto *Point* al objeto de tipo *Frame*.

Ejemplo:

```
Frame F (       “Frame1” ,
                & Point_uno ,
                & Base_uno );

F.set_Point ( & Point_dos )

//Visualizamos el resultado
cout << F.get_Point ()-> get_name << endl;
```

```
Point2
```

`void set_name ( string new_name );`

Método que asigna un nuevo nombre al objeto de tipo *Frame*

Ejemplo:

```
Frame F (       “Frame1” ,
                & Point_uno ,
                & Base_uno );

F.set_name ( “Frame_1_” )

//Visualizamos el resultado
cout << F.get_Base ()-> get_name << endl;
```

```
Frame_1_
```

- *Destructor*

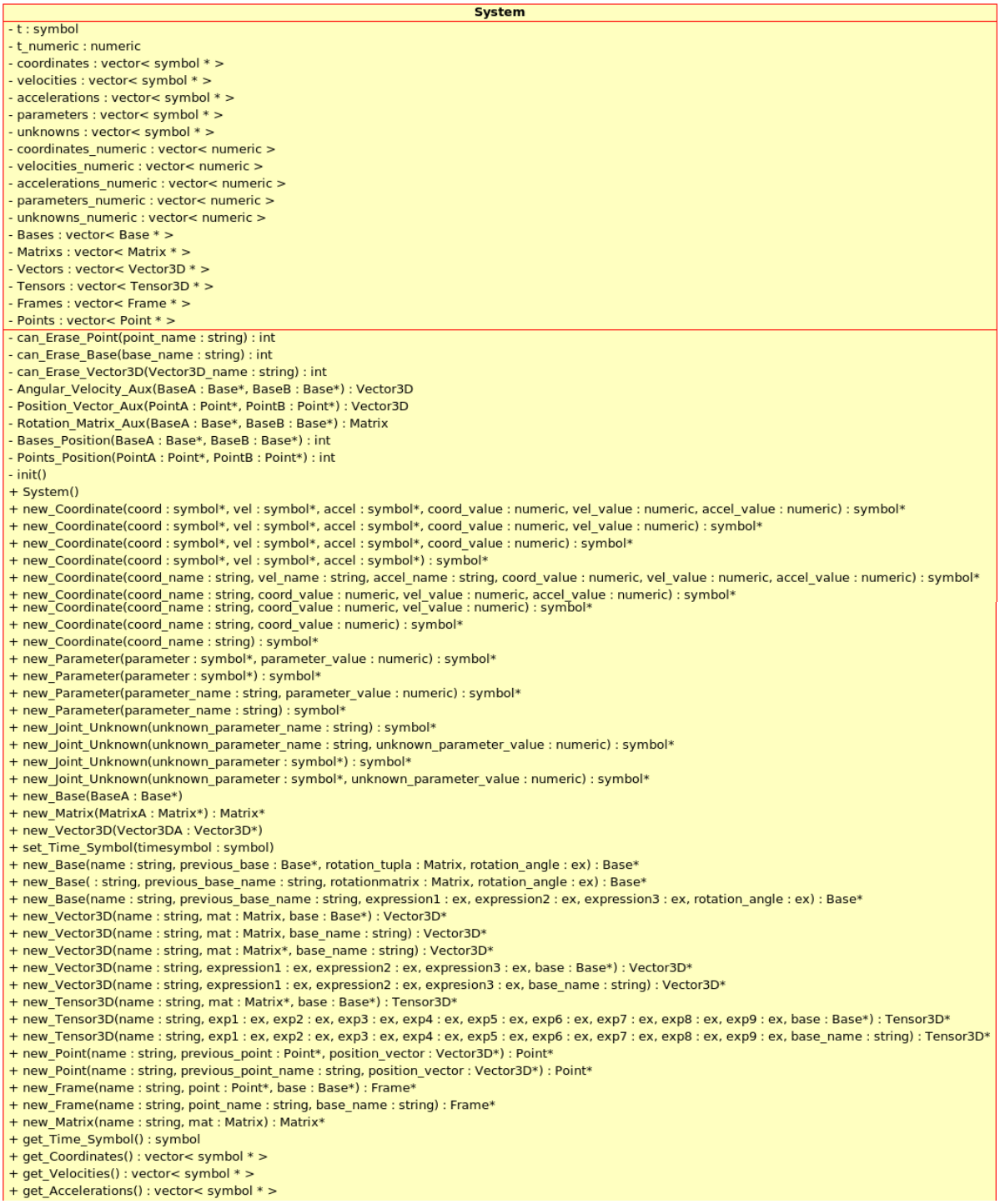
`~Frame ( void );`

Destructor de la clase *Frame*.

3.8 - Clase System

La clase *System* es una de las clases más importantes de la librería desarrollada, ya que en esta se almacenan todos los objetos para que puedan interrelacionarse entre sí, además de dar la posibilidad de realizar múltiples operaciones con estos objetos.

3.8.1 - Diagrama de clase



+ get_Parameters() : vector< symbol * >
+ get_Unknowns() : vector< symbol * >
+ get_Bases() : vector< Base * >
+ get_Matrixs() : vector< Matrix * >
+ get_Vectors() : vector< Vector3D * >
+ get_Tensors() : vector< Tensor3D * >
+ get_Points() : vector< Point * >
+ get_Frames() : vector< Frame * >
+ Coordinates() : Matrix
+ Accelerations() : Matrix
+ Velocities() : Matrix
+ Joint_Unknowns() : Matrix
+ get_Coordinate(coordinate_name : string) : symbol*
+ get_Velocity(velocity_name : string) : symbol*
+ get_Acceleration(acceleration_name : string) : symbol*
+ get_Parameter(parameter_name : string) : symbol*
+ get_Unknown(unknown_parameter_name : string) : symbol*
+ get_Base(base_name : string) : Base*
+ get_Frame(frame_name : string) : Frame*
+ get_Matrix(Matrix_name : string) : Matrix*
+ get_Vector3D(vector3D_name : string) : Vector3D*
+ get_Point(point_name : string) : Point*
+ Reduced_Base(BaseA_name : string, BaseB_name : string) : Base*
+ Reduced_Base(BaseA : Base*, BaseB : Base*) : Base*
+ Reduced_Point(PointA_name : string, PointB_name : string) : Point*
+ Reduced_Point(PointA : Point*, PointB : Point*) : Point*
+ Rotation_Matrix(BaseA : Base*, BaseB : Base*) : Matrix
+ Position_Vector(PointA : Point*, PointB : Point*) : Vector3D
+ Position_Vector(PointA_name : string, PointB_name : string) : Vector3D
+ Angular_Velocity(BaseA : Base*, BaseB : Base*) : Vector3D
+ Angular_Velocity(base_frame_nameA : string, base_frame_nameB : string) : Vector3D
+ Angular_Velocity_Tensor(BaseA : Base*, BaseB : Base*) : Tensor3D
+ Angular_Acceleration(base_frame_nameA : string, base_frame_nameB : string) : Vector3D
+ remove_Matrix(matrix_name : string)
+ remove_Vector3D(vector3D_name : string)
+ remove_Point(point_name : string)
+ remove_Base(base_name : string)
+ dt(expression : ex) : ex
+ is_dt_zero(expression : ex) : bool
+ Dt(MatrixA : Matrix) : Matrix
+ Dt(Vector3DA : Vector3D, base : Base*) : Vector3D
+ Dt(Vector3DA : Vector3D, frame : Frame*) : Vector3D
+ Dt(Vector3DA : Vector3D, base_frame_name : string) : Vector3D
+ get_Time_Value() : numeric
+ set_Time_Value(time_value : numeric)
+ jacobian(MatrixA : Matrix, MatrixB : Matrix) : Matrix
+ jacobian(: ex, MatrixA : Matrix) : Matrix
+ jacobian(MatrixA : Matrix, symbolA : symbol) : Matrix
+ jacobian(expression : ex, symbolA : symbol) : ex
+ diff(expression : ex, symbolA : symbol) : ex
+ diff(Matrixa : Matrix, symbolA : symbol) : Matrix
+ diff(Vector3DA : Vector3D, symbolA : symbol) : Vector3D
+ diff(Tensor3DA : Tensor3D, symbolA : symbol) : Tensor3D
+ numeric_evaluate(expression : ex) : ex
+ evaluate_Matrix(MatrixA : Matrix) : Matrix
+ evaluate_Array(MatrixA : Matrix) : double**
+ print_Array(rows : long, cols : long, array : double**) : string
+ export_var_def_C()
+ export_var_def_H()
+ export_var_init_C()
+ export_atom_def_C(atom_list : lst)
+ export_gen_coord_vect_def_H()
+ export_gen_coord_vect_init_C()
+ export_gen_vel_vect_init_H()
+ export_gen_vel_vect_init_C()
+ export_gen_accel_vect_init_H()
+ export_gen_accel_vect_init_C()
+ export_gen_accel_vect_init_C()
+ export_unknowns_vect_init_H()
+ export_unknowns_vect_init_C()
+ export_Column_Matrix_C(: string, : string, : Matrix, : lst)
+ export_Matrix_C(: string, : string, : Matrix, : lst)
+ export_write_state_file_header_C()
+ export_write_state_file_C()
+ ~ System()

3.8.2 - Elementos privados

- *Atributos*

```
symbol t;
```

Atributo que identifica al tiempo en el objeto *System*. El tiempo es del tipo *GiNaC::symbol*.

```
numeric t_numeric;
```

Atributo que identifica el valor numérico del tiempo en el objeto System. Este valor es del tipo *GiNaC::numeric*.

```
vector < symbol * > coordinates;
```

Vector que almacena punteros a coordenadas en el objeto *System*. Estas coordenadas son de tipo *GiNaC::symbol*.

```
vector < symbol * > velocities;
```

Vector que almacena punteros a velocidades. Estas velocidades son de tipo *GiNaC::symbol*.

```
vector < symbol * > accelerations;
```

Vector que almacena punteros a aceleraciones. Estas velocidades son de tipo *GiNaC::symbol*.

```
vector < symbol * > parameters;
```

Vector que almacena punteros a parámetros. Estos parámetros son de tipo *GiNaC::symbol*.

```
vector < symbol * > unknowns;
```

Vector que almacena punteros a parámetros desconocidos. Estos parámetros desconocidos son de tipo *GiNaC::symbol*.

```
vector < numeric > coordinates_numeric;
```

Vector que almacena los valores numéricos de las coordenadas en el objeto *System*. Estos valores numéricos son de tipo *GiNaC::numeric*.

```
vector < numeric > velocities_numeric;
```

Vector que almacena los valores numéricos de las velocidades en el objeto *System*. Estos valores numéricos son de tipo *GiNaC::numeric*.

```
vector < numeric > accelerations_numeric;
```

Vector que almacena los valores numéricos de las aceleraciones en el objeto *System*. Estos valores numéricos son de tipo *GiNaC::numeric*.

```
vector < numeric > parameters_numeric;
```

Vector que almacena los valores numéricos de los parámetros en el objeto *System*. Estos valores numéricos son de tipo *GiNaC::numeric*.

```
vector < numeric > unknowns_numeric;
```

Vector que almacena los valores numéricos de los parámetros desconocidos en el objeto *System*. Estos valores numéricos son de tipo *GiNaC::numeric*.

```
vector < Base * > Bases;
```

Vector que almacena punteros a objetos de tipo *Base* introducidos en el objeto *System*.

```
vector < Matrix * > Matrixes;
```

Vector que almacena punteros a objetos *Matrix* introducidos en el objeto *System*.

```
vector < Vector3D * > Vectors;
```

Vector que almacena punteros a objetos *Vector3D* introducidos en el objeto *System*.

```
vector < Tensor3D * > Tensors;
```

Vector que almacena punteros a objetos *Tensor3D* introducidos en el objeto *System*.

```
vector < Frame * > Frames;
```

Vector que almacena punteros a objetos *Frame* introducidos en el objeto *System*.

```
vector < Point * > Points;
```

Vector que almacena punteros a objetos *Point* introducidos en el objeto *System*.

- *Métodos privados*

```
int can_Erase_Point ( string point_name );
```

Método que nos dice si un objeto *Point* puede ser borrado del objeto Syst**e**m, debido a las interrelaciones que puede haber entre los objetos.

```
int can_Erase_Base ( string base_name );
```

Método que nos dice si un objeto *Base* puede ser borrado del objeto *System*, debido a las interelaciones que puede haber entre los objetos.

```
int can_Erase_Vector3D ( string Vector3D_name );
```

Método que nos dice si un objeto *Vector3D* puede ser borrado del objeto *System*, debido a las interelaciones que puede haber entre los objetos.

```
Vector3D Angular_Velocity_Aux ( Base * BaseA , Base * BaseB );
```

Método auxiliar que ayuda a calcular la velocidad angular en el método público *Angular_Velocity*. La función de este método es la de ir sumando las velocidades angulares de las bases desde el objeto *Base* que se pasa en segunda posición que es la base origen, hasta el objeto *Base* que se pasa en primera posición que es el destino y devolver el resultado en forma de *Vector3D*.

```
Vector3D Position_Vector_Aux( Point * PointA , Point * PointB );
```

Método auxiliar que ayuda a calcular el vector de posición en el método público *Position_Vector*. La función de este método es la de ir sumando los vectores de posición de los objetos *Point*, desde el segundo objeto *Point* que se pasa como parámetro o punto origen, hasta el objeto *Point* que se pasa en primera posición o destino y devolver el resultado en forma de objeto *Vector3D*.

```
Matrix Rotation_Matrix_Aux ( Base * BaseA , Base * BaseB );
```

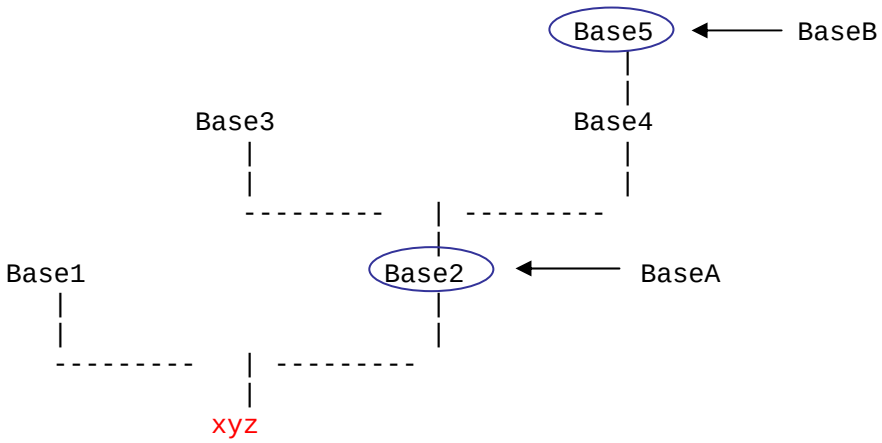
Método auxiliar que ayuda a calcular la matriz de rotación en el método público *Rotatino_Matrix*. La función de este método es la de ir multiplicando las matrices de rotación de las bases desde el segundo objeto *Base* que se pasa como parámetro u origen, hasta el primer objeto *Base* que se pasa como parámetro o destino y devolver el resultado en forma de objeto *Matrix*.

```
int Bases_Position ( Base * BaseA , Base * BaseB );
```

Este método nos dice si un objeto *Base* está en primer o segundo lugar empezando a buscar desde el objeto *Base* cuyo nombre es *"xyz"*. Si retorna 0, la primera *Base* se encuentra más cerca del objeto *Base* con nombre *"xyz"* que el segundo objeto *Base*, en caso contrario, retornará 1.

Ejemplo:

Ante el siguiente árbol de objetos Base



podemos ver los siguientes ejemplos de funcionamiento

```
//Visualizamos los resultados.
cout << Bases_Position ( & Base2 , & Base5 ) << endl;
```

```
0
```

```
cout << Bases_Position ( & Base5 , & Base2 ) << endl;
```

```
1
```

```
int Points_Position ( Point * PointA , Point * PointB );
```

Este método nos dice si un objeto *Point* está en primer o segundo lugar empezando desde el *Point* cuyo nombre es *"O"*. Si retorna 0, el primer *Point* esta más cerca del objeto *Point* con nombre *"0"* que el segundo *Point*, en caso contrario, retornará 1. El funcionamiento de este método es muy similar al de *Bases_Position*.

```
void init( void ( * ) ( char * ) );
```

Método que inicializa los atributos de la clase *System*. Recibe como parámetro un puntero a la función que será la encargada de gestionar los errores que se produzcan.

3.8.3 - Elementos públicos

- *Constructores*

```
System ( void );
```

Constructor de la clase *System*, que genera un sistema con la función de gestión de errores por defecto llamada *printError*.

```
System ( void ( * ) ( char * ) );
```

Constructor de la clase *System*, que genera un sistema con la función de gestión de errores que le pasamos como parámetro.

- *Métodos públicos*

```
symbol * new_Coordinate (      symbol * coordinate ,
                               symbol * velocity ,
                               symbol * acceleration ,
                               numeric coordinate_value,
                               numeric velocity_value ,
                               numeric acceleration_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración siendo todos ellos de tipo puntero a objeto GiNaC::symbol, seguidas de sus correspondientes valores numéricos de tipo *GiNaC::numeric*.

Ejemplo:

```
symbol x ( “x” );
symbol dx ( “dx” );
symbol ddx ( “ddx” );

/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del método new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    & x , & dx , & ddx ,
                                             1 , 2 , 3 );
```

Se crea:

Coordenada *"x"* con el valor 1
Velocidad *"dx"* con el valor 2
Aceleración *"ddx"* con el valor 3

```
symbol * new_Coordinate (      symbol * coordinate ,
                               symbol* velocity ,
                               symbol * acceleration ,
                               numeric coordinate_value ,
                               numeric velocity_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración siendo todos ellos de tipo puntero a objeto GiNaC::symbol, seguidas de los valores numéricos de la coordenada y la velocidad, siendo el valor numérico de la aceleración 0 y el tipo de estos valores *GiNaC::numeric*.

Ejemplo:

```
symbol x ( "x" );
symbol dx ( "dx" );
symbol ddx ( "ddx" );

/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del metodo new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    & x , & dx , & ddx ,
                                             1 , 2 );
```

Se crea:

Coordenada *"x"* con el valor 1
Velocidad *"dx"* con el valor 2
Aceleración *"ddx"* con el valor 0

```
symbol * new_Coordinate (      symbol * coordinate ,
                               symbol * velocity ,
                               symbol * acceleration ,
                               numeric coordinate_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración, siendo todos ellos de tipo puntero a objeto GiNaC::symbol, seguidas del valor numérico de la coordenada, siendo el valor numérico de la velocidad y de la aceleración 0, y el tipo del valor de la coordenada *GiNaC::numeric*.

Ejemplo:

```
symbol x ( "x" );
```

```
symbol dx ( “dx” );
symbol ddx ( “ddx” );

/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del método new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    & x , & dx , & ddx ,
                                             4 );
```

Se crea:

Coordenada *“x”* con el valor 4
Velocidad *“dx”* con el valor 0
Aceleración *“ddx”* con el valor 0

```
symbol * new_Coordinate (      symbol * coordinate ,
                               symbol * velocity ,
                               symbol * acceleration );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración, siendo todos ellos de tipo puntero a objeto GiNaC::symbol. El valor numérico de la coordenada, la velocidad y la aceleración será 0.

Ejemplo:

```
symbol x ( “x” );
symbol dx ( “dx” );
symbol ddx ( “ddx” );

/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del método new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate ( & x , & dx , & ddx );
```

Se crea:

Coordenada *“x”* con el valor 0
Velocidad *“dx”* con el valor 0
Aceleración *“ddx”* con el valor 0

```
symbol * new_Coordinate (      string coordinate_name ,
                               string velocity_name ,
                               string acceleration_name ,
                               numeric coordinate_value ,
                               numeric velocity_value ,
                               numeric acceleration_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración representados mediante cadenas de texto de tipo *stl::string*, seguidas de sus

correspondientes valores numéricos de tipo *GiNaC::numeric*.

Ejemplo:

```
/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del metodo new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    “x” , ”dx” , ”ddx” ,
                                             1 , 2 , 3 );
```

Se crea:

Coordenada *“x”* con el valor 1
Velocidad *“dx”* con el valor 2
Aceleración *“ddx”* con el valor 3

```
symbol * new_Coordinate(       string coordinate_name ,
                               numeric coordinate_value ,
                               numeric velocity_value ,
                               numeric acceleration_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración, seguidas de sus correspondientes valores numéricos de tipo *GiNaC::numeric*. La cadena de texto, representa a la coordenada y la velocidad y la aceleración se generan automáticamente añadiendo el carácter *“d”* al nombre de la coordenada para la velocidad y añadiendo los caracteres *“dd”* al nombre de la coordenada para la aceleración.

Ejemplo:

```
/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del metodo new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    “x” ,
                                             1 , 2 , 3 );
```

Se crea:

Coordenada *“x”* con el valor 1
Velocidad *“dx”* con el valor 2
Aceleración *“ddx”* con el valor 3

```
symbol * new_Coordinate (      string coordinate_name ,
                               numeric coordinate_value ,
                               numeric velocity_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración, seguidas de los valores numéricos de la coordenada y de la velocidad, siendo 0 el

valor asignado a la aceleración. La cadena de texto, representa a la coordenada y la velocidad y la aceleración se generan automáticamente añadiendo el carácter *"d"* al nombre de la coordenada para la velocidad y añadiendo los caracteres *"dd"* al nombre de la coordenada para la aceleración.

Ejemplo:

```
/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del metodo new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    "x" ,
                                             1 , 2 );
```

Se crea:

Coordenada *"x"* con el valor 1
Velocidad *"dx"* con el valor 2
Aceleración *"ddx"* con el valor 0

```
symbol * new_Coordinate (      string coordinate_name ,
                               numeric coordinate_value );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una aceleración, seguidas del valor numérico de la coordenada, siendo 0 el valor asignado a la velocidad y a la aceleración. La cadena de texto, representa a la coordenada y la velocidad y la aceleración se generan automáticamente añadiendo el carácter *"d"* al nombre de la coordenada para la velocidad y añadiendo los caracteres *"dd"* al nombre de la coordenada para la aceleración.

Ejemplo:

```
/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del metodo new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate (    "x" ,
                                             1 );
```

Se crea:

Coordenada *"x"* con el valor 1
Velocidad *"dx"* con el valor 0
Aceleración *"ddx"* con el valor 0

```
symbol * new_Coordinate ( string coordinate_name );
```

Método que introduce en el objeto *System* una coordenada, una velocidad y una

aceleración, siendo 0 el valor asignado a las mismas. La cadena de texto, representa a la coordenada y la velocidad y la aceleración se generan automáticamente añadiendo el carácter *"d"* al nombre de la coordenada para la velocidad y añadiendo los caracteres *"dd"* al nombre de la coordenada para la aceleración.

Ejemplo:

```
/*
Las coordenadas se pueden crear igualmente si no asignamos la
salida del metodo new_Coordinate a ninguna variable.
*/
symbol coord_uno = * sys.new_Coordinate ( "x" );
```

Se crea:

Coordenada *"x"* con el valor 0
Velocidad *"dx"* con el valor 0
Aceleración *"ddx"* con el valor 0

```
Symbol  * new_Parameter (      symbol * parameter ,
                               numeric parameter_value );
```

Método que introduce en el objeto *System* un parámetro de tipo puntero a *GiNaC::symbol* seguido de su valor numérico de tipo *GiNaC::numeric*.

Ejemplo:

```
symbol p( "p" );

/*
El parámetro se puede crear igualmente si no asignamos la salida
del método new_Parameter a ninguna variable.
*/
symbol coord_uno = * sys.new_Parameter (     & p ,
                                             1 );
```

Se crea:

Parámetro *"p"* con el valor 1

```
Symbol * new_Parameter ( symbol * parameter );
```

Método que introduce en el objeto *System* un parámetro de tipo puntero a *GiNaC::symbol*, cuyo valor numérico es 0.

Ejemplo:

```
symbol p ( “p” );

/*
El parámetro se puede crear igualmente si no asignamos la salida
del método new_Parameter a ninguna variable.
*/
symbol param_uno = * sys.new_Parameter ( & p );
```

Se crea:

Parámetro *“p”* con el valor 0

```
Symbol * new_Parameter (        string parameter_name ,
                                numeric parameter_value );
```

Método que introduce en el objeto *System* un parámetro representado por una cadena de texto de tipo *stl::string* seguido de su valor numérico de tipo *GiNaC::numeric*.

Ejemplo:

```
/*
El parámetro se puede crear igualmente si no asignamos la salida
del método new_Parameter a ninguna variable.
*/
symbol param_uno = * sys.new_Parameter (     “p” ,
                                             1 );
```

Se crea:

Parámetro *“p”* con el valor 1

```
symbol * new_Parameter( string parameter_name );
```

Método que introduce en el objeto *System* un parámetro representado por una cadena de texto de tipo *stl::string* cuyo valor numérico es 0.

Ejemplo:

```
/*
El parámetro se puede crear igualmente si no asignamos la salida
del método new_Parameter a ninguna variable.
*/
symbol param_uno = * sys.new_Parameter( “p” );
```

Se crea:

Parámetro *“p”* con el valor 0

```
symbol * new_Joint_Unknown ( string unknown_parameter_name );
```

Método que introduce en el objeto *System* un parámetro desconocido representado por una cadena de texto de tipo *stl::string*, cuyo valor numérico es 0.

Ejemplo:

```
/*
El parámetro desconocido se puede crear igualmente si no asignamos
la salida del método new_Joint_Unknown a ninguna variable.
*/
symbol unknown_param_uno = * sys.new_Joint_Unknown ( “u” );
```

Se crea:

Parámetro *“u”* con el valor 0

```
symbol * new_Joint_Unknown (   string unknown_parameter_name ,
                               numeric unknown_parameter_value );
```

Método que introduce en el objeto *System* a un parámetro desconocido representado por una cadena de texto de tipo *stl::string* seguido de su valor numérico de tipo *GiNaC::numeric*.

Ejemplo:

```
/*
El parámetro desconocido se puede crear igualmente si no asignamos
la salida del método new_Joint_Unknown a ninguna variable.
*/
symbol unknown_param_uno = * sys.new_Joint_Unknown (     “u” ,
                                                         5 );
```

Se crea:

Parámetro *“u”* con el valor 5

```
symbol * new_Joint_Unknown ( symbol * unknown_parameter );
```

Método que introduce en el objeto *System* un parámetro desconocido de tipo puntero a objeto *GiNaC::symbol*, cuyo valor numérico es 0.

Ejemplo:

```
symbol u ( “u” );
```

```
/*
El parámetro desconocido se puede crear igualmente si no asignamos
la salida del método new_Joint_Unknown a ninguna variable.
*/
symbol unknown_param_uno = * sys.new_Joint_Unknown ( & u );
```

Se crea:

Parámetro "*u*" con el valor 0

```
symbol * new_Joint_Unknown (   symbol * unknown_parameter ,
                               numeric unknown_parameter_value );
```

Método que introduce en el objeto *System* un parámetro desconocido de tipo puntero a tipo *GiNaC::symbol* seguido de su valor numérico de tipo *GiNaC::numeric*.

Ejemplo:

```
symbol u ( "u" );

/*
El parámetro desconocido se puede crear igualmente si no asignamos
la salida del método new_Joint_Unknown a ninguna variable.
*/
symbol unknown_param_uno = * sys.new_Joint_Unknown (     & u ,
                                                         3 );
```

Se crea:

Parámetro "*u*" con el valor 3

```
void new_Base( Base * BaseA );
```

Método que introduce en el sistema un puntero a un objeto de tipo *Base*.

Ejemplo:

```
//Con este método añadimos la Base Base_uno al objeto System.
sys.new_Base( & Base_uno );
```

```
void new_Matrix( Matrix * MatrixA );
```

Método que introduce en el objeto *System* a un puntero a un objeto de tipo *Matrix*.

Ejemplo:

```
//Con este método añadimos la Matrix Matrix_uno al objeto
```

```
System.
sys.new_Matrix ( & Matrix_uno );
```

void new_Vector3D(Vector3D * Vector3DA);

Método que introduce en el objeto *System* un puntero a un objeto de tipo *Vector3D*.

Ejemplo:

```
//Con este método añadimos el Vector3D Vector3D_uno al objeto
System.
sys.new_Vector3D ( & Vector3D_uno );
```

void set_Time_Symbol (symbol time);

Método que modifica en el objeto *System* el símbolo que representa el tiempo.

Ejemplo:

```
symbol h ( "h" );
sys.set_Time_Symbol ( h );
```

```
Base* new_Base (   string name ,
                   Base * previous_base ,
                   Matrix rotation_tupla ,
                   ex rotation_angle );
```

Método que genera un nuevo objeto de tipo *Base* introduciendo como parámetro el nombre de la misma, su base predecesora, la matriz de rotación y el ángulo de rotación para finalmente introducirla en el sistema.

Ejemplo:

```
/*
Con este método añadimos una nueva Base al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Base Base_uno = * sys.new_Base (    "Base1" ,
                                    sys.get_Base ( "xyz" ) ,
                                    Matrix_uno ,
                                    expression1 );
```

```
Base* new_Base (   string name ,
                   string previous_base_name ,
                   Matrix rotation_matrix ,
                   ex rotation_angle );
```

Método que genera una nueva base introduciendo como parámetro el nombre de la misma, el nombre de su base predecesora, la matriz de rotación y el ángulo de rotación para finalmente introducirla en el sistema.

Ejemplo:

```
/*
Con este método añadimos una nueva Base al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Base Base_uno = * sys.new_Base (    “Base1” ,
                                    “xyz” ,
                                    Matrix_uno ,
                                    expression1 );
```

```
Base * new_Base (  string name ,
                   string previous_base_name ,
                   ex expression1 ,
                   ex expression2 ,
                   ex expression3 ,
                   ex rotation_angle );
```

Método que genera una nueva base introduciendo como parámetro el nombre de la misma, el nombre de su base predecesora, los valores de la matriz de rotación (3x1) y el ángulo de rotación para finalmente introducirla en el sistema.

Ejemplo:

```
/*
Con este método añadimos una nueva Base al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Base Base_uno = * sys.new_Base (    “Base1” ,
                                    “xyz” ,
                                    1 ,
                                    1 ,
                                    1 ,
                                    expression1 );
```

```
Vector3D * new_Vector3D (      string name ,
                               matrix mat ,
                               Base * base );
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, la matriz, y la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Vector3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Vector3D Vector3D_uno = * sys.new_Vector3D (    “Vector3D1” ,
                                                matrix_uno ,
                                                & Base_uno );
```

```
Vector3D * new_Vector3D (       string name ,
                                matrix mat ,
                                string base_name );
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, la matriz, y el nombre de la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Vector3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Vector3D Vector3D_uno = * sys.new_Vector3D (    “Vector3D1” ,
                                                matrix_uno ,
                                                ”Base1” );
```

```
Vector3D * new_Vector3D (       string name ,
                                Matrix * mat ,
                                string base_name );
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, la Matrix, y el nombre de la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Vector3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Vector3D Vector3D_uno = * sys.new_Vector3D (    “Vector3D1” ,
                                                & Matrix_uno ,
                                                ”Base1” );
```

```
Vector3D* new_Vector3D (        string name ,
                                ex expression1 ,
                                ex expression2 ,
                                ex expression3 ,
                                Base * base );
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, la matriz, y el nombre de la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Vector3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Vector3D Vector3D_uno = * sys.new_Vector3D (    “Vector3D1” ,
                                                1 ,
                                                1 ,
                                                1 ,
                                                & Base_uno );
```

```
Vector3D* new_Vector3D (        string name ,
                                ex expression1 ,
                                ex expression2 ,
                                ex expression3 ,
                                string base_name );
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, los 3 valores que va a contener, y el nombre de la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Vector3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Vector3D Vector3D_uno = * sys.new_Vector3D (    “Vector3D1” ,
                                                1 ,
                                                1 ,
                                                1 ,
                                                ”Base1” );
```

```
Tensor3D * new_Tensor3D (       string name ,
                                Matrix * mat ,
                                Base * base );
```

Método que genera un nuevo *Tensor3D* introduciendo como parámetros su nombre, la *Matrix*, y la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Tensor3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Tensor3D Tensor3D_uno = * sys.new_Tensor3D (    “Tensor3D1” ,
                                                & Matrix_uno ,
                                                & Base_uno );
```

```
Tensor3D* new_Tensor3D (        string name ,
                                ex exp1 , ex exp2 , ex exp3 ,
                                ex exp4 , ex exp5 , ex exp6 ,
                                ex exp7 , ex exp8 , ex exp9 ,
                                Base * base );
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, los 9 valores que va a contener (3x3), y la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Tensor3D al objeto System.
No es necesario recoger la salida del método para que funcione
correctamente.
*/
Tensor3D Tensor3D_uno = * sys.new_Tensor3D (    “Tensor3D1” ,
                                                1 , 2 , 3 ,
                                                4 , 5 , 6 ,
                                                7 , 8 , 9 ,
                                                & Base_uno );
```

```
Tensor3D* new_Tensor3D (        string name ,
                                ex exp1 , ex exp2 , ex exp3 ,
                                ex exp4 , ex exp5 , ex exp6 ,
                                ex exp7 , ex exp8 , ex exp9 ,
                                string base_name);
```

Método que genera un nuevo *Vector3D* introduciendo como parámetros su nombre, los 9 valores que va a contener (3x3), y el nombre de la base a la que esta asociado, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Tensor3D al objeto System.
No es necesario recoger la salida del método para que funcione
```

```
correctamente.
*/
Tensor3D Tensor3D_uno = * sys.new_Tensor3D (    “Tensor3D1” ,
                                                1 , 2 , 3 ,
                                                4 , 5 , 6 ,
                                                7 , 8 , 9 ,
                                                ”Base1” );
```

```
Point* new_Point (      string name ,
                        Point * previous_point ,
                        Vector3D * position_vector );
```

Método que genera un nuevo Point introduciendo como parámetros su nombre, su punto predecesor y su vector de posición, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Point al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Point Point_uno = * sys.new_Point ( “Point1” ,
                                    sys.get_Point ( “O” ) ,
                                    & Vector3D_uno  );
```

```
Point* new_Point (      string name ,
                        string previous_point_name ,
                        Vector3D * position_vector );
```

Método que genera un nuevo Point introduciendo como parámetros su nombre, el nombre de su punto predecesor y su vector de posición, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Point al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Point Point_uno = * sys.new_Point ( “Point1” ,
                                    “O” ,
                                    & Vector3D_uno );
```

```
Frame * new_Frame (     string name ,
                        Point * point ,
                        Base * base );
```

Método que genera un nuevo *Frame* introduciendo como parámetros su nombre, un

punto y su base, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Frame al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Frame Frame_uno = * sys.new_Frame ( Frame1” ,
                                    & Point_uno ,
                                    & Base_uno );
```

```
Frame * new_Frame (     string name ,
                        string point_name ,
                        string base_name );
```

Método que genera un nuevo *Frame* introduciendo como parámetros su nombre, el nombre de un punto y el nombre de su base, para finalmente introducirlo en el sistema.

Ejemplo:

```
/*
Con este método añadimos un nuevo Frame al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente.
*/
Frame Frame_uno = * sys.new_Frame ( “Frame1” ,
                                    ”Point1” ,
                                    ”Base1” );
```

```
Matrix* new_Matrix (    string name ,
                        Matrix mat );
```

Método que genera una nueva *Matrix* introduciendo como parámetros su nombre y un objeto de tipo *Matrix*, para finalmente introducirla en el sistema.

Ejemplo:

```
/*
Con este método añadimos una nueva Matrix al objeto System. No
es necesario recoger la salida del método para que funcione
correctamente
*/
Matrix Matrix_uno = * sys.new_Matrix (    “Matrix1” ,
                                          Matrix_uno );
```

symbol get_Time_Symbol(void);

Método que retorna el símbolo que representa al tiempo en el objeto *System*.

vector < symbol * > get_Coordinates (void);

Método que retorna un objeto de tipo *stl::vector* con las coordenadas que se han introducido en el objeto *System*.

vector < symbol * > get_Velocities (void);

Método que retorna un objeto de tipo *stl::vector* con las velocidades que se han introducido en el objeto *System*.

vector < symbol * > get_Accelerations (void);

Método que retorna un objeto de tipo *stl::vector* con las aceleraciones que se han introducido en el objeto *System*.

vector < symbol * > get_Parameters (void);

Método que retorna un objeto de tipo *stl::vector* con los parámetros que se han introducido en el objeto *System*.

vector < symbol * > get_Unknowns (void);

Método que retorna un objeto de tipo *stl::vector* con los parámetros desconocidos que se han introducido en el objeto *System*.

vector < Base * > get_Bases (void);

Método que retorna un objeto de tipo *stl::vector* con los objetos *Base* que se han introducido en el objeto *System*.

vector < Matrix * > get_Matrixes (void);

Método que retorna un objeto de tipo *stl::vector* de los objetos *Matrix* que se han introducido en el objeto *System*.

vector < Vector3D * > get_Vectors (void);

Método que retorna un objeto de tipo *stl::vector* con los objetos *Vector3D* que se han introducido en el objeto *System*.

```
vector < Tensor3D * > get_Tensors ( void );
```

Método que retorna un objeto de tipo *stl::vector* con los objetos *Tensor3D* que se han introducido en el *System*.

```
vector < Point * > get_Points ( void );
```

Método que retorna un objeto de tipo *stl::vector* con los objetos *Point* que se han introducido en el objeto *System*.

```
vector < Frame * > get_Frames ( void );
```

Método que retorna un objeto de tipo stl::vector con los objetos Frame que se han introducido en el objeto *System*.

```
Matrix Coordinates ( void );
```

Método que retorna un objeto de tipo *Matrix* con las coordenadas que se han introducido en el sistema. De esta forma nos puede resultar mas sencillo más mostrarlas por pantalla o trabajar con ellas.

```
Matrix Velocities ( void );
```

Método que retorna un objeto de tipo *Matrix* con las velocidades que se han introducido en el objeto *System*.

```
Matrix Accelerations ( void );
```

Método que retorna un objeto de tipo *Matrix* con las aceleraciones que se han introducido en el objeto *System*.

```
Matrix Joint_Unknowns ( void );
```

Método que retorna un objeto de tipo *Matrix* con los parámetros desconocidos que se han introducido en el objeto *System*.

numeric get_Time_Value (void);

Método que retorna el valor numérico que tiene asignada la variable que representa al tiempo en el objeto *System*.

void set_Time_Value (numeric time_value);

Método que asigna un valor numérico a la variable que representa al tiempo en el objeto *System*.

Symbol * get_Coordinate (string coordinate_name);

Método que retorna un puntero a la coordenada perteneciente al objeto *System*, cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
symbol coord = * sys.get_Coordinate ( “x” );

//Visualizamos el contenido por pantalla
cout << coord._get_name () << endl;
```

```
x
```

symbol * get_Velocity (string velocity_name);

Método que retorna un puntero a la velocidad perteneciente al objeto *System*, cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
symbol veloc = * sys.get_Velocity ( “dx” );

//Visualizamos el contenido por pantalla
cout << veloc._get_name () << endl;
```

```
dx
```

symbol * get_Acceleration (string acceleration_name);

Método que retorna un puntero a la aceleración perteneciente al objeto *System*, cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
symbol accel = * sys.get_Acceleration ( “ddx” );

//Visualizamos el contenido por pantalla
cout << accel._get_name () << endl;
```

```
ddx
```

`symbol * get_Parameter ( string parameter_name );`

Método que retorna un puntero al parámetro perteneciente al objeto *System*, cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
symbol param = * sys.get_Parameter ( “p” );

//Visualizamos el contenido por pantalla
cout << param._get_name () << endl;
```

```
p
```

`symbol * get_Unknown ( string unknown_parameter_name );`

Método que retorna un puntero al parámetro desconocido perteneciente al objeto *System*, cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
symbol unknown = * sys.get_Unknown ( “u” );

//Visualizamos el contenido por pantalla
cout << unknown._get_name () << endl;
```

```
u
```

`Base * get_Base ( string base_name );`

Método que retorna un puntero al objeto *Base* perteneciente al objeto *System*, cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
Base B1 = * sys.get_Base ( “Base1” );

//Visualizamos el contenido por pantalla
```

```
cout << B1._get_name () << endl;
```

```
Base1
```

`Frame * get_Frame ( string frame_name );`

Método que retorna un puntero al objeto *Frame* perteneciente al sistema cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
Frame F1 = * sys.get_Frame ( “Frame1” );

//Visualizamos el contenido por pantalla
cout << F1._get_name () << endl;
```

```
Frame1
```

`Matrix * get_Matrix ( string matrix_name );`

Método que retorna un puntero al objeto *Matrix* perteneciente al sistema cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
Matrix M1 = * sys.get_Matrix ( “Matrix1” );

//Visualizamos el contenido por pantalla
cout << M1._get_name () << endl;
```

```
Matrix1
```

`Vector3D * get_Vector3D ( string vector3D_name );`

Método que retorna un puntero al objeto *Vector3D* perteneciente al sistema cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
Vector3D V1 = *sys.get_Vector3D(“Vector3D1”);

//Visualizamos el contenido por pantalla
cout << V1._get_name() << endl;
```

```
Vector3D1
```

```
Point * get_Point ( string point_name );
```

Método que retorna un puntero al objeto *Point* perteneciente al sistema cuyo nombre coincide con la cadena que se pasa como parámetro.

Ejemplo:

```
Point P1 = * sys.get_Point ( “Point1” );

//Visualizamos el contenido por pantalla
cout << P1._get_name () << endl;
```

```
Point1
```

```
Base * Reduced_Base (    Base * BaseA ,
                         Base * BaseB );
```

Método que calcula la base reducida entre los objetos *Base* pasadas como parámetros. A continuación podemos ver la implementación de este método así como una explicación más detallada y un ejemplo

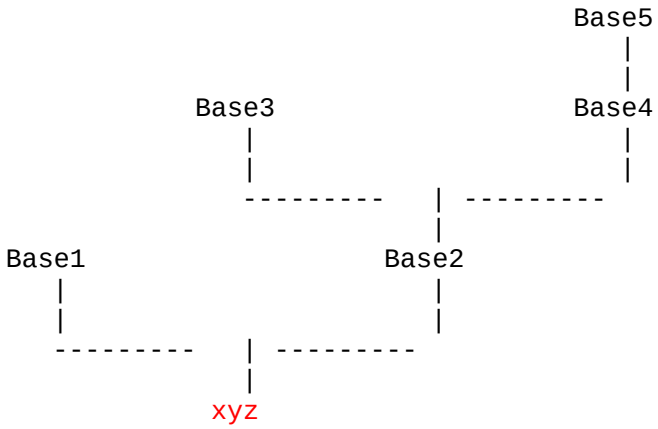
```
/*
Return the pointer of one Base result of reduce two Bases ( these common base in
the tree )
*/
Base * System::Reduced_Base ( Base * BaseA , Base * BaseB ) {
      Base * bauxA = BaseA;
      Base * bauxB = BaseB;

      vector < Base * > ramaA;
      vector < Base * > ramaB;

      ramaA.push_back ( BaseA );
      ramaB.push_back ( BaseB );

      while ( BaseA-> get_name() != "xyz" ) {
            ramaA.push_back ( get_Base ( BaseA->
                                    get_Previous_Base ()->get_name () ) );
            BaseA = get_Base ( BaseA-> get_Previous_Base ()-> get_name () );
      }
      while ( BaseB-> get_name () != "xyz" ){
            ramaB.push_back ( BaseB-> get_Previous_Base () );
            BaseB = get_Base ( BaseB-> get_Previous_Base ()-> get_name() );
      }

      int i;
      vector < Base * > ramaAaux;
      vector < Base * > ramaBaux;
```

```
      for ( i = ramaA.size () - 1 ; i >= 0 ; i-- ){
            ramaAaux.push_back ( ramaA.back () );
            ramaA.pop_back ( void );
      }

      for ( i = ramaB.size () - 1 ; i >= 0 ; i-- ){
            ramaBaux.push_back ( ramaB.back () );
            ramaB.pop_back ( void );
      }

      //Now run this vectors in pairs and stop when these pairs are'n equals
      i = 0;
      int encontrado = 0;
      while (( i < ramaAaux.size () ) and
             ( i < ramaBaux.size () ) and ( encontrado == 0 ) )
            if ( ramaAaux[i]-> get_name () == ramaBaux[i]-> get_name () )
                  i++;
            else encontrado=1;

      Base * b = ramaAaux[i-1];

      BaseA = bauxA;
      BaseB = bauxB;

      //The Bases stays in the same arm
      if ( ( b == BaseA ) || ( b == BaseB ) ) {
            if ( Bases_Position ( BaseA , BaseB ) == 0 ) {
                  if ( gravity == DOWN ){ return BaseA;}
                  else if ( gravity == UP ) {return BaseB;}
            }else{
                  if ( gravity == DOWN ) {return BaseB;}
                  else if ( gravity == UP ) {return BaseA;}
            }
      }else{
            return b;
      }
}
```

En el momento en el que creamos una base, debemos indicarle, cual es su base predecesora, con lo que se va creando un árbol de bases. En nuestro sistema siempre existirá un objeto *Base* cuyo nombre es *"xyz"*. Partiendo de esto, si al método "*Reduced_Base"* le introducimos dos bases, el método nos devolverá la primera base que tengan en común.

También, algo a tener en cuenta en el resultado que podemos obtener, es la gravedad *"gravity"*, ya que dependiendo de que su valor sea "*UP"* o "*DOWN"* cogeremos la base superior o inferior si dos bases se encuentran en la misma rama.

Podemos ver mejor esto mediante unos ejemplos gráficos.



En este árbol de *Bases* podemos ver la base xyz que es la base principal que siempre va a existir en el objeto *System*. También podemos ver otros objetos Base formando el árbol de forma que por ejemplo la base previa de *Base3* es *Base2* y la base previa de *Base5* es Base4.

Ejemplo 1:

Haciendo uso del diagrama anterior, en el siguiente ejemplo las bases de las cuales se quiere obtener la base reducida se encuentran en diferentes ramas, con lo que la gravedad, no afectara al resultado.

```
Base b1_reduced = * sys.Reduced_Base ( “Base3” , ”Base5” );
Base b2_reduced = * sys.Reduced_Base ( “Base1” , ”Base4” );
Base b3_reduced = * sys.Reduced_Base ( “Base2” , ”Base5” );

cout << b1_reduced.get_name () << endl;
```

```
Base2
```

```
cout << b2_reduced.get_name () << endl;
```

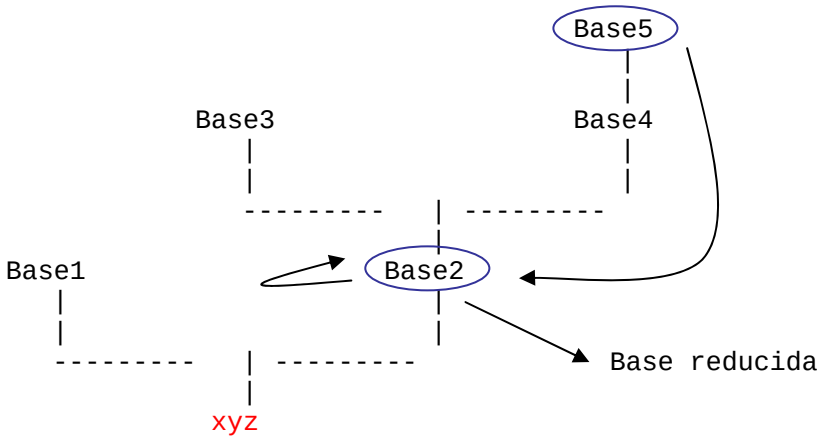
```
xyz
```

```
cout << b3_reduced.get_name () << endl;
```

```
Base2
```

Ejemplo 2:

En el siguiente ejemplo, queremos obtener la base reducida entre las bases *Base2* y *Base5*, con diferentes valores de la variable *“gravity”* que hace referencia a la gravedad. En este caso a diferencia del ejemplo anterior, al tratarse de bases que se encuentran en la misma rama, el resultado será diferente para los diferentes valores de *“gravity”*.




```
gravity = UP
Base b1_reduced = * sys.Reduced_Base ( “Base5” , ”Base2” );
gravity = DOWN
Base b2_reduced = * sys.Reduced_Base ( “Base5” , ”Base2” );

cout << b1_reduced.get_name () << endl;
```

```
Base5
```

```
cout << b2_reduced.get_name () << endl;
```

```
Base2
```

```
Base* Reduced_Base (     string BaseA_namme ,
                         string BaseB_name );
```

Método que calcula la base reducida entre las *Bases* cuyos nombres son pasados como parámetros, de igual funcionamiento que el método anteriormente visto.

```
Point * Reduced_Point (  Point * PointA ,
                         Point * PointB );
```

Método que calcula el punto reducido entre los objetos *Point* pasados como parámetros. La técnica usada para el cálculo de *Reduced_Point* es muy semejante a la usada para la localización de *Reduced_Base*.

```
Point * Reduced_Point (  string PointA_name ,
                         string PointB_name );
```

Método que calcula el punto reducido entre los objetos *Point* cuyos nombres son pasados como parámetros.

```
Matrix Rotation_Matrix (        Base * BaseA ,
                                Base * BaseB );
```

Método que calcula la matriz de rotación entre los objetos *Base* pasadas como parámetros. La *BaseA* será el objeto *Base* destino y la *BaseB* será el objeto *Base origen.*

Para que las operaciones de este método se realicen de una forma más cómoda, se utiliza el método *Rotation_Matrix_Aux*, el cual es el que verdaderamente realiza las operaciones. El método *Rotation_Matrix* es el que decide partiendo de como están situadas las bases en el árbol de bases, que operaciones deben hacerse.

A continuación podemos observar el método *Rotation_Matrix_Aux* y como va realizando las multiplicaciones de la rotation_matrix de las bases.

```
/*
Auxiliar method for calculate Rotation_Matrix ( BaseA is Reduced_Base )
*/
Matrix System::Rotation_Matrix_Aux ( Base * BaseA , Base * BaseB ) {
      Matrix rt = BaseB-> rotation_matrix () ;
      Matrix bAux;
      while ( BaseB-> get_Previous_Base ()-> get_name () !=
              BaseA-> get_name () ) {
            BaseB = BaseB-> get_Previous_Base () ;
            bAux = BaseB-> rotation_matrix () ;
            rt = rt * bAux;
      }
      return rt;
}
```

Ahora podemos observar el método *Rotation_Matrix* el cual una vez colocada la gravedad a nivel bajo (tenemos que recordar que para realizar este cálculo no debe afectar la gravedad), se decide, según sea la posición de la base en el árbol la operación a realizar.

```
/*
Return the Rotation Matrix between two Bases ( BaseA --> BaseB )
*/
Matrix System::Rotation_Matrix ( Base * BaseA , Base * BaseB ) {
      if ( BaseA != BaseB ) {
            Base * reducedbase = NULL;

            //The gravity no affect here

            if ( gravity == UP ) {
                  gravity = DOWN;
                  reducedbase = Reduced_Base ( BaseA , BaseB );
                  gravity = UP;
            }else
                  reducedbase = Reduced_Base ( BaseA , BaseB );


            if ( reducedbase == BaseA )
                  return Rotation_Matrix_Aux ( BaseA , BaseB );
            else
                  if ( reducedbase == BaseB )
                        return Rotation_Matrix_Aux ( BaseB , BaseA ).transpose () ;
```

```
                else
                        return Rotation_Matrix_Aux ( reducedbase , BaseB ) *
                        Rotation_Matrix_Aux ( reducedbase , BaseA ).transpose () ;
        }else{
                Matrix Aux ( 3 , 3 , lst ( 1 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 1 ) );
                return Aux;
        }
}
```

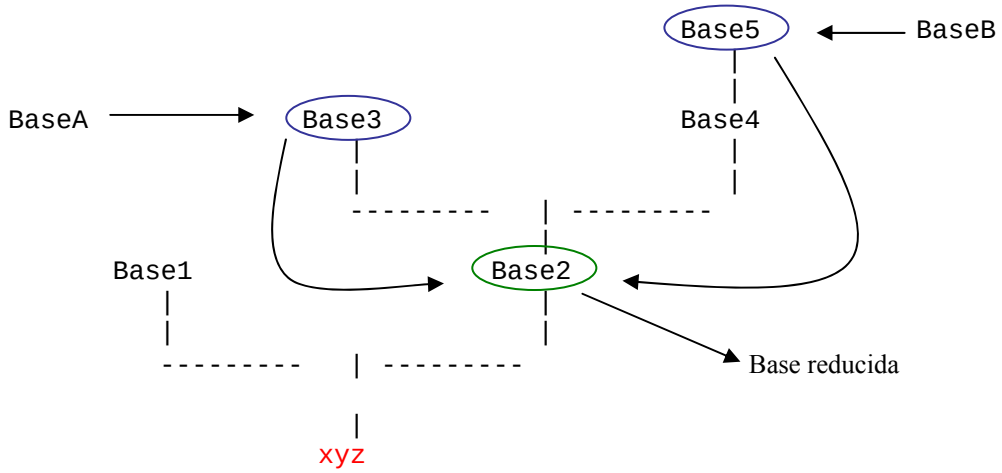
En caso de que queramos calcular la matriz de rotación de dos bases iguales, el resultado seria directamente la matriz identidad de grado 3.

Ejemplo:

Partiendo de las siguientes Bases y coordenadas.

```
symbol x = * sys.new_Coordinate ( "x" , 0 , 0 , 0 );
symbol theta = * sys.new_Coordinate ( "theta" , 3.1416/2.0 , 0 , 0);

Base B1 = *sys.new_Base ( "Base1" , "xyz" , 0 , 1 , 0 , -theta );
Base B2 = *sys.new_Base ( "Base2" , "xyz" , 2 , 1 , 0 , -x );
Base B3 = *sys.new_Base ( "Base3" , "Base2" , 0 , 1 , 0 , -theta );
Base B4 = *sys.new_Base ( "Base4" , "Base2" , 3 , 1 , 1 , -x );
Base B5 = *sys.new_Base ( "Base5" , "Base4" , 0 , 1 , 0 , -x );
```

Se crea el siguiente árbol de objetos Base.



Obtendremos los siguientes resultados

```
cout << sys.Rotation_Matrix ( & B3 , & B5 ) << endl;
```

```
[3*sin(-theta)*cos(-x)-3*sin(-theta)*cos(-x)*cos(-11*x)+3*cos(-
theta)*sin(-x)-8*cos(-x)*cos(-theta)*cos(-11*x)-sin(-11*x)*cos(-
theta)*sin(-x)+9*cos(-x)*cos(-theta)+sin(-theta)*sin(-11*x)*cos(-
x)+sin(-theta)*sin(-x)-3*cos(-theta)*cos(-11*x)*sin(-x),-sin(-
11*x)*cos(-x)-cos(-11*x)*sin(-x)+3*cos(-x)+sin(-x)-3*cos(-x)*cos(-
11*x)+3*sin(-11*x)*sin(-x),-9*sin(-theta)*cos(-x)+8*sin(-
theta)*cos(-x)*cos(-11*x)+cos(-theta)*sin(-x)-3*cos(-x)*cos(-
```

```
theta)*cos(-11*x)+sin(-11*x)*cos(-x)*cos(-theta)+sin(-theta)*sin(-
11*x)*sin(-x)+3*cos(-x)*cos(-theta)-3*sin(-theta)*sin(-x)+3*sin(-
theta)*cos(-11*x)*sin(-x);
-3*sin(-theta)*sin(-11*x)+sin(-theta)-3*cos(-theta)*cos(-
11*x)+3*cos(-theta)+sin(-11*x)*cos(-theta)-sin(-theta)*cos(-
11*x),1,-sin(-theta)*sin(-11*x)-3*sin(-theta)-cos(-theta)*cos(-
11*x)+cos(-theta)-3*sin(-11*x)*cos(-theta)+3*sin(-theta)*cos(-11*x);
sin(-theta)*cos(-x)-9*cos(-theta)*sin(-x)-3*cos(-x)*cos(-
theta)*cos(-11*x)-sin(-11*x)*cos(-x)*cos(-theta)-sin(-theta)*sin(-
11*x)*sin(-x)+3*cos(-x)*cos(-theta)-3*sin(-theta)*sin(-x)+8*cos(-
theta)*cos(-11*x)*sin(-x)+3*sin(-theta)*cos(-11*x)*sin(-x),3*sin(-
11*x)*cos(-x)+3*cos(-11*x)*sin(-x)+cos(-x)-3*sin(-x)-cos(-x)*cos(-
11*x)+sin(-11*x)*sin(-x),-3*sin(-theta)*cos(-x)+3*sin(-theta)*cos(-
x)*cos(-11*x)-3*cos(-theta)*sin(-x)-sin(-11*x)*cos(-theta)*sin(-
x)+cos(-x)*cos(-theta)+sin(-theta)*sin(-11*x)*cos(-x)+9*sin(-
theta)*sin(-x)+3*cos(-theta)*cos(-11*x)*sin(-x)-8*sin(-theta)*cos(-
11*x)*sin(-x)]
```

Ahora podemos observar una de las razones de la atomización, y es la de que el resultado anterior es muy grande y contiene muchas operaciones repetitivas, sin embargo con el siguiente resultado, es mucho más eficaz trabajar con la expresión.

```
[atom19,atom11,atom20;
atom21,atom14,atom22;
atom23,atom17,atom24]
```

```
Vector3D Position_Vector (     Point * PointA ,
                               Point * PointB );
```

Método que calcula el vector de posición entre dos punteros a objetos *Point pasados* como parámetros. Este método funciona de forma similar al método *Rotation_Matrix*.

```
Vector3D Position_Vector(      string PointA_name ,
                               string PointB_name );
```

Método que calcula el vector de posición entre dos objetos *Point* cuyos nombres son pasados como parámetros.

```
Vector3D Angular_Velocity (    Base * BaseA ,
                               Base * BaseB );
```

Método que calcula la velocidad angular entre dos punteros a objetos *Base* pasadas como parámetros. Este método funciona de forma similar al método *Rotation_Matrix*.

```
Vector3D Angular_Velocity (    string base_frame_nameA ,
                               string base_frame_nameB );
```

Método que calcula la velocidad angular entre dos objetos de tipo *Base* o *Frame* cuyo nombre se pasa como parámetro. Si por su nombre, se determina que el objeto es de tipo

Frame, se usara el objeto *Base* que posee este para realizar los cálculos.

```
Tensor3D Angular_Velocity_Tensor (   Base * BaseA ,
                                     Base * BaseB );
```

Método que calcula la velocidad angular del objeto *Tensor3D* entre dos punteros a objetos *Base* pasados como parámetros.

```
Vector3D Angular_Acceleration (      string base_frame_nameA ,
                                     string base_frame_nameB );
```

Método que calcula la aceleración angular entre dos objetos de tipo *Base* o *Frame* cuyo nombre se pasa como parámetro. Si por su nombre, se determina que el objeto es de tipo *Frame*, se usara el objeto *Base* que posee este para realizar los cálculos.

```
void remove_Matrix ( string matrix_name );
```

Método que elimina del objeto *System*, el objeto de tipo *Matrix* cuyo nombre coincide con la cadena de caracteres que se pasa como parámetro. Esto será posible si ningún otro objeto del sistema la referencia.

```
void remove_Vector3D ( string vector3D_name );
```

Método que elimina del objeto *System*, el objeto de tipo Vector3D cuyo nombre coincide con la cadena de caracteres que se pasa como parámetro. Esto será posible si ningún otro objeto del sistema la referencia.

```
void remove_Point ( string point_name );
```

Método que elimina del objeto *System,* el objeto de tipo *Point* cuyo nombre coincide con la cadena de caracteres que se pasa como parámetro. Esto será posible si ningún otro objeto del sistema la referencia.

```
void remove_Base ( string base_name );
```

Método que elimina del objeto *System*, el objeto de tipo *Base* cuyo nombre coincide con la cadena de caracteres que se pasa como parámetro. Esto será posible si ningún otro objeto del sistema la referencia.

```
bool is_dt_zero( ex expression );
```

Método que nos dice si la derivada de una expression de tipo *GiNaC::ex* con respecto a todas las coordenadas, velocidades y aceleraciones que contiene el objeto *System* es

igual a 0 o no.

```
ex dt( ex expression );
```

Método que retorna la derivada de la expresión que se introduce como parámetro. Esta derivada es el resultado de derivar la expresión con respecto a las coordenadas, velocidades y aceleraciones que contiene el objeto *System* en ese momento.

```
Matrix Dt ( Matrix MatrixA );
```

Método que retorna la derivada del objeto *Matrix* que se introduce como parámetro.

```
Vector3D Dt (      Vector3D Vector3DA ,
                   Base * base );
```

Método que calcula la derivada de un objeto *Vector3D* con respecto a un objeto *Base*.

```
Vector3D Dt (      Vector3D Vector3DA ,
                   Frame * frame );
```

Método que calcula la derivada de un objeto *Vector3D* con respecto a un objeto *Frame*.

```
Vector3D Dt (      Vector3D Vector3DA ,
                   string base_frame_name );
```

Método que calcula la derivada de un *Vector3D* con respecto a *Frame* o una *Base* cuyo nombre es pasado como parámetro. El método realizará una búsqueda para averiguar si el nombre que se ha pasado como parámetro pertenece a un objeto de tipo *Base* o de tipo *Frame*.

```
Matrix jacobina (  Matrix MatrixA  ,
                   Matrix MatrixB );
```

Método que calcula la matriz jacobina de dos objetos Matrix pasados como parámetros.

Ejemplo:

```
symbol x ( “x” );

Matrix A (  1 , 3 ,
```

```
            lst ( x , 0 , x ) );

Matrix B (  3 , 1 ,
            lst ( 1 ,
                  x ,
                  1 ) );

//Visualización del resultado
cout << sys.jacobian (A , B) << endl;
```

```
[0,1,0;
0,0,0;
0,1,0]
```

Matrix jacobian (ex expression, Matrix MatrixA);

Método que calcula la matriz jacobina entre un objeto de *tipo GiINAC::ex* y uno de tipo *Matrix* pasados como parámetros.

Matrix jacobian (Matrix MatrixA , symbol symbolA);

Método que calcula la matriz jacobina entre un objeto de *tipo GiINAC::symbol* y uno de tipo *Matrix* pasados como parámetros.

ex jacobian (ex expression , symbol symbolA);

Método que calcula la matriz jacobina entre un objeto de tipo *GiINAC::ex* y uno de tipo *GiNaC::symbol* pasados como parámetros.

ex diff (ex expression , symbol symbolA);

Método que calcula la derivada de una expresión de tipo *GiINAC::ex* con respecto a un objeto de tipo *GiNaC::symbol*. Esta forma de realizar la derivada nos permite manejar la atomización de las expresiones derivadas.

Ejemplo:

```
symbol x ( “x” );

ex expresión = 2 x + x ^ 2;

//Visualización del resultado
cout << sys.diff ( expresión , x ) << endl;
```

```
2+2x
```

Matrix diff (Matrix MatrixA , symbol symbolA);

Método que calcula la derivada de todos los elementos de un objeto de tipo *Matrix* con respecto a un objeto de tipo *GiNaC::symbol*..

Ejemplo:

```
symbol x ( “x” );

ex expresión1 = 2 x + x ^ 2;
ex expresión2 = 5 x - 5;


Matrix A ( 2 , 2 , &expression1 , 9 , &expression2 , 3 );

//Visualización del resultado
cout << sys.diff ( A ,  x ) << endl;
```

```
  [2+2x,0;
  5,0]
```

`Vector3D diff ( Vector3D Vector3DA , symbol symbolA );`

Método que calcula la derivada de todos los elementod de un objeto de tipo *Vector3D* con respecto a un objeto de tipo *GiNaC::symbol*..

`Tensor3D diff ( Tensor3D Tensor3DA , symbol symbolA );`

Método que calcula la derivada de todos los elementod de un objeto de tipo *Vector3D* con respecto a un objeto de tipo *GiNaC::symbol*..

`ex numeric_evaluate ( ex expression );`

Evalúa numéricamente una expresión.

A continuación podemos observar el método *numeric_evaluate*, y una explicación mas detallada de como realiza su función.

```
/*
This method evaluate one expression , substituting the variables from her value
*/
ex System::numeric_evaluate ( ex expression ) {
      try{
            expression = unatomize_ex ( expression );
            for ( int i = 0 ; i < coordinates.size () ; i++ ) {
                  expression = evalf ( expression.subs
                              ( * coordinates[i] == coordinates_numeric[i] ) );
                  expression = evalf ( expression.subs
                              ( * velocities[i] == velocities_numeric[i] ) );
                  expression = evalf ( expression.subs
                              ( * accelerations[i] == accelerations_numeric[i] ) );
```

```
            }

            for ( int i = 0 ; i < parameters.size () ; i++ )
                    expression = evalf ( expression.subs
                                ( * parameters[i] == parameters_numeric[i] ) );

            for ( int i = 0 ; i < unknowns.size () ; i++ )
                    expression = evalf ( expression.subs
                                ( * unknowns[i] == unknowns_numeric[i] ) );
      }catch ( exception & p ) {
            outError ( "ERR- in values evaluation" );
      }
      return expression;

}
```

Este método se utiliza para evaluar numéricamente una expresión y para ello, hay que sustituir las diferentes variables por su valor numérico. Primero substituimos primero las posibles coordenadas, velocidades y aceleraciones por sus correspondientes valores, luego los parámetros por sus valores y finalmente los parámetros desconocidos por sus valores. Como resultado, obtendremos una expresión cuyo valor será el resultado de sustituir sus variables por sus valores.

Ejemplo:

```
symbol x = * sys.new_Coordinate ( "x" , 1 );
symbol y = * sys.new_Coordinate ( "y" , 2 );
symbol z = * sys.new_Coordinate ( "z" , 3 );

ex expresion = ( 2 * y + z ) * x;

//Visualización del resultado
cout << sys.numeric_evaluate(expresion) << endl;
```

```
  7.0
```

`Matrix evaluate_Matrix ( Matrix MatrixA );`

Evalúa numéricamente una *Matrix*, *Vector3D* o *Tensor3D* y retorna el resultado en una *Matrix*. La implementación de este método consiste en llamadas sucesivas al método *numeric_evaluation*.

Ejemplo:

```
symbol x = * sys.new_Coordinate ( "x" , 1 );
symbol y = * sys.new_Coordinate ( "y" , 2 );
symbol z = * sys.new_Coordinate ( "z" , 3 );

ex expresion1 = ( 2 * y + z ) * x;
ex expresion2 = ( y + z ) * 2x;
ex expresion3 = 2 * y ;
```

```
  Matrix A ( 2 , 2 );

  A =       expresion1, 0 ,
            expresion2 , expresion3;

//Visualización del resultado
cout << sys.numeric_evaluate(A) << endl;
```

```
[7.0,0;
10,4]
```

double ** evaluate_Array (Matrix MatrixA);

Evalúa numéricamente una *Matrix*, *Vector3D* o *Tensor3D* y retorna el resultado en un array de doubles. Este método es útil a la hora de exportar los resultados y que puedan ser leídos por otras librerías que no soportan el objeto *Matrix* que se ha desarrollado en esta librería

string print_Array (long rows ,long cols ,
double ** array);

Muestra por pantalla, correctamente formateado un array de doubles, indicando para ello el numero de filas y de columnas del mismo.

Ejemplo:

```
  symbol x = * sys.new_Coordinate ( "x" , 1 );
  symbol y = * sys.new_Coordinate ( "y" , 2 );
  symbol z = * sys.new_Coordinate ( "z" , 3 );

  ex expresion1 = ( 2 * y + z ) * x;
  ex expresion2 = ( y + z ) * 2x;
  ex expresion3 = 2 * y ;

  Matrix A ( 2 , 2 );

  A =       expresion1, 0 ,
            expresion2 , expresion3;

//Visualización del resultado
cout << sys.print_Array( 2 , 2 , sys.evaluate_Array(A)) << endl;
```

```
[7.0,0;
10,4]
```

- *Métodos de exportación*

void export_var_def_C (void);

Método que genera un fichero en lenguaje C llamado `var_def.c` en el que se declaran los parámetros del objeto *System* como variables de tipo double, y la variable correspondiente al tiempo.

Ejemplo:

```
//----------var_def.c lib3D_MEC expoted-----------

//----------time-----------

double t;
//----------parameters-----------

double d;
double l;
double r;
double mblock;
double mpendulum;
double g;
double I1;
double I2;
double I3;
```

void export_var_def_H (void);

Método que genera un fichero en lenguaje C llamado `var_def.h` en el que se declaran los parámetros del objeto System como variables de tipo *double*, y la variable correspondiente al tiempo, de forma que puedan ser leídos por otros ficheros diferentes. Esto lo conseguimos escribiendo la palabra reservada *"extern"* delante de las variables que deseamos que se vean desde el exterior.

void export_var_init_C (void);

Método que genera un fichero en lenguaje C llamado `var_init.c` en el que se asignan valores numéricos a los parámetros que están almacenados en el objeto System, y que se han declarado mediante los métodos *export_var_def_C* y *export_var_def_H*.

void export_atom_def_C (lst atom_list);

Método que genera un fichero en lenguaje C llamado `atom_def.c`, en el que se declaran los átomos creados con sus valores correspondientes y luego mediante la palabra reservada *"export"*, para que puedan ser utilizados desde otros ficheros.

void export_gen_coord_vect_def_H (void);

Método que genera un fichero en lenguaje C llamado `gen_coord_vect_def.h` en el que se declaran las coordenadas del objeto System como variables de tipo *double*, y la variable correspondiente al tiempo, colocando delante de estas declaraciones la palabra

reservada *"extern"* para indicar que estas variables, van a poder ser accedidas desde el exterior.

`void export_gen_coord_vect_init_C ( void );`

Método que genera un fichero en lenguaje C llamado `gen_coord_vect_init.c` en el que se declaran las coordenadas del objeto System como variables de tipo *double*, y la variable correspondiente al tiempo, colocando delante de estas declaraciones la palabra reservada *"extern"* para indicar que estas variables, van a poder ser accedidas desde el exterior.

`void export_gen_vel_vect_init_H ( void );`

Método que genera un fichero en lenguaje C llamado `gen_coord_vect_def.h,` en el que se declaran las coordenadas del objeto System como variables de tipo *double*, y la variable correspondiente al tiempo. También podemos encontrar una constante que nos indica el número de velocidades que contiene el objeto System en ese instante.

`void export_gen_vel_vect_init_C ( void );`

Método que genera un fichero en lenguaje C llamado `gen_vel_vect_init.c` , en el que se definen constantes de tipo double cuyo nombre corresponde al nombre de las velocidades que se han introducido en el objeto *System* y sus valores son sus valores numéricos correspondientes. También se genera una función llamada `void gen_vel_vect_init(void)` que mediante un bucle, introduce en un gsl::vector todas las velocidades que contiene el objeto System.

`void export_gen_accel_vect_init_H ( void );`

Método que genera un fichero en lenguaje C llamado `gen_accel_vect_def.h,` en el que se declaran como variables externas el símbolo que representa al tiempo objeto System y todas las aceleraciones introducidas en este sistema. Todas estas variables son del tipo double. También podemos encontrar una constante que nos indica el número de aceleraciones que contiene el objeto System en ese instante.

`void export_gen_accel_vect_init_C ( void );`

Método que genera un fichero en lenguaje C llamado `gen_accel_vect_init.c` en el que se definen constantes con el nombre de las aceleraciones que contiene el objeto *System* y cuyo valor es el asignado en la introducción de las mismas . También se genera una función llamada `void gen_accel_vect_init(void),` que mediante un bucle , introduce en un gsl::vector todas las aceleraciones que contiene el objeto System.

`void export_unknowns_vect_init_H ( void );`

Método que genera un fichero en lenguaje C llamado `unknowns_vect_def.h` en el que se declaran como variables externas el símbolo que representa al tiempo objeto *System* y todas los parámetros desconocidos introducidas en este sistema. Todas estas variables son

del tipo double. También podemos encontrar una constante que nos indica el número de parámetros desconocidos que contiene el objeto *System* en ese instante.

```
void export_unknowns_vect_init_C ( void );
```

Método que genera un fichero en lenguaje C llamado `unknowns_vect_init.c`, en el que se definen constantes con el nombre de los parámetros desconocidos que contiene el objeto *System* y cuyo valor es el asignado en la introducción de las mismas . También se genera una función llamada `void void unknowns_vect_init(void),` que mediante un bucle , introduce en un gsl::vector todas los parámetros desconocidos que contiene el objeto *System*.

```
void export_Column_Matrix_C (  string function_name ,
                               string vector_name ,
                               Matrix Col_matrix ,
                               lst matrix_atom_list );
```

Método que genera un fichero en lenguaje C llamado como el primer parámetro que se pasa al método terminado en `.c` en el que se escriben todos los átomos de la variable *GiNac::lst* llamada *matrix_atom_list* que no están en la variable de tipo *Matrix* llamada *Col_matrix*. La variable de tipo Matrix, debe ser de dimensiones *nx1*.

```
void export_Matrix_C (   string function_name ,
                         string marix_name ,
                         Matrix mat ,
                         lst matrix_atom_list );
```

Método que genera un fichero en lenguaje C llamado como el primer parámetro que se pasa la método terminado en `.c` en el qie se escriben todos los átomos de la variable *GiNac::lst* llamada *matrix_atom_list* que no están en la variable de tipo *Matrix* llamada *Col_matrix*.

```
void export_write_state_file_header_C ( void );
```

Método que genera un fichero en lenguaje C llamado `write_state_file_header.c` en el que se crea una función llamada `void write_state_file_header(FILE * state_file)` en el la que se escriben separados mediante tabulaciones todas las coordenadas, velocidades, aceleraciones, parámetros y parámetros desconocidos que contiene el objeto *System*

```
void export_write_state_file_C ( void );
```

Método que genera un fichero en lenguaje C llamado `write_state_file.c` en el que se crea una función llamada `void write_state_file(FILE * state_file)` en el que se defiunen como constantes numéricas todos las coordenadas, velocidades, aceleraciones, parámetros y parámetros desconocidos que contiene el objeto System. Dicho de otra forma, se escriben los valores numéricos de las variables escritas en el método `“export_write_state_file_header_C”.`

- *Destructor*

~System (void);

Destructor de la clase *System*.

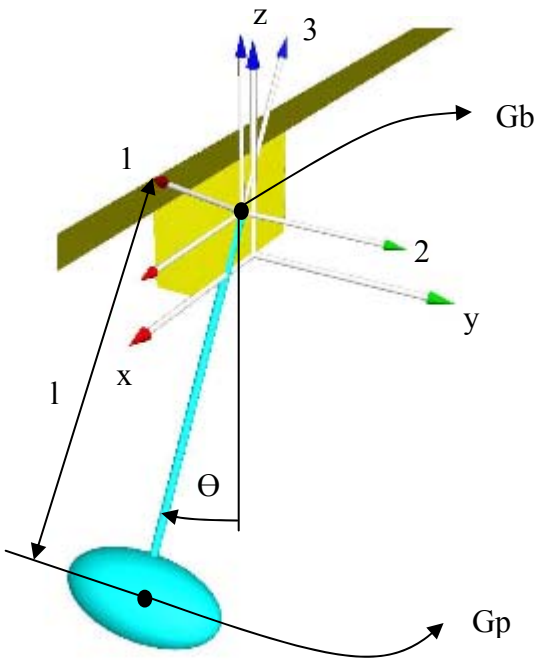
- *Funciones*

Una función de la clase System de gran utilidad es la función *Search_Object*, que busca una *Matrix*, *Point*, *Frame*, etc..., cual quiera que sea el tipo de objeto, en el sistema. Para ello se le tiene que pasar el vector en el que debe buscar, pero lo interesante de esta función es que este puede ser de cualquier tipo. Para ello se ha utilizado una plantilla que permite utilizar cualquier tipo de objeto siempre que los objetos que se quieran usar tengan la misma interfaz. Finalmente si se encuentra el objeto representado por el nombre que se pasa como parámetro, se retorna, y si no se retorna *NULL*

```
/*
Function that return the search symbol in the System in one vector
*/
template < class T >
T Search_Object ( vector < T > vect ,string name ) {
          int I = 0 ;
          int encontrado = 0;
          while ( ( i < vect.size () ) and ( encontrado == 0 ) )
              if ( name != vect[i]-> get_name () )
                    i++;
              else encontrado = 1;
          if ( encontrado == 1 )
              return vect[i];
          else
              return NULL;
}
```

4 - Ejemplo amplio

A continuación podemos ver un programa implementado con la librería lib3d_mec_GiNaC para ver de una forma más global, el tipo de problemas que se pueden resolver con la misma. Con este ejemplo fue presentada la librería en el congreso MULTIBODY DYNAMICS 2007, CCOMAS Thematic Conference en Milan.




```
#include <fstream>
#include "lib_3d_mec_ginac/lib_3d_mec_ginac.h"

using namespace std;

void printError(char* arg){
      printf("%s\n", arg);
}


int main(int argc, char *argv[]) {

if (argc!=3){
            printf(     "Error: The program %s takes two parameters: Gravity
                        (UP/DOWN) and Atomize (YES/NO)\n",argv[0]);
            exit(1);
      }

if (0==strcmp(argv[1],"DOWN")){
      gravity=DOWN;
      cout << "DOWN" << endl;
}
else{
```

```
      if (0==strcmp(argv[1],"UP")){
            gravity=UP;
            cout << "UP" << endl;
      }
      else
      {
            printf(     "Error: The program %s takes two parameters: Gravity
                        (UP/DOWN) and Atomize (YES/NO)\n",argv[0]);
            exit(1);
      }
}

if (0==strcmp(argv[2],"YES")){
            atomization=YES;
            cout << "YES" << endl;
}
else{
      if (0==strcmp(argv[2],"NO")){
            atomization=NO;
            cout << "NO" << endl;
      }
      else
      {
            printf(     "Error: The program %s takes two parameters: Gravity
                        (UP/DOWN) and Atomize (YES/NO)\n",argv[0]);
            exit(1);
      }
}

      double integration_time=strtod(argv[1], NULL);
      double delta_t= strtod(argv[2], NULL);
      long int k,steps;

      printf("integration_time %g delta_t %g\n", integration_time, delta_t);

//System definition

      System sys(&printError);
      cout << atomization << endl;
      cout << gravity << endl;

//Coordinate definition

      symbol x = *sys.new_Coordinate("x",0,0,0);
      symbol theta = *sys.new_Coordinate("theta", 3.1416/2.0,0,0);

//Kinematical parameter definition

      symbol d = *sys.new_Parameter("d",1);
      symbol l = *sys.new_Parameter("l",1);
      symbol r = *sys.new_Parameter("r",1);

//Define Base

      sys.new_Base("BPendulum","xyz",0,1,0, -theta);

//Define Vectors

      Vector3D OGb = *sys.new_Vector3D(   "OGb",
                                          x,
                                          0,
```

```
                                        d/2,
                                        "xyz");

      Vector3D OA = *sys.new_Vector3D(  "OA",
                                        x,
                                        0,
                                        0,
                                        "xyz");

      Vector3D GbGp=*sys.new_Vector3D(  "GbGp",
                                        0,
                                        0,
                                        -l,
                                        "BPendulum");

//Define Points

      Point * Gb = sys.new_Point("Gb","O",&OGb);
      Point * A = sys.new_Point("A","O",&OA);
      Point * Gp = sys.new_Point("Gp","Gb",&GbGp);

//Define Frames

      Frame * Block = sys.new_Frame("Block","Gb","xyz");
      Frame * Pendulum=sys.new_Frame("Pendulum","Gp","BPendulum");


//Dynamical Parameter Definition

      symbol  mblock = *sys.new_Parameter("mblock",1);
      symbol  mpendulum = *sys.new_Parameter("mpendulum",1);
      symbol  g = *sys.new_Parameter("g",10);
      symbol  I1 = *sys.new_Parameter("I1",1);
      symbol  I2 = *sys.new_Parameter("I2",1);
      symbol  I3 = *sys.new_Parameter("I3",1);
      Tensor3D  IpendulumGp = *sys.new_Tensor3D(     "IpendulumGp",
                                                     (ex)I1,(ex)0,(ex)0,
                                                     (ex)0,(ex)I2,(ex)0,
                                                     (ex)0,(ex)0,(ex)I3,
                                                     "BPendulum");
//Joint Unknown Definition

      symbol  Fgby = *sys.new_Joint_Unknown("Fgby");
      symbol  Fgbz = *sys.new_Joint_Unknown("Fgbz");
      symbol  Mgbx = *sys.new_Joint_Unknown("Mgbx");
      symbol  Mgby = *sys.new_Joint_Unknown("Mgby");
      symbol  Mgbz = *sys.new_Joint_Unknown("Mgbz");

      symbol  Fbpx = *sys.new_Joint_Unknown("Fbpx");
      symbol  Fbpy = *sys.new_Joint_Unknown("Fbpy");
      symbol  Fbpz = *sys.new_Joint_Unknown("Fbpz");
      symbol  Mbpx = *sys.new_Joint_Unknown("Mbpx");
      symbol  Mbpz = *sys.new_Joint_Unknown("Mbpz");

//Joint Torsor Definition

      Vector3D F_GB = *sys.new_Vector3D(  "F_GB",
                                        0,
                                        Fgby,
                                        Fgbz,
                                        "xyz");
```

```
      Vector3D M_GB_A = *sys.new_Vector3D(      "M_GB_A",
                                                Mgbx,
                                                Mgby,
                                                Mgbz,
                                                "xyz");

      Vector3D F_BP = *sys.new_Vector3D(  "F_BP",
                                          Fbpx,
                                          Fbpy,
                                          Fbpz,
                                          "xyz");

      Vector3D M_BP_Gb = *sys.new_Vector3D(     "M_BP_Gb",
                                                Mbpx,
                                                0,
                                                Mbpz,
                                                "xyz");

//Constitutive Forces and moments Definition

      Vector3D Block_gravity = *sys.new_Vector3D(     "mblock*g",
                                                      0,
                                                      0,
                                                      -mblock * g,
                                                      "xyz");

      Vector3D Pendulum_gravity = *sys.new_Vector3D(  "mpendulum*g",
                                                      0,
                                                      0,
                                                      -mpendulum * g,
                                                      "xyz");

//Define Velocity and Acceleration of Point Gb

      Vector3D VabsGb = sys.Dt(OGb,"abs");
      Vector3D AabsGb = sys.Dt(VabsGb,"abs");

//Define Acceleration of Point Gp

      Vector3D OGp = sys.Position_Vector("O","Gp");
      Vector3D VabsGp = sys.Dt(OGp,"abs");
      Vector3D AabsGp = sys.Dt(VabsGp,"abs");

//Angular moment and Inertia Moment of pendulum

      Vector3D OmegaPendulum = sys.Angular_Velocity("xyz","BPendulum");
      Vector3D H_Gp = IpendulumGp * OmegaPendulum;
      Vector3D Iner_Moment_Pendulum_Gp = -sys.Dt(H_Gp,"xyz");

//Dynamic Equations

      Matrix equation_1_3 = -mblock * AabsGb + Block_gravity + F_GB - F_BP;
      Matrix equation_2_3 = ((sys.Position_Vector("Gb","A") ^ F_GB) +  M_GB_A)-
                                                                      M_BP_Gb;
      Matrix equation_3_3 = -mpendulum * AabsGp + Pendulum_gravity + F_BP;
      Matrix equation_4_3 = Iner_Moment_Pendulum_Gp +
                            ((sys.Position_Vector("Gp","Gb") ^ F_BP) +_BP_Gb);

      Matrix Dynamic_Equations(     4,1,
                                    &equation_1_3,
```

```
                                    &equation_2_3,
                                    &equation_3_3,
                                    &equation_4_3);

      Matrix q = sys.Coordinates();
      Matrix dq = sys.Velocities();
      Matrix ddq = sys.Accelerations();
      Matrix epsilon = sys.Joint_Unknowns();

      Matrix M=sys.jacobian(Dynamic_Equations.transpose(),ddq);
      Matrix V=sys.jacobian(Dynamic_Equations.transpose(),epsilon);
      Matrix Q=-(Dynamic_Equations-(M*ddq+V*epsilon));
      Matrix MV(1,2,&M,&V);

//Output

      cout << q << endl;
      cout << dq << endl;
      cout << ddq << endl;
      cout << epsilon << endl;
      cout << Dynamic_Equations << endl;
      cout << MV << endl;
      cout << Q << endl;
      for (int i=0; (i < atoms.size())&& (i >0); ++i) {
            cout << i << " " << atoms.size()-1 << endl;
            cout << atoms[i] << " = " << atom_expressions[i] << endl;
      }

//Export C code for Direct Simulation

      sys.export_var_def_C();//->"var_def.c"
      sys.export_var_def_H();//->"var_def.c"
      sys.export_var_init_C();//->"var_init.c"
      sys.export_gen_coord_vect_def_H();//->"gen_coord_vect_def.c"
      sys.export_gen_coord_vect_init_C();//->"gen_coord_vect_init.c"
      sys.export_gen_vel_vect_init_H();//->"gen_vel_vect_init.c"
      sys.export_gen_vel_vect_init_C();//->"gen_vel_vect_init.c"
      sys.export_gen_accel_vect_init_H();//->"gen_accel_vect_init.c"
      sys.export_gen_accel_vect_init_C();//->"gen_accel_vect_init.c"
      sys.export_unknowns_vect_init_H();//->"unknowns_vect_init.c"
      sys.export_unknowns_vect_init_C();//->"unknowns_vect_init.c"
      lst used_atom_list, new_atom_list_Q, new_atom_list_MV;
      new_atom_list_Q=atom_list(Q, used_atom_list);
      new_atom_list_MV=atom_list(MV, used_atom_list);
      sys.export_atom_def_C(used_atom_list);//->"atoms_def.c" and "atoms_def.h"

//Repeated evaluation of atoms avoided if calling sequence is used.

      sys.export_Column_Matrix_C("Q", "Phi", Q, new_atom_list_Q);//->"Q.c" and
"Q.h" with Q(Phi);
      sys.export_Matrix_C("MV", "J", MV, new_atom_list_MV);//->"MV.c" and "MV.h"
with MV(J);


      sys.export_write_state_file_header_C();
      sys.export_write_state_file_C();

//Compile exported code

      system("gcc -DHAVE_INLINE=1 -DGSL_RANGE_CHECK_OFF=1 -o main  main.c
var_def.c var_init.c gen_coord_vect_def.c gen_coord_vect_init.c
```

```
gen_vel_vect_def.c gen_vel_vect_init.c gen_accel_vect_def.c
gen_accel_vect_init.c unknowns_vect_def.c unknowns_vect_init.c Q.c MV.c
write_state_file.c write_state_file_header.c qr_nm.c newton_raphson_qr.c
printing.c  -lgsl -lgslcblas -lm -lcln");

//Execute compiled program

      system("./main 10 0.001");

//Show graphics of simulation

      system("gnuplot plot.gnuplot");

      return 0;
}
```

Salidas

En el siguiente ejemplo hemos ejecutado el programa con valores *DOWN* para *"gravity"* y *NO* para *"atomization"*, y hemos obtenido los siguientes resultados

```
DOWN
NO
integration_time 0 delta_t 0
0
0

[x;
theta]

[dx;
dtheta]

[ddx;
ddtheta]

[Fgby;
Fgbz;
Mgbx;
Mgby;
Mgbz;
Fbpx;
Fbpy;
Fbpz;
Mbpx;
Mbpz]

[-Fbpx-mblock*ddx;
-Fbpy+Fgby;
-Fbpz-mblock*g+Fgbz;
1/2*d*Fgby+Mgbx-Mbpx;
Mgby;
Mgbz-Mbpz;
Fbpx-(l*cos(-theta)*ddtheta+l*dtheta^2*sin(-theta)+ddx)*mpendulum;
Fbpy;
Fbpz-(l*dtheta^2*cos(-theta)-l*sin(-theta)*ddtheta)*mpendulum-g*mpendulum;
Mbpx-l*cos(-theta)*Fbpy;
```

```
I2*ddtheta-l*Fbpz*sin(-theta)+l*Fbpx*cos(-theta);
l*sin(-theta)*Fbpy+Mbpz]

[-mblock , 0 , 0 , 0 , 0 , 0 , 0 , -1 , 0 , 0 , 0 , 0;
0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , -1 , 0 , 0 , 0;
0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , -1 , 0 , 0;
0 , 0 , 1/2*d , 0 , 1 , 0 , 0 , 0 , 0 , 0 , -1 , 0;
0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , -1;
-mpendulum , -l*cos(-theta)*mpendulum , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0;
0 , l*sin(-theta)*mpendulum , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , -l*cos(-theta) , 0 , 1 , 0;
0 , I2 , 0 , 0 , 0 , 0 , 0 , l*cos(-theta) , 0 , -l*sin(-theta) , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , l*sin(-theta) , 0 , 0 , 1]

[0;
0;
mblock*g;
0;
0;
0;
-mpendulum*ddx-l*cos(-theta)*ddtheta*mpendulum+(l*cos(-
theta)*ddtheta+l*dtheta^2*sin(-theta)+ddx)*mpendulum;
0;
(l*dtheta^2*cos(-theta)-l*sin(-theta)*ddtheta)*mpendulum+g*mpendulum+l*sin(-
theta)*ddtheta*mpendulum;
0;
0;
0]

integration_time 10 delta_t 0.001

q=
0
1.5708

dq=
0
0

Phi=
0
0
10
0
0
0
0
0
10
0
0
0

J=
-1      0       0       0       0       0       0       -1      0       0
0       0
0       0       1       0       0       0       0       0       -1      0
0       0
```

```
0       0       0       1       0       0       0       0       0       -1
0       0
0       0       0.5     0       1       0       0       0       0       0
-1      0
0       0       0       0       0       1       0       0       0       0
0       0
0       0       0       0       0       0       1       0       0       0
0       -1
-1      3.6732051033466e-06     0       0       0       0       0       1
0       0       0       0
0       0       0       0       0       0       0       0       1       0
0       0
0       -0.99999999999325       0       0       0       0       0       0
0       1       0       0
0       0       0       0       0       0       0       0
3.6732051033466e-06     0       1       0
0       1       0       0       0       0       0       -3.6732051033466e-06
0       0.99999999999325        0       0
0       0       0       0       0       0       0       0       -
0.99999999999325        0       0       1

pd=
        0       0       0       0       0       0       0       0       0
0       0       0

ddqunknowns=
0
0
0
0
0
0
0
0
0
0
0
0

pd=
        8       9       7       0       10      1       11      5       2
3       4       6

ddqunknowns=
-9.1830127590131e-06
-4.9999999999831
1.7342016077436e-15
15.000000000051
1.2965926985877e-15
2.5638943521033e-16
-1.2948923332718e-17
9.1830127577598e-06
2.6460087602881e-16
5.0000000000506
1.1731856335984e-15
-9.4039467031208e-16

ddq=
-9.1830127590131e-06
-4.9999999999831
```

```
lambda1=
1.7342016077436e-15
15.000000000051
1.2965926985877e-15
2.5638943521033e-16
-1.2948923332718e-17
9.1830127577598e-06
2.6460087602881e-16
5.0000000000506
1.1731856335984e-15
-9.4039467031208e-16

Going to Integrate
integration_time=10,  deta_t=0.001, steps=10000
t = 0, 134531840 of 10000
```

Ahora podemos ver el resultado de la ejecución del código anterir pero esta vez con el valor *YES* en la variable *"atomization"*.

```
DOWN
YES
integration_time 0 delta_t 0
1
0

[x;
theta]

[dx;
dtheta]

[ddx;
ddtheta]

[Fgby;
Fgbz;
Mgbx;
Mgby;
Mgbz;
Fbpx;
Fbpy;
Fbpz;
Mbpx;
Mbpz]

[atom23;
atom24;
atom25;
atom28;
Mgby;
atom29;
atom33;
Fbpy;
atom34;
atom40;
atom42;
atom41]
```

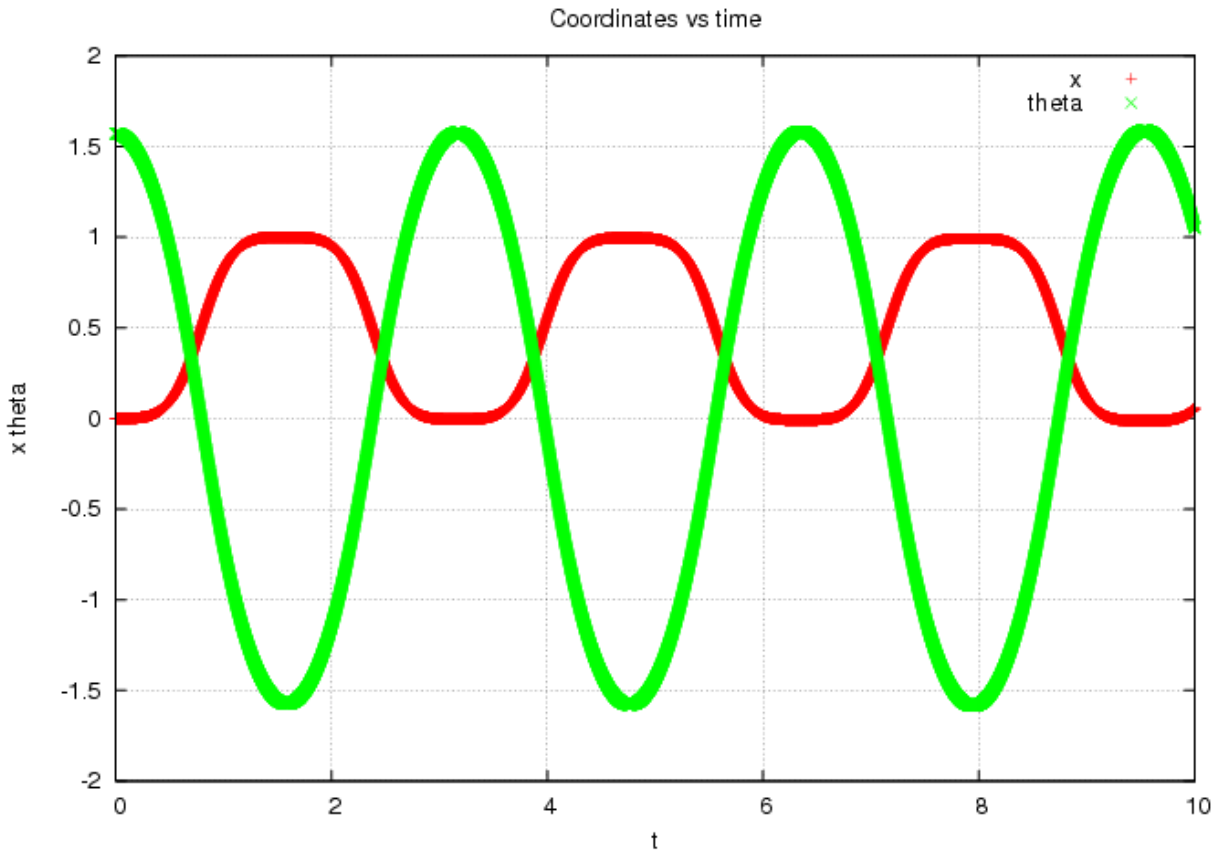
```
[-mblock , 0 , 0 , 0 , 0 , 0 , -1 , 0 , 0 , 0 , 0;
0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , -1 , 0 , 0 , 0;
0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , -1 , 0 , 0;
0 , 0 , 1/2*d , 0 , 1 , 0 , 0 , 0 , 0 , 0 , -1 , 0;
0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , -1;
-mpendulum , atom43 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0;
0 , atom44 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , atom6 , 0 , 1 , 0;
0 , I2 , 0 , 0 , 0 , 0 , 0 , -atom6 , 0 , atom5 , 0 , 0;
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , -atom5 , 0 , 0 , 1]

[0;
0;
-atom0;
0;
0;
0;
-atom52;
0;
-atom53;
0;
0;
0]
integration_time 10 delta_t 0.001

q=
0
1.5708

dq=
0
0

Phi=
0
0
10
0
0
0
0
0
10
0
0
0

J=
-1      0       0       0       0       0       0       -1      0       0
0       0
0       0       1       0       0       0       0       0       -1      0
0       0
0       0       0       1       0       0       0       0       0       -1
0       0
0       0       0.5     0       1       0       0       0       0       0
-1      0
0       0       0       0       0       1       0       0       0       0
0       0
```

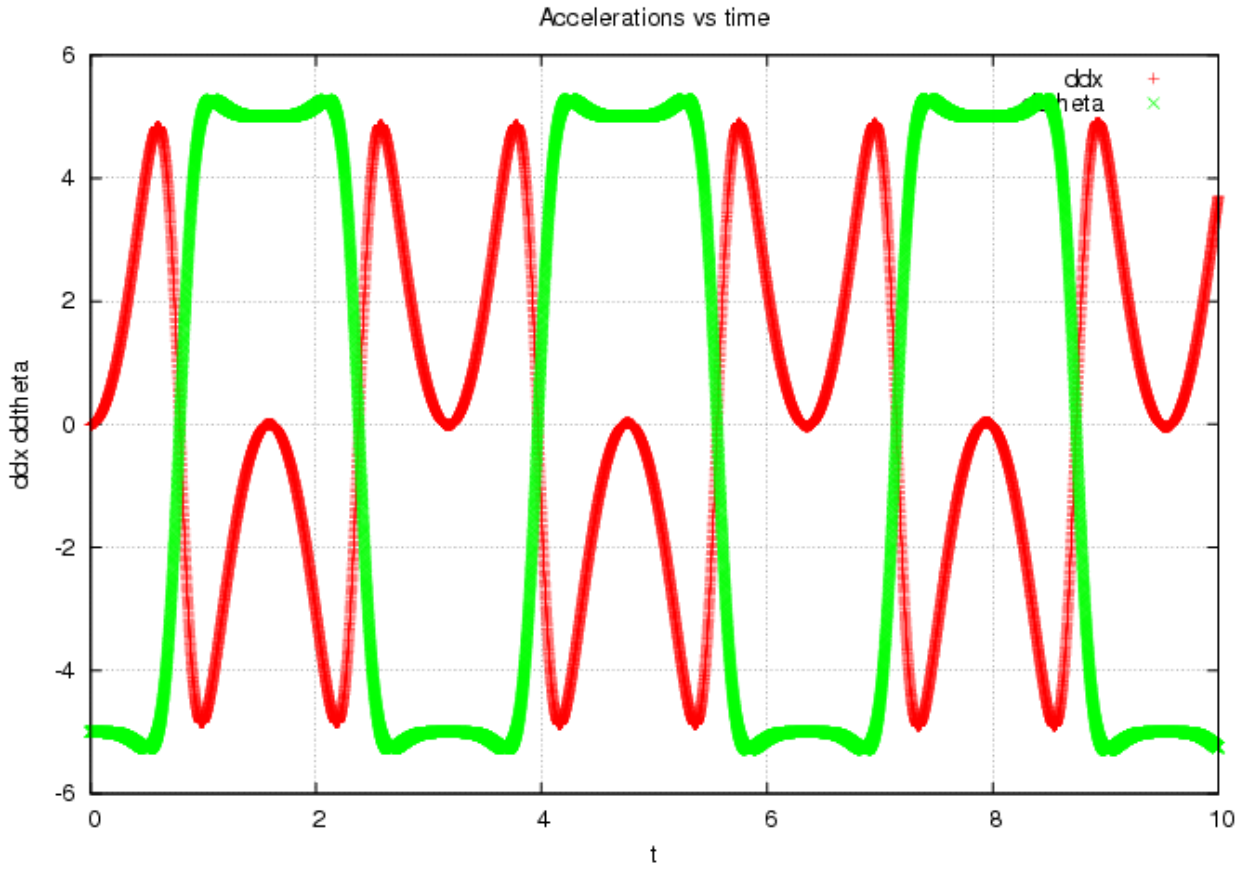
```
0       0       0       0       0       0       1       0       0       0
0       -1
-1      3.6732051033466e-06     0       0       0       0       0       1
0       0       0       0
0       0       0       0       0       0       0       0       1       0
0       0
0       -0.99999999999325       0       0       0       0       0       0
0       1       0       0
0       0       0       0       0       0       0       0
3.6732051033466e-06     0       1       0
0       1       0       0       0       0       0       -3.6732051033466e-06
0       0.99999999999325        0       0
0       0       0       0       0       0       0       0       -
0.99999999999325        0       0       1

pd=
        0       0       0       0       0       0       0       0       0
0       0       0

ddqunknowns=
0
0
0
0
0
0
0
0
0
0
0
0

pd=
        8       9       7       0       10      1       11      5       2
3       4       6

ddqunknowns=
-9.1830127590131e-06
-4.9999999999831
1.7342016077436e-15
15.000000000051
1.2965926985877e-15
2.5638943521033e-16
-1.2948923332718e-17
9.1830127577598e-06
2.6460087602881e-16
5.0000000000506
1.1731856335984e-15
-9.4039467031208e-16

ddq=
-9.1830127590131e-06
-4.9999999999831

lambda1=
1.7342016077436e-15
15.000000000051
1.2965926985877e-15
2.5638943521033e-16
-1.2948923332718e-17
```

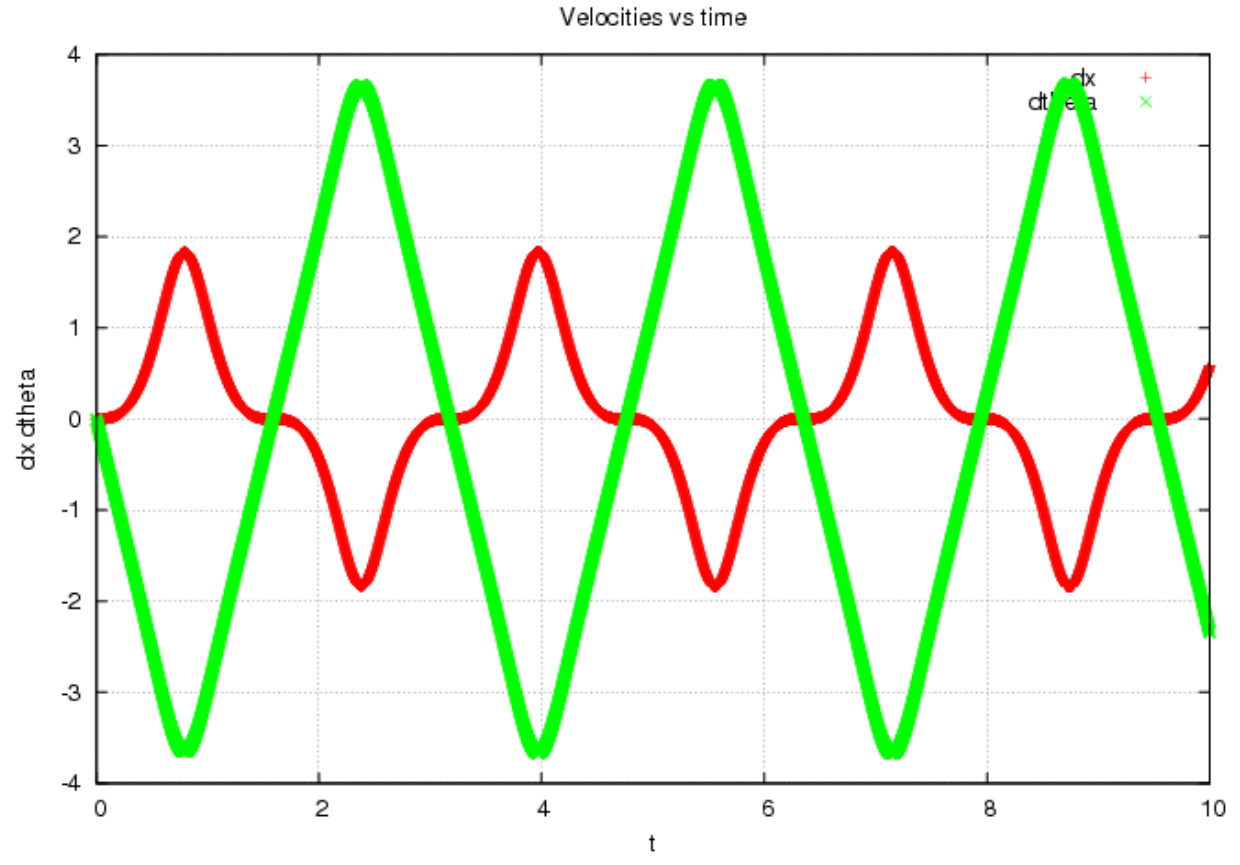
```
9.1830127577598e-06
2.6460087602881e-16
5.0000000000506
1.1731856335984e-15
-9.4039467031208e-16

Going to Integrate
integration_time=10,  deta_t=0.001, steps=10000
t = 0, 134531696 of 10000
```

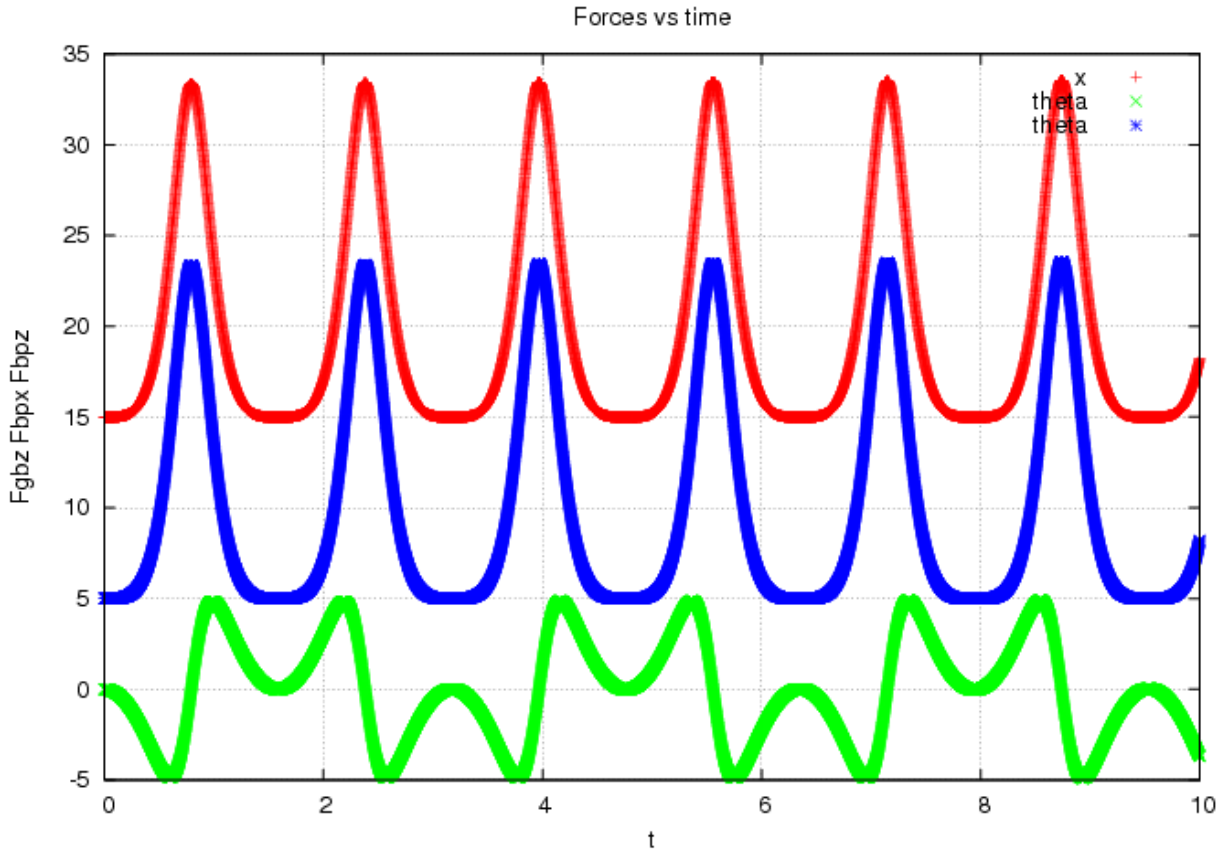
Gráficas

Al final de la ejecución del programa anterior, tambien podremos ver automáticamente por pantalla las siguientes gráficas, generadas con los resultados anteriormente mostrados.




Accelerations vs time
ddx
dtheta
ddx ddtheta
t
6
4
2
0
-2
-4
-6
0
2
4
6
8
10


Velocities vs time
dx
dtheta
dx dtheta
t
4
3
2
1
0
-1
-2
-3
-4
0
2
4
6
8
10


Forces vs time
x
theta
theta
Fgbz Fbpx Fbpz
t
35
30
25
20
15
10
5
0
-5
0
2
4
6
8
10

5 - Instalación y uso

Para que podamos instalar la librería lib3d_mec_GiNaC correctamente y poder ejecutar programas que hagan uso de la misma, debemos seguir los siguientes pasos:

Tener instaladas las librerías GiNaC, GLS y CLN para poder compilar la librería y el programa GNUPLOT para poder visualizar resultados mediante gráficas. Para ello podemos descargarnos los paquetes binarios de dichas librerías mediante los siguientes scripts.

```
   $ apt-get install libginac1.3c2a libginac-dev libcln4 libcln-dev
gnuplot libgsl0  libgsl0-dev
```

Una vez hemos instalado correctamente las librerías en nuestro ordenador, ahora podemos proceder a instalar la librería lib3d_mec_GiNaC, para ello, donde se encuentre la librería, ejecutamos las siguientes instrucciones

```
   $ ./autogen.sh –preffix=”directorio donde realizar la instalacion”
   $ ./configure –preffix=”directorio donde realizar la instalacion”
   $ make
   $ make install
```

Según el sistema operativo o distribución con la que estemos trabajando, a lo mejor es necesaria la instalación de alguna librería o programa adicional. En este caso, nos aparecerían mensajes indicándonos de cuales se tratan.

Una vez instalada la librería correctamente, debemos indicar cual es la localización de la librería. En el caso de que hallamos omitido las opciones *“–preffix”* en el apartado anterior, no sera necesario el siguiente paso.

```
   $ export LD LIBRARY PATH=”directorio de instalacion”/lib/
   $ export PKG CONFIG PATH=”directorio de instalacion”
/lib/pkgconfig/
```

Finalmente podremos realizar la compilación de un programa que haga uso de la librería lib3d_mec_GiNaC pudiendo hacer uso de todos sus objetos y funciones. Para ello escribimos la siguiente instrucción.

```
   $ g++ -o nombre programa nombre programa.cc `pkg-config --cflags -
-libs libmecginac-1.0 ginac cln`
```

Una vez compilado, podemos ejecutarlo y ver los resultados, escribiendo "./" delante del nombre que le hemos dado a nuestro programa y escribiendo los posibles parámetros que sean necesarios.

```
$./nombre_programa [parametro1 parametro2 …]
```

Adicionalemente en el código de las aplicaciones podemos hacer uso de las constantes `LIB3D_MEC_GINAC_MAJOR_VERSION` y `LIB3D_MEC_GINAC_MINOR_VERSION` las cuales nos indicarán la versión de la librería que estamos utilizando y que nos puede ser util a la hora de declarar o inicializar variables o constantes o funciones según sea la versión de que dispongamos.

6 - Bibliografía

- *“Desarrollo de software con C++”, Russel Winder I.S.B.N. 84-7978-21-8-8*
- *“Thinking in C++, Volume 1: Introduction to Standard C++”, Bruce Eckel I.S.B.N. 0-13-979809-9*
- *“Thinking in C++, Vol. 2: Practical Programming”, Bruce Eckel, Chuck Allison I.S.B.N.: 0-13-122552-9*
- *“GiNaC tutorial” http://www.ginac.de/tutorial.pdf*
- *“GiNaC Reference Manual 1.4.0” http://www.ginac.de/reference.pdf*
- *"Kinematic and Dynamic Simulation of Multibody Systems. The Real-Time Challenge", ISBN 0-387-94096-0, 440 pp., Springer-Verlag, New-York, 1994*
- *“Mecánica de La Partícula y del Sólido Rígido, J. Agulló”, Publicaciones OK Punt, I.S.B.N.: 84-920850-5-3, 2000*
- *Apuntes propios asignatura Mecánica de 2º II*
- *“Numerical Recipes in C, The Art of Scientific Computing”. William H. Press, “et al.”. CAMBRIDGE UNIVERSITY PRESS.*

7 – Fecha y firma

Este proyecto ha sido realizado por LUIS MIGUEL ARRONDO MARTÍNEZ , estudiante de INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Fdo: Luis Miguel Arrondo Martínez
Pamplona, Noviembre de 2007