Introducción al Desarrollo de Interfaces Gráficas de Usuario para la Simulación de Fenómenos Físicos en la Infraestructura de Análisis de Datos ROOT.

Andrés Mateus Vargas Hernández.

Julian Andrés Salamanca Bernal. PhD.

Índice general

Capítulo 1

Interfaces de usuario, ROOT y el paradigma de programación orientada a objetos

1.1. Introducción

> "The irresistible beauty of programming consists in the reduction of complex formal processes to a very small set of primitive operations."
>
> (Dewar y Schonberg, 2008)

La escritura de software es una de las áreas más interesantes de aplicación de los principios básicos de la lógica matemática. Los elementos fundamentales del software más complejo son la definición de unas determinadas variables y decisiones a partir de proposiciones lógicas simples y compuestas que determinan la evolución de esas variables: el software es entonces un proceso.

De la misma manera que una ecuación diferencial establece la evolución de unas variables a partir de unos valores iniciales o unas condiciones de frontera, el software determina la evolución de unas variables a partir de una interacción, de un diálogo con el usuario. El desarrollo de software está centrado en la solución de un problema; el problema en sí mismo es el fundamento del software y su razón de ser.

Para el usuario final no es relevante *cómo trabaja* el software (para éste no es relevante los procesos internos que el software ha de realizar con el fin de ser útil). El usuario promedio diariamente ejecuta tareas con programas cuyo funcionamiento interno ignora. Él conoce cómo dialoga con el software para ejecutar sus tareas; él sabe que para guardar un archivo debe hacer clic sobre un botón que tiene un dibujo de un *diskette* en su interior; no le resulta necesario saber que una vez pulsa ese botón se producen una serie de procesos que comunican al software del equipo con su

hardware, y que a través de otro software que se conoce como el núcleo del sistema, existe una traducción de toda la información que él introduce, en forma de unos y ceros; que luego a partir de la aplicación de principios electromagnéticos es posible leer y escribir en un soporte magnético como el disco duro, para el usuario final lo relevante es cómo soluciona su problema particular a partir del uso del software. Es en este punto donde entra en juego el papel de las interfaces de usuario, estos son mecanismos de interacción entre el usuario y la máquina.

En la enseñanza de un fenómeno físico, la interfaz del software no puede convertirse en el objeto de estudio, ya que el objeto de estudio es el fenómeno físico; lo que sí representa el software es un recurso para lograr entender ese fenómeno. Es importante entonces, que la interfaz proporciona al software sea tan intuitiva como sea posible, que ésta reduzca al mínimo el trabajo que el usuario debe realizar, los manuales que debe leer con el fin de entender *cómo usar* el software, en palabras de Cortes (1997)"un buen software es aquel que no sólo funciona sino que también es fácil aprender (cómo usarlo)".

En la escritura de software se tienen diferentes niveles; se cuenta con lenguajes de programación de bajo, medio y alto nivel, cuya principal diferencia radica en la forma en que el programador (quien escribe software) se expresa. En lenguajes de bajo nivel, el programador se expresa mediante referencias al hardware de la máquina; por el contrario, los lenguajes catalogados como de alto nivel el lenguaje es muy parecido al lenguaje natural (en inglés) así como normalmente se expresan condicionales: si se cumple una condición se desarrolla éste procedimiento, si no se cumple se ejecuta éste otro. Es necesario recordar que por más natural que se lea un software, todas las instrucciones deben pasar por un proceso de traducción a lenguaje máquina (nuevamente, a unos y ceros) lo que se logra a través de un proceso de compilación. Es preciso mencionar que también existen lenguajes que pueden ser de alto y bajo nivel de manera simultánea como lo es C++.

La utilización lenguajes de programación (en particular de alto nivel), permiten al programador abstraerse de los procedimientos que deben ocurrir para que el software, que está escribiendo, funcione; es decir, él no necesita saber qué pasa en el interior de la máquina para que se muestre un mensaje en la consola del sistema gracias a que existen *interfaces* que se encargan de esas tareas (por ejemplo, en C++, él agrega una instrucción del tipo `cout << "hola mundo";` y el mensaje es mostrado). Es así como el programador es a la vez usuario: "*Los usuarios pueden concentrarse en su problema particular. Ellos no tienen que ser expertos en la escritura de interfaces de usuario, expertos en gráficos, o en sistemas de redes para usar la infraestructura que provee esos servicios.*"(Rademakers, s.f.); es decir, que el programador es usuario de un conjunto de librerías (un conjunto de funciones que facilitan la ejecución de determinadas tareas), y éstas a su vez, pueden hacer parte de una infraestructura de desarrollo de software, la cual es otra vez, un conjunto de herramientas que el programador puede utilizar (y acomodar) de modo tal que sirvan a la escritura de software: entre más amplia sea la infraestructura más sencillo y conciso es el proceso de escritura y más funcional es la

infraestructura.

Con todo esto es posible ver que existen diferentes niveles, tanto en la ejecución como en la programación de *software*, desde el más fundamental que se encarga de la comunicación entre el software y el *hardware* (el núcleo del sistema) hasta la ejecución de un software determinado por parte del usuario final. El objetivo del presente escrito, es entonces, lograr que el lector entienda el mecanismo de un tipo especial de interfaz de usuario conocida cómo interfaz gráfica, y cómo puede él a través del uso de una infraestructura de desarrollo particular (ROOT) diseñar e implementar (escribir o crear) interfaces gráficas de usuario enfocadas en la simulación de fenómenos físicos. El presente documento se desarrolla en el marco de la ejecución del proyecto de investigación *Interfaces gráficas en ROOT para la enseñanza de la física* (Salamanca, 2011).

1.2. Mecanismos de interacción usuario-máquina

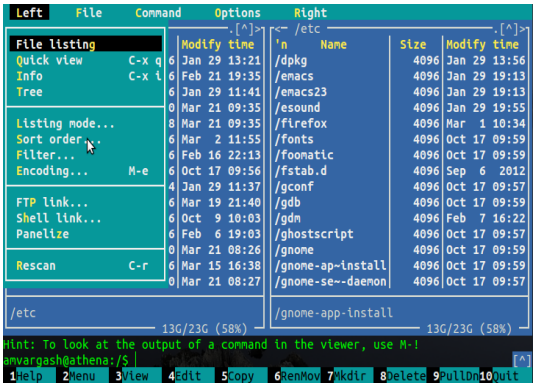
Existen básicamente dos métodos a través de los cuales el usuario interactúa con los procesos lógicos que se producen debido a la ejecución del software. En el caso de las simulaciones, ésta interacción se evidencia en la solicitud de información por parte del programa al usuario acerca de las condiciones en las que se reproducirá determinado experimento. Dicha *solicitud* de información puede darse a través de una *interfaz de línea de comandos* (*CLI* por las siglas en inglés de *Command Line Interface*) en donde el usuario interacciona con el software a través de una terminal o consola. éste tipo de interfaz requiere menos tiempo de desarrollo y presenta mayor portabilidad entre diferentes sistemas operativos (Manoukis y Anderson, 2008). Esta interfaz puede ser:

- *Interactiva*: Cuando el software solicita de manera secuencial la información al usuario. Permite que una vez realizada la acción, se continúe impartiendo instrucciones adicionales al programa. Un ejemplo de ello es la manipulación de funciones y datos experimentales como *gnuplot,* software de cálculo simbólico como *maxima*, o software para cálculo numérico como *octave*.[1]
- *Sintáctica*: En una línea inicial, que responde a determinada sintaxis, se establece totalmente los parámetros de ejecución del programa. Una vez se ejecuta la acción solicitada, finaliza la ejecución del programa. Por ejemplo, la utilidad *GNU units* permite realizar la conversión de unidades entre diferentes sistemas métricos, si ejecuta `$units '1.5 kg*m/s' 'g*cm/s'`, transformará la medida $1{,}5\frac{kg*m}{s}$ de unidades en el sistema internacional a unidades del sistema cegesimal, el programa imprime en pantalla la respuesta ($15*10^3\frac{g*cm}{s}$) y termina la ejecución.

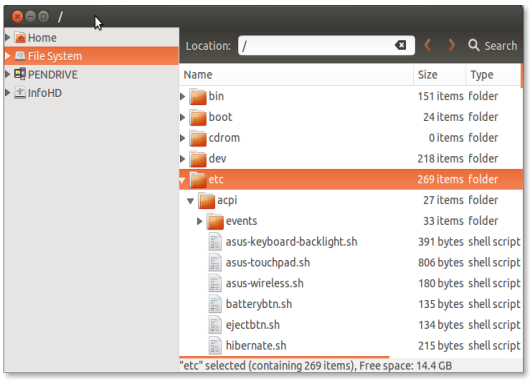
[1]Tenga en cuenta que estos programas también cuentan con interfaces gráficas como *gnuplot-qt*, *wxmaxima* o *qt-octave,* aquí se hace referencia al software base los cuales están basados en línea de comandos.

Una característica importante de la interfaz de líneas de comando, que se constituye en la principal diferencia con otros mecanismos de interacción, es que ésta sólo admite que el diálogo entre el usuario y la máquina se de a través del teclado; no se puede interactuar con ésta interfaz a través de dispositivos como el *mouse* (Cortes, 1997), lo que implica que el usuario debe conocer la sintaxis del comando con el fin de ejecutar determinadas tareas. También es posible hacerlo a través de una *Interfaz Gráfica de Usuario* (*GUI* por las siglas en inglés de *Graphical User Interface*) la cual se presenta de una manera intuitiva al usuario, ya que no está obligado a conocer los comandos del software, ni su sintaxis (Smith, 1997). En palabras de Manoukis y Anderson (2008), "*Las Interfaces Gráficas de Usuario proporcionan accesibilidad inmediata y un modo de interacción familiar para la mayoría de usuarios*", lo que permite que en un tiempo reducido el usuario logre entender el modo en que se utilizan los diferentes objetos que son presentados en la interfaz, como son los botones, cuadros de texto, listas, opciones de selección, etc.

Dentro del las interfaces gráficas de usuario se encuentran incluidas las Interfaces Textuales de Usuario (*TUI* por las siglas en inglés de *Textual User Interface*), un tipo especial de interfaz que utiliza símbolos y letras para representar objetos como botones o menús. La figura 1.1a muestra una interfaz gráfica basada en texto, mientras que la figura 1.1b muestra una interfaz gráfica basada en imágenes. Los diversos sistemas operativos evolucionaron desde una interfaz de línea de comandos e interfaces de usuarios basadas en texto hacia una interfaz gráfica que proporciona al usuario una experiencia más intuitiva requiriendo un menor tiempo para la familiarización del usuario con el funcionamiento básico del software.



(a) Interfaz de Usuario Textual (TUI): Administrador de archivos *mc*



(b) Interfaz Gráfica de Usuario (GUI): Administrador de archivos *nautilus*

Figura 1.1: Interfaces Gráficas de Usuario

1.3. La infraestructura de análisis de datos ROOT

El nacimiento de ROOT fue liderado por Rene Bruns y Fons Rademakers con el fin de crear una plataforma robusta que permitiera el análisis de enormes cantidades de datos proporcionados en experimentos realizados por el Centro de Estudio en Altas Energías y Física Nuclear (CERN por sus siglas en francés) organización de la cual recibe soporte oficial desde el año 2003 (Rademakers, Goto, Canal, y Brun, 2003). ROOT es una infraestructura cuyo desarrollo inicia en el año 1995 (Rademakers y cols., 2003), la cual recibe el apoyo de instituciones como *Fermilab* desde 1998 y que actualmente (luego de 18 años) cuenta con una alta frecuencia de actualización y desarrollo (ROOT-Team, s.f.-c), mostrada en la amplia participación de la lista de correo *roottalk*[2]. La capacidad de *ROOT* se ha utilizado en diversas áreas como el procesamiento de imágenes satelitáles del clima con el proyecto *Focus Solar*, o la investigación en economía con proyectos como *Forex Automaton* (ver (ROOT-Team, s.f.-a)).

ROOT cuenta con más de 2000 clases diseñadas para la solución de problemas en el área de la física, escritas en C++ uno de los lenguajes más utilizados en la comunidad científica para la ejecución de tareas que requieren cálculo intensivo y alto desempeño (Prabhu y cols., 2011). Entre ellas se destacan: la representación gráfica de una cantidad arbitraría datos a través histogramas, ajuste de curvas, clases para generar y graficar funciones en dos, tres dimensiones y más dimensiones a través del uso de escalas y cronómetros; generación de números y eventos aleatorios; capacidad de procesamiento en paralelo en procesadores con múltiples hilos (*threading*), y procesamiento en red a través de redes *master*/*slaver*/*worker*; generación automática de documentación; procesamiento de bases de datos a través de SQL, XML, interfaces con diferentes lenguajes de programación como Python y Ruby (mediaciones); trabajo con geometría; compresión de datos mediante árboles, librerías para la aplicación de métodos en álgebra lineal; interfaces gráficas de usuario, entre otras funciones que *diariamente* se siguen ampliando. Con todo lo anterior, ROOT representa una infraestructura (*framework* en inglés) de software.

Si bien es considerada una infraestructura para el análisis de datos, el autor de éste escrito considera que perfectamente puede ser considerada una infraestructura para el desarrollo de software, pertinente para el desarrollo de simulaciones ya que el programador necesita escribir menos código para solucionar determinado problema si utiliza los servicios que le proporciona ROOT, al ser distribuido bajo la licencia LGPL el programador puede a su vez realizar modificaciones a las rutinas proporcionadas para adecuarlas a las necesidades particulares de la solución del problema, ROOT se desarrolla bajo el paradigma de la *Programación Orientada a Objetos,* inclusive de acuerdo con lo declarado por Rene Brun un posible acrónimo de ROOT es "Rapid Object-Oriented Technology"(Brun, 1998), lo que permite que puedan incorporarse fácilmente funcionalidades adicionales y la reutilización del código para la solución de

[2] *roottalk@cern.ch* es la lista de correo de usuarios de *ROOT.*

actividades variadas.

ROOT incorpora a su vez un interprete de C++ llamado CINT, lo que aumenta la eficiencia del proceso de programación al presentar la opción de ejecutar el código fuente sin necesidad de compilarlo. A su vez presenta una nomenclatura sobre la declaración de clases y tipos de variables.

En general, las clases son nombradas comenzando con la letra T (T de Tipo (Rademakers, s.f.)), como por ejemplo, `TCanvas`, `TLine`, `TEllipse`, y `TGraph` utilizadas para la creación de lienzos, líneas, elipses y diagramas respectivamente. Las clases para el desarrollo de interfaces gráficas de usuario se identifican con una letra adicional G; por ejemplo, la clase que permite generar un botón que contiene texto en su interior es la clase `TGTextButton`, la clase que permite crear una etiqueta es la clase `TGLabel`, una entrada numérica `TGNumberEntry`, etc.

En los tipos de variables primitivos de C++, por convención, se suele agregar el sufijo `_t`. Cabe anotar que también se pueden utilizar las variables tipo provenientes de C++. La tabla 1.1 muestra las correspondencias más utilizadas entre las variables tipo primitivas de C++ y las utilizadas por convención en ROOT[3].

C++	ROOT	C++	ROOT	C++	ROOT
double	Double_t	char	Char_t	unsigned char	UChar_t
int	Int_t	const char	Option_t	unsigned int	UInt_t
float	Float_t	long	Long_t	unsigend long	ULong_t
bool	Bool_t	short	Short_t	unsigned short	UShort_t

Cuadro 1.1: Correspondencias entre los tipos de variables primitivos de C++ y ROOT

Cabe resaltar, que cuando se hace referencia al valor de verdad falso (o verdadero) en una variable de tipo `Bool_t`, se utiliza `kFALSE` (o `kTRUE`); sin embargo, nuevamente, es posible utilizar `false` y `true` tal como se suele hacer en C++ primitivo. Cuando se utilizan palabras clave (*keywords*) que comienzan con `k`, por lo general se hace referencia a constantes que representan opciones, por ejemplo `kFALSE`, es la representación de cero (`0`), y `kTRUE` es la representación de uno (`1`), de la misma forma que lo hacen `false` y `true`; (para mayor referencia consultar la guía del usuario (ROOT-Team, s.f.-b) –Sección: *Getting Started* subsección: *Conventions*-).

1.3.1. Un *tour* por la documentación de ROOT

En el siguiente enlace es posible encontrar el código fuente, un archivo comprimido que contiene la guía de referencia, los binarios para diferentes plataformas entre las cuales se encuentra GNU/Linux, Microsoft Windows, y Mac OS: `http://root.cern.ch/`

[3]Para mayor información puede consultar la documentación de referencia `http://root.cern.ch/root/html/ListOfTypes.html`

drupal/content/downloading-root, seleccionando la versión que se quiera instalar (se recomienda el uso de la versión etiquetada como Pro -*Production*-, la versión estable más reciente).

Existen dos tipos de documentación, la *documentación de usuario* o *guía del usuario y la documentación de referencia. La documentación del usuario* ofrece una visión general de la infraestructura *ROOT*, de su filosofía, objetivo y funcionalidad, en la mayoría de los casos muestra ejemplos específicos del modo de usar las clases, así como *screenshots* de los resultados, éste archivo tiene una frecuencia de actualización baja, escrito especialmente y diseñado para ser leído de manera no necesariamente secuencial pero sí estructurada, es escrito por humanos para humanos.

La documentación de referencia que si bien también es escrita por humanos es generada de manera automática en cada lanzamiento de una versión *pro* (lo cual aumenta su frecuencia de actualización con respecto a su contraparte), a partir de los comentarios en los archivos de código fuente de la infraestructura, ésta documentación no resulta tan accesible como lo resulta la documentación del usuario, ya que es, como su nombre lo indica de *referencia*, es consultada cuando se tiene una pregunta específica acerca de determinada clase, por ejemplo:

- ¿Cuales son los parámetros por defecto que adquiere una instancia de la clase `TGHSlider`?
- ¿Qué tipo de variable devuelve el método `Draw()` de la clase `TEllipse`?
- ¿Qué señales son emitidas por la clase `TGComboBox`?

Es posible consultar la documentación de referencia en línea o descargar un archivo comprimido el enlace `http://root.cern.ch/drupal/content/downloading-root`.

Definamos las siguientes variables de entorno para hacer referencia a los directorios de *ROOT.*

- `ROOTSYS`: En éste directorio se encuentra de instalación de la infraestructura ROOT.
- `ROOTDOC`: En éste directorio se encuentra la documentación de referencia, si no ha realizado la descarga de la documentación ésta variable hace referencia a la url `http://root.cern.ch/root/html/`

En el archivo `$ROOTDOC/ClassIndex.html` se encuentra un índice de todas las clases que ofrece la infraestructura, acompañadas de una breve descripción. `$ROOTDOC/GUI_GUI_Index.html` presenta un subíndice de las clases específicamente diseñadas para la escritura de interfaces gráficas de usuario. Al acceder a este índice y seleccionar alguna de las clases mostradas, será dirigido a una página que contiene la descripción de la clase, un listado de los métodos y variables disponibles para la clase seleccionada, la estructura de herencia de la clase, y ocasionalmente ejemplos. Seleccionando un

método podrá ver la descripción de su función[4] Un recuadro flotante en la esquina superior izquierda (ver figura 1.2) mostrará la siguiente información:

- `library`: Contiene el nombre de la librería compartida en la cual se encuentra la definición de la clase una vez ha sido compilada. Cada clase es compilada en una librería que contiene clases comunes. La mayoría de las clases que se abordan en el presente trabajo se encontrará en la librería `libGui`. Las librerías compiladas se encuentran en el directorio `$ROOTSYS/lib/`

- `#include`: Contiene la línea de código que carga la clase para poder crear objetos de éste tipo al momento de escribir aplicaciones o prototipos que utilicen la infraestructura.

- `Show inherited`: Ésta opción muestra todos los métodos y variables que han sido heredadas de otras clases.

- `Show non-public`: Al marcar ésta opción se muestran los métodos y variables de tipo `private` y `protected` tanto de la clase seleccionada como de su ascendencia.

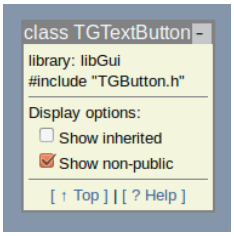


Figura 1.2: Ventana flotante de la documentación de referencia de ROOT.

Si se sigue el enlace marcado en el título de la descripción del método, puede ser redirigido bien sea al archivo de cabecera (`*.h`) de la clase (en el caso en el que la definición del método esté en la misma cabecera) o al archivo en el que se encuentran implementados (`*.cxx`).

[4] En ocasiones la descripción del método puede ser deducida de la variable que devuelve al ser ejecutado.

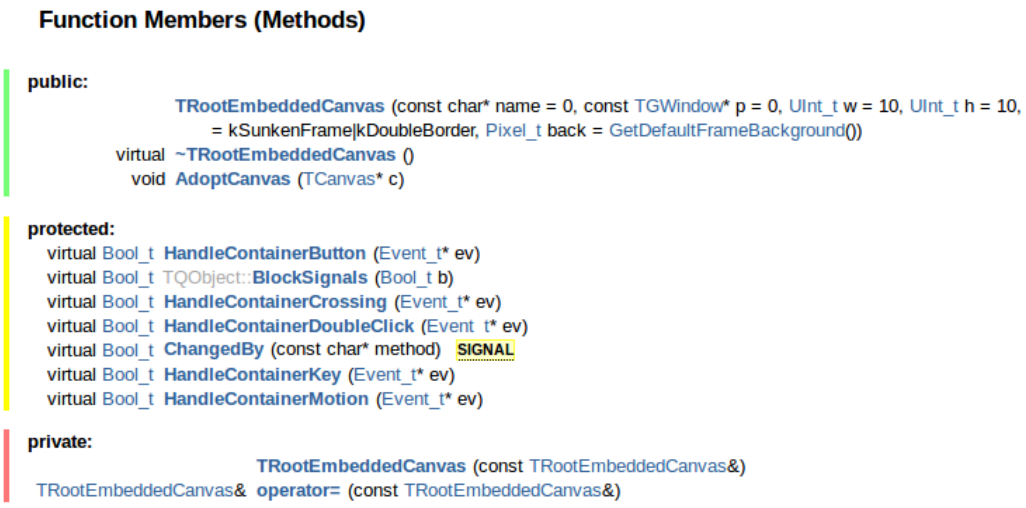


Figura 1.3: Resumen de la Documentación de Referencia de la clase TGMainFrame.

En el directorio `$ROOTDOC/src` se encuentra el código fuente (escrito en formato web) de todas las clases de la infraestructura. Allí se encuentran tanto los archivos de cabecera (`*.h`) (donde se declaran los métodos), como los archivos donde se implementan los métodos (`*.cxx`). Por ejemplo, para la clase `TGMainFrame` el resumen se encontrará en `$ROOTDOC/TGMainFrame.html`, el cual contiene una breve descripción de la clase, ejemplos (ocasionalmente) y todos los métodos de la clase (dependiendo de las opciones seleccionadas en la ventana flotante de la referencia). La globalidad de los métodos es indicada por el color de la margen izquierda en la figura 1.3, los métodos de tipo `public` se encuentran cobijados por la línea verde, los de tipo `protected` por la línea amarilla y los de tipo `private` por la línea roja, esto también se aplica a las variables globales utilizadas en la clase (conocidas como *data members*, no se muestran en la figura 1.3).

Continuando con la ubicación de la documentación de referencia de la clase `TGMainFrame`, el archivo de cabecera se encontrará en `$ROOTDOC/src/TGMainFrame.h.html`, y el archivo de definición de los métodos en `$ROOTDOC/src/TGMainFrame.cxx.html`.

Las señales (correspondientes al mecanismo *signal-slot* abordado en la sección 2.5.1) de la clase se pueden identificar porque contienen la etiqueta **SIGNAL** tanto en el archivo de resumen de la clase como en el archivo de cabecera; por ejemplo, en la figura 1.3 (ver figura 1.3) se puede observar que el método `ChangedBy()` emite una señal.

1.4. Términos en la programación orientada a objetos

A lo largo del presente texto se utilizan de manera frecuente términos relacionados con la programación; en particular, con el paradigma de programación orientada a

objetos. Es indispensable tener claro a qué se hace referencia cuando se utiliza un término; por esto, en la presente sección se relaciona cada uno de los términos con su significado y con su interpretación en éste contexto. Términos como clases, métodos, instancias, herencia, entre otros son fundamentales para entender éste paradigma. Se han escrito libros completos acerca de la programación orientada objetos, y existen múltiples lenguajes que hacen uso de ésta manera de plantear una solución a un problema [5]. Aquí se describirán a grandes rasgos los conceptos y su relación con nuestro objeto de estudio: las interfaces gráficas de usuario.

1.4.1. Clases, métodos, instancias, herencia, e Implementacion en C++.

La programación orientada a objetos es un paradigma en el que se aborda los problemas de la programación como problemas del mundo real. En particular, la OOP (siglas de *Object Oriented Programming*) es una perspectiva física del mundo. Si pensamos en un objeto (un carro, una silla, un hombre, etc), éste objeto puede ser definido por un conjunto de variables (alto, ancho, color, masa, etc) y es así como caracterizamos un objeto en el mundo real.

Es necesario recordar que los objetos son de determinado tipo o de determinada *clase*. En programación, una *clase* representa una idea. Por ejemplo, la idea de Carro es una idea general ya que pensamos en un conjunto de variables que pueden definir un carro; sabemos que tiene unas dimensiones espaciales, una altura, una anchura, una forma determinada por una geometría, un color o combinaciones de colores. También tendríamos otras como la capacidad máxima de aceleración, la eficiencia en el consumo de combustible, la estabilidad en curvas, la posición, cantidad de movimiento, y aceleración. Estas y otras variables que definirían completamente un carro (acompañadas por unos métodos) conformarían lo que en programación se definiría como la *clase* carro.

Fijémonos que aún no hemos asignado valores a esas variables[6]; por tanto, *una clase define tanto las variables a través de las cuales es posible caracterizar una idea, como los métodos a través de los cuales es posible leer y manipular el valor de esas variables*. Si las *Clases* son ideas generales, a lo particular, se le llama *objeto* o *instancia*. Por ejemplo, pensar en el último carro que usted observó al llegar a su trabajo, ese carro es un *objeto* o una *instancia* de la clase carro, ya que éste no es general; el conjunto de variables que definen un carro han adquirido un valor específico. Se puede pensar en

[5] para una introducción extensa a la programación orientada a objetos en C++ puede consultar (Eckel, 2000), una breve descripción de los beneficios del uso del paradigma de programación orientada a objetos descrito por uno de los creadores de *ROOT* consulte (Rademakers, s.f.)) .

[6] En programación para caracterizar una variable tenemos básicamente dos procedimientos: declaración y asignación. En la declaración de una variable se asigna un nombre a la variable y un tipo de variable, digamos que tenemos la variable Altura y que es de tipo float, en ese caso la sentencia que utilizaríamos sería (float Altura;). El segundo procedimiento es la asignación de un número a esa variable (i.e Altura = 1.75), podemos cambiar ese valor de tal manera que la variable Altura tendrá diferentes valores en instantes distintos.

los objetos como asignaciones específicas de las variables que conforman una clase; es decir, un determinado valor de altura, un valor determinado de anchura, una geometría determinada, un color determinado.

Para ilustrar lo anterior, se realizará la implementación de la clase `Velocidad`. Dos métodos fundamentales de toda clase (que el lector recordará) son su *constructor*, que es un método que se identifica porque se llama tal como se llama la clase; y el *destructor*, que se identifica por llamarse como la clase, pero precedido pero por una tilde (~). Esto se ilustra en la clase `Velocidad` en las líneas 6 y 7 :

```
1  class Velocidad {
2   protected:
3    Double_t fVx; //Componente horizontal
4    Double_t fVy; //Componente vertical
5   public:
6    Velocidad(Double_t vx, Double_t vy);
7    ~Velocidad() {}
8    Double_t GetVx() { return fVx; }
9    Double_t GetVy() { return fVy; }
10   void SetVx(Double_t vx) { fVx = vx; }
11   void SetVy(Double_t vy) { fVy = vy; }
12 };
```

Mediante el constructor, se asignan unos valores a cada una de las variables que definen el objeto, cuando se crea una nueva instancia. Como se puede observar, la anterior definición de la clase velocidad guarda la estructura de una clase en C++:

```
class NombreDeClase {
  ...
  //Cuerpo
  ...
};
```

Los métodos así como las variables de las clases tienen cierto nivel de globalidad. Los métodos de tipo `protected` son métodos que sólo pueden ser utilizados o llamados al interior de la clases o de clases que heredan sus métodos. Los métodos de tipo `private` son aquellos que son utilizados exclusivamente por la clase en la cual están declarados y definidos. Éstos dos tipos de métodos están relacionados con los procesos internos de la clase, que por conveniencia sólo deben ser accesibles a los desarrolladores de la clase; con esto es posible, por ejemplo, hacer más eficiente la ejecución de un método sin afectar al usuario de la clase. Se debe resaltar que los métodos de tipo `public` son métodos que pueden ser llamados desde cualquier lugar del código, desde otros programas, o donde exista una instancia de la clase. Estas clasificaciones (`private`,`public`, y `protected`) se aplica de igual forma para las variables de la clase.

En la definición de la clase `Velocidad`, se declaran inicialmente dos variables: `fVx` y `fVy`, que son de tipo `Double_t`. Luego se definen los métodos: `GetVx()`, `GetVy()`. La palabra *Get* traduce obtener, y es lo que se puede observar allí, ya que el método `GetVx()` devuelve el valor que tiene la variable `fVx`. Los métodos `SetVx()` y `SetVy()` permiten establecer (o fijar) el valor de las variables `fVx` y `fVy` respectivamente.

Ahora bien, se dice que una clase *hereda* los métodos de otra cuando por medio de la primera (la clase que hereda) es posible utilizar los métodos y variables de la segunda (la clase de la cual se hereda). La estructura de la clase que hereda:

```
class NombreDeClaseQueHereda:ClaseDeLaCualHereda{
  ...
  //Cuerpo
  ...
};
```

El programador tiene entonces la tarea de crear la clase `CantidadDeMovimiento`. Como ya se cuenta con una clase `Velocidad`, se puede heredar los métodos y variables de ésta para así poder utilizarlos. Por ejemplo, obtener el valor de la componente en x de la velocidad (`fVx`) a través de la clase `CantidadDeMovimiento` es uno de ellos.

En el caso particular que se ha planteado la clase CantidadDeMovimiento queda definida:

```
1  class CantidadDeMovimiento : Velocidad {
2   protected:
3    Double_t fM; //Variable para la Masa del cuerpo
4    Double_t fPx, fPy; //Componentes
5   public:
6    CantidadDeMovimiento(Double_t M, Double_t vx, Double_t vy);
7    ~CantidadDeMovimiento() {}
8    Double_t GetPx() { return fM*GetVx(); }
9    Double_t GetPy() { return fM*GetVy(); }
10   void SetM(Double_t m) { fM = m; }
11 };
```

Aquí la cantidad de movimiento está definida clásicamente por la expresión $\vec{p} = m\vec{v}$, donde m es la masa de un cuerpo y $\vec{v}$ su velocidad. La escritura de la clase `CantidadDeMovimiento` se vio reducida considerablemente debido a la existencia de la clase `Velocidad`. Fíjese que la clase `CantidadDeMovimiento` hace una llamada al método `GetVx()`, el cual no ha sido definido en ésta clase, sino que ha sido heredado de la clase `Velocidad`. Únicamente con el fin de apreciar la economía se muestra a continuación cómo quedaría definida la clase `CantidadDeMovimiento` si ésta no hubiera heredado las variables y métodos de la clase `Velocidad`:

```
1  class CantidadDeMovimiento {
2   protected:
3    Double_t fM;
4    Double_t fVx, fVy, fPx, fPy;
5   public:
6    CantidadDeMovimiento(Double_t M, Double_t vx, Double_t vy);
7    ~CantidadDeMovimiento();
8    Double_t GetPx() { return fM*fVx; }
9    Double_t GetPy() { return fM*fVy; }
10   Double_t GetVx() { return fVx; }
11   Double_t GetVy() { return fVy; }
12   void SetVx(Double_t Vx) { fVx = Vx; }
13   void SetVy(Double_t Vy) { fVy = Vy; }
14 };
```

Por otro lado, la escritura de éstas dos clases es simple debido a que cada método, (excepto el constructor y el destructor), está definido y declarado en la misma línea. La definición especifica la rutina que ejecuta el método (por ejemplo, línea 8: `{ return fM*fVx; }`) y la declaración establece el tipo de valor devuelto, el nombre del método y sus argumentos (por ejemplo, línea 8: `Double_t GetPx()`). Ahora bien, ¿qué pasa si

el método no se define en una sola línea?. Én este caso podemos agregar un nuevo método que se llame `GetP()`, que devuelva la magnitud (el módulo) de la cantidad de movimiento en la clase `CantidadDeMovimiento`; se tendrá que modificar nuestra definición de la clase `CantidadDeMovimiento`, agregando el método (línea 10) y una nueva variable `fP` (línea 4) que guardará el módulo de $\vec{p}$:

```
1  class CantidadDeMovimiento : Velocidad {
2   protected:
3    Double_t fM; //Variable para la Masa del cuerpo
4    Double_t fPx, fPy, fP;
5   public:
6    CantidadDeMovimiento(Double_t M, Double_t vx, Double_t vy);
7    ~CantidadDeMovimiento();
8    Double_t GetPx() { return fM*GetVx(); }
9    Double_t GetPy() { return fM*GetVy(); }
10   Double_t GetP();
11 };
```

De la anterior definición vemos que la única diferencia del método que se ha agregado es qué éste no contiene la estructura:

```
Tipo NombreMetodo(Argumentos...){ instrucciones; }
```

Esto es porque el cuerpo del método que se encuentra delimitado por los corchetes `{}` se ha omitido, y se ha omitido intencionalmente ya que éste método no será definido en una sola línea, es decir, el método se ha *declarado* más no se ha *definido*. Sin embargo, el cuerpo del método debe implementarse en algún lado, y para ello se utiliza la siguiente estructura:

```
TipoDeMetodo NombreDeClase::NombreDeMetodo(Argumentos...){
  ...
  //Cuerpo
  ...
}
```

Para el caso del ejemplo, la definición del método `GetP()` será:

```
12 Double_t CantidadDeMovimiento::GetP() {
13   Double_t fPSquared;
14   fPSquared = fPx*fPx + fPy*fPy;
15   fP = sqrt(fPSquared);
16   return fP;
17 }
```

La *definición* del método se ha realizado extensa intencionalmente, con el único fin de justificar su no definición en una sola línea. Esta diferencia entre *declaración* y *definición* ha dado lugar a una diferenciación de archivos. Por lo general, se encuentra que en un archivo llamado la cabecera de la clase (*header* en inglés) identificado por su extensión `.h` contiene la declaración de la clase y todos los métodos que pueden ser definidos en una sola línea, y, un archivo que contenga la definición del resto de los métodos, identificado por la extensión `.cxx` (ocasionalmente `.cpp`).

Para considerar la declaración del ejemplo de la clase `CantidadDeMovimiento`, se contará con un archivo llamado `CantidadDeMovimiento.h` cuyo interior será:

```
1  #ifndef CDM
2  #define CDM
3
4  #include "Velocidad.h"
5
6  class CantidadDeMovimiento : Velocidad {
7   protected:
8    Double_t fM; //Variable para la Masa del cuerpo
9    Double_t fPx,fPy,fP;
10   ... //Más variables o Métodos
11  public:
12   Double_t GetPx() { return fM*GetVx() };
13   Double_t GetPy() { return fM*GetVy() };
14   Double_t GetP();
15   ... //Más variables o Métodos
16 };
17 #endif
```

Se han agregado unas instrucciones las cuales comienzan por el signo `#`, conocidas como las directivas del preprocesador (*Preprocessor Directives*[7]). Por ejemplo, `#ifndef CDM` significa que si no esta definido `CDM` (un nombre arbitrario de una variable, `CDM` de Cantidad de Movimiento), defina (`#define CDM`). En este caso `CDM` adquiere un valor nulo pero queda definida. Es importante mencionar que la directiva `#ifndef` debe ir siempre acompañada con la directriz `#endif` para finalizar la estructura condicional. Resumiendo, si no está definida `CDM` (línea 1), defínala (línea 2). Recuerde que la clase `CantidadDeMovimiento` hereda los métodos de la clase `Velocidad` por lo que se incluye la cabecera de ésta clase (mediante `#include "Velocidad.h"` (línea 4), de la línea 9 a la línea 16 se declara la clase `CantidadDeMovimiento`, y finaliza el condicional (línea 17).

Ahora, para considerar la definición de los métodos, se crea un archivo identificado por la extensión `.cxx` (ocasionalmente `.cpp`), en el ejemplo sería `CantidadDeMovimiento.cxx`, el cual contendría:

```
1  #include "CantidadDeMovimiento.h"
2
3  CantidadDeMovimiento::CantidadDeMovimiento
4                  (Double_t M, Double_t vx, Double_t vy){
5    fM = M; fVx = vx; fVy = vy;
6    fPx = fM*fVx; fPy = fM*fVy;
7  }
8  Double_t CantidadDeMovimiento::GetP() {
9    Double_t fPSquared;
10   fPSquared = fPx*fPx + fPy*fPy;
11   fP = sqrt(fPSquared);
12   return fP;
13 }
14 Tipo Clase::OtroMetodo(argumentos...){
15   ...
16   //instrucciones;
17   ...
18 }
```

Fíjese que en la línea 1 se realiza la inclusión de la cabecera de la clase misma. De la línea 3 a la línea 13 se ha definido el constructor, con el fin de mostrar la función básica

[7]Para mayor información consulte `http://www.cplusplus.com/doc/tutorial/preprocessor/`

del mismo: inicializar (i.e definir por primera vez) el valor de las variables de la clase. Se hace hincapié en que no es absolutamente necesario realizar la separación de archivos en cabecera (que contiene la declaración de la clase) y archivo de definición (el cual contiene la implementación de los métodos), es posible declarar y definir inclusive varias clases en un mismo archivo de manera secuencial siempre que se esté trabajando mediante el interprete; sin embargo, ésta separación es necesaria para el proceso de compilación (abordado en el apéndice C), es decir, si trabajamos mediante el interprete, se podría tener un archivo para realizar la declaración y la definición de la clase:

```
#ifndef CDM
#define CDM

#include "Velocidad.h"

class CantidadDeMovimiento : Velocidad {
 protected:
  Double_t fM; //Variable para la Masa del cuerpo
  Double_t fPx,fPy,fP;
  ... //M\'as variables o M\'etodos
 public:
  Double_t GetPx() { return fM*GetVx() };
  Double_t GetPy() { return fM*GetVy() };
  Double_t GetP();
  ... //M\'as variables o M\'etodos
};
#endif
CantidadDeMovimiento::CantidadDeMovimiento
                (Double_t M, Double_t vx, Double_t vy){
  fM = M; fVx = vx; fVy = vy;
  fPx = fM*fVx; fPy = fM*fVy;
}
Double_t CantidadDeMovimiento::GetP() {
  Double_t fPSquared;
  fPSquared = fPx*fPx + fPy*fPy;
  fP = sqrt(fPSquared);
  return fP;
}
Tipo Clase::OtroMetodo(argumentos...){
  ...
  //instrucciones;
  ...
}
```

1.4.2. Creación de instancias y llamado de métodos

En la sección anterior se ha visto la estructura básica de la declaración y la definición de clases, recuerde que las clases declaradas no han sido utilizadas, esto es, se han definido sus características, se ha definido la clase, sus variables y sus métodos; no obstante, no se han creado instancias de éstas y no se han utilizado objetos particulares que correspondan a dicha clase.

Al igual que cualquier variable, para utilizar una instancia de una clase es necesario dos procesos: un proceso de declaración y uno de asignación o creación[8]. Como se

[8]Cuando se hace uso del interprete que incorpora ROOT, se puede crear el objeto directamente sin

mencionó anteriormente, en la declaración se especifica que tipo de clase se va a utilizar y el nombre que tendrá el objeto. Un ejemplo de ello se presenta en la figura 1.4, donde se realiza la declaración de un objeto de tipo `TCanvas` cuyo nombre es `c1`.

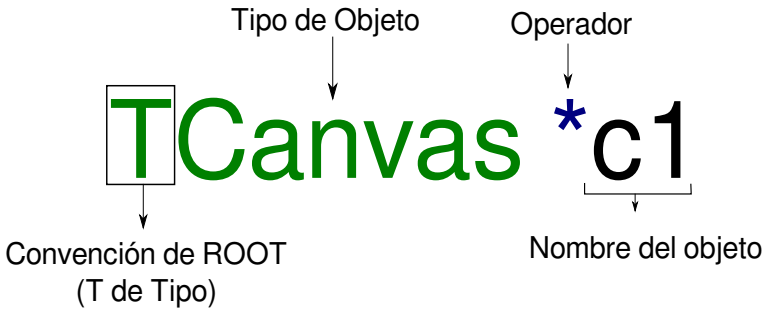


Figura 1.4: Ejemplo de la estructura utilizada para la declaración de una instancia.

Ahora, en la asignación se hace un llamado al constructor de la clase, el cual inicializa al objeto mediante la asignación de valores específicos a las variables que lo definen. Por ejemplo, la figura 1.5, se presenta la instrucción ejecutada para realizar la creación una *instancia* de la clase `TCanvas` cuyo nombre es `c1`. Ya que no se le han asignado argumentos al constructor, `c1` ha sido creado con los valores asignados por defecto. A éste constructor (que no contiene argumentos) se le conoce como el constructor por defecto (*default constructor*).

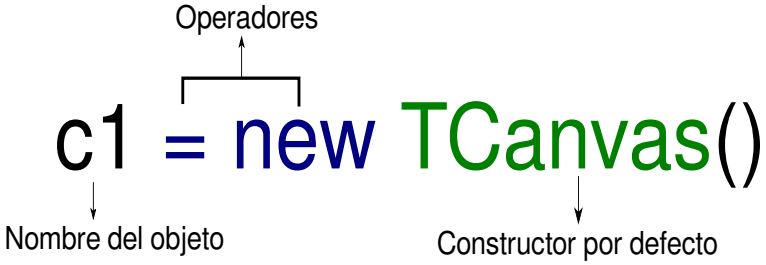


Figura 1.5: Ejemplo de la estructura de la creación de una instancia utilizando el constructor por defecto

Cuando se desea asignar valores específicos a esas variables mediante el constructor, se envían esos valores de acuerdo a su *sintaxis*[9]; es decir, respetando el orden en el valor que corresponde a cada variable. Por ejemplo, la figura 1.6, crea una instancia de la clase `TCanvas` llamada `c1`, donde el primer argumento es el nombre de la instancia; el segundo, el título de la ventana del lienzo; y los últimos dos argumentos corresponden a las dimensiones (ancho*alto) del lienzo en píxeles.

haber realizado su declaración.

[9]Recuerde que la sintaxis hace referencia al orden en que debe ser introducidos los argumentos de un método.

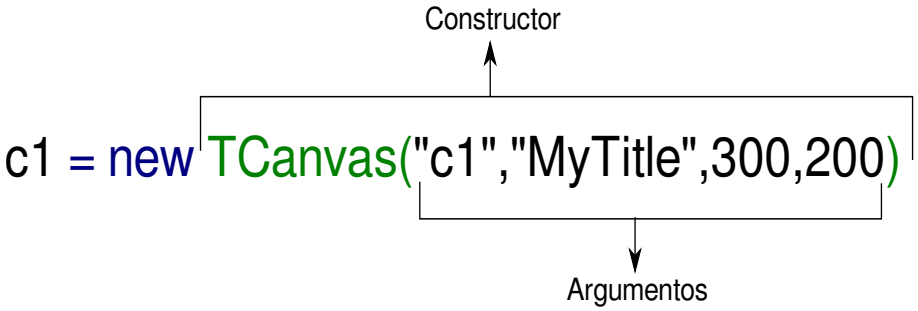


Figura 1.6: Ejemplo de la creación de una instancia a través de un constructor que contiene argumentos.

Es posible realizar tanto la declaración como la asignación de la instancia en una misma línea de código utilizando la estructura utilizada en la figura 1.7, note que allí se ha utilizado el constructor por defecto. Sin embargo, puede introducir los argumentos necesarios para configurar la instancia en el constructor respetando la sintaxis. Note que es necesario tener en cuenta la globalidad de los objetos; si realiza la declaración y la asignación en una misma línea deberá asumir que sólo puede manipular el objeto (i.e hacer llamados a sus métodos) en el interior de la función en la cual se ejecuta la línea.

```
TCanvas *c1 = new TCanvas()
```

Figura 1.7: Ejemplo de la declaración y creación de una instancia realizadas en una misma línea

Una vez ha sido definida la instancia (`c1` en los ejemplos), es posible utilizar los métodos de la clase, conocidos en el argot de C++ como funciones miembro *(Function Members)*, para esto es necesario tener presente la *sintaxis* de los métodos. La sintaxis para cada uno de los métodos de las clases de ROOT puede ser consultada en la documentación de referencia presentada en la sección 1.3.1. Por ejemplo, si se ejecuta la siguiente sentencia mostrada en la figura 1.8. Se estará ejecutando el método `SetWindowSize()`, el cual está definido en la clase `TCanvas` (debido a que c1 es una *instancia* de ésta clase) y hemos pasado a éste método 2 argumentos: el ancho y la altura del lienzo en píxeles (variables de tipo entero).

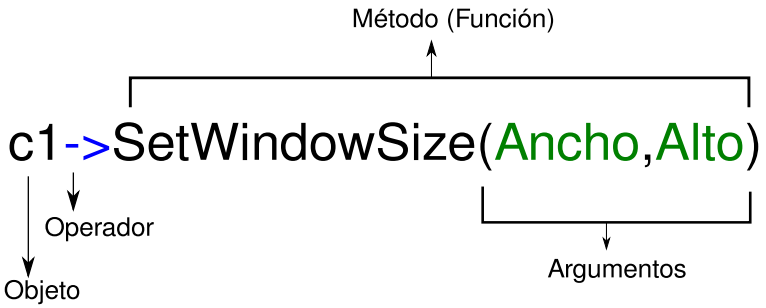


Figura 1.8: Ejemplo que muestra los elementos de la ejecución de métodos públicos a través de una instancia

Éste método es de tipo `public` ya que estamos llamando el método desde fuera de la definición de la *clase*. La función de `SetWindowSize()` es modificar el tamaño del lienzo, el cual ha sido asignado en el constructor (o el valor que ha sido asignado por defecto en el caso de haber invocado el constructor por defecto).

Ahora, si intentáramos ejecutar un método declarado de tipo `private` en la clase, como lo es `CreatePainter()`, mediante la sentencia `c1->CreatePainter()`, obtendremos un error: `can not call private or protected function`, debido a que el método sólo puede ser utilizado por los procedimientos internos de la clase.

En este capítulo se expusieron las características básicas de las sentencias que serán utilizadas a lo largo del texto, así como la definición de conceptos y la terminología utilizada en la programación orientada a objetos: el modo en que se implementa una clase, lo que representa una instancia u objeto y para qué nos sirve la herencia en programación.

En el próximo capítulo estudiaremos cómo se implementan y configuran los diferentes elementos que componen una interfaz gráfica.

Capítulo 2

Diseño e implementación de interfaces gráficas de usuario

> "But how do we ensure that we design the interface in the best way possible? the short answer is that we do not yet know!."
>
> (Smith, 1997)

2.1. Aspectos a tener en cuenta

Antes de realizar la implementación de una interfaz, resulta necesario realizar un diseño y un esquema que nos permita visualizar la interfaz, tanto el aspecto gráfico con el modo en que funcionará. Este procedimiento permitirá clarificar y simplificar el proceso de implementación y escritura de código.

Para llevar a cabo el diseño de una interfaz gráfica para la simulación de un fenómeno físico, deben tenerse en cuenta las siguientes preguntas:

- **¿Cuál es la necesidad que suple el desarrollo de la interfaz?** Como se mencionó en la introducción, toda pieza de software surge como respuesta a un problema: el software es el modo de satisfacer una necesidad. Antes de llevar a cabo cualquier implementación es necesario preguntarse qué se pretende con ella. En el caso de las simulaciones, una necesidad puede ser la visualización de la evolución de una variable o un conjunto de variables, no sólo mediante diagramas sino también a través de representaciones (por ejemplo, un vector representado a través de un segmento de recta dirigido, cuando requiera estudiar el comportamiento de un campo vectorial). También es posible que lo que se requiera sea un conjunto de datos que permitan realizar el contraste de una práctica experimental con los resultados obtenidos mediante el modelo matemático, para llevar a cabo el

análisis de elementos semejantes y discrepantes.
El planteamiento de la necesidad implica establecer el modelo que será estudiado; en el caso que éste modelo se encuentre planteado en términos de ecuaciones diferenciales es necesario proponer, si es el caso, un método numérico a través del cual realizar el cálculo. Si la ecuación diferencial tiene solución analítica, puede utilizar la solución en vez de resolver numéricamente la ecuación diferencial (realizando evaluaciones sucesivas de la solución analítica para diferentes valores de la(s) variable(s) dependientes).

- **¿Qué parámetros de la simulación puede establecer el usuario?** La implementación de simulaciones requiere por lo general un gran número de variables. Por ejemplo, la solución numérica de ecuaciones diferenciales a través de la familia de métodos *Runge-Kutta,* requiere la reserva en memoria de los coeficientes de la tabla de Buttcher; sin embargo, estos coeficientes harán parte de los procedimientos internos (que por su naturaleza no es necesario que el usuario los establezca). No obstante, condiciones de proceso (como el intervalo o "paso" de los métodos numéricos o el mismo método numérico a utilizar en la simulación) o las condiciones iniciales del problema pueden ser establecidas por el usuario. A éste tipo de variables (las cuales puede fijar el usuario) se les reconoce como parámetros de la simulación, por lo que, una vez establecido el fenómeno a simular, se realiza una lista de los parámetros que el usuario podrá fijar a través de la interfaz gráfica.

- **¿Qué características tienen estos parámetros?** Luego de haber establecido todas las variables que pueden ser establecidas a través de la interfaz, es necesario enumerar las características de cada variable; si es un número, ¿en qué intervalo puede encontrarse?. Por ejemplo, para la introducción de un valor de la masa de un cuerpo, nos interesará que el método de introducción o el objeto utilizado en la interfaz impida que el usuario asigne una cantidad negativa a ese valor. Es necesario indicar la cantidad de cifras decimales que se requieren y un valor por defecto (el valor que tendrá la variable al iniciar el programa).

- **¿Existe una relación estrecha entre los parámetros que permita agruparlos?** Cuando se trata de parámetros, es posible agrupar determinado conjunto de variables. Por ejemplo, si se va a simular un choque entre dos o más discos, es posible agrupar la masa, la velocidad y la posición inicial de cada disco.

- **¿Qué objeto de la infraestructura es el más adecuado para el ingreso del valor del parámetro por parte del usuario?** Cualquier infraestructura de desarrollo de software tiene una cantidad limitada de objetos para desarrollar la interfaz gráfica; se encontrarán botones, cajas de texto, listas de selección, etc[1]. Es necesario proponer, para cada uno de los parámetros que se podrán fijar a través de

[1] Observe la sección "*Widgets in Detail*" de la Guía del Usuario de ROOT (ROOT-Team, s.f.-b) para un completo listado de éstos elementos

la interfaz gráfica, un posible método de introducción. Por ejemplo, uno de los parámetros en una simulación de un cuerpo que cae libremente, un parámetro es la aceleración gravitacional del planeta en el que ocurrirá el fenómeno; el desarrollador puede optar por usar una entrada numérica que le permita al usuario introducir un número, o una lista de selección donde el usuario pueda escoger el valor de una lista predefinida como la aceleración gravitacional de los planetas.

- **¿Incorporará la interfaz un mecanismo de visualización?** En el diseño de la interfaz, es necesario definir si ésta incorporará un mecanismo de visualización de la información que se está procesando. La interfaz de la simulación puede tener un espacio donde se muestra i.e, una animación del fenómeno, o un espacio donde de diagramas de fase o energía. Es necesario entonces definir qué elementos de visualización tendrá la interfaz.

A partir de éstas preguntas, el diseño del esquema de la interfaz permitirá ubicar cada elemento (botones, lienzos, entradas numéricas, etc) en la ventana. Si la simulación requiere de varias ventanas, deberá realizar un esquema para cada una. Se recomienda hacer un bosquejo de la interfaz, como se esquematiza en la figura 2.1.

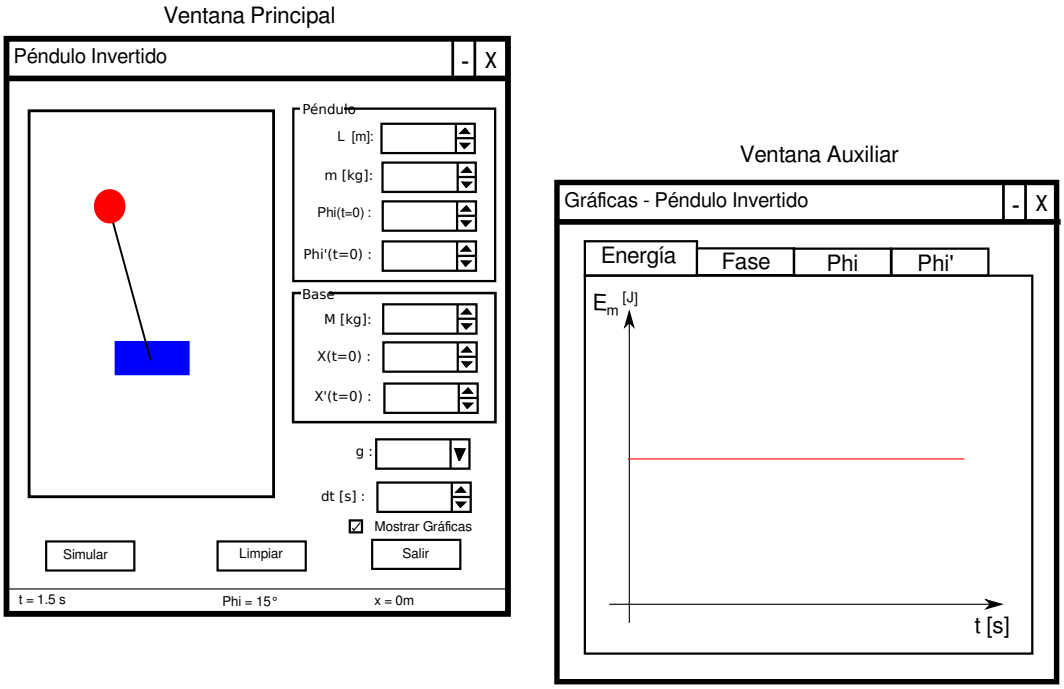


Figura 2.1: Esquema de una interfaz gráfica de usuario para la simulación del péndulo invertido

- **¿Cuál es el comportamiento de la interfaz gráfica?** Luego de realizar el esquema gráfico, es necesario describir en palabras el funcionamiento de la interfaz: indicar qué acción se realiza cuando se modifica un elemento de la interfaz; por ejemplo, qué se espera que realice el programa cuando se pulsa un botón determinado,

cuando se introduce un número, cuando se cierra una ventana. Es importante prevenir que la respuesta a este último (cerrar la ventana) puede no ser tan simple: Imagine que la simulación permite exportar un conjunto de datos; en este caso, puede interesarle que una vez pulsado el botón cerrar se muestre un cuadro de diálogo que le pregunte al usuario si desea o no guardar los datos. Esta acción debe verificar que efectivamente existen datos para guardar; de otro modo, no tendría sentido realizar la pregunta.
Este último paso del diseño, de describir el comportamiento de la interfaz, no sólo establece un objetivo, sino que también permite aclarar la forma en que debe ser implementada la interfaz. Puede utilizar un diagrama de flujo en donde plantee de manera general el funcionamiento del programa.

Un aspecto importante en el diseño y la implementación de algoritmos es la modularidad: realizar la separación de tareas complejas en procedimientos cortos e interdependientes facilita no sólo la lectura del código sino su modificación, adaptación a nuevos problemas y el mantenimiento. Aruliah y cols. (2012) presentan una serie de útiles prácticas a la hora de escribir *software*.

2.2. La ventana principal

La ventana principal, que debe ser una instancia de la clase `TGMainFrame`, actúa como un contenedor en cuyo interior se agrupan otros objetos que componen la interfaz. Por defecto, la ventana principal, es un contenedor que agrupa en orden vertical los objetos que se van añadiendo i.e, si se han creado dos botones y luego se añaden a la ventana, estos aparecerán uno tras otro verticalmente.

Cada objeto de la interfaz gráfica está contenido en el interior de otro (embebido); incluso, la ventana principal está embebida en un espacio proporcionado por el servidor gráfico del sistema operativo: Ésta relación de objeto embebido – objeto que embebe, se conoce como relación padre-hijo (*parent-child* en inglés). Suponga que creó un botón y quiere embeberlo en la ventana principal. En éste caso, debe entenderse que la ventana principal es el padre del botón y el botón es el hijo de la ventana principal. Esta distinción es importante debido a que cualquier movimiento o posicionamiento de un objeto hijo está dado en relación al objeto padre y no necesariamente a la ventana principal.

Se aclara que si existen ventanas auxiliares, hijas de la ventana principal, como la mostrada en la figura 2.1, es posible crearlas a través de las clase `TGTransientFrame`. Los métodos de estas clases, en general, coinciden con los métodos de `TGMainFrame`.

2.2.1. Creación y configuración de la ventana principal

El constructor de la clase `TGMainFrame` de acuerdo a la documentación de referencia está definido como:

```
TGMainFrame(const TGWindow* p = 0, UInt_t w = 1, UInt_t h = 1, UInt_t options = kVerticalFrame)
```

donde, el primer argumento corresponde al padre (`p` de *parent*); el segundo, al ancho dado en píxeles y de tipo entero (`w` de *width*), el siguiente, la altura en píxeles (de tipo entero, `h` de *high*), y el último (`options`) hace referencia a la organización de los hijos en el interior de la ventana. Por defecto, como se mencionó anteriormente, los hijos se ubicarán de manera vertical (`kVerticalFrame`)[2]. Es posible cambiar éste valor según las opciones descritas en `http://root.cern.ch/root/html/src/TGFrame.h.html#70`. Es necesario tener presente que ésta ubicación o patrón de ubicación sólo se aplicará a los hijos de la ventana principal. Se puede ver que todos los argumentos del constructor tienen parámetros por defecto; por lo tanto, si ejecutáramos la instrucción:

```
fMain = new TGMainFrame();
```

se crearía una instancia de la clase `TGMainFrame` llamada `fMain`, cuyo tamaño es `1px * 1px` y que organizará a sus hijos de manera vertical. Es posible aumentar la eficiencia del código si se establecen los parámetros en el constructor directamente, en vez de crear una instancia con los atributos por defecto y luego, a través de sus métodos, cambiar cada atributo. Por ejemplo,

```
root[] fMain = new TGMainFrame(gClient->GetRoot(),400,300,kVerticalFrame);
```

el objeto `fMain` corresponderá a una ventana de $400 * 300$ píxeles, donde los elementos se agruparán verticalmente. En la ventana principal el padre es proporcionado por el servidor gráfico del sistema operativo, lo que corresponde al primer argumento `gClient->GetRoot()`. Fíjese que las dimensiones de la ventana principal permiten que ésta pueda ser observada sin dificultad en la pantalla, mientras que si se utiliza el constructor por defecto (donde las dimensiones de la ventana son de `1px * 1px`), la ventana no podría ser visualizada en pantalla con facilidad.

Para establecer el título de la ventana se utiliza el método `SetWindowName()`:

```
root[] fMain->SetWindowName(``T\355tulo de la ventana'');
```

El título asignado es entonces "Título de la ventana". Para caracteres especiales es necesario utilizar la representación correspondiente en el sistema octal (ver equivalencias en `http://www.programasprogramacion.com/caracteres.php`). Para este caso, la i tildada de la palabra "Título" está representada en el sistema octal por el número 355.

Una característica importante de la ventana principal es el icono, el cual es establecido a través del método `SetIconPixmap()`:

[2]La opción `kHorizontalFrame` permitirá agregar los elementos de manera secuencial.

```
root[] fMain->SetIconPixmap(``$ROOTSYS/icons/info.gif'');
```

donde es posible ver que el argumento de éste método es la ruta del archivo que contiene el icono. Pueden utilizarse diversos formatos para la imagen del icono tal como se definen en `http://root.cern.ch/root/html/TImage.html#datamembers`.

Para mostrar el resultado de las instrucciones anteriores se ejecuta la instrucción:

```
root[] fMain->MapWindow();
```

lo que hace visible la ventana principal. Es necesario ejecutarlo luego de haber configurado todos los elementos de la interfaz. El resultado de la ejecución de las instrucciones anteriores se muestran en la figura 2.2.

Figura 2.2: Resultado de la creación y configuración de la ventana principal

Si se necesita cambiar el tamaño de la ventana, se ejecuta el método `SetWMSizeHints`:

```
root[] fMain->SetWMSizeHints(500,500,800,600,10,10);
```

Éste método permite establecer la medida tanto mínima como máxima de la ventana. Los primeros dos argumentos proporcionan la medida mínima de la ventana en píxeles (ancho*alto), los dos siguientes, la medida máxima que puede adquirir la ventana, y los dos últimos proporcionan el paso en el que es incrementado el tamaño de la ventana cuando se cambia su tamaño, arrastrando con el puntero alguno de sus bordes. Si se establece que el tamaño mínimo es equivalente al tamaño máximo, el botón de maximizar desaparecerá. La instrucción anterior modifica la ventana de tal manera que y tamaño mínimo es $500 * 500$ píxeles y su tamaño máximo es $800 * 600$ píxeles.

De este modo se ha creado y configurado la ventana principal para una aplicación. Es importante resaltar que el proceso se compone de la creación de una instancia y la configuración/modificación de sus propiedades.

2.3. Organización y estructura de la interfaz gráfica

Una vez configurada la ventana principal, es necesario crear y organizar los demás objetos (como botones, lienzos, entradas de texto, etc) en su interior. Hay dos métodos

para llevar a cabo este procedimiento: el primero es especificando la coordenada en la que será ubicado el objeto, y el otro consiste en la creación de "indicaciones de diseño" que organicen de manera automática los objetos a partir de opciones especificadas por el programador.

Para ubicar los objetos en el interior de la ventana principal especificando la coordenada, se debe ejecutar en la ventana principal (o en el padre correspondiente) el método `SetLayoutBroken()`. Como en el ejemplo ilustrado a continuación.

```
1  {
2    //Se crea y configura la ventana principal
3    fMain = new TGMainFrame(gClient->GetRoot(),120,100,kVerticalFrame);
4    fMain->SetWMSizeHints(120,110,120,100,0,0);
5    fMain->SetWindowName("Ejemplo SetLayoutBroken");
6    //Se ejecuta el método SetLayoutBroken
7    fMain->SetLayoutBroken(kTRUE);
8
9    //Se crea un botón de texto
10   fButton = new TGTextButton(fMain,"SetLayoutBroken");
11   //Se modifica su tamaño y se mueve
12   fButton->Resize(100,20);
13   fButton->Move(10,70);
14
15   //Se muestran los resultados
16   fMain->AddFrame(fButton);
17   fMain->MapSubwindows();
18   fMain->MapWindow();
19 }
```

Note que el método `Move(int x, int y)` permite ubicar de manera arbitraria cada uno de los objetos. Por ejemplo: `fButton->Move(10,70);`, ubica el objeto `fButton` 10 píxeles a la derecha y 70 píxeles hacia abajo de la esquina superior izquierda de la ventana. Tenga en cuenta que: el origen del sistema coordenado se encuentra en la esquina superior izquierda de la ventana principal (o del objeto padre que contendrá el elemento que se quiere ubicar) y, la dirección de crecimiento es de izquierda a derecha en la coordenada horizontal y de arriba hacia a abajo en la coordenada vertical. El resultado se muestra en la figura 2.3.

Figura 2.3: Resultado de la implementación ubicando elementos en la interfaz ejecutando el método `SetLayoutBroken` en la ventana principal.

Éste modo de ubicación de los objetos en la interfaz no es el más adecuado en la mayoría de las situaciones, ya que si se tiene una ventana principal que puede cambiar de tamaño (por ejemplo se puede maximizar), los objetos no se adecuarían al nuevo tamaño, debido a que se les ha dado instrucciones específicas acerca de su ubicación. Este método tampoco es el más cómodo debido a se requiere manipular

manualmente la ubicación de los objetos, por lo que se recomienda el uso de la clase `TGLayoutHints()`, la cual permite ubicar de manera automática los objetos dando parámetros de ubicación.

2.3.1. Ubicación y adecuación a través de `TGLayoutHints`

Otro método para manejar la ubicación y el tamaño de los objetos en la interfaz gráfica es a través de la clase `TGLayoutHints`, la cual permite seleccionar alineaciones como centrar horizontal o verticalmente e inclusive expandir un objeto. Tenga en cuenta que la ubicación y las opciones de centrado/expansión son en relación al padre del objeto.

Inicialmente, se crea un objeto del tipo `TGLayoutHints` donde se especifican las opciones que se le dan a un objeto. El constructor de la clase `TGLayoutHints` tiene la siguiente sintaxis:

```
TGLayoutHints(ULong_t hints = kLHintsNormal, Int_t padleft = 0, Int_t padright = 0, Int_t padtop = 0, Int_t p
```

El primer argumento representa el tipo de alineación o la opción de adecuación:

- `kLHintsExpandX`: Expande horizontalmente el objeto hijo en el objeto padre.
- `kLHintsExpandY`: Expande verticalmente el objeto hijo en el objeto padre.
- `kLHintsCenterX` : Centra horizontalmente el objeto hijo en el objeto padre.
- `kLHintsCenterY` : Centra verticalmente el objeto hijo en el objeto padre.

Los siguientes cuatro argumentos del constructor de la clase `TGLayoutHints`, representan las márgenes dadas en píxeles en el siguiente orden: izquierda, derecha, superior, e inferior. Las instancias de tipo `TGLayoutHints`, tal como su nombre sugiere, crean indicaciones de diseño que son útiles cuando se desea que un objeto se adecue a las características de su padre; por ejemplo, cuando se maximiza, o cuando se cambia el tamaño de una ventana, tanto la posición como el tamaño de los objetos debe cambiar de tal manera que se adecuen al cambio.

Para ilustrar lo anterior se realizará a continuación se presenta un primer ejemplo en donde se crea un botón, el cual se expande en el interior de una ventana principal:

Inicialmente se crea la ventana principal:

```
//Se crea y configura una ventana principal
root[] fMain = new TGMainFrame();
root[] fMain->SetWMSizeHints(200,50,400,100,10,10);
```

Aquí, la ventana principal `fMain` tiene un tamaño inicial (y mínimo) de $200 * 50$ píxeles, y puede extenderse hasta un tamaño de $400 * 100$ píxeles. Seguidamente, se crea un botón que contenga el texto "Objeto Expandido":

```
root[] fButton = new TGTextButton(fMain,''Objeto Expandido'');
```

Luego se crea una indicación de diseño, esto es, una instancia de la clase `TGLayoutHints`, la cual se llamará `Expandir`:

```
root[] Expandir = new TGLayoutHints(kLHintsExpandX|kLHintsExpandY,5,5,5,5);
```

Fíjese que se han introducido de manera simultánea instrucciones para expandir, tanto horizontal como verticalmente a través de la barra vertical `|`, también se establece que las márgenes son de 5 píxeles en cada extremo.

Una vez definida la indicación de diseño (`Expandir`), es necesario a través del método `AddFrame()` del objeto padre (`fMain`), incluir el botón `fButton` en la ventana principal:

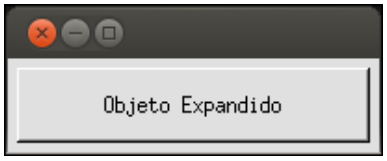
```
root[] fMain->AddFrame(fButton,Expandir);
```

de esta manera, se establece que el botón `fButton` cuyo padre es el objeto `fMain` se ajustará a la ventana principal, utilizando la indicación de diseño `Expandir`; así, no importa si la ventana principal (el objeto `fMain`) cambia de tamaño, el botón siempre se expandirá tanto horizontal como verticalmente guardando márgenes de 5 píxeles en cada extremo. Finalmente, se ejecutan los métodos necesarios para mostrar los resultados:

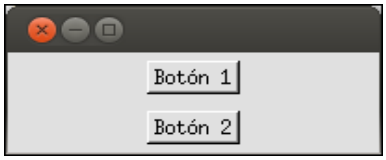
```
//Se mapea los objetos de la ventana
root[] fMain->MapSubwindows();
//Se activa el mecanismo de las indicaciones de diseño
root[] fMain->Resize();
//Se muestra la ventana
root[] fMain->MapWindow();
```

Se debe notar que es necesario ejecutar el método `Resize()` de la ventana principal Éste método permite inicializar las indicaciones de diseño de tal forma que puedan ser utilizadas. Siempre que se haga uso de instancias `TGLayoutHints`, se debe invocar el método `Resize()` en la ventana principal (sin argumentos).

El resultado de la ejecución de las instrucciones anteriores que representan este primer ejemplo se visualiza en la figura 2.4a.



(a) Un botón ha sido expandido en su padre



(b) Se han centrado dos botones a través de la misma indicación de diseño

Figura 2.4: Ejemplo de usos de instancias de la clase TGLayoutHints

Una de las características de las indicaciones de diseño es que es posible reutilizarlas. Para ilustrar esto se propone el siguiente ejemplo, en el cual se crean dos botones centrados y en secuencia vertical, utilizando la misma indicación de diseño:

```
{

  //Se crea y configura una ventana principal
  fMain = new TGMainFrame();
  fMain->SetWMSizeHints(500,150,800,150,10,10);

  //Se cre un botón
  fButton1 = new TGTextButton(fMain,"Ejemplo 1");
  fButton2 = new TGTextButton(fMain,"Ejemplo 2");

  //Se crea una indicación de diseño
  Centrar = new TGLayoutHints(kLHintsCenterX,5,5,5,5);

  //Se agrega el botón y se establece
  //la indicación de diseño que usa.
  fMain->AddFrame(fButton1,Centrar);
  fMain->AddFrame(fButton2,Centrar);

  //Se mapea los objetos de la ventana
  fMain->MapSubwindows();
  fMain->Resize();
  //Se muestra la ventana
  fMain->MapWindow();

}
```

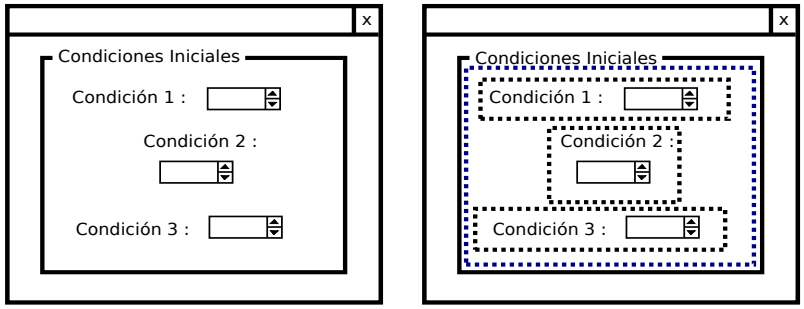
Aquí se muestran dos botones centrados horizontalmente en una ventana y ubicados secuencialmente de manera vertical (recuerde que la ubicación por defecto es vertical).

El resultado se muestra en la figura 2.4b.

Si se desea que estos objetos sigan centrados, pero que sean continuos horizontalmente se reemplaza la línea `fMain = new TGMainFrame();` por `fMain = new TGMainFrame(gClient->GetRoot(),500,500,kHorizontalFrame);` donde se ha alterado el valor por defecto, para permitir que los objetos, que se van añadiendo a `fMain` lo hagan horizontalmente.

2.3.2. Medios y jerarquía de agrupación en la interfaz gráfica

Como se mencionó anteriormente, pueden existir diferentes padres, no es necesario que el padre de cada elemento sea la ventana principal. En ocasiones, resulta conveniente realizar una jerarquía de padres para organizar los objetos en la interfaz. Si se necesita agrupar unas variables que tienen características comunes, por ejemplo, las condiciones iniciales de un experimento que serán agrupadas como se muestra en la figura 2.5a.



(a) Agrupación de Objetos (b) Marcos de agrupación

Figura 2.5: Agrupación y jerarquía de objetos

Es necesario observar que hay varios elementos: existen etiquetas con un texto (`TGLabel`), entradas numéricas (`TGNumberEntry`), un marco que agrupa a todos estos elementos en su interior (`TGGroupFrame`) y que tiene el título "Condiciones Iniciales". Para la disposición que se muestra en la figura 2.5a se utilizaron diferentes objetos. El primer objeto corresponde marco delineado cuyo título es "Condiciones Iniciales" mediante una instancia de `TGGroupFrame`. Dentro de este se crea un marco, línea punteada en azul en la figura 2.5b, y dentro de éste, otros 3 marcos organizados verticalmente (línea negra). Dentro de éstos tres marcos se crean dos objetos que corresponden a una etiqueta y una entrada numérica, En el primero, la disposición de los objetos es horizontalmente, en el segundo, los objetos se disponen de manera vertical, y en el tercero, se disponen nuevamente de manera horizontal (ver figura 2.5b).

Fíjese que la figura 2.5b permite identificar la jerarquía de padres: la ventana principal es el padre de aquel objeto cuyo título es "Condiciones Iniciales", a su vez, éste es padre del marco delineado en azul. Éste último es padre de los otros 3 marcos delineados en negro, y cada marco delineado en negro tiene dos hijos: una etiqueta y una entrada numérica.

Para implementar el diseño de la interfaz gráfica descrita anteriormente, se debe configurar una ventana principal:

```
root[] fMain = new TGMainFrame();
root[] fMain->SetWMSizeHints(200,150,400,300,10,10);
```

Luego se crea el hijo de la ventana principal que corresponde al marco que agrupa los objetos que permiten introducir las condiciones iniciales:

```
root[] GFCondIni = new TGGroupFrame(fMain,''Condiciones Iniciales'');
```

donde el primer argumento es el padre (la ventana principal, `fMain`), y el segundo, el título del marco (Condiciones iniciales). Seguidamente como se indica en la figura 2.5b se crea el marco delineado en azul que albergará a sus hijos de manera vertical. Para esto, se crea una instancia de la clase `TGVerticalFrame`:

```
root[] VFM = new TGVerticalFrame(GFCondIni);
```

donde el argumento que ha sido dado en el constructor de la clase es un apuntador al padre del objeto, que en este caso es `GFCondIni`.

Continuando con la implementación, sigue un marco que agrupa horizontalmente la primera etiqueta, con el texto "Condición 1", y la primer entrada numérica. Éste marco es una instancia de la clase `TGHorizontalFrame`:

```
root[] MarcoH1 = new TGHorizontalFrame(VFM);
```

Nuevamente, se ha pasado como argumento al constructor el apuntador al padre del objeto, llamado `MarcoH1` cuyo padre es el objeto `VFM` (delineado en azul en la figura 2.5b).

Dentro de `MarcoH1` se crean dos objetos: una etiqueta y una entrada numérica. La etiqueta es una instancia de la clase `TGLabel`:

```
root[] Etiqueta1 = new TGLabel(MarcoH1,''Condici\363n 1 :'');
```

donde los argumentos del constructor corresponden al padre de la etiqueta al texto que lo identificará, respectivamente. La entrada numérica es una instancia de la clase `TGNumberEntry`:

```
root[] Entrada1 = new TGNumberEntry(MarcoH1);
```

donde el argumento que se ha pasado al constructor de la clase `TGNumberEntry` corresponde al padre de la entrada numérica.

Una vez creados el padre `MarcoH1` y sus respectivos hijos `Etiqueta1`, `Entrada1`, estos objetos son agregados a su padre mediante el método `AddFrame()`[3]:

```
root[] MarcoH1->AddFrame(Etiqueta1);
root[] MarcoH1->AddFrame(Entrada1);
```

El orden de adición es importante ya que determinará el orden en el que se ubican los hijos en el padre. En este caso, primero irá la etiqueta y luego la entrada numérica.

Para el segundo marco `MarcoV2`, que representa las Condición inicial 2 se sigue la implementación que se utilizó para el primer marco `MarcoH1`, con la diferencia que se usa una instancia de la clase `TGVerticalFrame`:

```
root[] MarcoV2 = new TGVerticalFrame(VFM);
root[] Etiqueta2 = new TGLabel(MarcoV2,''Condi\363n 2 : '');
root[] Entrada2 = new TGNumberEntry(MarcoV2);
root[] MarcoV2->AddFrame(Etiqueta2);
root[] MarcoV2->AddFrame(Entrada2);
```

Éste marco, agrupa a sus hijos de manera vertical.

El último marco,`MarcoH3`, que corresponde a la condición 3, agrupa a sus hijos de manera horizontal:

```
root[] MarcoH3 = new TGHorizontalFrame(VFM);
root[] Etiqueta3 = new TGLabel(MarcoH3,''Condici\363n 3 : '');
root[] Entrada3 = new TGNumberEntry(MarcoH3);
root[] MarcoH3->AddFrame(Etiqueta3);
root[] MarcoH3->AddFrame(Entrada3);
```

[3] `AddFrame()` añade el hijo (especificado en su argumento) al padre.

En este punto, es importante configurar las indicaciones de diseño, creando dos instancias de `TGLayoutHints`, una permitirá centrar los objetos (`Centrar`) y la otra permitirá expandir objetos `Expandir`, permitiendo ambas configurar las márgenes respectivas:

```
root[] Centrar = new TGLayoutHints(kLHintsCenterX|kLHintsCenterY,5,5,5,5);
root[] Expandir = new TGLayoutHints(kLHintsExpandX|kLHintsExpandY,10,10,10,10);
```

El objeto `Centrar` se utiliza para ubicar a los marcos `MarcoH1`,`MarcoV2`,y `MarcoH3` dentro de su padre (el objeto `VFM`):

```
root[] VFM->AddFrame(MarcoH1,Centrar);
root[] VFM->AddFrame(MarcoV2,Centrar);
root[] VFM->AddFrame(MarcoH3,Centrar);
```

y el objeto `Expandir`, se utilizará para expandir el objeto `VFM` en su padre `GFCondIni`:

```
root[] GFCondIni->AddFrame(VFM,Expandir);
```

Se agrega `GFCondIni` a su padre (`fMain`) :

```
root[] fMain->AddFrame(GFCondIni,Expandir);
```

Se ejecutan los métodos necesarios para activar las indicaciones de diseño y para mostrar la ventana:

```
root[] fMain->MapSubwindows();
root[] fMain->Resize();
root[] fMain->MapWindow();
```

En ésta sección se aprecia el valor que tiene el esquema requerido en el diseño de la interfaz, éste simple dibujo, clarifica y rige la implementación, diciéndole al programador la jerarquía de los padres, lo que a su vez ahorra tiempo en el momento de la implementación.

2.4. La visualización

En la simulación es posible disponer de un espacio en la interfaz gráfica, donde se visualice la evolución de determinadas variables del proceso. Se pueden crear gráficas de la relación entre determinadas variables o también es posible disponer de una animación donde en un espacio (el lienzo) se aprecie la evolución de las características del mismo (Por ejemplo, cuando se presenta una dilatación o contracción en las dimensiones de un cuerpo debido a la variación de su temperatura, éste cambio en las dimensiones puede ser representado en una animación). Este tipo de visualización requiere de control, es decir, no resulta conveniente realizarlas en un ciclo convencional, como un `while` o un `for`, debido a que estos ciclos truncan la ejecución del software hasta que las condiciones para finalizar el ciclo se cumplan, es decir, la ejecución de éstos ciclos bloquearán la posibilidad de ejecutar otras instrucciones en el programa

mientras los ciclos finalizan; el programa permanecerá ocupado hasta que termine los ciclos, lo cual no es conveniente en una interfaz gráfica ya que debe permitirse al usuario pausar/parar la simulación cuando éste así lo considere (es decir, el usuario puede ejecutar instrucciones sin necesidad de que la simulación halla terminado), y además, el tiempo de ejecución de estos ciclos dependen de la capacidad de procesamiento de la máquina en la cual son ejecutados.

Por lo tanto, para implementar un mecanismo de visualización es necesario una o más instancias de la clase `TTimer`, las cuales permiten ejecutar una rutina cada cierto intervalo de tiempo sin truncar la ejecución del software, por lo que en esta sección se aborda el uso de ésta clase.

En caso que la interfaz posea un espacio de visualización, los cuales son asociados a las clases `TCanvas` y `TRootEmbeddedCanvas`, donde se presente una animación o una gráfica, es necesario, antes de crear la animación, crear y configurar dicho espacio. Estos espacios pueden ser creados de manera independiente (que tienen ventana propia) o embebidos dentro de otra ventana.

2.4.1. Configuración de un lienzo independiente y creación de Figuras

Para crear un espacio de visualización independiente se crea una instancia, en general, de la clase `TCanvas`, la cual ha sido llamada `Lienzo` (por la traducción que hace este documento de la palabra inglesa *Canvas* al español):

```
root[] Lienzo = new TCanvas();
```

Ésta definición no tiene asignados argumentos en el constructor, lo cual inicializa una instancia con los valores por defecto en las dimensiones del `Lienzo` (espacio de visualización). Para modificar el tamaño del lienzo se ejecuta el método `SetWindowSize`:

```
root[] Lienzo->SetWindowSize(300,300);
```

donde el primer argumento indica el ancho y el segundo la altura de la ventana en píxeles (argumentos de tipo entero). Para establecer un sistema de coordenadas en el lienzo se utiliza el método `Range()`:

```
root[] Lienzo->Range(-10,-10,10,10);
```

donde, el primer argumento indica el valor que obtiene la coordenada horizontal del extremo izquierdo, el segundo la coordenada vertical del extremo inferior, con lo que los primeros dos argumentos formarían el punto coordenado de la esquina inferior izquierda. El tercer argumento indica el valor que adquirirá la coordenada horizontal del extremo derecho y finalmente, el cuarto argumento, corresponde al valor que adquiere la coordenada vertical del extremo superior; estos dos últimos argumentos representarán el punto coordenado de la esquina superior derecha, tal como se muestra en la figura 2.6.

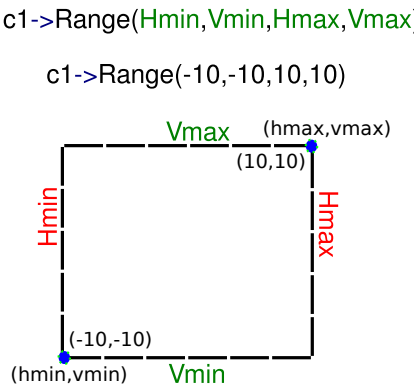


Figura 2.6: Sistema coordenado establecido en el lienzo a través del método `Range()`

Es preciso tener en cuenta que la variación tanto en la coordenada vertical como en la horizontal es de carácter lineal. Debido a que el lienzo es de dimensión cuadrada (300 píxeles de alto por 300 píxeles de ancho) y a la manera en que se ha dispuesto el valor de los extremos a través del método `Range()`, es posible establecer el origen (el punto $(0, 0)$) en el centro de la ventana. A partir de éste sistema coordenado es posible posicionar nuevos elementos en el interior del lienzo[4].

Trazo de líneas

Una vez dispuesto el lienzo, se pueden añadir sobre éste figuras geométricas. Por ejemplo, para trazar una línea que cruce desde el extremo inferior izquierdo hasta el extremo superior derecho se crea una instancia de la clase `TLine`:

```
root[] Linea1 = new TLine(-10,-10,10,10);
```

donde los primeros dos argumentos corresponden al punto (x, y) (referido al sistema coordenado del lienzo) donde comienza la linea y los dos últimos argumentos corresponderán al punto donde termina la línea. Ejecutada ésta línea se crea el objeto y se almacena en memoria. Para hacerlo visible se ejecuta el método `Draw()`:

```
root[] Linea1->Draw();
```

Trazo de flechas

Ahora, para trazar una flecha desde el extremo superior izquierdo hasta el extremo inferior derecho, es posible utilizar la clase `TArrow`:

```
root[] Flecha1 = new TArrow(-10,10,10,-10);
```

[4]Esto requiere que la resolución de su monitor se encuentre bien configurada. Si no es así, puede que al establecer un lienzo de dimensión cuadrada usted vea un rectángulo.

Los argumentos del constructor de la clase `TArrow` son los mismos de la clase `TLine`, sólo que aquí se distingue claramente el punto inicial y el punto final. Para plasmar la fecha en el lienzo se ejecuta:

```
root[] Flecha1->Draw()
```

Métodos para la modificación de flechas y líneas

Es posible cambiar las características de las flechas y las líneas, como el grosor, color, estilo (diríjase a `http://root.cern.ch/root/html/TAttLine.html` para observar los posibles estilos). En los dos casos (`TArrow`, `TLine`) se puede utilizar los siguientes métodos:

```
SetLineStyle(Int_t)
SetLineWidth(Int_t)
SetLineColor(Int_t)
```

El argumento es el entero que indica la opción. Recuerde que cada vez que se realiza un cambio sobre las características de un objeto debe ejecutar el método `Draw()` en el objeto, para poder visualizar dichos cambios. Adicionalmente, deberá ejecutar el método `Update()` en el lienzo en el cual se encuentra dibujado el objeto.

Se pueden modificar los puntos iniciales y finales, ejecutando los siguientes métodos:
Punto inicial:

```
SetX1(Double_t)
SetY1(Double_t)
```

Punto final:

```
SetX2(Double_t)
SetY2(Double_t)
```

donde `SetX1` y `SetX2` hacen referencia al eje horizontal y `SetY1` y `SetY2` al eje vertical.Los argumentos de los métodos anteriores corresponden al valor del punto en el eje horizontal (x) y en el eje vertical (y). Estos métodos existen tanto en la clase `TArrow` como en `TLine`. Por ejemplo, para cambiar el punto de partida de la flecha creada anteriormente (`Flecha1`) se ejecutaría:

```
root[] Flecha1->SetX1(0.0);
root[] Flecha1->SetY1(0.0);
```

Donde se fijaría el punto inicial de la flecha en el origen del sistema coordenado establecido en el lienzo. Para cambiar el punto final de la flecha y hacer visibles los cambios se ejecutaría:

```
root[] Flecha1->SetX2(1.0);
root[] Flecha1->SetY2(2.5);
root[] Flecha1->Draw();
root[] Lienzo->Update();
```

Así, la flecha quedaría modificada de tal manera que el punto coordenado en el cual inicia es $(0,0)$ y el punto coordenado final es $(1{,}0, 2{,}5)$.

Trazo de elipses/círculos

Para crear una elipse se utiliza la clase `TEllipse`, ejecutando:

```
root[] Elipse1 = new TEllipse(0,0,1,2,0,360);
```

Los primeros dos argumentos indican el punto (x, y) del sistema coordenado del lienzo, a donde están referidos tanto el semieje mayor como el semieje menor de la elipse (después puede cambiar éste punto a través de los métodos `SetX1()` y `SetY1()`). Los siguientes dos argumentos son los valores que adquieren los semiejes: el primero del centro al borde derecho/izquierdo de la elipse, medirá 1 unidad en éste caso (posteriormente es posible cambiar el valor de éste a través del método `SetR1()`), y el segundo medido del centro al borde superior/inferior, medirá 2 en éste caso (posteriormente es posible cambiar el valor de éste a través del método `SetR2()`); finalmente los dos últimos argumentos indican el ángulo inicial y el ángulo final del trazo, con lo que se pueden realizar no sólo elipses sino también círculos y segmentos de arco (elimine los trazos de los radios con el método `SetNoEdges()`). Existen otras clases como `TArc` que permiten dibujar círculos y radios; sin embargo, aquí se abordará únicamente la clase `TEllipse` debido a que es una clase general que permite cubrir incluso las funcionalidades de `TArc`. Nuevamente, para hacer visible la elipse `Elipse1` en el lienzo es necesario ejecutar su método `Draw()`:

```
root[] Elipse1->Draw();
```

Trazo de rectángulos

Para crear un rectángulo se utiliza la clase `TBox`. Por ejemplo:

```
root[] Rect1 = new TBox(-9,-9,-1,-1);
root[] Rect1->Draw();
```

Los primeros dos argumentos indican el punto (x, y) del extremo inferior izquierdo, los últimos dos argumentos indican el punto del extremo superior derecho. Para las últimos dos clases vistas (`TEllipse` y `TBox`) es posible seleccionar el color de relleno mediante el método (Ver `http://root.cern.ch/root/html/TAttFill.html#F1`):

```
SetFillColor(int)
```

También puede cambiar el patrón de relleno (ver opciones en: `http://root.cern.ch/root/html/TAttFill.html#TAttFill:fFillStyle`) mediante el método:

```
SetFillStyle(int)
```

Animando objetos

Hasta el momento los elementos creados han sido estáticos. Para crear una animación es necesario crear una instancia de la clase `TTimer`. Un objeto de

tipo `TTimer` permite (entre otras) la ejecución de una función (conocida como *función comando*) cada determinado intervalo de tiempo (dado en el constructor en milisegundos).

Para ilustrar el uso de una instancia de `TTimer`, se realizará una animación en la cual se crea un círculo (a través de la clase `TEllipse`) y se modifica tanto su radio (haciendo uso de los métodos `SetR1()` y `SetR2()`) como su color de relleno (a través del método `SetFillColor()`), cada $200ms$.

Inicialmente se declaran los objetos que serán utilizados y se cargan sus correspondientes cabeceras (esto último es opcional siempre que se trabaje con el interprete), como se ilustra a continuación:

```
1  #include <TCanvas.h>
2  #include <TEllipse.h>
3  #include <TTimer.h>
4
5  TCanvas *Lienzo;
6  TEllipse *Circulo;
7  TTimer *Cronometro;
8  Int_t i;
9  Double_t Radio;
```

Luego se define la función que se ejecutará cada intervalo establecido en la instancia de `TTimer`. En éste caso hemos llamado a ésta función `Animar()`:

```
10 void Animar(){
11
12   Radio = Circulo->GetR1();
13   //en cada ejecución se incrementa el radio
14   //en 0.1 unidades
15   Radio += 0.1;
16
17   //i es un contador de cuántas veces
18   //se ha ejecutado ésta función
19   i++;
20
21   //Se cambia el color de relleno
22   //del círculo aprovechando el
23   //contador i
24   Circulo->SetFillColor(i);
25   //Se establece el nuevo radio
26   //al ser un círculo r1=r2
27   Circulo->SetR1(Radio);
28   Circulo->SetR2(Radio);
29   //Se dibuja en el lienzo
30   Circulo->Draw();
31   //Se actualiza el lienzo
32   Lienzo->Update();
33
34   //Detiene el cronómetro si se alcanza
35   //un radio de 8 unidades
36   if(Circulo->GetR1() >= 8) Cronometro->TurnOff();
37
38 }
```

Finalmente, se crea la función principal del macro [5] que será llamado`ttimer.C`, por lo que la función se debe llamar `ttimer`:

[5] En ROOT, un archivo que contiene código fuente que puede ser ejecutado por el interprete, es

```
38  void ttimer(){
39
40    //Se crea y configura el lienzo
41    Lienzo = new TCanvas();
42    Lienzo->SetWindowSize(500,500);
43    Lienzo->Range(-10,-10,10,10);
44
45    //Se crea el círculo
46    Circulo = new TEllipse(0,0,1,1,0,360);
47    Circulo->Draw();
48    //Se crea un cronómetro que ejecutará
49    //una función cada 200ms
50    Cronometro = new TTimer(200);
51    i = 0;
52    //Se establece la función a ejecutar
53    //en cada intervalo
54    Cronometro->SetCommand("Desplazar()");
55    Cronometro->TurnOn(); //Activar TTimer
56
57  }
```

Es importante aclarar que a diferencia de la disposición de objetos en la interfaz gráfica, aquí no es necesario especificar en qué lienzo será plasmado determinado objeto, debido a que éste se dibujará en el lienzo que esté activo al momento de ejecutar el método `Draw()`. El lienzo es activado a través del método `cd()`. Para ilustrar esta idea se crean dos lienzos y dos objetos:

```
1   {
2     //Se crean dos lienzos
3     Lienzo1 = new TCanvas();
4     Lienzo2 = new TCanvas();
5
6     //Se configuran sus características principales
7     Lienzo1->SetWindowSize(300,300);
8     Lienzo1->Range(-10,-10,10,10);
9     Lienzo2->SetWindowSize(300,300);
10    Lienzo2->Range(-10,-10,10,10);
11
12    //Se crea un círculo y una línea
13    Circulo = new TEllipse(0,0,2,2,0,360);
14    Linea = new TLine(-10,-10,10,10);
15
16    //Se activa el Lienzo1
17    Lienzo1->cd()
18    //Se dibuja el círculo
19    Circulo->Draw()
20
21    //Se activa el Lienzo2
22    Lienzo2->cd()
23    //Se dibuja la línea
24    Linea->Draw()
25  }
```

se dibujará el círculo (objeto `Circulo`) en el `Lienzo1` y la línea (objeto `Linea`) en el `Lienzo2`. Si no se hubiera ejecutado el método `cd()`, ambos objetos se hubieran

llamado macro. Al ejecutar el macro, el punto de inicio de ésta ejecución es la función cuyo nombre corresponde al del macro. Por ejemplo, si se tiene un macro llamado `Macro.C`, la ejecución iniciará en la función `Macro()`. Ésta función, representaría el punto de partida del macro, es decir, cumple el cometido de la función `main()` en un programa tradicional escrito en C++.

dibujado sobre el `Lienzo2` debido a que por defecto el lienzo activo es el último que ha sido creado.

2.4.2. Visualización en un lienzo embebido

Cuando la visualización se encuentra embebida en una ventana, se debe crear el lienzo mediante la clase `TRootEmbeddedCanvas`. Para ilustrar lo anterior, se crea y configura una ventana principal que contendrá el lienzo:

```
root[] fMain = new TGMainFrame(gClient->GetRoot(),300,300,kHorizontalFrame);
root[] fMain->SetWindowName(``Lienzo embebido'');
```

y se crea una instancia de la clase `TRootEmbeddedCanvas`:

```
root[] fECanvas = new TRootEmbeddedCanvas(``fECanvas'',fMain,280,280);
```

cuyo primer argumento es el nombre de la instancia; el segundo, el padre del objeto que en éste ejemplo será la ventana principal (`fMain`), y, el tercer y cuarto argumento especifican el tamaño (ancho*alto). En el interior de `FECanvas` se debe especificar un lienzo particular `fCanvas`, en el cual se plasmarán los objetos (líneas, rectángulos, círculos, etc):

```
root[] TCanvas *fCanvas = fECanvas->GetCanvas();
```

Note que `fECanvas` resulta ser un mediador entre un lienzo `fCanvas` y la ventana principal.

El método `GetCanvas()` ejecutado en `fECanvas` devuelve un apuntador a dicho lienzo. Fíjese que hay dos objetos uno que es `fECanvas` y el otro que es `fCanvas`, éste último es únicamente un acceso directo al lienzo que crea `TRootEmbeddedCanvas`, es en éste sentido que la clase `TRootEmbeddedCanvas` representa un mediador entre el padre y el lienzo embebido; a partir de la instrucción anterior se puede configurar el lienzo `fCanvas` y crear objetos (dibujar) sobre él de la misma forma que se realizó en la sección anterior:

```
root[] fCanvas->Range(-10,-10,10,10);
root[] fCanvas->cd();

//Se crea un círculo
root[] Circ = new TEllipse(0,0,5,5,0,360);
root[] Circ->SetFillColor(2);
root[] Circ->Draw();

//Se crea un cuadro con un
//patrón de relleno
root[] Rect1 = new TBox(-2,-2,2,2);
root[] Rect1->SetFillStyle(3024);
root[] Rect1->Draw();
```

Finalmente se incorpora `fECanvas` en la ventana principal, y se ejecutan los métodos habituales para mostrar los ventana:

```
root[] Expandir = new TGLayoutHints(kLHintsExpandX|kLHintsExpandY,10,10,10,10);
root[] fMain->AddFrame(fECanvas,Expandir);
root[] fMain->MapSubwindows();
root[] fMain->Resize();
root[] fMain->MapWindow();
```

El resultado se muestra en la figura 2.7.

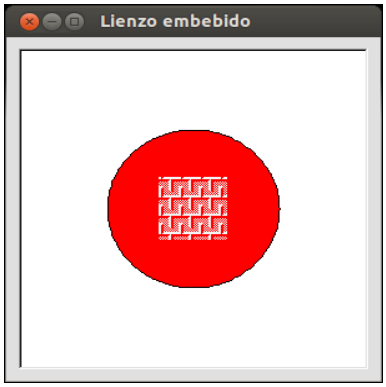


Figura 2.7: Lienzo embebido en una ventana principal, que contiene un círculo y un rectángulo plasmados en su interior

2.4.3. Configuración básica de una gráfica

En esta sección se ofrece una rápida descripción de cómo se implementa y cómo se aplica la configuración básica de una gráfica en dos dimensiones. Una descripción detallada que muestra la implementación de gráficas en dos y tres dimensiones, así como la adecuación de parámetros como márgenes de error o la superposición de gráficas y procedimientos como ajustes de curvas, es ofrecida en el Capítulo 4 de la guía del usuario de ROOT (ROOT-Team, s.f.-b).

Realizar una gráfica con las clases que ROOT ofrece es un procedimiento sencillo. Para esto es necesario una instancia de la clase `TGraph`:

```
root[] g1 = new TGraph();
```

Para introducir los datos en la gráfica existen diferentes formas; sin embargo, el método clave es `SetPoint()`. Por ejemplo, para realizar la gráfica de la función $\sqrt{x}$, se puede implementar la siguientes líneas de código:

```
root[] for (Int_t i = 0 ; i < 50; i++){
root[]  g1->SetPoint(i,i*2,sqrt(i));
root[] }
```

El primer argumento en `SetPoint()` indica el índice del punto coordenado, en tanto que el segundo y tercer argumento indican los valores en los ejes horizontal x, y vertical y respectivamente; así, el ejemplo anterior graficará el siguiente conjunto de datos:

Índice	x	$f(x)$
1	2	$\sqrt{2}$
2	4	$\sqrt{4}$
3	6	$\sqrt{6}$
...	...	...
i	$2i$	$\sqrt{2i}$

Para realizar el cambio del título de la gráfica y la etiqueta de los ejes[6] se ejecuta:

```
root[] g1->SetTitle(``Nuevo T\355tulo'')
root[] g1->GetXaxis()->SetTitle(``x'');
root[] g1->GetYaxis()->SetTitle(``#sqrt{x}'');
```

Finalmente igual que la elipse, el rectángulo o la línea, para plasmar la gráfica en el lienzo se invoka el método `Draw()`:

```
root[] g1->Draw(``AC*'');
```

La cadena de caracteres que se pasa como argumento al método `Draw()` corresponde a opciones del gráfico. En éste caso, la opción `A` (de *axis*), la opción `C` (de *curve*) permite unir los puntos mediante una curva y la opción `*` es la representación de los puntos evaluados. Esto se ilustra en la figura2.8.

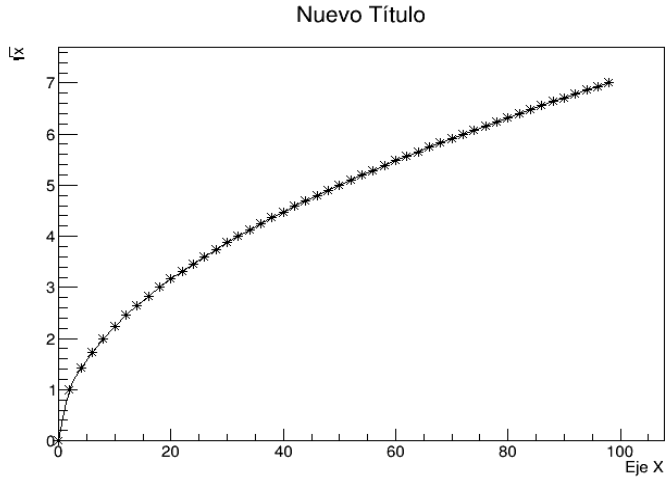


Figura 2.8: Ejemplo de un diagrama que cuenta con el formato básico.

[6]Para la escritura de fórmulas complejas puede utilizar LaTeX , básicamente reemplazando el *backslash* (\) por el símbolo numeral (#), tanto en el título como en los ejes (para mayor información consulte la sección *Text and Latex Mathematical Expressions* de la guía del usuario (ROOT-Team, s.f.-b))

2.5. Comunicación entre objetos y conexión de la interfaz

2.5.1. El mecanismo *signal-slot*

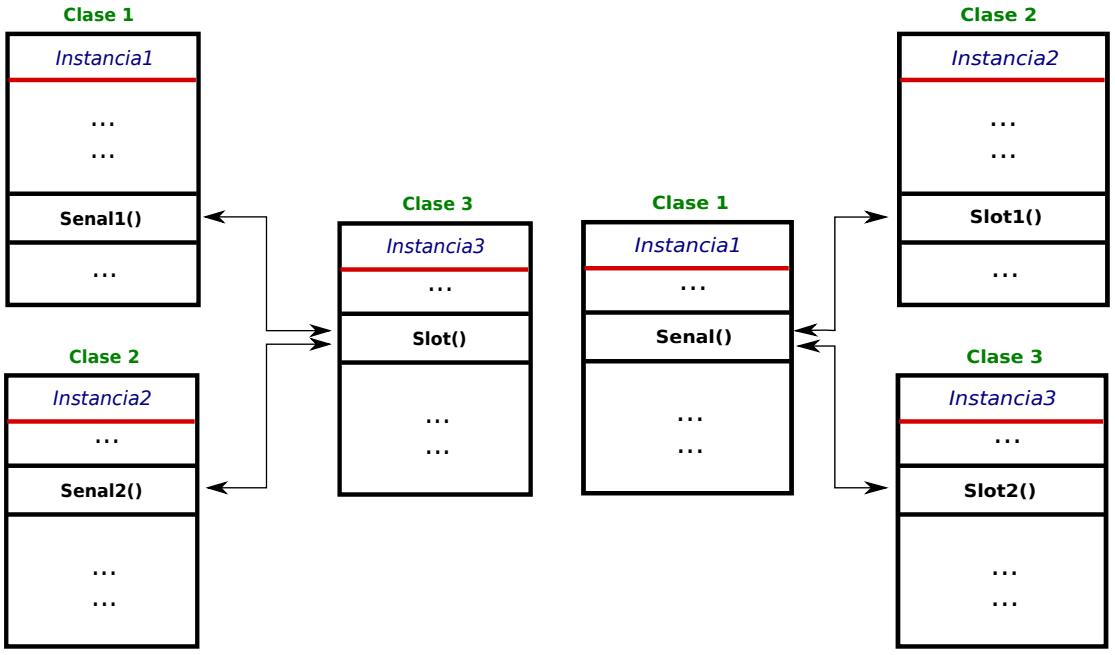
En la implementación de interfaces gráficas de usuario mediante la infraestructura de análisis de datos *ROOT*, se cuenta con el mecanismo desarrollado por *Qt*[7] para la comunicación entre objetos que se conoce como *signal-slot*. Éste mecanismo es una flexible, útil, e intuitiva herramienta para realizar la conexión de los diferentes elementos de la interfaz gráfica, permitiendo su funcionalidad.

En el desarrollo de una interfaz gráfica se requiere la conexión de los elementos gráficos, llámense ventanas, botones, etiquetas, etc, con los procesos internos del programa. Si tiene una interfaz gráfica de una simulación que cuenta con un botón que se titula "Ejecutar" éste botón no hará nada a menos que la pulsación de ese botón este *conectada* en la programación de la simulación con un proceso determinado. Éste proceso es simplemente un bloque de código, un método de carácter público definido en una clase determinada (éste método se le asigna el nombre de *slot*).

Cuando ocurre un evento como la pulsación de un botón o el cambio en el tamaño de una ventana, se ejecuta un método en la clase sobre la cual se produce ese evento. Por ejemplo, cuando ocurre un evento como el haber presionado un botón, se ejecuta un método, la ejecución de éste método produce una señal, es decir, el botón envía información sin un destino fijo mediante la cual todos los demás objetos pueden enterarse que ha sucedido un evento en el botón. La señal (*signal*) es la forma mediante la cual un objeto se comunica con otros para informar su cambio de estado; en el ejemplo se informa a los demás objetos que determinado botón de la interfaz ha sido presionado. Una vez ésta señal es emitida, se puede *conectar* la recepción de la señal con un método de otro objeto, el cual una vez conectado recibe el nombre de *slot*.

La conexión de la señal emitida por un objeto no es unívoca necesariamente, es posible conectar una señal con múltiples *slots* (figura 2.9b); esto conllevará que cuando se produzca un evento que genere una señal, se ejecuten varios métodos. También es posible conectar múltiples señales a un *slot* (figura 2.9a), lo que implica que cuando se generen eventos en diferentes objetos (lo cual produciría múltiples señales) se ejecute un método en particular.

[7] Qt es una infraestructura libre y multiplataforma para el desarrollo de interfaces gráficas de usuario. Ésta infraestructura se encuentra implementada en diferentes lenguajes de programación, como Java (Qt-Jambi), Python (PyQt) y C++ (Qt).



(a) Conexión de múltiples señales conectadas a un mismo *slot*.

(b) Conexión de múltiples *slots* conectados a una misma señal

Figura 2.9: La conexión no es necesariamente unívoca en el mecanismo *signal-slot*.

Existen señales más complejas, que contienen mayor información acerca del estado del objeto que ha producido la señal; por ejemplo, cuando la interfaz cuenta con una lista de selección, no sólo nos interesa saber que se ha seleccionado una opción en la lista, sino que también nos interesa saber qué opción ha sido seleccionada. En éste caso, la señal proveerá la información de qué opción ha sido seleccionada, lo cual resulta útil para el programador ya que la respuesta del software puede depender de la opción que seleccionó el usuario.

El mecanismo de *signal-slot* permite conectar la emisión de señales con bloques de código de tal manera que los objetos obtengan el comportamiento deseado en la interfaz. Es importante resaltar que debido a que los *slots* hacen parte de un sistema de comunicación entre diferentes objetos deben ser declarados bajo la directiva `public`.

Hasta el momento se ha visto cómo crear ventanas y como se organizan elementos en ésta; no obstante, estos elementos han permanecido hasta ahora desconectados. A continuación se muestran dos ejemplos del mecanismo *signal-slot*. En el primero de ellos se ve una señal que no proporciona mayor información, donde se creará una ventana con un botón en su interior que una vez pulsado, cierra la ventana. Luego se presenta un ejemplo de una señal que contiene información acerca del evento: se creará una lista de selección, que una vez el usuario seleccione una opción de la lista, se cambiará el título de la ventana de acuerdo con la opción seleccionada.

2.5.2. Ejemplo de una señal sin información adicional: `VUnBoton`

Al implementar una interfaz gráfica es necesario definir una clase, que resulta ser, un tipo de objeto restringido y particular. Ésta clase representa la interfaz gráfica del programa de tal manera que sea posible posteriormente crear una instancia de esta clase. Crear una instancia de ésta clase es análogo a abrir el programa (la apertura de un programa implica la creación de una instancia del mismo).

El ejemplo consiste en una ventana que contiene un botón en su interior que permite cerrar esa ventana[8]. La implementación es iniciada con la inclusión de dos librerías básicas que permiten el uso del mecanismo *signal-slot*. Si bien ROOT mediante su interprete CINT realiza la precarga una gran cantidad de librerías, para todas las interfaces gráficas es necesario realizar la carga "manual" de:

```
1  #include <TGClient.h>
2  #include <RQ_OBJECT.h>
```

Luego se escribe la cabecera de la clase que contiene las declaraciones de los métodos, tal como se vio en la sección 1.4.1:

```
3  class VUnBoton {
4
5  public:
6    VUnBoton(Int_t w = 200, Int_t h = 200);
7
8  };
```

Aquí se ha declarado una clase llamada `VUnBoton` (de VentanaConUnBoton), que tiene sólo un método, el cual es de carácter público. El método definido es el constructor de la clase que tiene dos argumentos de tipo entero con un valor por defecto de 200, el primero de ellos (`w`) representa el ancho y el segundo representa la altura de la ventana principal. Entonces, la definición del constructor podría ser codificada, así:

```
9  void VUnBoton::VUnBoton(Int_t w,Int_t h){
10
11   //Se crea la ventana principal
12   fMain = new TGMainFrame(gClient->GetRoot(),w,h);
13   //el título de la ventana principal será
14   //Ventana 1
15   fMain->SetWindowName("Ventana 1");
16
17   //Se crea un botón de texto
18   fButton = new TGTextButton(fMain,"Bot\363n");
19   //Se conecta la señal Clicked() del botón
20   //Con el método CloseWindow() de la clase
21   //TGMainFrame
22   fButton->Connect("Clicked()","TGMainFrame",fMain,"CloseWindow()");
23   //Se cambia el tamaño del botón
24   fButton->Resize(w-10,h-10);
25   //Se cambia de posición
26   fButton->Move(5,5);
27
28   //Se añade el botón a la ventana principal
```

[8]La declaración y definición de la clase será realizada en un archivo llamado `UnBoton.C`

```
29      fMain->AddFrame(fButton);
30      //Se mapea todos los objetos de la ventana
31      //principal, en éste caso sólo el botón
32      fMain->MapSubwindows();
33      //se muestra la ventana principal
34      fMain->MapWindow();
35
36    }
```

Cuando se crea una instancia de tipo `VUnBoton`, se ejecutan las instrucciones contenidas en el constructor de ésta clase; así, de acuerdo con la definición dada se creará una ventana cuyo título será `Ventana 1`, con un ancho de `w` y una altura de `h` píxeles. Ésta ventana contendrá un botón de texto (`TGTextButton`) que contiene la palabra *Botón* que, al ser presionado, cerrará la ventana.

Un método de vital importancia en la conexión de *signals* con *slots* es el método `Connect()`, cuya sintaxis corresponde a:

```
Bool_t Connect(const char* signal, const char* receiver_class, void* receiver, const char* slot)
```

donde el primer argumento es el nombre de la señal, el segundo, corresponde al tipo de objeto (la clase) que contiene el *slot*, el tercero, es un apuntador al objeto que recibe la señal, y el cuarto argumento es el nombre del *slot*, es decir, el nombre del método que se ejecutará cuando la señal sea emitida. Por tanto:

```
22    fButton->Connect("Clicked()","TGMainFrame",fMain,"CloseWindow()");
```

se leería así: se ejecuta el método `Connect()` en el botón `fButton` para conectar la señal `Clicked()` con el método `CloseWindow()`. La señal `Clicked()` es emitida cada vez que el botón `fButton` es presionado. Note que `CloseWindow()` es un método que pertenece a una clase particular, en este caso, la clase `TGMainFrame` y se ejecuta a través de una instancia de dicha clase, en este caso, la instancia corresponde a `fMain` (ver figura2.10).

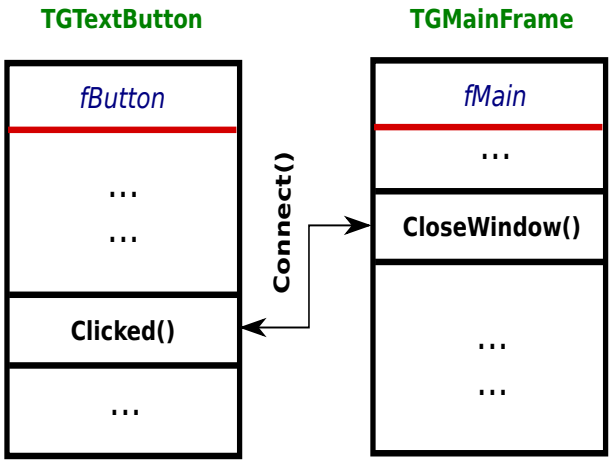


Figura 2.10: Esquema de la conexión *signal-slot*. Una vez pulsado el botón `fButton` se emite la señal `Clicked()`, a través del método `Connect()` se especifica que una vez se produce esa señal se debe ejecutar el método `CloseWindow()` el cual cierra la ventana `fMain`.

Siempre que se requiera conectar *signals* con *slots* es necesario ejecutar el método `Connect()` en el objeto que emite la señal. Cabe anotar que se puede conectar (como se mencionó anteriormente) varios señales a un *slot*, y varios *slots* a una señal.

Finalmente, para crear una instancia de la clase que ha sido escrita (i.e ejecutar el programa) se ejecutaría:

```
root[] #include ''VUnBoton.C''
root[] fVentanaConUnBoton = new VUnBoton(400,200);
```

La primera línea incluye la definición de la clase implementada en ésta sección, y la segunda, crea una instancia de la clase `VUnBoton` llamada `fVentanaConUnBoton`.

2.5.3. Ejemplo de una señal con información adicional: `TituloDeLista`

Las señales pueden contener información que permite adecuar la respuesta a las instrucciones dadas por el usuario. Para ilustrar lo anterior, se realiza una interfaz gráfica que contiene una lista de opciones, cada vez que el usuario selecciona una de las opciones de la lista, se modificará el título de la ventana de acuerdo a la opción seleccionada por el usuario. Para este ejemplo, la señal no sólo informa que una opción ha sido seleccionada (que corresponde al caso de una señal sin información adicional), sino que especifica qué opción seleccionó el usuario (lo que corresponde a una señal con información adicional).

Inicialmente, se declara la clase que se llamará `TituloDeLista` con sus respectivos métodos y objetos globales[9]:

[9]La declaración y definición de ésta clase será realizada en el archivo `Lista.C`

```
1  #include <TGClient.h>
2  #include <RQ_OBJECT.h>
3
4  class TituloDeLista {
5
6  private:
7    void CambiarTitulo(Int_t id);
8    TGMainFrame *fMain;
9
10 public:
11   TituloDeLista();
12
13 };
```

La declaración de la ventana principal `fMain` tiene un nivel de globalidad `private`, dado que se requiere que las propiedades de la ventana principal, entre las cuales se encuentra su título, puedan ser modificadas en cualquiera de los métodos que corresponden a la clase `TituloDeLista`. El método `CambiarTitulo()` será asignado como *slot*, estos métodos deben ser . Se define entonces el constructor de la clase:

```
14 TituloDeLista::TituloDeLista(){
15
16   //Se crea y configura la ventana principal
17   fMain = new TGMainFrame(gClient->GetRoot(),200,80);
18   fMain->SetLayoutBroken(kTRUE);
19
20   //Se crea una lista de selección
21   Lista = new TGComboBox(fMain);
22   //Se modifica el tamaño de la Lista
23   //y luego se ubica en la ventana
24   Lista->Resize(160,40);
25   Lista->Move(20,20);
26
27   //Se añaden items a la lista
28   Lista->AddEntry("Opci\363n 1",1);
29   Lista->AddEntry("Opci\363n 2",2);
30   Lista->AddEntry("Opci\363n 3",3);
31
32   //Se conecta la señal
33   Lista->Connect("Selected(Int_t)","TituloDeLista",
34                  this,"CambiarTitulo(Int_t)");
35
36   //Se selecciona la opción que
37   //tendrá al iniciar
38   Lista->Select(3);
39
40   //Se ejecutan los métodos
41   //para hacer visible los objetos
42   fMain->AddFrame(Lista);
43   fMain->MapSubwindows();
44   fMain->MapWindow();
45
46 }
```

donde ha sido incluida una instancia de la clase `TGComboBox`, el cual es un objeto que presenta una lista expandible. El argumento que se ha especificado en su constructor indica el padre del objeto, que pare este caso, corresponde a la ventana principal `fMain`. Los ítems de la lista se añaden a través del método `AddEntry()` que contiene dos argumentos: el primero es el texto que se mostrará en el ítem, y el segundo, es un número

identificador (de tipo entero) que representará la opción. Por ejemplo, si el identificador es 3 se hace referencia a la "Opción 3".

En la ejecución del método `Connect()`, utiliza ahora la señal `Selected(Int_t)` la cual contiene un argumento (Note que a diferencia de la señal `Clicked()` abordada en la sección 2.5.2). Éste argumento es la información adicional que emite la señal; para este caso es un argumento es de tipo entero (`Int_t`) que representa el número identificador de la opción seleccionada. Por ejemplo, si el usuario seleccionó "Opción 1" de la lista, la señal contendrá el número identificador de ésta opción (1 en este caso). Ésta señal (`Selected(Int_t)`) es conectada al método `CambiarTitulo(Int_t id)`, definido en la clase `TituloDeLista` (la clase que se está implementando). Éste método tiene un argumento de tipo entero (`id`); es decir, al realizar la conexión de la señal con el *slot*, el argumento es transmitido de la señal al *slot*. De ésta manera, el *slot* puede utilizar el argumento para ajustar la respuesta del programa. Este ajuste de la respuesta del programa se evidencia en la selección del título que le corresponde a la ventana principal:

```
15  void TituloDeLista::CambiarTitulo(Int_t id){
16
17    switch(id){
18    case 1:{
19      fMain->SetWindowName("Opci\363n 1");
20    }
21      break;
22    case 2:{
23      fMain->SetWindowName("Opci\363n 2");
24    }
25      break;
26    case 3:{
27      fMain->SetWindowName("Opci\363n 3");
28    }
29    }
30    return kTRUE;
31  }
```

Aquí se guarda en una variable de tipo entero (`id`), la información transmitida por la señal (el identificador de la opción seleccionada) y, mediante una estructura de tipo `switch` se establece el título de la ventana: si el identificador `id` es el número 1 ($id = 1$) entonces se cambia el título de la ventana a "Opción 1"; si el identificador no equivale a 1, sino a 2, el título de la ventana será "Opción 2", y así sucesivamente.

Para crear una instancia de la clase que ha sido escrita se ejecutaría:

```
root[] #include ``Lista.C''
root[] fSeleccionTitulo = new TituloDeLista()
```

Capítulo 3

Casos de estudio

En el capítulo anterior se expuso sobre las diferentes partes que componen una interfaz gráfica, y, la creación y configuración básica de sus componentes: unos objetos organizados de manera jerárquica en una ventana principal; un espacio de visualización tanto de una animación como de un diagrama, y, la conexión básica de elementos como botones a través del mecanismo de *signal-slot*.

En este capítulo se realizará un compendio práctico de la información contenida en capítulos anteriores, a través de la explicación del procedimiento ejecutado para el diseño y la implementación de interfaces gráficas para dos fenómenos físicos particulares: la caída libre y el movimiento de un cuerpo esférico sumergido en un fluido bajo la acción de la gravedad. Al finalizar la presentación de cada caso se propone un conjunto de actividades que pueden ser desarrolladas modificando el código fuente que será explicado a lo largo de cada sección.

3.1. La caída libre

Propósito y modelo matemático

En este casó se realizará una interfaz gráfica que permita la visualización del movimiento de un cuerpo que se deja caer desde el reposo y que se encuentra bajo la acción de la gravedad. Ésta interfaz permitirá observar cómo varia la altura en función del tiempo. El modelo matemático que rige el fenómeno de la caída libre de un cuerpo establece que:

$$m\frac{d^2y}{dt} = -mg \tag{3.1}$$

donde m representa la masa del cuerpo, y es una función del tiempo que representa la altura del cuerpo en un instante determinado, y g la aceleración de la gravedad. Los parámetros que deben ser introducidos por el usuario son: la masa del cuerpo, el valor de g y dos condiciones iniciales $y(t = 0s)$. Como se ha mencionado anteriormente, el

cuerpo cae desde el reposo por lo que $y'(t = 0s) = 0\frac{m}{s}$. Debido a que no se requiere diferenciar el conjunto de variables no se utilizará ningún método de agrupación en la interfaz gráfica.

Parámetros, características y medio de introducción

Evaluando las características de los parámetros de la simulación y las condiciones iniciales del experimento, así como las restricciones a las que estará sujeta la simulación de acuerdo a lo planteado en la sección 2.1, para éste ejemplo se establece que:

- La masa m del cuerpo:
  - Cantidad positiva y diferente de cero.
  - Precisión en $0{,}001kg$.
  - El valor por defecto será de $1kg$.
- La aceleración g de la gravedad:
  - Cantidad positiva
  - Precisión en $0{,}001\frac{m}{s}$
  - El valor por defecto será $9{,}870\frac{m}{s^2}$
- La altura inicial del cuerpo $y(t = 0)$:
  - Matemáticamente no existe restricción. No obstante, se establece que es una cantidad positiva y será establecido un límite donde la altura del cuerpo es $0m$ (el suelo), idealmente se espera que el cuerpo toque el suelo y allí termine la simulación.
  - La precisión en $0{,}01m$
  - El valor por defecto será $10{,}00m$

Si bien se pueden utilizar diferentes elementos gráficos para la introducción de las variables, se utilizará la entrada numérica (`TGNumberEntry`) para todos los datos de éste caso de estudio.

Esquema de la interfaz

El esquema diseñado para la interfaz gráfica se muestra en la figura 3.1.

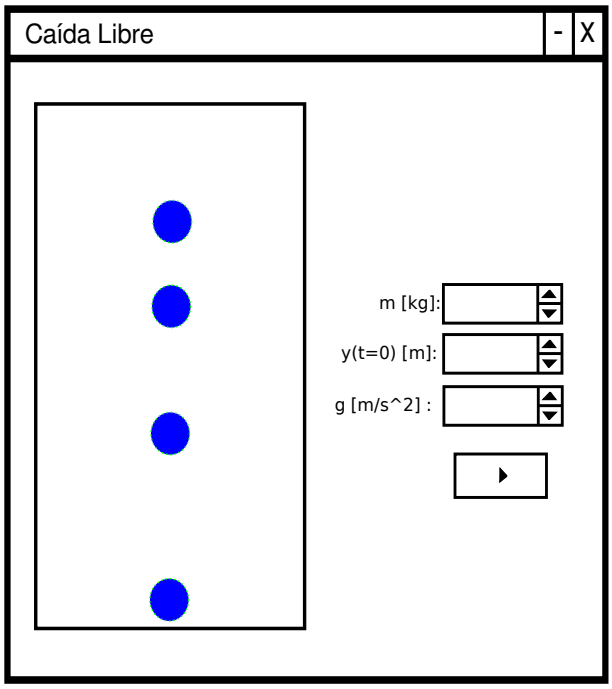


Figura 3.1: Esquema de la interfaz gráfica para el fenómeno de la caída libre.

Fíjese que la interfaz gráfica contará con un área de visualización, en la que se verá la caída del cuerpo (representado por el círculo azul) y los parámetros que permiten simular dicha caída (para éste caso m, y, y g). Además cuenta con un botón en la parte inferior izquierda que permitirá ejecutar la visualización.

A partir del esquema realizado es posible llevar a cabo la identificación de los marcos de agrupación que establecen la jerarquía de la interfaz, tal como fue expuesto en la sección 2.3.2. Existen marcos que agrupan las etiquetas y las entradas numéricas para los parámetros de la simulación de manera horizontal, un marco que contiene a estos marcos horizontales y al botón de manera vertical. Finalmente, la ventana principal, agrupa de manera horizontal al lienzo donde se verá la animación y al último marco mencionado.

Hasta aquí ya se han resuelto seis de las siete preguntas, planteadas por éste documento, asociadas a los aspectos de diseño de una interfaz gráfica de usuario.

Como se mencionó anteriormente, un diagrama de flujo proporciona una visión general del comportamiento de la interfaz gráfica de usuario, lo cual es una guía para el proceso de implementación. Éste no debe representar en ningún caso una restricción, ya que la propia implementación puede proporcionar elementos adicionales que pueden aumentar la funcionalidad del programa o la eficiencia del proceso. Lo antedicho nos permitirá resolver la séptima pregunta: ¿Cuál es el comportamiento de la interfaz?

Comportamiento de la interfaz

A partir del diseño ilustrado en la figura 3.1, el comportamiento de la interfaz se resume en el diagrama de flujo ilustrado en la figura 3.2.

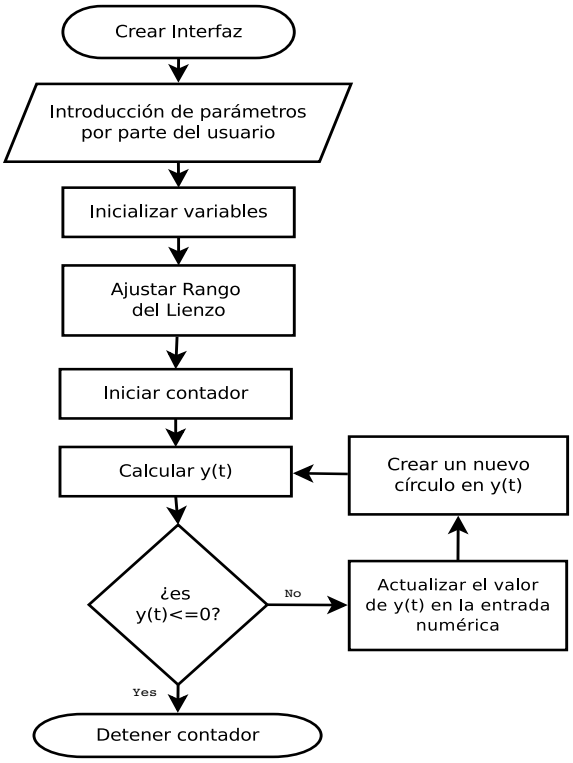


Figura 3.2: Diagrama de flujo: simluación del fenómeno de la caída libre

El diagrama de flujo presentado en la figura 3.2 muestra que la apertura de la simulación crea la interfaz gráfica. Una vez el usuario introduce el valor de los parámetros y pulsa el botón que permite dar inicio a la simulación, el programa se encargará de inicializar las variables, asignándoles el valor correspondiente dado por el usuario (ésta rutina se implementará en el método que será llamado `PBPlayClicked()` y corresponderá al *slot* de la señal `Clicked()` producida por el botón). A partir de la asignación del valor de la altura se realiza el ajuste del rango del lienzo, de modo tal que permita la visualización de la totalidad del fenómeno (éste ajuste será implementado en el método `AjustarRangoCanvas(Double_t h)`) y se inicia el contador. La activación del contador pone en marcha la animación, para lo que es necesario realizar el cálculo de $y(t)$ de acuerdo a la dinámica del sistema, plasmar la posición del cuerpo a través del círculo azul (ver figura 3.1), y modificar el valor de la altura en la entrada numérica correspondiente (esta rutina será parte del método llamado `MNumerico()`). Con el fin de dejar un rastro en el lienzo, se crea un nuevo círculo de las mismas características, y se repite el ciclo hasta que la altura adquiera un valor igual o inferior a cero.

La simulación se detendrá automáticamente cuando la altura del cuerpo sea cero.

El cero, en la visualización está representado por el borde inferior del espacio de visualización.

Implementación

Una vez se ha llevado a cabo el diseño de la simulación, se requiere realizar su implementación. Se inicia con la definición de una clase que será llamada `CaidaLibreGui`[1]:

```
1  #ifndef CaidaLibreGui_
2  #define CaidaLibreGui_
3
4  #include "TGNumberEntry.h"
5  #include "TRootEmbeddedCanvas.h"
6  #include "TCanvas.h"
7  #include "TEllipse.h"
8  #include "TTimer.h"
9  #include "TGClient.h"
10 #include "RQ_OBJECT.h"
11
12 class CaidaLibreGui
13 {
14
15   RQ_OBJECT("CaidaLibreGui");
16
17 private:
18   TGNumberEntry *NEMasa; //Entrada Numérica para la masa
19   TGNumberEntry *NEGravedad; //Entrada Numérica para la gravedad
20   TGNumberEntry *NEAltura; //Entrada Numérica para la altura del cuerpo
21   TRootEmbeddedCanvas *ECanvas; //Lienzo embebido
22   TCanvas *fCanvas; //Referencia al lienzo
23   TEllipse *E1; //Elipse que representará el cuerpo
24   TTimer *Contador; //Cronómetro
25   void AjustarRangoCanvas(Double_t h);
26   Double_t g;   // g = Aceleración gravitacional
27   Double_t Y;   // Y  = Y(t)
28   Double_t YP;  // YP  = Ẏ(t)
29   Double_t m;   // m = Masa
30   Double_t h;   // h = Altura Inicial
31   Double_t dt;  // dt = Paso (Intervalo de Tiempo)
32
33 public:
34   CaidaLibreGui();
35   ~CaidaLibreGui() {;}
36   void PBPlayClicked();
37   void MNumerico();
38   void Exit();
39
40   ClassDef(CaidaLibreGui,0);
41 };
42
43 #endif
```

Inicialmente se declaran 3 entradas numéricas (`TGNumberEntry`): una para la masa (`NEMasa` - de NumberEntryMasa), otra para la aceleración de la gravedad (`NEGravedad`) y la última para la altura (`NEAltura`).

[1] Esta es la cabecera de la clase que se encontrará en el archivo `CaidaLibreGui.h`

Todos los objetos que son declarados con globalidad privada (`private`) son globales dentro de la clase, es decir, se puede llamar o ejecutar métodos de éstas instancias dentro de *cualquier* función miembro de la clase `CaidaLibreGui`. Por ejemplo, un método del objeto `NEAltura` (como el que nos permite extraer el número que contiene esta entrada numérica) puede ser llamado desde el método (o función miembro) `PBPlayClicked()` y también desde su constructor (`CaidaLibreGui`).

También se realizó la declaración del lienzo que contendrá la animación, una elipse (`E1`) que será la representación gráfica del cuerpo y un contador (`TTimer`) que permitirá animar las representaciones gráficas tal como se vio en la sección 2.4.

Finalmente se declaran los métodos públicos básicos de toda clase, el constructor `CaidaLibreGui()` y el destructor de la clase `CaidaLibreGui()`, y los *slots* que serán utilizados (los cuales son de carácter público de acuerdo a lo expuesto en la sección 2.5.1)

Una vez declarados los métodos, objetos y variables, se inicia con la implementación del constructor de `CaidaLibreGui()`[2][3]:

```
1  #include "CaidaLibreGui.h"
2  CaidaLibreGui::CaidaLibreGui(){
3    //Se crea y configura la ventana principal
4    TGMainFrame *fMain = new TGMainFrame(gClient->GetRoot(),
5                                         350,300,kHorizontalFrame);
6    fMain->SetWMSizeHints(350,300,350,300,0,0);
7    fMain->SetWindowName("Caso de Estudio: Ca\355da Libre");
8    fMain->Connect("CloseWindow()","CaidaLibreGui",this,"Exit()");
9    //Se crea y configura el Lienzo para la animación
10   ECanvas = new TRootEmbeddedCanvas("ECanvas",fMain,155,280);
11   //Definimos un "Acceso Directo" al canvas del canvas embebido
12   fCanvas = ECanvas->GetCanvas();
13   AjustarRangoCanvas(10);
14   E1 = new TEllipse(0,5,0.5,0.5);
15   E1->SetFillColor(4);
16   E1->Draw();
17   //Se crea el espacio de los botones
18   TGVerticalFrame *VF1 = new TGVerticalFrame(fMain);
19   TGLayoutHints *CentrarEnX = new TGLayoutHints(kLHintsCenterX,2,2,2,2);
20   TGHorizontalFrame *HFMasa = new TGHorizontalFrame(VF1);
21   TGLabel *LMasa = new TGLabel(HFMasa,"Masa [kg] : ");
22   NEMasa = new TGNumberEntry(HFMasa);
23   NEMasa->SetNumAttr(TGNumberFormat::kNEAPositive);
24   NEMasa->SetNumStyle(TGNumberFormat::kNESRealThree);
25   NEMasa->SetNumber(1.000);
26   HFMasa->AddFrame(LMasa,CentrarEnX);
27   HFMasa->AddFrame(NEMasa,CentrarEnX);
```

Observe cómo se ha configurado la entrada numérica `NEMasa`: se ha restringido el valor que se puede introducir utilizando métodos como `SetNumAttr()` y `SetNumStyle()`.

`SetNumAttr()` establece los atributos del número que puede introducirse, cuyo argumento argumento un número entero que indica una opción; sin embargo, para hacer legible el código utilizamos una variable de tipo entero, `TGNumberFormat::kNEAPositive`[4],

[2] La definición de los métodos de ésta clase se encontrará en el archivo `CaidaLibreGui.cxx`

[3] Cada línea ha sido explicada en secciones anteriores, si tiene dudas acerca de qué significa una línea de código en particular, se recomienda retroceder a la sección correspondiente.

[4] La `k` indica que es una constante. `kNEAPositive` hace referencia a NumberEntryAttributePositive

que está definida en la clase `TGNumberFormat`. Esta la forma de implementar las restricciones determinadas en el diseño de acuerdo a las características del modelo para la masa m.

El siguiente método, (`SetNumStyle`), determina lo que en la implementación de ROOT se llamó el estilo (*Style*). Aquí se establece el tipo de número, que en este caso la opción `TGNumberFormat::kNESRealThree` hace referencia a un número real con tres decimales[5]. Una lista completa de las opciones tanto de estilo como de atributos para el formato de número puede consultarse en `http://root.cern.ch/root/html/TGNumberFormat.html`.

Finalmente, el método `SetNumber()` permite fijar el número que contendrá la entrada numérica al abrir el programa.

Continuando con la implementación del constructor:

```
28    TGHorizontalFrame *HFAltura = new TGHorizontalFrame(VF1);
29    TGLabel *LAltura = new TGLabel(HFAltura,"Altura [m] : ");
30    NEAltura = new TGNumberEntry(HFAltura);
31    NEAltura->SetNumber(10.00);
32    NEAltura->SetNumAttr(TGNumberFormat::kNEAPositive);
33    NEAltura->SetNumStyle(TGNumberFormat::kNESRealTwo);
34    HFAltura->AddFrame(LAltura,CentrarEnX);
35    HFAltura->AddFrame(NEAltura,CentrarEnX);
36    TGHorizontalFrame *HFGravedad = new TGHorizontalFrame(VF1);
37    TGLabel *LGravedad = new TGLabel(HFGravedad,"g [m/s^2] : ");
38    NEGravedad = new TGNumberEntry(HFGravedad);
39    NEGravedad->SetNumAttr(TGNumberFormat::kNEAPositive);
40    NEGravedad->SetNumStyle(TGNumberFormat::kNESRealThree);
41    NEGravedad->SetNumber(9.870);
42    HFGravedad->AddFrame(LGravedad,CentrarEnX);
43    HFGravedad->AddFrame(NEGravedad,CentrarEnX);
```

donde se crean y configuran las entradas numéricas para la altura y la gravedad.

Con el fin de agregar un botón de ejecución (ver figura 3.1), se utiliza la clase `TGPictureButton`. El primer argumento del constructor de ésta clase, corresponde padre del botón, y el segundo a la imagen que contendrá. Ésta imagen se obtiene a través de la instrucción `gClient->GetPicture(RutaDeAccesoALaImagen)`:

```
44    TGPictureButton *PBPlay = new TGPictureButton(VF1,
45                  gClient->GetPicture("$ROOTSYS/icons/arrow_right.xpm"));
46    PBPlay->Connect("Clicked()","CaidaLibreGui",this,"PBPlayClicked()");
```

Aquí, se realiza la conexión mediante el método `Connect()`, de modo que cuando el botón es presionado se ejecuta el método `PBPlayClicked()` de la clase `CaidaLibreGui`. Fíjese que el tercer argumento, `this`, es un apuntador a la instancia de la clase `CaidaLibreGui`. Siempre que se se conecten *signals* con *slots* que se encuentran definidos en la clase donde se está definiendo la interfaz gráfica, se utiliza el apuntador `this`[6].

[5] `kNESRealThree` es la contracción de NumberEntryStyleRealThree, y permite establecer la máxima precisión con la que es posible introducir la variable

[6] `this` corresponde a un apuntador a la instancia de la clase donde es llamado. Por ejemplo, para la ejecutar la simulación que está siendo implementada, debe crearse una instancia de la clase

Ahora, se acomodan los marcos que contienen los pares etiqueta-entrada numérica y el botón de ejecución en el marco vertical `VF1`, y, el lienzo embebido `ECanvas` y `VF1` en la ventana principal, tal como se elaboró sobre la jerarquía de padres e hijos en la sección 2.3.2.

```
47     TGLayoutHints *CentrarEnXY = new TGLayoutHints(kLHintsCenterX|
48                                                    kLHintsCenterY,2,2,2,2);
49     VF1->AddFrame(HFMasa,CentrarEnXY);
50     VF1->AddFrame(HFAltura,CentrarEnXY);
51     VF1->AddFrame(HFGravedad,CentrarEnXY);
52     VF1->AddFrame(PBPlay,CentrarEnXY);
53     fMain->AddFrame(ECanvas, new TGLayoutHints(kLHintsNoHints,
54                                                10,10,10,10));
55     fMain->AddFrame(VF1, new TGLayoutHints(kLHintsExpandX|kLHintsCenterY,
56                                            10,10,10,10));
```

Finalmente, se crea la instancia de `TTimer`, la cual permitirá realizar la animación y se agregan los métodos que permiten mostrar la ventana principal:

```
57     Contador = new TTimer(100);
58     Contador->Connect("Timeout()","CaidaLibreGui",this,"MNumerico()");
59     fMain->MapSubwindows();
60     fMain->Resize();
61     fMain->MapWindow();
62   }
```

Aquí, la asignación de la función *comando* (como se vio en tal sección 2.4.1) se realizó a través de una conexión *signal-slot*. Cada vez que pasa el intervalo de tiempo especificado en el constructor de `TTimer` (en este caso $100ms$), éste emite la señal `Timeout()`, dejando que la conexión de ésta señal tenga como slot el método `MNumerico()`, definido en la clase `CaidaLibreGui`[7].

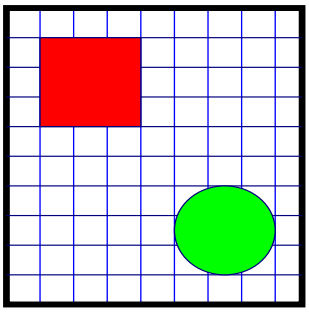
Definido el constructor, es necesario implementar los demás métodos. El método `AjustarRangoCanvas()` que se define a continuación, permite ajustar el rango del lienzo de tal manera que se adapte a la altura que ha determinado el usuario (si se establece un rango fijo y el usuario establece un número fuera de éste rango a la variable altura, la elipse que representa el cuerpo no será mostrada):

```
63   void CaidaLibreGui::AjustarRangoCanvas(Double_t h){
64     Double_t x0,x1,y0,y1,w;
65     y0 = 0.0;
66     y1 = h;
67     w = (Double_t)fCanvas->GetXsizeReal();
68     h = (Double_t)fCanvas->GetYsizeReal();
69     x0 = -(w/h)*(y1-y0)/2.0;
70     x1 = (w/h)*(y1-y0)/2.0;
71     fCanvas->Range(x0,y0,x1,y1);
72   }
```

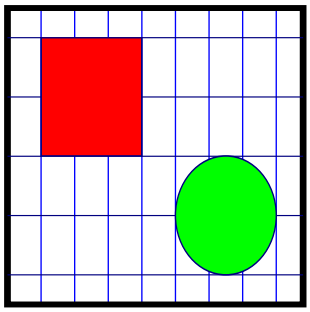
`CaidaLibreGui` (como se verá más adelante); suponga que la instancia de esa clase será llamada `App`, de este modo. `this` es un apuntador a `App`. Fíjese que no es posible reemplazar a `this` por `App` en la conexión, debido a que la instancia `App` no ha sido creada.

[7]El resultado es el mismo que cuando se asignaba una función comando a través del método `SetCommand()`, en la sección 2.4.1

Una función adicional que cumple el método `AjustarRangoCanvas()` es establecer que la relación $\Delta x/\Delta y$ (la cual representa la razón entre la distancia que representa cada píxel en la coordenada horizontal y vertical) sea aproximadamente igual a 1. De este modo, cuando se plasme un círculo en el lienzo, se verá plasmado un círculo en el lienzo y no una elipse. Ésta diferencia se presenta en la figura 3.3.



(a) Un Cuadrado y un Círculo en un lienzo cuya relación $\frac{\Delta x}{\Delta y} = 1$. La grilla permite notar que la distancia representada por cada píxel es la misma tanto en el eje vertical como en el horizontal.



(b) Un Cuadrado y un Círculo Plasmados en un Lienzo cuya Relación $\frac{\Delta x}{\Delta y} \neq 1$. La grilla muestra que la distancia representada por cada píxel es mayor en el eje vertical que en el eje horizontal.

Figura 3.3: Detalles en el ajuste del rango de un lienzo

El siguiente método a implementar es `PBPlayClicked()`. En el constructor de la clase `CaidaLibreGui` se ha establecido que éste método es el *slot* de la señal `Clicked()` del objeto `PBPlay`, es decir, cuando se presione el botón se ejecutará el siguiente bloque de código:

```
73  void CaidaLibreGui::PBPlayClicked(){
74    //Se cargan las variables
75    g = -1*NEGravedad->GetNumber();
76    m = NEMasa->GetNumber();
77    h = NEAltura->GetNumber();
78    dt = 0.1;
79    YP = 0.0;
80    Y = h;
81    AjustarRangoCanvas(h+(0.2*h));
82    fCanvas->Clear();
83    Contador->TurnOn();
84  }
```

donde: se cargan los valores introducidos en cada uno de los objetos de tipo `TGTextEntry` a través del método `GetNumber()`, se ajusta el rango del lienzo a través del método `AjustarRangoCanvas()`, se limpia `fCanvas` a través del método `Clear` (éste método es útil debido a que está provisto que la ejecución de la simulación muestre el rastro del cuerpo, con esto se limpia cualquier rastro de una ejecución previa), y finalmente, se activa el `Contador` mediante la ejecución de `Contador->TurnOn()`.

Para terminar con la implementación de ésta simulación, se define el método `MNumerico()`. De acuerdo a lo establecido en el constructor de la clase `CaidaLibreGui`, cada vez que el objeto `Contador` emita la señal `Timeout()`, se ejecutará, cada $100ms$, el método `MNumerico()` :

```
85  void CaidaLibreGui::MNumerico(){
86    YP += g*dt;
87    Y += YP*dt;
88    NEAltura->SetNumber(Y);
89    //Establecemos el radio del círculo
90    //proporcionales a la altura
91    E1->SetR1(0.05*h);
92    E1->SetR2(0.05*h);
93    E1->SetY1(Y);
94    E1->Draw();
95    E1 = new TEllipse(0,Y,0.05*h,0.05*h);
96    E1->SetFillColor(4);
97    fCanvas->Update();
98    if(Y<=0) Contador->TurnOff();
99  }
100 void CaidaLibreGui::Exit() { gApplication->Terminate(0); }
```

Éste método permite:

1. Calcular la velocidad del cuerpo (a través del método de *Euler*)
2. Calcular la nueva altura del cuerpo de acuerdo a la dinámica del problema.
3. Modificar la entrada numérica `NEAltura` y colocar en ésta el valor que tiene la altura en éste instante.
4. Modificar el radio del círculo
5. Ajustar la posición del círculo con la nueva altura
6. Dibujar el círculo
7. Crear y modificar un nuevo círculo (con el fin de crear un rastro).
8. Actualizar el lienzo para que muestre los cambios
9. Parar el `Contador` si el cuerpo ha alcanzado la altura mínima definida ($0m$)

De ésta forma se ha concluido la definición de la clase `CaidaLibreGui`.

Para ejecutar ésta simulación es necesario crear una instancia de ésta clase, mediante:

```
root[] #include ``CaidaLibreGui.cxx''
root[] fCaidaLibre = new CaidaLibreGui();
```

El resultado se muestra en la figura 3.4.

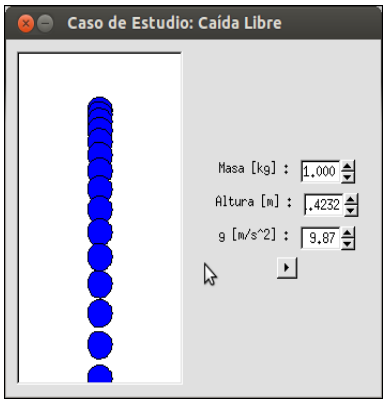


Figura 3.4: Interfaz gráfica para el fenómeno de la caída libre una vez ha sido implementada.

A través de la interfaz pueden modificarse los valores para los parámetros establecidos en el diseño: la masa, la altura y la aceleración de la gravedad. Para iniciar la simulación se debe pulsar el botón con el símbolo de reproducir. Si presiona el botón con una simulación que aún se encuentra en ejecución, el flujo del programa indica que tomará un nuevo valor para la altura inicial (el cual corresponderá al último valor adquirido antes de pulsar el botón) y continuará con la solución numérica del problema. Recuerde que ésta interfaz no cuenta con un mecanismo de pausa ya que es detenida automáticamente una vez la altura adquiera un valor menor o igual a $0{,}0m$. Si el valor es menor que cero no se registrará en la entrada numérica debido a que ésta está restringida a números positivos.

El objeto `Contador` a nivel de la visualización (en el ejemplo implementado) actúa como un estroboscopio, por lo que en caso de que se le asigne un valor pequeño a la altura, es necesario disminuir tanto el intervalo de tiempo que le ha sido asignado (lo que aumenta su frecuencia) como el valor del *paso* (variable `dt` en el código fuente) que ha sido establecido para la solución numérica.

Actividades propuestas

- Modifique el código fuente de la interfaz para incluir una entrada numérica que permita introducir el valor de la variable `dt`, y establezca que el intervalo de tiempo asignado al objeto `Contador` sea proporcional a `dt`. Tenga en cuenta que el valor de `dt` está dado en segundos y el valor asignado al contador está dado en milisegundos.

- Establezca una alternativa para que el usuario pueda pausar (y una vez pausada permita continuar) la simulación utilizando sólo los objetos que ya se encuentran en la interfaz e implementela en el código fuente.

- Tal como fue implementada la simulación, el valor de la masa no modifica la

representación del cuerpo en la animación. Asuma que la densidad de masa del "material" con el cual representamos el cuerpo es constante; en éste caso, un cambio en el valor de la masa implica un cambio directamente proporcional en el radio del cuerpo. Modifique el código fuente de tal manera que un cambio en el valor de la masa implique un cambio en la representación del cuerpo en la animación.

3.2. El movimiento en dos dimensiones de un cuerpo esférico sumergido en un fluido bajo la interacción gravitacional.

Propósito y modelo matemático

Considere ahora un cuerpo esférico que se mueve en dos dimensiones. El cuerpo está sumergido en un fluido y se encuentra bajo la acción de la gravedad. La interfaz gráfica permitirá visualizar el movimiento, además del comportamiento de los vectores velocidad y los diferentes vectores de fuerza. Igualmente, proveerá al usuario la opción de observar el comportamiento en un diagrama de las variables que medien en el experimento tales como la posición, la velocidad, la fuerza de fricción, la energía cinética, la energía mecánica y la energía potencial.

El diagrama de cuerpo libre del problema se muestra en la figura 3.5.

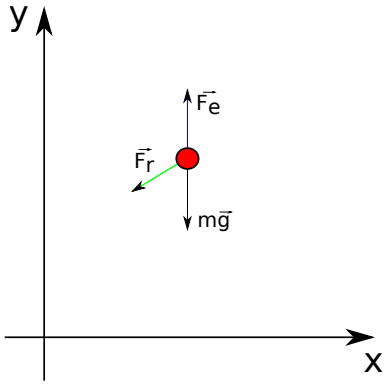


Figura 3.5: Diagrama de cuerpo libre para un cuerpo sumergido en un fluido sometido a una interacción gravitacional

El cuerpo se encuentra sometido a tres fuerzas, la fuerza de empuje $\vec{F}_e$, la fuerza de fricción $\vec{F}_r$, y el peso del cuerpo $m\vec{g}$

$$\sum \vec{F} = \vec{F}_e + \vec{F}_r + m\vec{g}$$

De acuerdo al principio de Arquímedes, la fuerza de empuje es directamente proporcional al peso del volumen del fluido desplazado al sumergir el cuerpo:

$$\vec{F_e} = \rho \left(\frac{4}{3} \pi r^3 \right) g\hat{y}$$

donde ρ es la densidad de masa del medio, r el radio de la esfera y g la aceleración de la gravedad. Se modela la fuerza de fricción de tal manera que su magnitud sea proporcional al cuadrado de la velocidad:

$$\vec{F_r} = -\frac{1}{2} c\rho \left(\pi r^2 \right) v\dot{x}\hat{x} - \frac{1}{2} c\rho \left(\pi r^2 \right) v\dot{y}\hat{y}$$

donde c es una constante adimensional, la cual caracteriza la interacción del cuerpo con el fluido en el que está sumergido (esta constante es conocida como *coeficiente de arrastre*), y v, la magnitud de la velocidad dada por:

$$v = \sqrt{\dot{x}^2 + \dot{y}^2}$$

De este modo, se obtienen dos ecuaciones diferenciales que rigen el movimiento del cuerpo:

$$g \left(\rho \left(\frac{4}{3} \pi r^3 \right) - m \right) - \frac{1}{2} c\rho \left(\pi r^2 \right) v\dot{y} = m\ddot{y}$$

$$-\frac{1}{2} c\rho \left(\pi r^2 \right) v\dot{x} = m\ddot{x}$$

Las ecuaciones presentadas no tienen solución por métodos analíticos debido a su no-linealidad, por lo que se simulará utilizando el método *Runge-Kutta* de cuarto orden.

Parámetros, características y medio de introducción

Evaluando las características de los parámetros de la simulación y las condiciones iniciales del experimento, así como las restricciones a las que estará sujeta la simulación, se determina que:

- La masa m del cuerpo:
  - Cantidad positiva y diferente de cero
  - Precisión de $0{,}001kg$
  - El valor por defecto será $0{,}100kg$
  - Se usará una entrada numérica
  - Corresponderá al grupo "Propiedades del cuerpo"
- El radio r del cuerpo:
  - Cantidad positiva mayor que cero

- Precisión de $0{,}001m$
- El valor por defecto será $0{,}100m$
- Se usará una entrada numérica
- Corresponderá al grupo "Propiedades del cuerpo"

- La densidad ρ del medio :
  - Cantidad positiva mayor que cero
  - Precisión en $0{,}001\frac{kg}{m^3}$
  - El valor por defecto será $0{,}010\frac{kg}{m^3}$
  - Se proporcionará una lista de opciones, sin embargo el usuario tendrá la posibilidad de introducir el valor a través de una entrada numérica
  - Corresponderá al grupo "Propiedades del medio"
- El coeficiente c de arrastre:
  - Número positivo mayor que cero
  - El valor por defecto será $0{,}500$
  - Será introducido a través de una entrada numérica.
  - Corresponderá al grupo "Propiedades del medio"
- El valor de la aceleración gravitacional g:
  - Cantidad positiva mayor que cero
  - Precisión de $0{,}001\frac{m}{s^2}$
  - El valor por defecto será $9{,}780\frac{m}{s^2}$
  - Se proporcionará una lista de opciones que incluyen los planetas del sistema solar, la luna terrestre y el sol. No obstante, el usuario tendrá la posibilidad de introducir el valor a través de una entrada numérica
  - Corresponderá al grupo "Parámetros de la simulación"
- El valor del paso dt:
  - Cantidad positiva y diferente cero
  - El valor por defecto será $5ms$
  - Se usará una entrada numérica para su ingreso
  - Corresponderá al grupo "Parámetros de la simulación"

- La posición inicial en el eje horizontal $x(t=0)$:
  - Una cantidad real
  - Precisión de $0{,}001m$
  - El valor por defecto será $2{,}000m$
  - Se usará una entrada numérica para su ingreso
  - Corresponderá al grupo "Condiciones iniciales"
- La posición inicial en el eje vertical $y(t=0)$:
  - Cantidad real
  - Precisión de $0{,}001mm$
  - El valor por defecto será $1{,}000m$
  - Se usará una entrada numérica para su ingreso
  - Corresponderá al grupo "Condiciones iniciales"
- La velocidad inicial $\vec{v}$ definida:
  - El valor de la rapidez inicial $v(t=0)$:
    - Cantidad positiva o cero
    - El valor por defecto será $20\frac{m}{s}$
    - Se usará una entrada numérica para su ingreso
  - El ángulo del lanzamiento α:
    - El valor por defecto será $80°$
    - Se usará una entrada numérica para su ingreso
  - Corresponderá al grupo "Condiciones iniciales"

Adicionalmente, se agregará la opción de mostrar los vectores de fuerza y velocidad en la animación, así como la posibilidad de mostrar el rastro del cuerpo. Antes de iniciar la simulación el usuario debe seleccionar el par coordenado de variables que desea visualizar. Entre las variables que pueden ser representadas en el diagrama figurarán las componentes de la posición, el tiempo, la energía mecánica, la energía cinética y la energía potencial, así como las componentes y la magnitud de la fuerza de fricción y de la velocidad.

Esquema de la interfaz

El esquema de la interfaz gráfica se muestra en la figura 3.6.

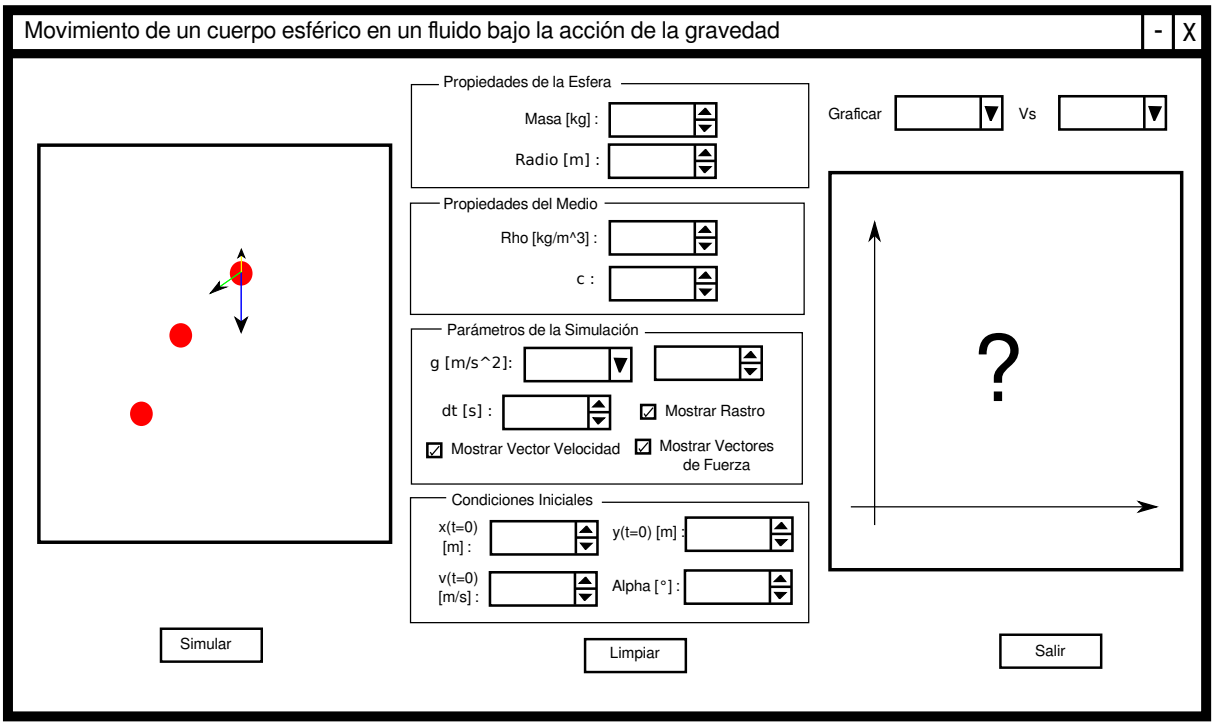


Figura 3.6: Esquema de la interfaz gráfica para la simulación del movimiento de un cuerpo esférico en un fluido bajo la acción de la gravedad.

En el esquema realizado en la figura 3.6, es posible ver que la simulación contendrá una animación que representará la posición del cuerpo en el espacio (izquierda), los diferentes objetos que permiten para la configuración de las variables de la simulación (centro) y un diagrama (derecha) cuyas variables dependientes e independientes pueden ser establecidas a través de dos listas de selección ubicadas en la esquina superior derecha.

Comportamiento de la interfaz

La interfaz gráfica permitirá al usuario seleccionar qué variables desea visualizar en el diagrama (ubicada en el costado derecho de la figura 3.6) así como si desea o no plasmar los vectores de fuerza o velocidad.

Tan pronto el usuario modifica el valor del radio o la posición inicial, éste cambio se verá reflejado en el lienzo que contiene la simulación, para lo cual se implementarán los métodos `NERadioValueSet()`, `NEY0ValueSet()`, y `NEX0ValueSet()`. Estos métodos serán los *slots* que responderán a la señal `ValueSet(Long_t)`, la cual es emitida cuando el valor de una entrada numérica ha sido modificado.

Cuando el usuario selecciona las variables del gráfico se emite la señal `Selected(Int_t)`, la cual es conectada a los métodos `CBYGraphSelected(Int_t)` y `CBXGraphSelected(Int_t)` que permiten asignar a un apuntador la posición en memoria de la variable seleccionada.

Debido a que el usuario cuenta con dos métodos de ingresar las variables densidad ρ y aceleración de la gravedad g (pueden ser seleccionadas de una lista o introducidas a través de una entrada numérica), se usará únicamente el valor de la entrada numérica, es decir, si el usuario selecciona una opción de la lista, ésta acción modificará el valor de la entrada numérica de acuerdo a la opción de la lista seleccionada por el usuario y la edición manual de la entrada se habilitará únicamente si ha sido seleccionada la opción "Otro..." de la lista. Lo anterior se hará implementando los *slots* `CBRhoSelected(Int_t Opt)` y `CBGrSelected(Int_t Opt)` que responderán a la señal `Selected(Int_t)` de la lista de selección de la densidad y la gravedad respectivamente.

Como se mencionó anteriormente, la lista de selección de la aceleración de la gravedad contendrá los valores para los planetas del sistema solar, mientras que la lista de selección de la densidad contendrá valores de la densidad del Aire a $0°$C , del oxígeno y del helio a $20°$C, y del vapor de agua, a $1atm$ de presión.

El diagrama de flujo presentado en la figura 3.7 muestra que la apertura de la simulación crea la interfaz gráfica. Para ésta simulación existen tres estados:

- Simulación no iniciada: implica que no se ha realizado la inicialización de las variables, representada por el número -1.
- Simulación pausada: implica que ya se realizó la inicialización de variables y que se puede continuar la simulación desde su último estado mecánico, representada por el número 0.
- Simulación en proceso: la simulación se encuentra en estado activo, ejecutando la rutina de solución numérica representada por el número 1.

El estado de la simulación será guardado en una variable de tipo `Int_t` llamada `IsActive`.

Una vez el usuario presiona el botón Simular (ver figura 3.6): se inicializará el valor de las variables con los datos suministrados por el usuario, y se establecerá el valor de la variable `IsActive` en 1 (Simulación en proceso), se desactivará la edición de las entradas numéricas (esto se implementará en el método llamado `Disable()`) y se activará el contador, ésta subrutina se implementará en el método `TBPlayClicked()`.

La activación del contador iniciará la solución numérica con el fin de establecer la posición del cuerpo (resolviendo las ecuaciones diferenciales, que rigen la dinámica del fenómeno, a través del método Runge-Kutta de cuarto orden) y la animación. En la animación se verificarán qué opciones de visualización fueron seleccionadas por el usuario (por ejemplo si seleccionó mostrar el rastro de la posición del cuerpo o los vectores de fuerza/velocidad) y se ejecutarán los procedimientos necesarios para llevar a cabo la animación y la gráfica de las variables seleccionadas para el diagrama. Los

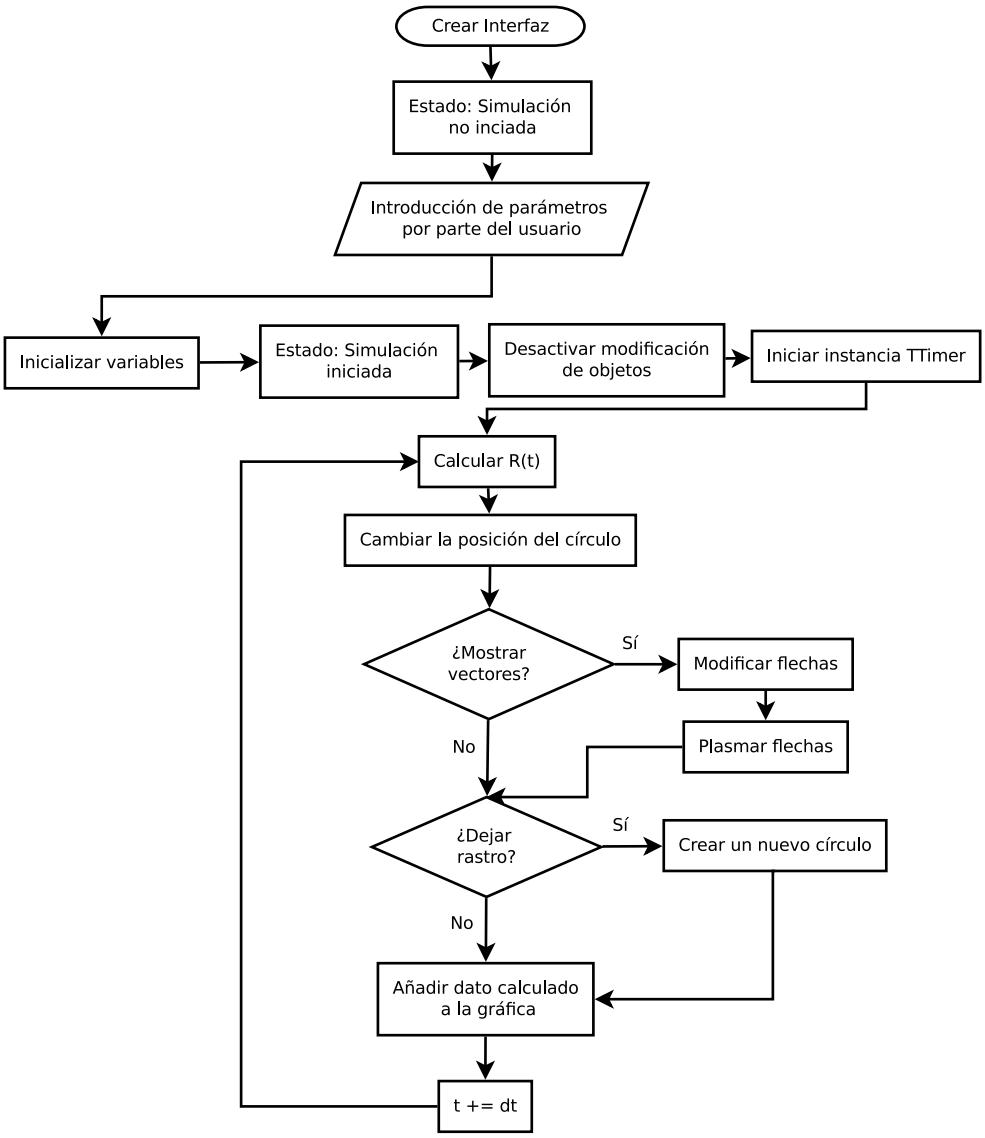


Figura 3.7: Diagrama de flujo: simulación de un movimiento de un cuerpo sumergido en un fluido bajo la acción de la gravedad

procedimientos requeridos para la solución numérica y para la visualización (animación y diagrama) se implementarán en el método llamado `DoDraw()`.

Una vez la simulación es iniciada, el texto de botón que contiene la palabra "Simular" (ver figura 3.6) se cambiará por "Pausar". Si el usuario presiona el botón pausar, se detendrá el contador y el valor de la variable `IsActive` sera 0 (Simulación pausada), en este instante se cambiará el texto del botón a "Continuar".

Si el botón limpiar es presionado: se detendrá el contador, se activará la edición de las entradas numéricas, se creará un nuevo diagrama, se limpiaran los lienzos y se establecerá el estado de la simulación en -1 (simulación no iniciada). Esta subrutina se implementará en el método `TBClearClicked()`.

Al presionar el botón salir, se detendrá el contador y se cerrará la aplicación.

Implementación

Para iniciar la implementación de la simulación, se incluyen las cabeceras necesarias para poder utilizar los diferentes elementos que se encuentran en el esquema de la interfaz[8]:

```
1  #ifndef __NPM__
2  #define __NPM__
3
4  #include "TGClient.h"
5  #include "RQ_OBJECT.h"
6  #include "TGButton.h"
7  #include "TGFrame.h"
8  #include "TEllipse.h"
9  #include "TArrow.h"
10 #include "TTimer.h"
11 #include "TRootEmbeddedCanvas.h"
12 #include "TCanvas.h"
13 #include "TGNumberEntry.h"
14 #include "TGComboBox.h"
15 #include "TGraph.h"
16 #include "TGLabel.h"
17 #include "TMath.h"
```

Se da lugar a la declaración de la clase que definirá la interfaz gráfica para la simulación, la cual será llamada `WMainNPM`, declarando cada uno de los objetos que se incluye en la interfaz:

```
18 class WMainNPM {
19   RQ_OBJECT("WMainNPM");
20   //Elementos de la interfaz
21   TGMainFrame *fMain; //Ventana principal
22   TRootEmbeddedCanvas *ECGraph; //Lienzo para las gráficas
23   TCanvas *CGPos;
24   TRootEmbeddedCanvas *ECGPos; //Lienzo para la animación
25   TCanvas *CGraph;
26   TGNumberEntry *NEMass; //Entrada numérica para la masa
27   TGNumberEntry *NERadio; // para el radio
28   TGNumberEntry *NERho; // para la densidad ρ
29   TGNumberEntry *NECPull; // para el coeficiente de arrastre
30   TGNumberEntry *NEGr; // para la gravedad
31   TGNumberEntry *NEDt; // para el intervalo de tiempo
32   TGNumberEntry *NEX0; // para la posición inicial en x
33   TGNumberEntry *NEY0; // para la posición inicial en y
34   TGNumberEntry *NEV0; // para la rapidez inicial
35   TGNumberEntry *NEAlpha; // para el ángulo del lanzamiento
36   TGComboBox *CBRho; //Lista de selección para la densidad ρ
37   TGComboBox *CBGr; //Lista de selección para la gravedad g
38   TGraph *GGraph; //Gráfica
39   TEllipse *EBody; //Elipse que representará el cuerpo
40   TGTextButton *TBPlay; //Botón para iniciar y pausar la simulación
41   TGTextButton *TBClear; //Botón para limpiar la ejecución
42   TGCheckButton *CHBRastro; //Opción para mostrar (o no) el rastro
43   TGCheckButton *CHBVVel; //Opción para mostrar (o no) el vector v⃗
44   //Opción para mostrar (o no) los vectores de fuerza:
```

[8]El archivo de cabecera de ésta clase se encontrará en `WMainNPM.h`

```
45     TGCheckButton *CHBVForce;
46     //Flechas que representan el vector velocidad
47     //y sus componentes
48     TArrow *AV,*AXP,*AYP;
49     //Flechas que representan las fuerzas que actúan
50     //sobre el cuerpo
51     TArrow *AW,*AFe,*AFrx,*AFry,*AFr;
```

Recuerde que los objetos que no han sido declarados bajo ninguna directiva (`private`, `protected` o `public`) son considerados privados por defecto. Se continua con la declaración de las variables que intervendrán en el cálculo numérico y que requieren globalidad de clase:

```
52     //Variables para la solución numérica
53     TTimer *Cont; //Contador
54     Double_t m; //m = masa del cuerpo
55     Double_t g; //g = valor de la aceleración gravitacional
56     Double_t Rho; //ρ = densidad del medio
57     Double_t R; //R = Radio del cuerpo
58     Double_t c; //c = coeficiente de arrastre
59     Double_t Alpha; //α = ángulo de lanzamiento
60     Double_t FW; //peso
61     Double_t FFe; //Fuerza de empuje
62     Double_t FFrx; //Componente en x de la fricción
63     Double_t FFry; //Componente en y de la fricción
64     Double_t FFr; //Fuerza de fricción (Resultante)
65     Double_t X; // X(t) = posición en el eje x del centro de masa
66     Double_t XP; // Ẋ(t) = componente en x de la velocidad
67     Double_t Y; // Y(t) =  posición en el eje y del centro de masa
68     Double_t YP; // Ẏ(t) = componente en x de la velocidad
69     Double_t t; // tiempo
70     Double_t dt; // paso (intervalo de tiempo)
71     Double_t V; // magnitud de v⃗
72     Double_t p; // magnitud de p⃗
73     Double_t U; //Energía potencial
74     Double_t K; //Energía cinética
75     Double_t E; //Energía mecánica
```

Se agregan unos apuntadores y unas variables auxiliares:

```
76     //Objetos auxiliares
77     //Apuntadores a variables dependiente e independiente del gráfico:
78     Double_t *GX,*GY;
79     //Variable auxiliar para determinar el estado de la simulación
80     Int_t IsActive;
81     //Funciones
82     //función que calcula el módule de un vector bidimensional
83     Double_t Mod(Double_t CX,Double_t CY);
84     Double_t XPP(Double_t, Double_t); //Ẍ(t)
85     Double_t YPP(Double_t, Double_t); //Ÿ(t)
86     //Funciones auxiliares
87     void Disable();
88     void Enable();
```

Se finaliza la declaración de la clase con los métodos públicos, donde se encuentran los *slots* requeridos para realizar la conexión de los elementos de la interfaz con su comportamiento:

```
87   public:
88     WMainNPM();
```

```
89     ~WMainNPM(){};
90     //Slots
91     void CBGrSelected(Int_t Opt);
92     void NERadioValueSet();
93     void NEY0ValueSet();
94     void NEX0ValueSet();
95     void CBXGraphSelected(Int_t);
96     void CBYGraphSelected(Int_t);
97     void CBRhoSelected(Int_t);
98     void TBClearClicked();
99     void TBExitClicked();
100    void TBPlayClicked();
101    void DoDraw();
102    ClassDef(WMainNPM,0);
103  };
104  #endif //__NPM__//
```

Una vez finaliza la declaración de la clase, es necesario definir cada uno de los métodos, iniciando por el constructor[9].

En el esquema ilustrado en la figura 3.6 es posible identificar tres columnas: la primera, contiene el lienzo de la animación; la segunda, contiene todas las opciones de configuración de los parámetros de la simulación y las opciones de la visualización; y la tercera, contiene el diagrama y los objetos que permiten establecer qué variables serán graficadas. Bajo las tres columnas, se encuentran en una misma fila los botones simular, limpiar y salir. Se construirán tres marcos: el primero, será un marco horizontal que contendrá las tres columnas; el segundo será un marco horizontal que contendrá los botones y el tercer marco de tipo vertical, contendrá en su interior a los dos marcos anteriores (de acuerdo a lo visto en la sección 2.3.2):

```
1   #include "WMainNPM.h"
2   #include "TApplication.h"
3
4   ClassImp(WMainNPM);
5
6   WMainNPM::WMainNPM(){
7
8     //Creación y configuración de la ventana principal
9     fMain = new TGMainFrame(gClient->GetRoot(),980,380);
10    fMain->Connect("CloseWindow()","WMainNPM",this,"TBExitClicked()");
11    fMain->SetWMSizeHints(980,380,980,380,0,0);
12    fMain->SetCleanup(kDeepCleanup);
13    fMain->SetWindowName("Movimiento No Tan Parab\363lico");
14
15    //Marco general
16    TGVerticalFrame *VFMain = new TGVerticalFrame(fMain);
17    //Marco que contendrá las 3 columnas
18    TGHorizontalFrame *VF0 = new TGHorizontalFrame(VFMain);
19
20    //Unas indicaciones de diseño generales
21    TGLayoutHints *LayoutA = new TGLayoutHints(kLHintsCenterX|
22                                               kLHintsCenterY,2,2,2,2);
23    TGLayoutHints *LayoutB = new TGLayoutHints(kLHintsExpandX,0,0,0,0);
```

Se implementa la primera columna (que contiene el lienzo de la animación), aprovechando este segmento para crear todo lo relacionado con visualización del movimiento del cuerpo:

[9]La definición de los diferentes métodos de ésta clase se encontrará en el archivo `WMainNPM.cxx`

```
24    //Columna1
25    //Lienzo que contendrá la animación
26    TGVerticalFrame *VFCol1 = new TGVerticalFrame(VF0);
27    ECGPos = new TRootEmbeddedCanvas("ECGPos",VFCol1,300,300);
28    CGPos = ECGPos->GetCanvas();
29    CGPos->Range(0,0,10,10);
30    //Círculo que representará el cuerpo
31    EBody = new TEllipse();
32    EBody->SetFillColor(2);
```

Se crea una flecha (instancia de `TArrow`) de color rojo, que representará el vector velocidad y dos líneas punteadas que mostrarán las componentes:

```
33    AV = new TArrow();
34    AV->SetLineColor(2);
35    AV->SetLineWidth(2);
36    AV->SetArrowSize(0.04);
37    AV->SetOption(">");
38    //Componente en x
39    AXP = new TArrow();
40    AXP->SetLineColor(2);
41    //Componentes punteadas
42    AXP->SetLineStyle(3);
43    //Componente en y
44    AYP = new TArrow();
45    AYP->SetLineColor(2);
46    AYP->SetLineStyle(3);
```

una flecha de color negro para el vector peso:

```
47    AW = new TArrow();
48    AW->SetLineWidth(2);
49    AW->SetArrowSize(0.04);
50    AW->SetOption(">");
```

una flecha de color verde para el vector de la fuerza de empuje:

```
51    AFe = new TArrow();
52    AFe->SetArrowSize(0.04);
53    AFe->SetOption(">");
54    AFe->SetLineWidth(2);
55    AFe->SetLineColor(3);
```

una flecha de color azul para la fricción y sus componentes en líneas punteadas:

```
56    //Componente  en x de la fricción
57    AFrx = new TArrow();
58    AFrx->SetLineColor(4);
59    AFrx->SetLineStyle(3);
60    //Componente  en y de la fricción
61    AFry = new TArrow();
62    AFry->SetLineStyle(3);
63    AFry->SetLineColor(4);
64    //Fuerza de Fricción
65    AFr = new TArrow();
66    AFr->SetArrowSize(0.04);
67    AFr->SetOption(">");
68    AFr->SetLineColor(4);
69    AFr->SetLineWidth(2);
```

y se agrega el lienzo al padre `VFCol1` centrado a través de la indicación de diseño `LayoutA`:

```
70      VFCol1->AddFrame(ECGPos,LayoutA);
71      //Agregamos la columna al contenedor
72      VF0->AddFrame(VFCol1);
```

La segunda columna es implementada, iniciando con el grupo de variables que caracterizan el cuerpo. Se agrega uno a uno los elementos encontrados en éste grupo y se configuran las entradas numéricas para asignar las respectivas restricciones:

```
73      //Columna 2
74      TGVerticalFrame *VFCol2 = new TGVerticalFrame(VF0);
75      //Variables del cuerpo
76      TGGroupFrame *GFBody = new TGGroupFrame(VFCol2,
77                                              "Propiedades Esfera");
78      TGHorizontalFrame *HF2 = new TGHorizontalFrame(GFBody);
79      TGLabel *LMass = new TGLabel(HF2,"Masa [kg]:");
80      NEMass = new TGNumberEntry(HF2);
81      NEMass->SetNumStyle(TGNumberFormat::kNESRealThree);
82      NEMass->SetNumAttr(TGNumberFormat::kNEAPositive);
83      NEMass->SetNumber(0.100);
84      NEMass->Resize(70,20);
85      HF2->AddFrame(LMass,LayoutA);
86      HF2->AddFrame(NEMass,LayoutA);
87      TGHorizontalFrame *HF3 = new TGHorizontalFrame(GFBody);
88      TGLabel *LRadio = new TGLabel(HF3,"Radio [m]:");
89      NERadio = new TGNumberEntry(HF3);
90      NERadio->Resize(70,20);
91      NERadio->SetNumStyle(TGNumberFormat::kNESRealThree);
92      NERadio->SetNumAttr(TGNumberFormat::kNEAPositive);
93      NERadio->SetNumber(0.1);
94      NERadio->Connect("ValueSet(Long_t)","WMainNPM",
95                       this,"NERadioValueSet()");
96      HF3->AddFrame(LRadio,LayoutA);
97      HF3->AddFrame(NERadio,LayoutA);
98      GFBody->AddFrame(HF2,LayoutA);
99      GFBody->AddFrame(HF3,LayoutA);
100     VFCol2->AddFrame(GFBody,LayoutB);
```

El segundo grupo de la columna 2 contiene las variables que caracterizan al medio en el cual se encuentra sumergido el cuerpo:

```
101     //Variables del medio
102     TGGroupFrame *GFMed = new TGGroupFrame(VFCol2,
103                                            "Propiedades del Medio");
```

La densidad ρ puede ser introducida a través de una una entrada numérica y una lista de selección:

```
104     TGHorizontalFrame *HF4 = new TGHorizontalFrame(GFMed);
105     TGLabel *LRho = new TGLabel(HF4,"Rho [kg/m^3]:");
106     NERho = new TGNumberEntry(HF4);
107     NERho->SetNumStyle(TGNumberFormat::kNESRealThree);
108     NERho->SetNumAttr(TGNumberFormat::kNEAPositive);
109     NERho->SetNumber(0.010);
110     NERho->Resize(70,20);
111     CBRho = new TGComboBox(HF4);
112     CBRho->AddEntry("Aire 0\260C 1atm",0);//1.293
113     CBRho->AddEntry("Aire 20\260C 1atm",1);//1.2045
114     CBRho->AddEntry("Oxigeno 20\260C 1atm",2);//1.331
115     CBRho->AddEntry("Helio 20\260C 1atm",3);//0.1664
116     CBRho->AddEntry("Vapor de Agua",4);//0.804
117     CBRho->AddEntry("Otro...",5);
```

```
118     CBRho->Connect("Selected(Int_t)","WMainNPM",
119                    this,"CBRhoSelected(Int_t)");
120     CBRho->Select(1);
121     CBRho->Resize(120,20);
122     HF4->AddFrame(LRho,LayoutA);
123     HF4->AddFrame(CBRho,LayoutA);
124     HF4->AddFrame(NERho,LayoutA);
```

y el coeficiente de arrastre c a través de una entrada numérica:

```
125     TGHorizontalFrame *HF5 = new TGHorizontalFrame(GFMed);
126     TGLabel *LCPull = new TGLabel(HF5,"c : ");
127     NECPull = new TGNumberEntry(HF5);
128     NECPull->SetNumStyle(TGNumberFormat::kNESRealThree);
129     NECPull->SetNumber(0.5);
130     HF5->AddFrame(LCPull,LayoutA);
131     HF5->AddFrame(NECPull,LayoutA);
```

y se agregan los elementos a su respectivo padre:

```
132     GFMed->AddFrame(HF4,LayoutB);
133     GFMed->AddFrame(HF5,LayoutB);
134     VFCol2->AddFrame(GFMed,LayoutB);
```

El tercer grupo de la columna 2 (figura 3.6) contiene el valor de la gravedad, las opciones de la animación, y el paso o intervalo de tiempo utilizado para la solución mediante el método numérico. Inicialmente se crea una instancia de `TGGroupFrame`:

```
135     //Parámetros simulación
136     TGGroupFrame *GFPar = new TGGroupFrame(VFCol2,
137                                            "Par\341metros Simulaci\363n");
```

Se crean y configuran los objetos que permiten introducir la aceleración de la gravedad g (una entrada numérica y una lista de selección),

```
138     TGHorizontalFrame *HF6 = new TGHorizontalFrame(GFPar);
139     TGLabel *LGr = new TGLabel(HF6,"g [m/s^2]: ");
140     NEGr = new TGNumberEntry(HF6);
141     NEGr->Resize(70,20);
142     NEGr->SetNumStyle(TGNumberFormat::kNESRealThree);
143     NEGr->SetNumAttr(TGNumberFormat::kNEAPositive);
144     NEGr->SetNumber(TMath::Gn());
145     NEGr->SetState(kFALSE);
146     CBGr = new TGComboBox(HF6);
147     CBGr->AddEntry("Sol (274.0)",0);
148     CBGr->AddEntry("Mercurio (3.701)",1);
149     CBGr->AddEntry("Venus (8.870)",2);
150     CBGr->AddEntry("Tierra (9.780)",3);
151     CBGr->AddEntry("Luna (1.62)",4);
152     CBGr->AddEntry("Marte (3.690)",5);
153     CBGr->AddEntry("Jupiter (23.12)",6);
154     CBGr->AddEntry("Saturno (8.96)",7);
155     CBGr->AddEntry("Urano (8.69)",8);
156     CBGr->AddEntry("Neptuno (11.00)",9);
157     CBGr->AddEntry("Otro... ",10);
158     CBGr->Connect("Selected(Int_t)","WMainNPM",
159                   this,"CBGrSelected(Int_t)");
160     CBGr->Select(3);
161     CBGr->Resize(120,20);
162     HF6->AddFrame(LGr,LayoutA);
163     HF6->AddFrame(CBGr,LayoutA);
164     HF6->AddFrame(NEGr,LayoutA);
165     GFPar->AddFrame(HF6,LayoutB);
```

el intervalo de tiempo dt mediante una entrada numérica,

```
166     TGHorizontalFrame *HF7 = new TGHorizontalFrame(GFPar);
167     TGLabel *LDt = new TGLabel(HF7,"dt [s]:");
168     NEDt = new TGNumberEntry(HF7);
169     NEDt->SetNumStyle(TGNumberFormat::kNESRealThree);
170     NEDt->SetNumAttr(TGNumberFormat::kNEAPositive);
171     NEDt->SetNumber(0.005);
172     HF7->AddFrame(LDt,LayoutB);
173     HF7->AddFrame(NEDt,LayoutB);
174     GFPar->AddFrame(HF7,LayoutB);
```

Además de los objetos que han sido implementados en otras secciones, a continuación se encuentra una instancia de la clase `TGCheckButton` que permite usar un botón de verificación. El primer argumento del constructor de ésta clase es el padre del objeto y el segundo es el texto del botón (ver figura 3.6). Por medio del método `SetState()` es posible fijar el estado del botón, es decir, marcarlo o desmarcarlo con las opciones `kButtonDown` o `kButtonUp` respectivamente:

```
175     CHBRastro = new TGCheckButton(HF7,"Mostrar Rastro");
176     CHBRastro->SetState(kButtonDown);
177     HF7->AddFrame(CHBRastro,LayoutA);
178     TGHorizontalFrame *HF10 = new TGHorizontalFrame(GFPar);
179     CHBVVel = new TGCheckButton(HF10,"Mostrar Vector Velocidad");
180     CHBVVel->SetState(kButtonDown);
181     CHBVForce = new TGCheckButton(HF10,"Mostrar Vectores de Fuerza");
182     CHBVForce->SetState(kButtonDown);
183     HF10->AddFrame(CHBVVel,LayoutA);
184     HF10->AddFrame(CHBVForce,LayoutA);
185     GFPar->AddFrame(HF10,LayoutB);
186     VFCol2->AddFrame(GFPar,LayoutB);
```

El último grupo de la 2 columna (figura 3.6) corresponde a los valores de las condiciones iniciales:

```
187     //Condiciones Iniciales
188     TGGroupFrame *GFInitCond = new TGGroupFrame(VFCol2,
189                                                 "Condiciones Iniciales");
```

Se crean y configuran los objetos que permiten introducir la posición inicial, (entradas numéricas para el eje horizontal x y el eje vertical y),

```
190     TGHorizontalFrame *HF0 = new TGHorizontalFrame(GFInitCond);
191     TGLabel *LX0 = new TGLabel(HF0,"x(t=0) [m]:");
192     NEX0 = new TGNumberEntry(HF0);
193     NEX0->SetNumStyle(TGNumberFormat::kNESRealThree);
194     NEX0->SetNumber(2.0);
195     NEX0->Resize(70,20);
196     NEX0->Connect("ValueSet(Long_t)","WMainNPM",
197                   this,"NEX0ValueSet()");
198     TGLabel *LY0 = new TGLabel(HF0,"y(t=0) [m]:");
199     NEY0 = new TGNumberEntry(HF0);
200     NEY0->Resize(70,20);
201     NEY0->SetNumStyle(TGNumberFormat::kNESRealThree);
202     NEY0->SetNumber(1.0);
203     NEY0->Connect("ValueSet(Long_t)","WMainNPM",
204                   this,"NEY0ValueSet()");
205     HF0->AddFrame(LX0,LayoutA);
206     HF0->AddFrame(NEX0,LayoutA);
207     HF0->AddFrame(LY0,LayoutA);
208     HF0->AddFrame(NEY0,LayoutA);
209     GFInitCond->AddFrame(HF0,LayoutB);
```

la velocidad inicial mediante entradas numéricas para el eje horizontal x, el eje vertical y, y el ángulo α del vector respecto a la horizontal :

```
210     TGHorizontalFrame *HF1 = new TGHorizontalFrame(GFInitCond);
211     TGLabel *LV0 = new TGLabel(HF1,"v(t=0) [m/s]:");
212     NEV0 = new TGNumberEntry(HF1);
213     NEV0->Resize(70,20);
214     NEV0->SetNumStyle(TGNumberFormat::kNESRealThree);
215     NEV0->SetNumAttr(TGNumberFormat::kNEAPositive);
216     NEV0->SetNumber(20.0);
217     TGLabel *LAlpha = new TGLabel(HF1,"Alpha [\260]:");
218     NEAlpha = new TGNumberEntry(HF1);
219     NEAlpha->SetNumStyle(TGNumberFormat::kNESRealThree);
220     NEAlpha->SetNumber(80.0);
221     NEAlpha->Resize(70,20);
222     HF1->AddFrame(LV0,LayoutA);
223     HF1->AddFrame(NEV0,LayoutA);
224     HF1->AddFrame(LAlpha,LayoutA);
225     HF1->AddFrame(NEAlpha,LayoutA);
226     GFInitCond->AddFrame(HF1,LayoutB);
227     VFCol2->AddFrame(GFInitCond,LayoutB);
228     VF0->AddFrame(VFCol2);
```

La tercera columna (figura 3.6) contiene las opciones de configuración para la generación del gráfico de variables:

```
229     //Columna 3
230     TGVerticalFrame *VFCol3 = new TGVerticalFrame(VF0);
```

Se crean y configura una lista de selección que permite introducir la variable dependiente (en el eje vertical y):

```
231     TGHorizontalFrame *HF8 = new TGHorizontalFrame(VFCol3);
232     TGLabel *LGraph = new TGLabel(HF8,"Graficar ");
233     TGComboBox *CBYGraph = new TGComboBox(HF8);
234     CBYGraph->Connect("Selected(Int_t)","WMainNPM",
235                       this,"CBYGraphSelected(Int_t)");
236     CBYGraph->AddEntry("x [m]",0);
237     CBYGraph->AddEntry("y [m]",1);
238     CBYGraph->AddEntry("t [s]",2);
239     CBYGraph->AddEntry("U [jul]",3);
240     CBYGraph->AddEntry("K [jul]",4);
241     CBYGraph->AddEntry("E [jul]",5);
242     CBYGraph->AddEntry("V [m/s]",6);
243     CBYGraph->AddEntry("Vx [m/s]",7);
244     CBYGraph->AddEntry("Vy [m/s]",8);
245     CBYGraph->AddEntry("Frx [N]",9);
246     CBYGraph->AddEntry("Fry [N]",10);
247     CBYGraph->AddEntry("Fr [N]",11);
248     CBYGraph->Select(1);
249     CBYGraph->Resize(80,20);
```

Se crea y configura la lista de selección para la variable independiente (en el eje horizontal x):

```
250     TGLabel *LVs = new TGLabel(HF8," Vs ");
251     TGComboBox *CBXGraph = new TGComboBox(HF8);
252     CBXGraph->Connect("Selected(Int_t)","WMainNPM",
253                       this,"CBXGraphSelected(Int_t)");
254     CBXGraph->AddEntry("x [m]",0);
255     CBXGraph->AddEntry("y [m]",1);
```

```
256     CBXGraph->AddEntry("t [s]",2);
257     CBXGraph->AddEntry("U [jul]",3);
258     CBXGraph->AddEntry("K [jul]",4);
259     CBXGraph->AddEntry("E [jul]",5);
260     CBXGraph->AddEntry("V [m/s]",6);
261     CBXGraph->AddEntry("Vx [m/s]",7);
262     CBXGraph->AddEntry("Vy [m/s]",8);
263     CBXGraph->AddEntry("Frx [N]",9);
264     CBXGraph->AddEntry("Fry [N]",10);
265     CBXGraph->AddEntry("Fr [N]",11);
266     CBXGraph->Select(0);
267     CBXGraph->Resize(80,20);
```

Se añaden los hijos a sus respectivos padres utilizando la indicación de diseño `LayoutA`:

```
268     HF8->AddFrame(LGraph,LayoutA);
269     HF8->AddFrame(CBYGraph,LayoutA);
270     HF8->AddFrame(LVs,LayoutA);
271     HF8->AddFrame(CBXGraph,LayoutA);
```

Se configura el lienzo donde estará dispuesta la gráfica :

```
272     ECGraph = new TRootEmbeddedCanvas("ECGPos",VFCol3,300,300);
273     CGraph = ECGraph->GetCanvas();
274     VFCol3->AddFrame(HF8,LayoutA);
275     VFCol3->AddFrame(ECGraph,LayoutA);
276     VF0->AddFrame(VFCol3);
```

Se crea y configura el marco que contiene los botones de control de la simulación:

```
277     TGHorizontalFrame *HF9 = new TGHorizontalFrame(VFMain);
278     TBPlay = new TGTextButton(HF9,"&Simular");
279     TBPlay->Connect("Clicked()","WMainNPM",this,"TBPlayClicked()");
280     TBClear = new TGTextButton(HF9,"&Limpiar");
281     TBClear->Connect("Clicked()","WMainNPM",this,"TBClearClicked()");
282     TGTextButton *TBExit = new TGTextButton(HF9,"S&alir");
283     TBExit->Connect("Clicked()","WMainNPM",this,"TBExitClicked()");
284     HF9->AddFrame(TBPlay,LayoutA);
285     HF9->AddFrame(TBClear,LayoutA);
286     HF9->AddFrame(TBExit,LayoutA);
```

Se añaden el marco y las columnas en el contenedor principal y se ejecutan los métodos para hacer visible la ventana:

```
287     VFMain->AddFrame(VF0,new TGLayoutHints(kLHintsExpandX,
288                                            1,1,1,1));
289     VFMain->AddFrame(HF9,new TGLayoutHints(kLHintsExpandX|
290                                            kLHintsExpandY,1,1,1,1));
291     fMain->AddFrame(VFMain,new TGLayoutHints(kLHintsExpandX|
292                                              kLHintsExpandY,1,1,1,1));
293     fMain->MapSubwindows();
294     fMain->Resize();
295     fMain->MapWindow();
```

La escritura del constructor finaliza con la creación del contador y el establecimiento del estado de la simulación.

```
296     Cont = new TTimer(100);
297     Cont->Connect("Timeout()","WMainNPM",this,"DoDraw()");
298     GGraph = new TGraph();
299     IsActive = -1;
300   }
```

Se implementa el *slot* que está ligado a la modificación del valor del radio, esto se presentará de manera directa en el lienzo de la animación:

```
301 void WMainNPM::NERadioValueSet(){
302   EBody->SetR1(NERadio->GetNumber());
303   EBody->SetR2(NERadio->GetNumber());
304   CGPos->cd();
305   EBody->Draw();
306   CGPos->Update();
307 }
```

Se implementan los *slots* que están ligados a la modificación del valor de la posición; esto se presentará de manera directa en el lienzo de la animación:

```
308 void WMainNPM::NEY0ValueSet(){
309   EBody->SetY1(NEY0->GetNumber());
310   CGPos->cd();
311   EBody->Draw();
312   CGPos->Update();
313 }
314 void WMainNPM::NEX0ValueSet(){
315   EBody->SetX1(NEX0->GetNumber());
316   CGPos->cd();
317   EBody->Draw();
318   CGPos->Update();
319 }
```

Se realiza la implementación del *slot* que responde a la selección del diagrama por parte del usuario de la variable dependiente, para lo cual se utilizan los apuntadores auxiliares `GX` haciendo referencia (operador `&`) a la variable que ha sido seleccionada:

```
320 void WMainNPM::CBXGraphSelected(Int_t Opt){
321   delete GGraph;
322   GGraph = new TGraph();
323   switch(Opt){
324   case 0: GX = &X;
325     break;
326   case 1: GX = &Y;
327     break;
328   case 2: GX = &t;
329     break;
330   case 3: GX = &U;
331     break;
332   case 4: GX = &K;
333     break;
334   case 5: GX = &E;
335     break;
336   case 6: GX = &V;
337     break;
338   case 7: GX = &XP;
339     break;
340   case 8: GX = &YP;
341     break;
342   case 9: GX = &FFrx;
343     break;
344   case 10: GX = &FFry;
345     break;
346   case 11: GX = &FFr;
347     break;
348   }
349 }
```

El proceso es análogo para la selección de la variable dependiente, cambiando únicamente el apuntador auxiliar `Gy`:

```
350  void WMainNPM::CBYGraphSelected(Int_t Opt){
351    delete GGraph;
352    GGraph = new TGraph();
353    switch(Opt){
354    case 0: GY = &X;
355      break;
356    case 1: GY = &Y;
357      break;
358    case 2: GY = &t;
359      break;
360    case 3: GY = &U;
361      break;
362    case 4: GY = &K;
363      break;
364    case 5: GY = &E;
365      break;
366    case 6: GY = &V;
367      break;
368    case 7: GY = &XP;
369      break;
370    case 8: GY = &YP;
371      break;
372    case 9: GY = &FFrx;
373      break;
374    case 10: GY = &FFry;
375      break;
376    case 11: GY = &FFr;
377      break;
378    }
379  }
```

Se establece el valor que debe adquirir la entrada numérica de la densidad ρ (o se habilita su edición directa) de acuerdo con la opción seleccionada por el usuario. El método `SetState()` permite fijar si el objeto se encuentra habilitado para la edición por parte del usuario:

```
380  void WMainNPM::CBRhoSelected(Int_t Opt){
381    NERho->SetState(kFALSE);
382    switch(Opt){
383    case 0: NERho->SetNumber(1.293); //Aire 0 C 1atm
384      break;
385    case 1: NERho->SetNumber(1.205); //Aire 20 C 1atm
386      break;
387    case 2: NERho->SetNumber(1.331); //Oxigeno 20 C 1atm
388      break;
389    case 3: NERho->SetNumber(0.1664); //Helio 20 C 1atm
390      break;
391    case 4: NERho->SetNumber(0.804); //Vapor de Agua
392      break;
393    case 5: NERho->SetState(kTRUE);
394      break;
395    }
396  }
```

De manera análoga se establece el comportamiento para la selección del valor que adquiere la entrada numérica que contendrá el valor de la aceleración gravitacional g:

```
397  void WMainNPM::CBGrSelected(Int_t Opt){
398    switch(Opt){
```

```
399     case 0: NEGr->SetNumber(274.0); //Sol
400       break;
401     case 1: NEGr->SetNumber(3.701); //Mercurio
402       break;
403     case 2: NEGr->SetNumber(8.870); //Venus
404       break;
405     case 3: NEGr->SetNumber(9.780); //Tierra
406       break;
407     case 4: NEGr->SetNumber(1.62); //Luna Terrestre
408       break;
409     case 5: NEGr->SetNumber(3.690); //Marte
410       break;
411     case 6: NEGr->SetNumber(23.12); //Jupiter
412       break;
413     case 7: NEGr->SetNumber(8.96); //Saturno
414       break;
415     case 8: NEGr->SetNumber(8.69); //Urano
416       break;
417     case 9: NEGr->SetNumber(11.00); //Neptuno
418       break;
419     case 10: NEGr->SetState(kTRUE); //Otro
420     break;
421     }
422   }
```

Con el fin de ahorrar líneas de código se establece un método que devolverá el módulo de un vector bidimensional dadas sus componentes:

```
423   Double_t WMainNPM::Mod(Double_t CX,Double_t CY){
424     return sqrt(pow(CX,2)+pow(CY,2));
425   }
```

El siguiente método devuelve el valor de la aceleración en el eje vertical y, dada las componentes de la posición del cuerpo según la ecuación 3.2:

$$\frac{\left(g\left(\rho\left(\frac{4}{3}\pi r^3\right)-m\right)-\frac{1}{2}c\rho\left(\pi r^2\right)v\dot{y}\right)}{m}=\ddot{y} \tag{3.2}$$

```
426   Double_t WMainNPM::YPP(Double_t XP, Double_t YP){
427     return (g*(Rho*(4.0*TMath::Pi()*pow(R,3)/3.0)-m)
428             - 0.5*c*Rho*TMath::Pi()*pow(R,2)*Mod(XP,YP)*YP)/m;
429   }
```

El siguiente método devuelve el valor de la aceleración en el eje horizontal x, dada las componentes de la posición del cuerpo según la ecuación 3.3:

$$\frac{\left(-\frac{1}{2}c\rho\left(\pi r^2\right)v\dot{x}\right)}{m}=\ddot{x} \tag{3.3}$$

```
430   Double_t WMainNPM::XPP(Double_t XP, Double_t YP){
431     return (-0.5*c*Rho*TMath::Pi()*pow(R,2)*Mod(XP,YP)*XP)/m;
432   }
```

El método que permite deshabilitar la modificación de los objetos de la interfaz por parte del usuario:

```
431  void WMainNPM::Disable(){
432      TBClear->SetEnabled(kFALSE);
433      NEMass->SetState(kFALSE);
434      NERadio->SetState(kFALSE);
435      NECPull->SetState(kFALSE);
436      CBRho->SetEnabled(kFALSE);
437      NERho->SetState(kFALSE);
438      CBGr->SetEnabled(kFALSE);
439      NEGr->SetState(kFALSE);
440      NEDt->SetState(kFALSE);
441      NEDt->SetState(kFALSE);
442      NEX0->SetState(kFALSE);
443      NEY0->SetState(kFALSE);
444      NEV0->SetState(kFALSE);
445      NEAlpha->SetState(kFALSE);
446  }
```

Se define un método que permite habilitar la modificación de los elementos de la interfaz por parte del usuario, éste método hará parte de la rutina ejecutada una vez el botón *Limpiar* es presionado:

```
449  void WMainNPM::Enable(){
450      TBClear->SetEnabled(kTRUE);
451      NEMass->SetState(kTRUE);
452      NERadio->SetState(kTRUE);
453      NECPull->SetState(kTRUE);
454      CBRho->SetEnabled(kTRUE);
455      CBGr->SetEnabled(kTRUE);
456      NEDt->SetState(kTRUE);
457      CHBRastro->SetEnabled(kTRUE);
458      CHBRastro->SetState(kButtonDown);
459      NEDt->SetState(kTRUE);
460      NEX0->SetState(kTRUE);
461      NEY0->SetState(kTRUE);
462      NEV0->SetState(kTRUE);
463      NEAlpha->SetState(kTRUE);
464  }
```

La siguiente rutina es el *slot* que corresponde al botón limpiar cuando emite la señal `Clicked()`.

```
465  void WMainNPM::TBClearClicked(){
466    Enable();
467    delete GGraph;
468    GGraph = new TGraph();
469    TBPlay->SetText("&Simular");
470    CGraph->Update();
471    CGPos->Clear();
472    CGPos->Update();
473    IsActive=-1;
474  }
```

y el botón *Salir*, que detiene la simulación y termina la aplicación:

```
475  void WMainNPM::TBExitClicked(){
476    Cont->TurnOff();
477    gApplication->Terminate(0);
478  }
```

Para terminar la aplicación se requiere detener la ejecución de la instancia de `TTimer` por lo que se ha ejecutado `Cont->TurnOff()`.

La siguiente es la rutina ejecutada una vez el botón *Simular* (o *Pausar* dependiendo del estado de la simulación) es presionado:

```
479  void WMainNPM::TBPlayClicked(){
480    if (IsActive == -1){
481      //Si no se ha iniciado el proceso, se inicializan variables
482      m = NEMass->GetNumber();
483      R = NERadio->GetNumber();
484      EBody->SetR1(R);
485      EBody->SetR2(R);
486      Rho = NERho->GetNumber();
487      c = NECPull->GetNumber();
488      g = NEGr->GetNumber();
489      t = 0;
490      dt = NEDt->GetNumber();
491      X = NEX0->GetNumber();
492      Y = NEY0->GetNumber();
493      Alpha = NEAlpha->GetNumber()*TMath::Pi()/180;
494      XP = NEV0->GetNumber()*cos(Alpha);
495      YP = NEV0->GetNumber()*sin(Alpha);
496      TBPlay->SetText("&Pausar");
497      FW = -m*g;
498      FFe = Rho*(4.0/3.0)*TMath::Pi()*pow(R,3)*g;
499      Cont->TurnOn();
500      Disable();
501      IsActive = 0;
502    }
503    else if (IsActive == 0){
504      //Si se pausa la simulación
505      Cont->TurnOff();
506      TBPlay->SetText("&Continuar");
507      TBClear->SetEnabled(kTRUE);
508      IsActive = 1;
509    }
510    else if (IsActive == 1){
511      //Si se despausa la simulación
512      TBPlay->SetText("&Pausar");
513      TBClear->SetEnabled(kFALSE);
514      IsActive = 0;
515      Cont->TurnOn();
516    }
517  }
```

El siguiente método, implementa la solución numérica de las ecuaciones diferenciales que rigen la dinámica del sistema. En ésta ocasión es implementado el método Runge-Kutta de cuarto orden. Fíjese que no todos los datos generados en la solución numérica son representados en la animación o en el gráfico[10].

```
518  void WMainNPM::DoDraw(){
519    Double_t kX1,kX2,kX3,kX4;
520    Double_t kY1,kY2,kY3,kY4;
521    Double_t kXP1,kXP2,kXP3,kXP4;
522    Double_t kYP1,kYP2,kYP3,kYP4;
523    for(Int_t i = 0; i < 20 ;i++){
524      V = Mod(XP,YP);
```

[10]Ésta decisión fue tomada debido a que la ejecución del método `Draw()` es un proceso computacionalmente exigente. Para aumentar la resolución (en referencia a la cantidad de datos) se puede cambiar el número de ciclos en la estructura `for` por un menor valor y aumentar el valor del intervalo de tiempo asignado a la instancia `TTimer`

```
525       p = m*V;
526       K = 0.5*m*pow(Mod(XP,YP),2);
527       U = m*g*Y;
528       E = K+U;
529       FFrx = -0.5*c*Rho*TMath::Pi()*pow(R,2)*V*XP;
530       FFry = -0.5*c*Rho*TMath::Pi()*pow(R,2)*V*YP;
531       FFr = Mod(FFrx,FFry);
532       kX1 = XP*dt;
533       kXP1 = XPP(XP,YP)*dt;
534       kY1 = YP*dt;
535       kYP1 = YPP(XP,YP)*dt;
536       kX2 = (XP+0.5*kXP1)*dt;
537       kXP2 = XPP(XP+0.5*kXP1,YP+0.5*kYP1)*dt;
538       kY2 = (YP+0.5*kYP1)*dt;
539       kYP2 = YPP(XP+0.5*kXP1,YP+0.5*kYP1)*dt;
540       kX3 = (XP+0.5*kXP2)*dt;
541       kXP3 = XPP(XP+0.5*kXP2,YP+0.5*kYP2)*dt;
542       kY3 = (YP+0.5*kYP1)*dt;
543       kYP3 = YPP(XP+0.5*kXP2,YP+0.5*kYP2)*dt;
544       kX4 = (XP+kXP3)*dt;
545       kXP4 = XPP(XP+kXP3,YP+kYP3)*dt;
546       kY4 = (YP+kYP3)*dt;
547       kYP4 = YPP(XP+kXP3,YP+kYP3)*dt;
548       X += (1.0/6.0)*kX1 + (1.0/3.0)*kX2 +
549            (1.0/3.0)*kX3 + (1.0/6.0)*kX4;
550       XP += (1.0/6.0)*kXP1 + (1.0/3.0)*kXP2 +
551             (1.0/3.0)*kXP3 + (1.0/6.0)*kXP4;
552       Y += (1.0/6.0)*kY1 + (1.0/3.0)*kY2 +
553            (1.0/3.0)*kY3 + (1.0/6.0)*kY4;
554       YP += (1.0/6.0)*kYP1 + (1.0/3.0)*kYP2 +
555             (1.0/3.0)*kYP3 + (1.0/6.0)*kYP4;
556       t += dt;
557     }
```

Se modifica la posición del cuerpo. y se implementa una opción que permite "Viajar sobre el cuerpo" (i.e centrar el cuerpo en el lienzo), puede descomentar la línea para habilitarla:

```
558     GGraph->SetPoint(GGraph->GetN(),*GX,*GY);
559     CGPos->cd();
560     //Viajar sobre el cuerpo
561     //CGPos->Range(X-10,Y-10,X+10,Y+10);
562     EBody->SetX1(X);
563     EBody->SetY1(Y);
```

Si se ha seleccionado la opción de mostrar el vector de velocidad (esto es si el estado del botón `CHBVVel` es `kButtonDown`), se modifica su punto de origen, así como la longitud de las flechas que representan sus componentes:

```
564     Float_t ks1 = 0.5;//Factor de Escala Velocidad
565     Float_t ks2 = 3.0;//Factor de Escala Fuerza
566     if (CHBVVel->IsDown()){
567       //Componente en x de la velocidad
568       AXP->SetX1(X);
569       AXP->SetY1(Y);
570       AXP->SetX2(X+ks1*XP);
571       AXP->SetY2(Y);
572       //Componente en y de la velocidad
573       AYP->SetX1(X);
574       AYP->SetY1(Y);
575       AYP->SetX2(X);
576       AYP->SetY2(Y+ks1*YP);
```

```
577       //Velocidad
578       AV->SetX1(X);
579       AV->SetY1(Y);
580       AV->SetX2(X+ks1*XP);
581       AV->SetY2(Y+ks1*YP);
582       //Dibujamos todos los cambios
583       AXP->Draw();
584       AYP->Draw();
585       AV->Draw();
586     }
```

Del mismo modo sucede con la opción de mostrar las flechas que representan los vectores de fuerza:

```
587     if (CHBVForce->IsDown()){
588       //Peso
589       AW->SetX1(X);
590       AW->SetY1(Y);
591       AW->SetX2(X);
592       AW->SetY2(Y+ks2*FW);
593       //Fuerza de empuje
594       AFe->SetX1(X);
595       AFe->SetY1(Y);
596       AFe->SetX2(X);
597       AFe->SetY2(Y+ks2*FFe);
598       //Componente en x fricción
599       AFrx->SetX1(X);
600       AFrx->SetY1(Y);
601       AFrx->SetX2(X+ks2*FFrx);
602       AFrx->SetY2(Y);
603       //Componente en y fricción
604       AFry->SetX1(X);
605       AFry->SetY1(Y);
606       AFry->SetX2(X);
607       AFry->SetY2(Y+ks2*FFry);
608       //Fricción
609       AFr->SetX1(X);
610       AFr->SetY1(Y);
611       AFr->SetX2(X+ks2*FFrx);
612       AFr->SetY2(Y+ks2*FFry);
613       //Dibujamos todo
614       AW->Draw();
615       AFr->Draw();
616       AFe->Draw();
617       AFrx->Draw();
618       AFry->Draw();
619     }
```

Si el usuario seleccionó la opción de rastro, se crea un nuevo círculo de las mismas características. La posición de éste círculo será modificada en la próxima ejecución de la rutina (recuerde que ésta rutina está relacionada con el ciclo del contador) y se ejecutan los métodos necesarios para plasmar los datos, tanto en la animación como en el gráfico:

```
620     EBody->Draw();
621     if (CHBRastro->IsDown()){
622       EBody = new TEllipse(X,Y,R,R);
623       EBody->SetFillColor(2);
624     }
625     CGPos->Update();
626     //Actualizamos la gráfica
```

```
627     CGraph->cd();
628     GGraph->Draw("AC*");
629     CGraph->Update();
630   }
```

Para ejecutar la simulación es necesario crear una instancia de la clase que ha sido definida:

```
root[] #include ''WMainNPM.cxx''
root[] fNPM = new WMainNPM();
```

cuyo resultado se muestra en la figura 3.8.

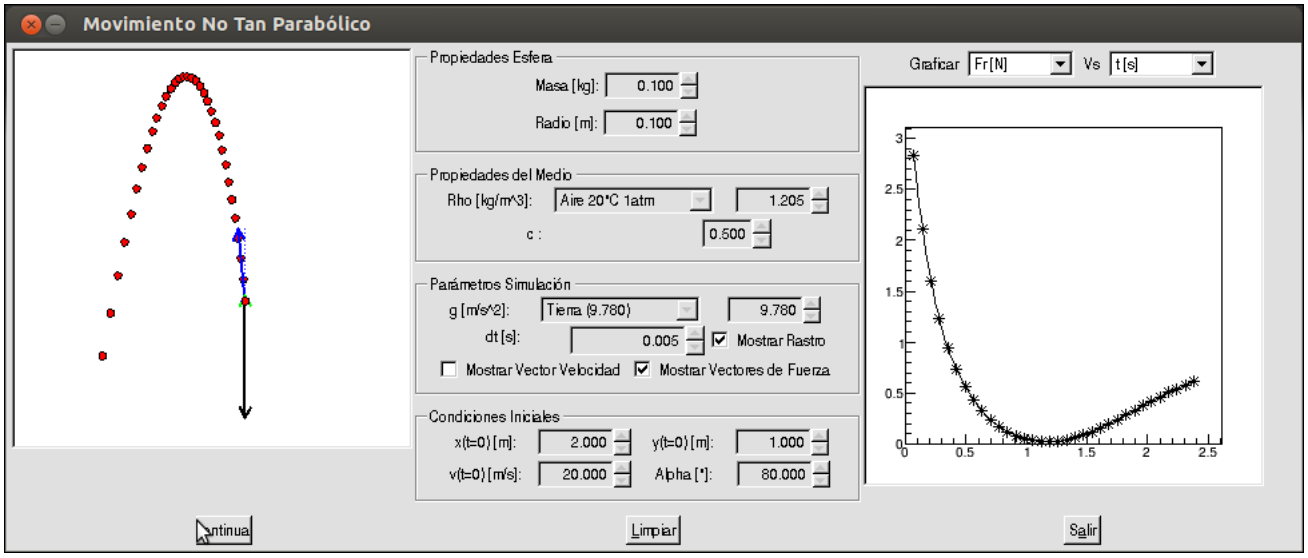


Figura 3.8: Interfaz gráfica para el fenómeno del movimiento de un cuerpo sumergido en un fluido bajo la acción de la gravedad una vez ha sido implementada.

El valor de los parámetros y de las condiciones iniciales pueden ser establecidos a través de los diferentes objetos de la interfaz. Es posible seleccionar la variable dependiente e independiente del diagrama a través de la listas de selección ubicadas en la parte superior derecha de la interfaz (ver figura 3.8); esto se debe realizar antes de ejecutar la simulación: si en la ejecución de la simulación se realiza el cambio de éstas variables, la gráfica será visualizada únicamente desde el instante en que se realizó el cambio.

Actividades propuestas

- A diferencia del caso de la interfaz creada para el fenómeno de la caída libre (sección 3.1), la animación de la presente interfaz gráfica no permite visualizar de manera completa el fenómeno (tenga en cuenta que "completa" es una característica determinada en tiempo de ejecución ya que el usuario es el que define cuando desea parar la simulación). A partir del código fuente utilizado para

la interfaz gráfica del fenómeno de caída libre, modifique la rutina de la presente interfaz de tal manera que permita ajustar el rango del lienzo de manera dinámica de modo que sea posible visualizar el fenómeno completamente.

- Añada a las variables que puede representar en la gráfica la fuerza neta que actúa sobre el cuerpo.
- No es posible identificar qué flecha representa qué vector. En el lienzo escriba una etiqueta que permita identificar qué flecha corresponde a cada vector.
- Si el usuario desmarca la opción de mostrar vectores (velocidad o fuerza) los vectores quedan plasmados en el rastro. Sin modificar el código fuente, explique cómo evitaría que esto sucediera.
- Tal como está implementada la interfaz, la escala de las flechas que representan los vectores es una condición estática. Diseñe un mecanismo que permita al usuario definir éste factor de escala. La única condición es que no puede utilizar ninguno de los objetos previamente utilizados.

Perspectivas y consideraciones finales

El desarrollo de software es un campo de estudio que requiere, pero a la vez desarrolla, un pensamiento lógico-matemático y una perspectiva física de la resolución de problemas, donde es posible representar objetos a través de un conjunto de variables que presentan una evolución dinámica en tiempo de ejecución determinada tanto por el algoritmo diseñado como por la interacción del usuario con éste proceso.

En la actualidad el desarrollo de software es una actividad indispensable en el proceso de investigación, desde la adquisición de datos, la representación gráfica y el análisis de estos hasta el proceso de edición y publicación.

Las interfaces gráficas de usuario facilitan el uso de software y el aprendizaje de éste al proporcionar una mediación a través de imágenes. En el ámbito de las simulaciones representan un mecanismo efectivo de representación a través del cual es posible visualizar y estudiar el comportamiento de modelos físico-matemáticos.

La consulta y modificación de código fuente es una característica indispensable en la enseñanza y el aprendizaje de resolución de problemas a través de algoritmos. La disponibilidad del código posibilita la evaluación de pares lo cual permite desarrollar de manera colaborativa una pieza de software.

La infraestructura de análisis de datos ROOT proporciona un completo conjunto de herramientas tales como el interprete que aumenta la eficiencia del proceso de escritura y testeo de código o el generador de diccionarios que facilita en gran manera el proceso de compilación, así como clases de representación gráfica, a través de las cuales se simplifica la implementación de interfaces gráficas de usuario especialmente

en el área de las simulaciones, tenga presente que se ha abordado sólo un pequeño segmento de ésta infraestructura. quedan muchos aspectos por explorar.

Referencias

Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., ... Wilson, P. (2012). Best Practices for Scientific Computing. *CoRR, abs/1210.0530*. Descargado de `http://arxiv.org/abs/1210.0530`

Brun, R. (1998, mayo). *What does ROOT stand for?* `http://root.cern.ch/root/roottalk/roottalk98/0718.html`.

Cortes, L. (1997, Mayo). Designing a Graphical User Interface . *Medical Computing Today*. `http://medicalcomputing.org/archives/0agui.php`. ((En-Línea; Consultado el 31 de Marzo de 2012))

Dewar, R. B., y Schonberg, E. (2008). Computer Science Education: Where Are the Software Engineers of Tomorrow? *Crosstalk: the Journal of Defense Software Engineering*.

Eckel, B. (2000). Thinking in C++. En (2.ª ed., cap. Introduction to Objects). `http://hep.uchicago.edu/~tangjian/Elog/CPP/thinkInCpp/vol_1/Chapter01.html`.

Manoukis, N. C., y Anderson, E. C. (2008). GuiLiner: A Configurable and Extensible Graphical User Interface for Scientific Analysis and Simulation Software. *CoRR, abs/0806.0314*. Descargado de `http://arxiv.org/abs/0806.0314`

Prabhu, P., Jablin, T. B., Raman, A., Zhang, Y., Huang, J., Kim, H., ... others (2011). A survey of the practice of computational science. En *State of the Practice Reports* (p. 19).

Rademakers, F. (s.f.). The Power of Object-Oriented Frameworks: Part1: Introduction to Framework Technology. *Interface Magazine*. `ftp://root.cern.ch/root/frameworks.ps.gz`.

Rademakers, F., Goto, M., Canal, P., y Brun, R. (2003). ROOT Status and Future Developments. *CoRR, cs.SE/0306078*. Descargado de `http://arxiv.org/abs/cs.SE/0306078`

ROOT-Team. (s.f.-a). *Example Applications.* `http://root.cern.ch/drupal/content/example-applications`.

ROOT-Team. (s.f.-b). *ROOT Users Guide.* `http://root.cern.ch/download/doc/ROOTUsersGuide.pdf`.

ROOT-Team. (s.f.-c). What Are We Working on. `http://root.cern.ch/drupal/content/what-are-we-working`.

Salamanca, J. (2011). *Interfaces Gráficas de Usuario en ROOT para la enseñanza de la física.* Proyecto de Investigación no Publicado. Universidad Distrital Francisco José de Caldas. (Proyecto de Investigación avalado por el Centro de Investigación y Desarrollo Científico de la Universidad Distrital.)

Smith, A. (1997). *Human computer factors*. McGraw-Hill.

Apéndice A

Instalando ROOT en el sistema GNU/Linux

Existen múltiples formas de instalar ROOT en un sistema GNU/Linux, incluso existen distribuciones como *Scientific Linux*[1] que vienen con ésta infraestructura en la instalación por defecto. La primera opción para instalar ROOT es consultar si existe una versión precompilada en los repositorios de la distribución. ROOT se distribuye en el metapaquete llamado `root-system`, para distribuciones basadas en Debian (Ubuntu, Trisquel, Mint etc) ejecute:

```
# apt-get install root-system
```

Si el paquete no se encuentra en los repositorios o si requiere la última versión existen dos opciones, instalar los binarios o compilar el código fuente, presentadas a continuación.

A.1. Instalar ROOT desde los binarios

Para instalar los binarios de ROOT, primero debe conocer la versión del compilador con la cual cuenta su ordenador para esto ejecute:

```
$ gcc --version
```

si no tiene instalado `gcc` en su ordenador puede instalarlo a través de:

```
# apt-get install build-essential
```

Diríjase a `http://root.cern.ch/drupal/content/downloading-root` (Sección *Binaries*) y descargue la versión de su preferencia (se recomienda la versión etiquetada con *Pro* de *Production* la cual es la última versión estable) al realizar la descarga tenga en cuenta la versión de `gcc` y la arquitectura del sistema si es de 32 bits descargue `ia32`, si cuenta con un sistema de 64 bits descargue `x86_64`, si desconoce la arquitectura que está instalada en su sistema ejecute:

```
$ uname -a
```

[1] https://www.scientificlinux.org/

Luego de esto es necesario enlazar las librerías que requiere la ejecución de ROOT. Con el fin de simplificar la explicación de éste procedimiento, será tomado un ejemplo particular: la instalación completa de ROOT 5.34/07 en una instalación limpia de la distribución GNU/Linux Ubuntu versión 12.10. Una vez determinada la versión del compilador y la arquitectura del sistema descargamos los binarios pertinentes:

```
$ wget -c ftp://root.cern.ch/root/root_v5.34.07.Linux-slc5-gcc4.3.tar.gz
```

Desempaquetamos el comprimido:

```
$ tar -xf root_v5.34.07.Linux-slc5-gcc4.3.tar.gz
```

Esto creará la carpeta `root`:

```
$ cd root
$ source bin/thisroot.sh
$ root -l
```

Esto mostrará:

```
root[0]
```

Parecería que ROOT es completamente funcional; sin embargo, para probarlo podemos crear una instancia de `TBrowser`:

```
root[] new TBrowser()
```

En nuestro caso no se creó la instancia sino que mostró los siguientes errores:

```
dlopen error: libjpeg.so.62: cannot open shared object file: No such file or directory
Load Error: Failed to load Dynamic link library /home/ubuntu/Documentos/root/lib/libASImage.so
...
...
```

La primera línea informa que no ha encontrado la librería `libjpeg.so.62`, como vemos ésta es una versión específica de la librería `libjpeg` por lo que puede que contemos con otra versión en el sistema, la buscamos a través de:

```
$ find /usr/lib/ -iname ``libjpeg*''
```

En el sistema en el que estamos trabajando devolvió lo siguiente:

```
/usr/lib/i386-linux-gnu/libjpeg.so.8
/usr/lib/i386-linux-gnu/libjpeg.so.8.0.2
```

si ejecutamos:

```
file /usr/lib/i386-linux-gnu/libjpeg.so.8
```

Veremos que éste archivo es un enlace a `libjpeg.so.8.0.2`, por lo que haremos un enlace a ésta última de tal manera que cuando ROOT llame a `libjpeg.so.62` se cargue la librería `libjpeg.so.8`, esto lo logramos ejecutando:

```
# ln -s /usr/lib/i386-linux-gnu/libjpeg.so.8.0.2 /usr/lib/libjpeg.so.62
```

Nuevamente probamos creando una instancia de `TBrowser` y mostró el siguiente error:

```
dlopen error: libtiff.so.3: cannot open shared object file: No such file or directory
Load Error: Failed to load Dynamic link library /home/ubuntu/Documentos/root/lib/libASImage.so
...
...
 *** Break *** segmentation violation
Root >
```

Repetimos el proceso de buscar y enlazar la librería `libtiff` :

```
$ find /usr/lib/ -iname "libtiff*"
/usr/lib/i386-linux-gnu/libtiff.so.5
/usr/lib/i386-linux-gnu/libtiff.so.5.1.0
$ ln -s /usr/lib/i386-linux-gnu/libtiff.so.5.1.0 /usr/lib/libtiff.so.3
```

Nuevamente probamos creando una instancia de `TBrowser` y creó la instancia, sin embargo, mostró el siguiente error:

```
root [0] new TBrowser()
dlopen error: libcrypto.so.6: cannot open shared object file: No such file or directory
Load Error: Failed to load Dynamic link library /home/ubuntu/Documentos/root/lib/libNet.so
...
...
```

En este caso, al realizar la búsqueda el comando `find` no devuelve nada; es decir, no tenemos instalada *ninguna* versión de esa librería, para eso primero se debe saber qué paquete contiene esa librería, nos dirigimos a: `http://packages.ubuntu.com/#search_contents` en *keywords* de la sección *Search the contents of packages*, colocamos el nombre de la librería más la extensión `.so` (de *shared object*). En este caso buscamos `libcrypto.so` y seleccionamos la opción *packages that contain files whose names contain the keyword,* esto devuelve el nombre de los paquetes que contienen archivos relacionados con nuestra búsqueda, en éste caso, instalaré el paquete `libssl-dev`:

```
# apt-get install libssl-dev
```

El paquete `libssl-dev` no contiene específicamente `libcrypto.so.6` así que repito el procedimiento para crear un enlace:

```
$ find /usr/lib/ -iname "libcrypto*"
/usr/lib/i386-linux-gnu/libcrypto.so
# ln -s /usr/lib/i386-linux-gnu/libcrypto.so /usr/lib/libcrypto.so.6
```

Repetimos la creación de la instancia `TBrowser` y nos encontramos con el último error:

```
dlopen error: libssl.so.6: cannot open shared object file: No such file or directory
Load Error: Failed to load Dynamic link library /home/ubuntu/Documentos/root/lib/libNet.so
...
...
```

La librería `libssl` es proveída por el último paquete que instalamos por lo que repetimos el procedimiento para crear el enlace respectivo:

```
$ find /usr/lib/ -iname ``libssl*''
...
/usr/lib/i386-linux-gnu/libssl.so
...
# ln -s /usr/lib/i386-linux-gnu/libssl.so /usr/lib/libssl.so.6
```

Finalmente la creación de `TBrowser` funciona perfectamente:

```
root[] new TBrowser()
(class TBrowser*)0x9bf4830
```

Para terminar la instalación, sin salir del directorio se ejecuta:

```
$ echo ``export ROOTSYS=`pwd`'' > ~/.bashrc
$ echo ``. $ROOTSYS/bin/thisroot.sh''~/.bashrc
```

Esto se hace con el fin de configurar las variables de entorno necesarias para la ejecución de ROOT al iniciar una sesión en *bash*. Así se concluye el proceso de instalación.

A.2. Compilando ROOT

Si sus repositorios cuentan con el metapaquete `root-system` puede descargar el código fuente ejecutando[2]:

```
$ apt-get source root-system
```

Igualmente si éste metapaquete se encuentra en los repositorios instale las dependencias para poder compilar el código:

```
# apt-get build-dep root-system
```

Si su distribución no cuenta con el metapaquete o si requiere instalar una versión diferente, diríjase a `http://root.cern.ch/drupal/content/downloading-root` (Sección *Source*) y descargue la versión de su preferencia[3], en éste ejemplo será la versión 5.34/07:

```
wget -c ftp://root.cern.ch/root/root_v5.34.07.source.tar.gz
```

Desempaquete el archivo:

```
$ tar -xf root_v5.34.07.source.tar.gz
$ cd root
```

[2]Debe tener los repositorios de código fuente activados en el archivo `/etc/apt/sources.list`

[3]Si busca involucrarse en el desarrollo de ROOT y obtener el código fuente que actualmente se encuentra en desarrollo descargue un clon del repositorio mediante: `git clone http://root.cern.ch/git/root.git`

Para ver las opciones con las que se puede compilar ROOT ejecute `$./configure -help` establezca las características que requiere y ejecute `$./configure -enable-X -disable-Y`. El *script* `configure` informará acerca de las dependencias no resueltas y los paquetes no instalados consulte la sección anterior para observar un método efectivo para buscar los paquetes que suplen las dependencias, finalmente ejecute:

```
$ make
```

Si cuenta con un computador con múltiples núcleos puede ejecutar `$ make -j n` con n el número de núcleos/hilos para agilizar (considerablemente) el proceso de compilación. Una vez compilado ejecute:

```
$ echo ``export ROOTSYS=`pwd`'' > ~/.bashrc
$ echo ``. $ROOTSYS/bin/thisroot.sh''~/.bashrc
```

Esto se hace con el fin de configurar las variables de entorno necesarias para la ejecución de *ROOT* al iniciar una sesión en *bash*. Así se concluye el proceso de instalación.

Apéndice B

Configurando emacs para trabajar con ROOT

B.1. Instalando complementos

B.1.1. Número de línea (linum.el)

Cuando se hace referencia a un segmento de código en un programa tanto el interprete como el compilador se expresan en términos del número de línea, así que es útil tener una columna que indique el número de línea:

```
File Edit Options Buffers Tools C++ nXhtml Help
  49 |  char STAll[50];
  50 |  void GetObjectsGraphs();
  51 |
  52 |public:
  53 |  WMainIP();
  54 |  ~WMainIP();
  55 |  void TBExitClicked();
  56 |  void TBPlayClicked();
  57 |  void TBClearClicked();
  58 |  void ReDraw();
  59 |  void DoDraw();
-UUU:----F1  WMainIP.h      66% L57   Git-master  (C++/l Abbrev)---
```

Una de las opciones es utilizar el complemento `linum`, si utiliza una versión de emacs mayor o igual a la versión 22 ejecute en su terminal:

```
wget -c http://web.student.tuwien.ac.at/~e0225855/linum/linum.el
```

o si trabaja con *xemacs* o una versión más antigua ejecute:

```
wget -c http://eweb.student.tuwien.ac.at/~e0225855/linum/compat/linum.el
```

luego ejecute:

```
$ mv linum.el ~/.emacs.d/
$ echo ``(require~'linum)'' > ~/.emacs
$ echo ``(global-linum-mode~t)'' > ~/.emacs
```

B.1.2. Complemento por y para ROOT (root-help.el)

ROOT cuenta con un complemento que permite hacer uso de funciones de ROOT directamente desde *emacs*, para instalarlo ejecute:

```
wget -c https://www.slac.stanford.edu/grp/eg/minos/ROOTSYS/pro/build/misc/root-help.el
mv root-help.el ~/.emacs.d/
echo (require 'root-help) >> ~/.emacs
```

Una vez realice este procedimiento puede:

- Crear la estructura de la función `main()`:
  Ejecute `$ emacs NombreAplicacion.cxx`
  Presione Alt + x y ejecute `root-main`
  presione `y`.
- Crear la estructura de la cabecera y el archivo de definición:
  Ejecute `$ emacs`
  Presione Alt + x y ejecute `root-class`
- Ejecutar ROOT en *emacs*:
  Presione Alt + x y ejecute `root-shell`
- Ejecutar un bloque de código:
  Seleccione el bloque de código,
  Presione Alt + x y ejecute `root-send-region-to-root`

Consulte las opciones disponibles en la documentación:

```
$ less ~/.emacs.d/root-help.el
```

Apéndice C

Compilando Aplicaciones desarrolladas en ROOT

El proceso de compilación es la traducción de un lenguaje de programación; en el caso de ROOT, la traducción de C++ a lenguaje máquina. Este proceso permite generar un ejecutable, el cual es optimizado para la máquina en la cual se compiló, para este proceso es necesario contar con los siguientes archivos:

- Archivos de cabecera de cada clase generada `*.h`
- Archivos de la definición de cada clase generada `*.cxx`
- Un diccionario de las clases generadas.
- Un archivo que contiene la definición de la función `main()`

C.1. Generando diccionarios para una clase

Para poder compilar código fuente que utiliza librerías de ROOT, es necesario generar el *diccionario* del conjunto de clases que han sido escritas. El diccionario contiene la información que requiere CINT acerca de una clase o variable. Este proceso genera dos archivos uno con extensión `.h` y el otro con extensión `.cxx`, los cuales requieren ser compilados posteriormente. La utilidad de ROOT llamada `rootcint` se encarga de generar el diccionario, para ello, es necesario pasarle como argumento la dirección de las cabeceras de las clases. Por ejemplo, si en la escritura del software el autor generó las clases llamadas `Clase1` y `Clase2` utilizaría el siguiente comando para generar el diccionario:

```
$ rootcint -f Dict.cxx -c Clase1.h Clase2.h
```

En este caso el diccionario será escrito en el archivo `Dict.cxx` (también se generará el archivo `Dict.h` de manera automática) que contendrá las definiciones necesarias para poder utilizar instancias de `Clase1` y `Clase2`, si cuenta con un mayor número de clases deberá añadirlas después del argumento `-c`.

C.2. Estructura del archivo que contiene la función main()

Todo programa candidato a ser compilado debe tener un punto de inicio, éste punto de inicio define las sentencias a ejecutar una vez se lanza el ejecutable. Por convención, éste punto de inicio es la función `main()`. ROOT cuenta con un complemento para *emacs* [1], el cual es capaz de generar varias estructuras entre ellas la del archivo de cabecera, el archivo de definición de una clase y el archivo de definición de la función `main()`.

La siguiente es la estructura del archivo que contiene esa definición el cual vamos a llamar `Aplicacion.cxx`.

```
1  #ifndef __CINT__
2  #ifndef ROOT_TApplication
3  #include "TApplication.h"
4  #endif /*ROOT_TApplication*/
5  #endif /*__CINT__*/
6
7  /*Incluir aquí las cabeceras necesarias*/
8
9  int Aplicacion()
10 {
11   /* Crear una instancia de la clase
12   que contiene la ventana principal */
13   return 0;
14 }
15
16 #ifndef __CINT__
17 int main(int argc, char** argv)
18 {
19   TApplication AplicacionApp("AplicacionApp", &argc, argv);
20   int retVal = Aplicacion();
21   AplicacionApp.Run();
22   return retVal;
23 }
24 #endif /*__CINT__*/
```

Ésta definición contiene inicialmente unos condicionales que definen si se incluyen o no unas cabeceras, luego contiene una función de aplicación en la cual está provisto entregar el flujo del programa a la clase que contiene la ventana principal, ya que allí se ha definido cómo se comporta el programa. En la función `main()` se redunda un poco, inicialmente se declara una instancia de la clase `TApplication` que se llama `AplicacionApp`, esto se hace con el fin de ejecutar una serie de procesos necesarios para para la inicialización del programa, entre los cuales se encuentra proveer la interfaz entre la aplicación y el sistema gráfico (recuerde que éste depende del sistema operativo, para una descripción detallada de los procesos ejecutados a través de `TApplication` por favor consultar la documentación de referencia `http://root.cern.ch/root/html/TApplication.html#TApplication:TApplication@2`).

[1] Consulte el Apéndice B para la instalación y uso básico del complemento.

C.3. Compilando y enlazando el código fuente en el ejecutable.

Una vez generado el diccionario y el código de la función `main()` es necesario compilar cada archivo de código fuente:

```
$ g++ `root-config --cflags` -Wall -c Clase1.cxx
$ g++ `root-config --cflags` -Wall -c Clase2.cxx
$ g++ `root-config --cflags` -Wall -c Dict.cxx
$ g++ `root-config --cflags` -Wall -c Aplicacion.cxx
```

La opción `-Wall` hace que la compilación genere un registro descriptivo. Por ejemplo, informará de variables que han sido declaradas y que no se han usado. `root-config -cflags` permite agregar los *flags*[2] necesarios para compilar código fuente que utiliza librerías de ROOT. Por ejemplo, para indicarle donde debe buscar las cabeceras indicadas en los *includes statements*. La ejecución de estas instrucciones generará los archivos `Clase1.o`, `Clase2.o`, `Dict.o` y `Aplicacion.o`. Finalmente, para obtener el ejecutable es necesario enlazar (*linking process*) éstos objetos, lo cual se logra a través de:

```
$ g++ -o Programa.exe Clase1.o Clase2.o Dict.o Aplicacion.o  `root-config --glibs`
```

El comando `root-config -glibs` genera los *flags* necesarios para hacer el enlace de los objetos en el ejecutable, de ésta manera se producirá `Programa.exe`, el cual es el ejecutable.

C.4. Errores comunes en el proceso de compilación.

A continuación se enumeran errores comunes que pueden ocurrir al momento de la compilación.

- `*:*:*: error: '*' does not name a type` posiblemente no ejecutó el `#include` para cargar la definición de la clase de la cual quiere crear una instancia.
- `*.cxx:*:*: error: '*' was not declared in this scope` puede estar asignando un valor a una variable que no ha declarado o puede que no haya declarado la variable con la globalidad que ésta requiere.
- Si declaró o está definiendo el constructor o el destructor con algún tipo, por ejemplo: `void Clase::Clase()` estaría declarando el constructor como de tipo `void` lo cual es inválido (debe reemplazar por `Clase::Clase()`) lo que producirá un error del tipo:

[2]Los argumentos que toma el compilador también son conocidos como *flags*

```
*.cxx:*:*: error: return type specification for constructor/destructor invalid
```

- `*:*:*: error: 'gApplication' was not declared in this scope` no ejecutó `#include ÏApplication.h¨`

Apéndice D

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains

nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for

verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

```
If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```

Referencias

Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., ... Wilson, P. (2012). Best Practices for Scientific Computing. *CoRR, abs/1210.0530*. Descargado de http://arxiv.org/abs/1210.0530

Brun, R. (1998, mayo). *What does ROOT stand for?* http://root.cern.ch/root/roottalk/roottalk98/0718.html.

Cortes, L. (1997, Mayo). Designing a Graphical User Interface . *Medical Computing Today*. http://medicalcomputing.org/archives/0agui.php. ((En-Línea; Consultado el 31 de Marzo de 2012))

Dewar, R. B., y Schonberg, E. (2008). Computer Science Education: Where Are the Software Engineers of Tomorrow? *Crosstalk: the Journal of Defense Software Engineering*.

Eckel, B. (2000). Thinking in C++. En (2.ª ed., cap. Introduction to Objects). http://hep.uchicago.edu/~tangjian/Elog/CPP/thinkInCpp/vol_1/Chapter01.html.

Manoukis, N. C., y Anderson, E. C. (2008). GuiLiner: A Configurable and Extensible Graphical User Interface for Scientific Analysis and Simulation Software. *CoRR, abs/0806.0314*. Descargado de http://arxiv.org/abs/0806.0314

Prabhu, P., Jablin, T. B., Raman, A., Zhang, Y., Huang, J., Kim, H., ... others (2011). A survey of the practice of computational science. En *State of the Practice Reports* (p. 19).

Rademakers, F. (s.f.). The Power of Object-Oriented Frameworks: Part1: Introduction to Framework Technology. *Interface Magazine*. ftp://root.cern.ch/root/frameworks.ps.gz.

Rademakers, F., Goto, M., Canal, P., y Brun, R. (2003). ROOT Status and Future Developments. *CoRR, cs.SE/0306078*. Descargado de http://arxiv.org/abs/cs.SE/0306078

ROOT-Team. (s.f.-a). *Example Applications.* http://root.cern.ch/drupal/content/example-applications.

ROOT-Team. (s.f.-b). *ROOT Users Guide.* http://root.cern.ch/download/doc/ROOTUsersGuide.pdf.

ROOT-Team. (s.f.-c). What Are We Working on. http://root.cern.ch/drupal/content/what-are-we-working.

Salamanca, J. (2011). *Interfaces Gráficas de Usuario en ROOT para la enseñanza de la física.* Proyecto de Investigación no Publicado. Universidad Distrital Francisco José de Caldas. (Proyecto de Investigación avalado por el Centro de Investigación y Desarrollo Científico de la Universidad Distrital.)

Smith, A. (1997). *Human computer factors*. McGraw-Hill.