SOFTWARE MANIPULADOR DEL ROBOT PORTA ENDOSCOPIO HIBOU PARA CIRUGIA LAPAROSCÓPICA

JUAN CAMILO MUÑOZ CAMAYO
JUAN FERNANDO ORDOÑEZ FAJARDO

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento De Electrónica, Instrumentación y Control
Línea de Investigación: Control y Robótica

Popayán, Febrero de 2012

SOFTWARE MANIPULADOR DEL ROBOT PORTA ENDOSCOPIO HIBOU PARA CIRUGIA LAPAROSCÓPICA

JUAN CAMILO MUÑOZ CAMAYO
JUAN FERNANDO ORDOÑEZ FAJARDO

Trabajo de grado presentado a la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca para la obtención del Título de

Ingeniero en Automática Industrial

Director: PhD. Oscar Andrés Vivas

Popayán, Febrero de 2012

Hoja de Aprobación

____

____

____

Director ________________________________

PhD. Oscar Andrés Vivas

Jurado ________________________________

Jurado ________________________________

Fecha de sustentación: Popayán, Marzo 2012

A……

Agradecimientos

A nuestros padres y hermanos por su apoyo incondicional en el transcurso de nuestros estudios de pregrado.

A nuestros amigos por su apoyo y compañía en los momentos difíciles

A Dios por acompañarnos en cada tropiezo que tuvimos y por ayudarnos en la superación de cada obstáculo.

Al Doctor Oscar Andrés Vivas por su colaboración en cada meta propuesta en el trabajo y por su interés como director.

Al Doctor Carlos Rengifo por su colaboración, consejos y conocimientos que nos ayudaron en un gran número de circunstancias que tuvimos en el desarrollo del proyecto.

Al Magister Jaime Díaz y al Magister Sergio Salinas por sus consejos en el desarrollo del proyecto.

Al Departamento de Electrónica, Instrumentación y Control por prestarnos sus servicios y por el profesionalismo de sus docentes.

A la Universidad del Cauca por su contribución administrativa y económica.

Resumen

En este proyecto se presentara el desarrollo de un software manipulador para el robot para cirugía laparoscópica (HIBOU), ya desarrollado en un trabajo de pregrado anterior, simulado bajo la plataforma de Matlab® y Virtual Reality. Este robot será manipulado a través de un joystick en un ambiente virtual creado a través del motor gráfico de renderizado Ogre 3D.

Se utiliza el lenguaje de programación C++ que permite a la aplicación trabajar de forma más rápida, bajo el entorno de desarrollo integrado Visual Studio. En este IDE se construye el robot en un ambiente virtual quirúrgico a partir de mallas obtenidas a través del software CAD Blender y Solid Edge®, este último fue la plataforma en la cual se construyó el robot.

Se implementa el modelo geométrico inverso (MGI), que permite calcular los valores de las variables articulares del HIBOU, dependiendo de la orientación y localización deseada del efector final en el espacio cartesiano. Además se implementan los modelos dinámicos, Modelo Dinámico Inverso (MDI) y Modelo Dinámico Directo (MDD), que contienen toda la información geométrica y dinámica del robot, obteniendo así todo el lazo de control que manipulara al robot.

Este desarrollo de software va de la mano con un segundo sub-proyecto encargado de la construcción de un dispositivo de mando externo, que tiene el objetivo de manipular la posición y orientación del efector final en el interior del abdomen insuflado.

Palabras Clave: Software manipulador, robótica quirúrgica, cirugía laparoscópica, robot porta endoscopio.

Abstract

This project will introduce the development of manipulating software for the laparoscopic surgery robot (HIBOU), which was developed in a previous undergraduate work, simulated under Matlab platform using Virtual Reality. This robot will be manipulated by a Joystick in a virtual environment created with the rendering graphics engine Ogre3D.

It uses the programming language C++, which allows the application to work in a more rapid, under the integrated development environment Visual Studio. This IDE will rebuild the robot in a virtual surgical environment from meshes obtained through the CAD software Blender and Solid Edge, on which the robot was built.

The inverse geometric model (MGI) will be implemented, which can calculate the values of the joint variables of the HIBOU, depending on the desired orientation and location of the end effector in the Cartesian space. Also, dynamic models are implemented, the Inverse Dynamic Model (IDM) and the Direct Dynamic Model (DDM), which contain all the geometric and dynamic information of the HIBOU, thus obtaining the entire loop control to manipulate the robot.

This software development goes hand in hand with a second sub-project, in charge of the construction of an external control device, which aims to manipulate the position and orientation of the end effector inside the insufflated abdomen.

Key words: manipulating software, robotic surgery, laparoscopic surgery, endoscope holder robot.

Contenido

INTRODUCCIÓN... 1

1. LA CIRUGÍA ROBÓTICA ACTUAL Y EL ROBOT HIBOU 4

1.1. LA ROBÓTICA APLICADA A LA MEDICINA 4
1.1.1. LA CIRUGÍA MINI-INVASIVA (CMI) 5
1.1.2. LA CIRUGÍA LAPAROSCÓPICA.. 6
1.1.3. ROBÓTICA QUIRÚRGICA 8
1.1.3.1. DA VINCI 8
1.1.3.2. ZEUS. 9
1.1.3.3. ROBODOC. 10
1.1.4. ROBOTS PORTA ENDOSCOPIOS 10
1.1.4.1. LAPMAN.. 11
1.1.4.2. ENDOASSIST. 11
1.1.4.3. LER. 12

1.2. ROBOT PORTA ENDOSCOPIO HIBOU 13
1.2.1. MODELADO MATEMATICO DEL ROBOT HIBOU 14
1.2.1.1. MODELO GEOMÉTRICO DIRECTO 14
1.2.1.2. MODELO GEOMETRICO INVERSO 15
1.2.1.3. DISEÑO DEL CUERPO DEL HIBOU. 17
1.2.1.4. MODELOS DINÁMICOS. 17

1.3. AMBIENTES VIRTUALES EN LA CIRUGÍA 19
1.3.1. LAP MENTOR 19
1.3.2. VIRTUAL LAPAROSCOPY SIMULATOR (LapSimÂ®)........................ 20
1.3.3. POSICIONAMIENTO DE LAPBOT EN UN AMBIENTE TRIDIMENSIONAL VIRTUAL USANDO INTERFAZ HÁPTICA 20

2. CONSTRUCCIÓN DEL AMBIENTE VIRTUAL 22

2.1. REQUERIMIENTOS DE LA APLICACIÓN 23

2.2. HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO DE LA APLICACIÓN ... 23
2.2.1. MICROSOFT VISUAL STUDIO ® [27] ... 24
2.2.2. OGRE3D [28] ... 24
2.2.3. BLENDER [29] ... 25
2.2.4. SOLID EDGE® [30] ... 25
2.2.5. MAKE HUMAN [31] ... 25

2.3. AMBIENTE VIRTUAL DE OGRE3D ... 27

2.4. CONSTRUCCIÓN DE MALLAS TRIDIMENSIONALES ... 28
2.4.1. Robot HIBOU ... 28
2.4.2. Cuarto quirúrgico ... 32
2.4.3. El paciente y el interior de su abdomen ... 33

2.5. RESULTADO DE LA CONSTRUCCIÓN DE MALLAS ... 35

3. ESQUEMA DE CONTROL A IMPLEMENTAR Y LA SOLUCIÓN DEL MGI ... 39

3.1. MODELO GEOMETRICO INVERSO EN C++ ... 42
3.1.1. METODO DE NM (NELDER - MEAD) ... 43
3.1.2. MGI CON EL METODO DE NELDER-MEAD ... 45

3.2. MODELO GEOMETRICO DIRECTO EN C++ ... 46

4. MODELOS DINÁMICOS Y EL LAZO DE CONTROL ... 48

4.1. MODELOS DINÁMICOS EN C++ ... 48
4.1.1. METODO DE RUNGE KUTTA DE CUARTO ORDEN ... 49
4.1.2. IMPLEMENTACION DEL RK4 EN C++ ... 51

4.2. LOOPCONTROL ... 53

4.3. COMPORTAMIENTO DEL LAZO DE CONTROL ... 55
4.3.1. CONSIGNA CIRCULAR ... 55
4.3.2. PRIMERA CONSIGNA GENERADA POR EL JOYSTICK ... 57
4.3.3. SEGUNDA CONSIGNA GENERADA POR EL JOYSTICK ... 58
4.3.4. INTEGRACION DEL DISPOSITIVO AUXILIAR ... 59
4.3.4.1. PRIMERA CONSIGNA DEL DISPOSITIVO AUXILIAR ... 59
4.3.4.2. SEGUNDA CONSIGNA DEL DISPOSITIVO DE MANDO AUXILIAR ... 61

4.4. COMPORTAMIENTOS A TENER EN CUENTA EN LA EJECUCIÓN DE LA APLICACIÓN 62

5. ASPECTOS IMPORTANTES A TENER EN CUENTA.................64

5.1. EN EL LAZO DE CONTROL.. 64

5.2. EN EL TIEMPO DE EJECUCIÓN DE LA APLICACIÓN............................ 65

6. PUBLICACIONES EN EL MARCO DE ESTA TESIS 66

6.1. Publicación en evento... 66

6.2. Publicación en revista... 66

7. CONCLUSIONES 67

REFERENCIAS BIBLIOGRÁFICAS 69

ANEXO A. ECUACIONES DE LOS MODELOS DEL ROBOT HIBOU 1

A.1. ECUACIONES PARA EL CÁLCULO DEL MGD 1

A.2. ECUACIONES PARA EL CÁLCULO DEL MGI 3

ANEXO B. TUTORIALES 4

B.1. INSTALACIÓN DE OGRE3D PARA VISUAL STUDIO.................................. 4

B.2. INSTALACIÓN DE BLENDER Y SU UTILIZACIÓN 9

B.3. COMANDOS FUNCIONALES EN EL SOFTWARE..................................... 13

B.4. Habilitando el joystick en Ogre3D... 13

ANEXO C. CODIGO UTILIZADO..16

C.1. CODIGO NECESARIO PARA EL RENDERIZADO DEL ROBOT 16

C.2. CODIGO PARA EL RENDERIZADO DEL CUARTO QUIRÚRGICO............ 17

C.3. FUNCION OBJETIVO EN MATLAB 17

C.4. FUNCIÓN OBJETIVO EN C++.. 18

C.5. ARCHIVO CABECERA Y CUERPO DEL NELDER-MEAD.......................... 20

C.6. FUNCION PARA INICIALIZAR EL ROBOT... 28

C.7. CODIGO COMPLETO DEL RK4.. 29

Lista de Tablas

Tabla 1.1. Parámetros geométricos del HIBOU [1]. 14
Tabla 1.2. Parámetros inerciales de base del HIBOU [1]. 18
Tabla 3.1. Relación de variables software con el diagrama de bloques................ 40
Tabla 3.2. Comparación de los métodos numéricos. 46
Tabla 4.1. Ecuaciones para la solución del MDD con RK4. 50
Tabla B.1. Comandos con el teclado.. 13

Lista de Figuras

Figura 1.1. Región abdominal con instrumental laparoscópico [10]. 6
Figura 1.2. Vista ampliada del procedimiento quirúrgico de extracción de vesícula biliar (cuerpo en verde) [8].. 7
Figura 1.3. Sistema quirúrgico Da Vinci (Standard) [10]. 9
Figura 1.4. Robot Zeus [12]... 10
Figura 1.5. Robodoc [13] 10
Figura 1.6. Robot LapMan [14]... 11
Figura 1.7. Endoassist [15]... 12
Figura 1.8. LER [16]. 12
Figura 1.9. Estructura cinemática del HIBOU [1]... 13
Figura 1.10. Movimiento de las articulaciones alrededor del punto de incisión (trocar) [1].. 15
Figura 1.11. Restricción de colinealidad del vector $\overline{P5P6}$ con $\overline{P5Pt}$ y $\overline{PtP6}$ [1]. 16
Figura 1.12. Dimensiones del HIBOU [1]. 17
Figura 1.13. Robot HIBOU en SolidEdge® [1] 17
Figura 1.14. Interfaz virtual del LapMentor [23] 19
Figura 1.15. Interfaz virtual del LapSim [24]. 20
Figura 1.16. Ambiente 3D del LapBot [25]. 21
Figura 2.1. Pieza sólida en Blender (Izquierda), conversión a malla (derecha)..... 25
Figura 2.2. Interfaz gráfica de usuario de MakeHuman... 26
Figura 2.3. Niveles jerárquicos del software a utilizar. 26
Figura 2.4. Diferencia de los ejes entre Matlab® y Ogre3D. 27
Figura 2.5. Ejes de las articulaciones del robot en su estructura cinemática. 28
Figura 2.6. Pieza de la articulación 3 (Izq.), pieza de la articulación pasiva 5 (centro) y pieza de la articulación 2 (der.). 29
Figura 2.7. Pieza de la articulación pasiva 4 (izq.) y pieza de la articulación 1 (der.)... 29
Figura 2.8. Pieza de la articulación 6 (Izq.) y pieza de la articulación 7 (der.)....... 29
Figura 2.9. Nodos en una pieza móvil. 31
Figura 2.10. Orientación sin nodo padre (Izq.); orientación con nodo padre (der.). 31
Figura 2.11. Robot completamente renderizado. 31

Figura 2.12. Robot HIBOU en Ogre3D con sus articulaciones en cero radianes. . 32
Figura 2.13. Ambiente quirúrgico de la aplicación. 32
Figura 2.14. Cuerpo humano del ambiente virtual. 34
Figura 2.15. Órganos al interior del abdomen del paciente, virtual (Izq.) y real [28] (der.). 34
Figura 2.16. Vista lateral derecha del ambiente virtual. 35
Figura 2.17. Vista lateral izquierda del ambiente virtual. 36
Figura 2.18. Vista de la articulación manual en dos diferentes posiciones. 36
Figura 2.19. Rango de visibilidad de una cámara en Ogre3D. 37
Figura 2.20. Anillo que limita el campo de trabajo 37
Figura 2.21. Vista del interior del abdomen. 38
Figura 3.1. Esquema de control CTC [18]. 39
Figura 3.2. Modelos que intervienen en el lazo de control. 40
Figura 3.3. Posiciones de las articulaciones t2 y t3. 43
Figura 3.4. Optimización de la función Banana con 44
Figura 3.5. Optimización de la función Banana con *Nelder Mead*. 44
Figura 3.6. Esquema de prueba del MGI. 46
Figura 3.7. Error del LM ante una consigna circular. 47
Figura 3.8. Error del NM ante una consigna circular. 47
Figura 4.1. Error ante consigna circular en C++. 55
Figura 4.2. Error ante consigna circular en Simulink®. 56
Figura 4.3. Error de seguimiento articular del lazo de control en Simulink®. 56
Figura 4.4. Error ante una entrada recta constante del Joystick. 58
Figura 4.5. Error ante una consigna con cambios de dirección, deteniendo el robot. 58
Figura 4.6. Consigna con movimientos curvos del casco. 60
Figura 4.7. Error para la primera consigna del casco. 60
Figura 4.8. Consigna lineal con el casco. 61
Figura 4.9. Error para la segunda consigna del casco. 62
Figura 6.1. Lazo de control del algoritmo implementado 64
Figura B.1. Ventana para extraer el paquete de Ogre 4
Figura B.2. Ventana de la instalación del *AppWizard*. 5
Figura B.3. Comandos para agregar la variable de entorno. 5
Figura B.4. Abrir nuevo proyecto. 6
Figura B.5. Seleccionar tipo de proyecto (Izq.), Asistente de aplicaciones Ogre (der.) 6
Figura B.6. Explorador de soluciones 7
Figura B.7. Ventana de inicio de aplicación de Ogre 8
Figura B.8. Aplicación prueba de Ogre3D. 8
Figura B.9. Interfaz de Blender para importar un archivo. 9
Figura B.10. Pieza importada a Blender. 10

Figura B.11. Propiedades de los materiales. 10
Figura B.12. Panel de colores y efectos de malla. 11
Figura B.13. Ingresando al script de OGRE Meshes... 11
Figura B.14. Interfaz de OGRE Meshes. 11
Figura B.15. Preferencias de OGRE Meshes. 12
Figura B.16. Pieza renderizada en Ogre3D.. 12

Lista de Abreviaturas

HIBOU: Robot porta endoscopio para cirugía laparoscópica.

3D: Tridimensional.

C++: Lenguaje de programación orientado a objetos.

IDE: Entorno de desarrollo integrado.

Ogre3D: (Object-Oriented Graphics Rendering Engine)

CAD: Diseño asistido por computadora

CMI: Cirugía Mini-Invasiva

CO2: Dióxido de Carbono.

MGD: Modelo Geométrico Directo.

MGI: Modelo Geométrico Inverso.

MDI: Modelo Dinámico Inverso.

MDD: Modelo Dinámico Directo.

SYMORO®: (Symbolic Modeling Robots)

MATLAB®: (Matrix Laboratory)

LAPBOT: Robot para Laparoscopia

LGPL: Licencia Publica General Menor.

LM: Método de Levenberg Marquardt.

NM: Método de Nelder Mead.

CTC: (Computed Torque Control)

RK4: Método de Runge Kutta de cuarto orden.

Introducción

Reducir los riesgos de un paciente en el quirófano al momento de ser sometido a cirugía es un tema el cual presenta una constante investigación, es por eso que la medicina ha recurrido a la búsqueda de nuevas tecnologías que asistan al cirujano para lograr resultados óptimos, como son cirugías sin inconvenientes, menor tiempo de recuperación de los pacientes y menores tiempos de cirugías, entre otras cosas.

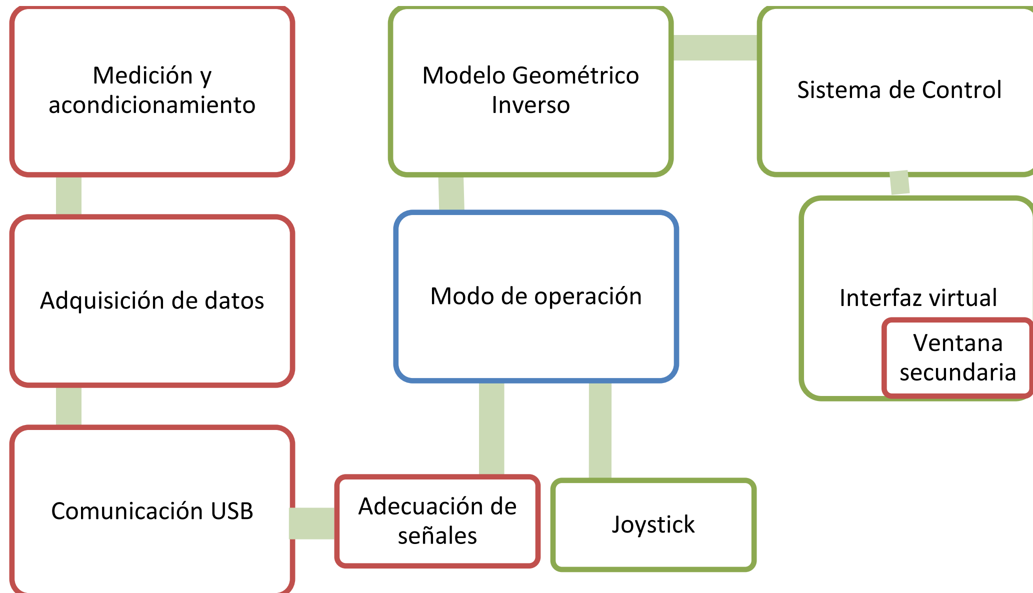
Una de estas tecnologías hace referencia a la robótica aplicada a la medicina la cual pretende unir las capacidades de un cirujano con el robot para mejorar los procedimientos quirúrgicos. El robot se convierte en un instrumento más, con la diferencia que esta es una herramienta inteligente que trata de compensar las diferencias y limitaciones que pueda tener el cirujano para realizar ciertas actuaciones.

De este modo, se hace posible la implementación de algunas técnicas de cirugía mínimamente invasiva gracias a la utilización de ayudas de soportes robotizados, consiguiendo minimizar la herida, reducir el tiempo de intervención y el de posterior recuperación.

Los robots presentan muchas formas de ayuda para el cirujano, pueden mejorar la percepción, memorizar posiciones, acceder a puntos determinados con gran precisión, entre otras cosas. Ayudas de este estilo suponen la diferencia en que algunas intervenciones se realicen o no. Las herramientas de asistencia médica pueden ir desde un brazo mecánico convencional hasta elementos de medida, como sensores que miden fuerza o cámaras de alta precisión que visualicen información de un modo más claro que como lo haría una cámara de televisión convencional.

El proyecto que se presenta a continuación está constituido por dos sub proyectos complementarios que mostraran el desarrollo de una aplicación capaz de manipular el robot porta endoscopio para cirugía laparoscópica (HIBOU) [1], por medio de un dispositivo de mando externo (casco), como se puede apreciar en el diagrama de bloques mostrado más abajo. El robot fue diseñado por estudiantes de la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad

del Cauca como trabajo de grado y fue presentado en simulación bajo Matlab® y Simulink® mediante una interfaz hombre-computador (joystick), en el año 2010 [1].



El objetivo de uno de los sub proyectos es el que se presenta en este trabajo, y es el mostrado en los bloques bordeados en color verde en el diagrama de arriba, el cual hace referencia a la implementación en C++ del lazo de control.

Se independizó el control y la simulación del robot (HIBOU), del ambiente virtual que tiene Simulink®, llamado Virtual Reality, y se colocó bajo un entorno 3D en C++. Fue utilizado el IDE (entorno de desarrollo integrado), Microsoft Visual Studio 2005 como herramienta de programación y OGRE 3D como motor gráfico base para la construcción del entorno que ayudaría a la visualización del proyecto conforme fuera avanzando. Luego todo fue integrado a un dispositivo de mando externo del cual se encargará el proyecto complementario.

La aplicación presenta en la ventana principal al HIBOU con la capacidad de ser manipulado por joystick y por el casco, realizando los movimientos de tal forma que se respete el paso por el trocar (pequeña incisión en el abdomen del paciente), y en la mini pantalla se ubica la cámara en la que se visualiza el interior del abdomen.

En el primer capítulo de este documento se dará un breve conocimiento de lo que existe hoy en la robótica médica tomando como referencia la cirugía mínimamente

invasiva y la robótica quirúrgica. Como primera instancia se presentan algunos conceptos básicos sobre esta temática, luego se mostrará con más detalle el robot porta endoscopio HIBOU y por último, los ambientes virtuales que más sobresalen de los simuladores médicos que se han desarrollado. Cabe aclarar que lo se presenta en este trabajo no es un simulador si no el software que manipulará al robot. En el segundo capítulo se exhibe la construcción y montaje del robot, detallando todas las herramientas que se utilizaron y cómo fueron utilizadas, para lograr posicionar al HIBOU en una ambiente tridimensional.

Más adelante en el capítulo 3 se hace mención de cómo se implementaron los distintos modelos del robot en el lenguaje C++ y cómo se solucionaron los inconvenientes de implementar modelos dinámicos en este lenguaje. En el capítulo 4 se habla acerca de cómo se comporta el HIBOU, ya con su esquema de control implementado, ante distintas entradas en el mando de control. Por último se concluirá acerca de los resultados y el trabajo realizado teniendo en cuenta que este proyecto es una labor previa para que en el futuro se realice la construcción real de un prototipo del robot porta endoscopio HIBOU y pueda ser manipulado por el software elaborado en este sub proyecto.

1. La cirugía robótica actual y el robot HIBOU

Los software de los robots tienen la característica de realizar la manipulación de ellos desde una interfaz de usuario amigable, como los son los ambientes virtuales. La función principal de estos tipos de software es realizar el control maestro-esclavo de forma remota permitiendo al usuario interactuar con él sin necesidad de hacer contacto [2].

A continuación se presenta un breve y detallado estado del arte de la robótica médica así como de los diferentes ambientes virtuales de simuladores, software de robots quirúrgicos y robots para cirugía laparoscópica utilizados, además del diseño del robot porta endoscopio (HIBOU), desarrollado en una tesis de pregrado anterior por estudiantes de la Universidad del Cauca.

1.1. LA ROBÓTICA APLICADA A LA MEDICINA

La robótica puede ser de gran utilidad en la medicina ya que es una herramienta muy apropiada en el momento de realizar cirugías complejas o donde el ojo humano le es difícil acceder. Algunos métodos sirven para la captación de imágenes, que permiten reconstruir modelos virtuales de los órganos que se necesitan examinar, para luego manipularlos y mejorar un diagnóstico o simular operaciones quirúrgicas.

Los grandes avances en informática y comunicaciones no solo se ven reflejados en la industria, también vemos un gran desarrollo en la medicina y de forma más particular en la cirugía, que ha dado como resultado un acelerado proceso de informatización de todas las áreas de la medicina. Así, los robots se están convirtiendo en un instrumento indispensable en esta ciencia, ya que son herramientas que han ayudado a mejorar múltiples procesos quirúrgicos [1].

La implementación de esta tecnología ha hecho que los mecanismos robotizados tengan mayor utilidad en técnicas avanzadas en donde los sentidos del ser

humano se encuentran altamente limitados. Es aquí en que las aplicaciones médicas entran a hacer parte de la investigación robótica.

Una de las áreas más tratadas por la robótica médica es la cirugía mínimamente invasiva que generó una revolución en la forma de realizar las intervenciones quirúrgicas, ya sea en las áreas de cirugía general, cirugía cardiovascular, cirugía pediátrica, ortopedia, urología, neurocirugía [3].

1.1.1. LA CIRUGÍA MINI-INVASIVA (CMI)

Podemos definir Cirugía Mínimamente Invasiva (CMI), también denominada de mínimo abordaje, como el conjunto de técnicas diagnósticas y terapéuticas que por visión directa, o endoscópica, o por otras técnicas de imagen, utiliza vías naturales o mínimos abordajes para introducir herramientas y actuar en diferentes partes del cuerpo humano [4].

El desarrollo de la Cirugía Mínimamente Invasiva se enmarca dentro de la historia reciente de la cirugía. Muchos autores señalan la colecistectomía laparoscópica, llevada a cabo por primera vez en 1985 por Muhe, en Alemania Occidental, como el evento que define el crecimiento explosivo de la Cirugía Mínimamente Invasiva moderna. Aunque existen referencias previas, algunas con carácter anecdótico, es a partir de los años ochenta cuando este tipo de cirugía vive su verdadero desarrollo y comienza su expansión [4].

La rapidez con la que se ha desarrollado esta técnica no tiene precedentes en la historia de la cirugía. Se podría poner como ejemplo la rápida evolución de la colecistectomía laparoscópica, que en menos de una década, en 1993, alcanzó en Estados Unidos un porcentaje del 67% frente a los procedimientos de cirugía abierta. Nunca hasta entonces se había producido una revolución tan grande en el campo de la cirugía, ni una nueva técnica había conseguido una aceptación universal tan rápida. De todos modos, la adopción de estas técnicas en otro tipo de operaciones diferentes a la colecistectomía ha sido mucho más lenta, debido fundamentalmente a la gran dificultad en el aprendizaje de estos procedimientos por parte del colectivo médico [4].

Las diferentes maneras en la que la CMI se encuentra clasificada dependen del lugar del cuerpo en el que se esté trabajando. En este proyecto se recrea la parte abdominal del cuerpo humano haciendo referencia a la cirugía endocavitaria laparoscópica.

1.1.2. LA CIRUGÍA LAPAROSCÓPICA

La laparoscopia es una técnica de endoscopia que permite la visión de la cavidad pélvica-abdominal con la ayuda de un tubo óptico. En este procedimiento quirúrgico se realizan únicamente pequeñas incisiones de aproximadamente 1 cm de diámetro. A diferencia de la cirugía tradicional, en lugar de mirar directamente al órgano del cuerpo que está siendo tratado, los médicos monitorizan el procedimiento a través de una videocámara especial llamada laparoscopio. Esta cámara se introduce a través de uno de los pequeños agujeros, mientras los otros instrumentos son manipulados desde otros puntos de incisión [5].

En la laparoscopia es necesario separar la pared abdominal de los órganos, elevando el abdomen para que el cirujano tenga acceso al interior del paciente y pueda manipular los instrumentos con relativa facilidad. Actualmente se emplea CO_2 para este fin [6]. En la figura 1.1, se puede apreciar la iluminación con fuente de luz fría del laparoscopio, los instrumentos que el cirujano manipula para realizar la operación y un conducto utilizado para insuflar gas inerte con el fin de distender la cavidad abdominal [1].

Es importante destacar que la laparoscopia, antes de ser una técnica quirúrgica, es un método de exploración, que consta de un componente visual que permite magnificar (de 10 a 20 veces) la cavidad abdominal, y un componente táctil que no es una característica inmediata, pero que permite identificar el campo de trabajo [7].

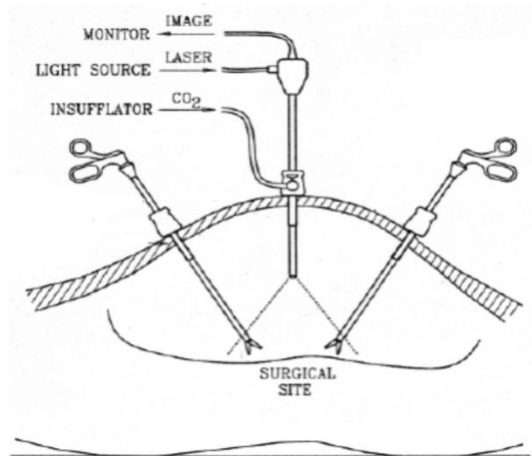


Figura 1.1. Región abdominal con instrumental laparoscópico [10].

En la actualidad los procedimientos laparoscópicos asistidos por robots involucran las partes del cuerpo en el interior del abdomen como son: vesícula biliar, esófago, estómago, útero, riñón, apéndice y colon [8]. En la figura 1.2, se detalla el procedimiento de extracción de vesícula biliar y las herramientas utilizadas para llevarla a cabo.

Figura 1.2. Vista ampliada del procedimiento quirúrgico de extracción de vesícula biliar (cuerpo en verde) [8].

Además de la colecistectomía vista en la figura 1.2, actualmente se realizan varios procedimientos quirúrgicos laparoscópicos, siendo los más comunes [5].

- **Apendicectomía:** extirpación de un apéndice inflamado o infectado (apendicitis).
- **Histerectomía:** cirugía para extirpar el útero (matriz).
- **Colectomía asistida:** extirpación o resección de una parte enferma del intestino grueso (colon).
- **Ligadura de trompas:** es una forma permanente de evitar el embarazo cerrando las trompas de Falopio de una mujer.
- **Gastrectomía:** extirpación parcial o total del estómago, en caso de presentarse problemas gástricos crónicos como úlceras o cáncer.
- **Prostatectomía:** intervención para extraer la totalidad o parte de la glándula prostática (próstata), para la curación del cáncer, la preservación de la continencia o la función sexual.
- **Hepatectomía:** procedimiento para diagnosticar, tratar o evacuar las lesiones y tumores en el hígado.
- **Vasectomía:** procedimiento para lograr la esterilización masculina. Consiste en cortar y ligar los conductos deferentes, impidiendo el recorrido del semen.

Es conveniente mencionar que por medio de otras incisiones, aparte de las del sistema de visión (endoscopio), se insertan pinzas, tijeras, cauterizadores o cualquier otro instrumento que se requiera [5].

1.1.3. ROBÓTICA QUIRÚRGICA

Las ventajas más notables de los robots quirúrgicos son la precisión y la miniaturización. Estos robots son utilizados, entre otros, en el ámbito de la cirugía cardiaca, gastrointestinal, pediátrica o de la neurocirugía. Entre las aplicaciones de la robótica quirúrgica, podemos destacar dos: la tele-cirugía y la cirugía mínimamente invasiva [9].

Los robots quirúrgicos pueden ser clasificados de muchas maneras:

- Por su nivel de autonomía, por ejemplo pre-programado, tele-operado o control cooperativo restringido
- Por objetivo anatómico o técnica, como lo pueden ser cardíaca, intravascular, percutánea, laparoscópica, o microcirugía.
- Los que operan pensado en el entorno a modo de escáner, y en la sala de operaciones convencional.

La meta de la robótica quirúrgica es no reemplazar al cirujano por un robot, por el contrario es proporcionarle un nuevo juego de herramientas muy versátiles que extiendan las habilidades del cirujano con el fin de mejorar los tratamientos en los pacientes [3]. Ahora mostramos algunos robots que han representado a la robótica quirúrgica en lo que va de su historia.

1.1.3.1. DA VINCI

El sistema quirúrgico Da Vinci es un sistema desarrollado por *Intuitive Surgical,* que consiste en un robot diseñado para posibilitar cirugías complejas con invasiones mínimas al cuerpo humano, con mayor visión, precisión, destreza y control. El sistema es usado especialmente para operaciones de próstata, reparaciones de válvulas cardiacas y procedimientos quirúrgicos ginecológicos [10]. El sistema consta de tres partes principales:

1. La consola del cirujano, que está controlada por el mismo cirujano sentado en una posición cómoda y ergonómica.

2. El robot quirúrgico, que se sitúa junto a la mesa de operaciones en la que está el paciente y del que salen dos brazos que realizan directamente el procedimiento.
3. El sistema de visión, que es el tercer brazo del robot quirúrgico, sostiene una cámara endoscópica en 3D de alta calidad.

Un cuarto brazo puede emplearse para reemplazar a un asistente. La consola del cirujano consiste en un visualizador que presenta imágenes 3D obtenidas a partir de la cámara endoscópica que está dentro del cuerpo del paciente. El término "manipulación amo-esclavo" se refiere a la consola del cirujano, equipada con manipuladores que controlan los movimientos de los brazos quirúrgicos que sostienen los instrumentos y, como segundo, el manipulador de la cámara endoscópica durante el procedimiento [11].

Figura 1.3. Sistema quirúrgico Da Vinci (Standard) [10].

1.1.3.2. ZEUS.

El Proyecto Zeus nació en Goleta CA en Estados Unidos, y fue desarrollado por Computer Motion en 1997. Es un sistema *master-slave*, que consiste en una unidad de control, dos brazos mecánicos y el robot *Aesop*, todo en conjunto forman el robot Zeus. El robot Zeus contiene una computadora que se encuentra dentro de la consola del cirujano, que controla los tres brazos, esta es la que lleva el rastro de la posición tridimensional de la punta de cada instrumento y de la cámara el cirujano, observándose la operación con un sistema de imágenes tridimensionales. El brazo robótico que controla cámara era accionado mediante comandos de voz por el cirujano, en la figura 1.4 se puede observar el Zeus [12].

Figura 1.4. Robot Zeus [12].

1.1.3.3. ROBODOC.

Éste robot es desarrollado por *Integrated Surgical System Inc.* en California Estados Unidos, está compuesto por un robot SCARA de cinco grados de libertad, equipado con fresa de alta velocidad, un monitor de sala con Pedant y el armario de control. Este sistema robotizado sirve para realizar operaciones, como el vaciado de hueso en prótesis de cadera y rodilla, sus principales ventajas son: precisión del 98%, no necesita cementación, minimiza micromovientos entre el implante y hueso [13].

Figura 1.5. Robodoc [13]

1.1.4. ROBOTS PORTA ENDOSCOPIOS

Un robot laparoscópico pertenece al grupo de los robots clínicos y su tarea específica consiste en auxiliar al médico cirujano cuando está practicando una

cirugía por la técnica de laparoscopia, contribuyendo con la fijación y el control de los movimientos del laparoscópio.

La función del robot es sostener y posicionar un instrumento óptico alargado (laparoscópio) que se introduce en el organismo a través de un orificio en el abdomen o en la zona donde se requiere explorar. Este instrumento óptico funciona como una cámara de video y permite ver lo que hay en el interior del organismo. El cirujano le da instrucciones al robot para que se desplace y así ubique la zona de interés [1]. A continuación se hace referencia a algunos de los robots porta endoscopios que existen en el mercado.

1.1.4.1. LAPMAN

Es un robot porta endoscopio fabricado por MEDSYS en Belgica, que puede ser fácilmente trasladado desde y hacia la mesa de operaciones. Su eje tiene un laparoscópio que se mueve mediante órdenes a distancia, estas órdenes son enviadas a través del mando de control RF LapStick®. LapMan asegura la estabilidad de la imagen, el control de los movimientos de la cámara recae únicamente en el cirujano y libera un asistente de cirugía para ampliar sus funcionas de apoyo [14].

Figura 1.6. Robot LapMan [14].

1.1.4.2. ENDOASSIST.

Es un robot fabricado por *Armstrong Healthcare Ltd.* en Inglaterra Es un robot porta endoscopio controlado por los movimientos de la cabeza del cirujano. Este robot permite fácilmente insertar el sistema de video dentro de la cavidad pélvica-abdominal; la naturaleza de su interfaz humano-computador permite lograr el dominio del robot en poco tiempo [15].

Figura 1.7. Endoassist [15].

1.1.4.3. LER.

Light Endoscopio Robot, robot creado por TIMC, Grenoble, Francia. Brazo mecánico articulado unido a un computador, el cual se encarga de interpretar órdenes verbales simples que el robot convierte en movimientos de la cámara laparoscópica. Este sistema consta de tres partes: una base móvil, un laparoscopio de flexión, y un manipulador externo. La particularidad de este robot es su reducido tamaño por tanto es muy útil, ya que se puede acoplar en cualquier quirófano sin necesitar de grandes espacios [16].

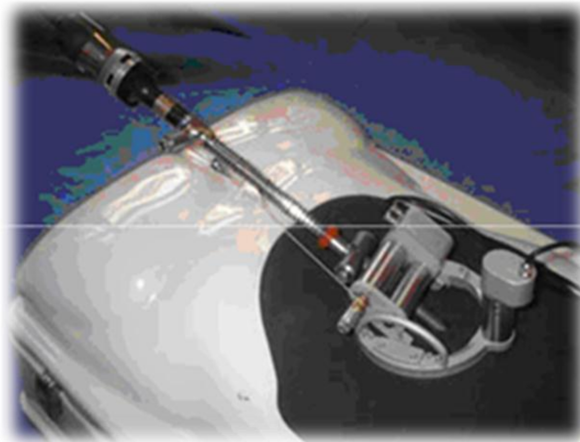


Figura 1.8. LER [16].

1.2. ROBOT PORTA ENDOSCOPIO HIBOU

El robot porta endoscopio HIBOU es un robot serie que fue diseñado, en una tesis anterior, a partir del estudio de las estructuras cinemáticas existentes. En dicho análisis se identificaron las articulaciones que proporcionan el posicionamiento y las que proporcionan la orientación de la cámara, llegando a la conclusión que para lograr una estructura que pueda posicionar y orientar correctamente el efector final del robot en cualquier punto del espacio cartesiano, al interior del abdomen y respetando el paso por el trocar, se hacen necesarias siete articulaciones, de las cuales cinco son motorizadas y dos son no motorizadas (articulaciones pasivas). La técnica de articulaciones pasivas es utilizada para asegurar el paso por el trocar u orificio abdominal. La estructura cinemática del robot HIBOU se puede observar en la figura 1.9.

El punto de incisión abdominal (trocar) se representa con un círculo entre las articulaciones 5 y 6, de manera que las articulaciones 6 y 7 quedarían dentro del abdomen del paciente. De otra parte, debido a la disposición física de las articulaciones 6 y 7, la distancia D7 se hace igual a cero.

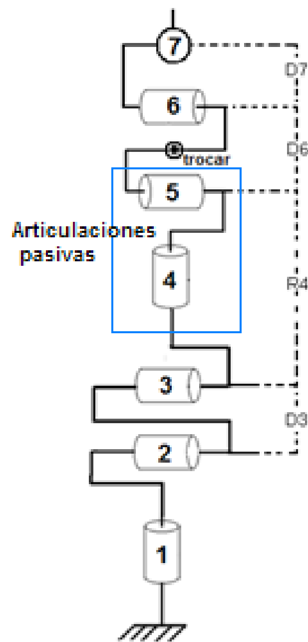


Figura 1.9. Estructura cinemática del HIBOU [1].

1.2.1. MODELADO MATEMATICO DEL ROBOT HIBOU

Para el cálculo de los modelados matemáticos de la anterior estructura cinemática se utilizó el método de Khalil–Kleinfinger [17], [18]. Utilizando este método se obtiene la tabla de parámetros geométricos que se muestra a continuación (tabla 1.1).

j	ant	M	σ	A	D	θ	R
1	0	1	0	0	0	t1	0
2	1	1	0	-90	0	t2	0
3	2	1	0	0	D3	t3	0
4	3	0	0	90	0	t4	R4
5	4	0	0	-90	0	t5	0
6	5	1	0	0	D6	t6	0
7	6	1	0	-90	D7	t7	0

Tabla 1.1. Parámetros geométricos del HIBOU [1].

Dónde:

- **j:** Número de la articulación.
- **σ:** Indica si la articulación es rotoide (0) o prismática (1).
- **μ:** Indica si la articulación es activa (motorizada = 1) o pasiva (no motorizada=0).
- **α:** Ángulo entre los ejes Z_{j-1} y Z_j, correspondiente a una rotación alrededor de $X_{j-1.}$
- **d:** Distancia entre Z_{j-1} y Z_j a lo largo de $X_{j-1.}$
- **θ:** Ángulo entre los ejes X_{j-1} y X_j correspondiente a una rotación alrededor de $Z_{j.}$
- **r:** Distancia entre X_{j-1} y X_j a lo largo de $Z_{j.}$

1.2.1.1. MODELO GEOMÉTRICO DIRECTO

A partir de esta tabla se obtiene el modelo geométrico directo (MGD, matriz de transformación ${}^{0}T_{7}$), que permite calcular la posición y orientación del efector final del robot en el espacio cartesiano, a partir de las posiciones articulares de cada grado de libertad del robot en cualquier momento. Este modelo permite saber la posición exacta de la cámara sin hacer uso de un sistema de visión. En (1.1), se puede apreciar la matriz de transformación del sistema cero tomado como la base

del robot, al sistema 7, que es donde se encuentra el efector final, *A* es una matriz 3x3 que representa la orientación y **P**(x, y, z) es un vector columna que representa la posición del instrumento [1].

$$ {}^{0}T_{7} = \begin{bmatrix} & {}^{0}A_{7} & & {}^{0}P_{7} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.1} $$

Las ecuaciones que representan al vector de posición *P* y la matriz de orientación A, se pueden encontrar en el Anexo A.

1.2.1.2. MODELO GEOMETRICO INVERSO

El MGI permite encontrar el valor de las variables de cada articulación sabiendo la posición y la orientación en el sistema de coordenadas cartesianas de la cámara

Para encontrar el MGI se hizo uso del método de Paul [19], integrado en este método la restricción del paso por el trocar, la cual consiste en limitar los movimientos de posicionamiento de tal forma que el cuerpo del robot que se introduce dentro del paciente siempre tenga un punto de rotación fijo, que será el mismo punto por donde se hace la incisión en el abdomen (trocar).

Las articulaciones a las que afecta la restricción son las que se encuentran antes y después del trocar como aparece en la figura 1.10, siendo la quinta y sexta articulación las que se encuentran justo antes y después de la incisión. Los vectores de posición que representan a esas articulaciones son los vectores *P5* y *P6*, respectivamente, además el punto de incisión es identificado con el vector *Pt* [1].

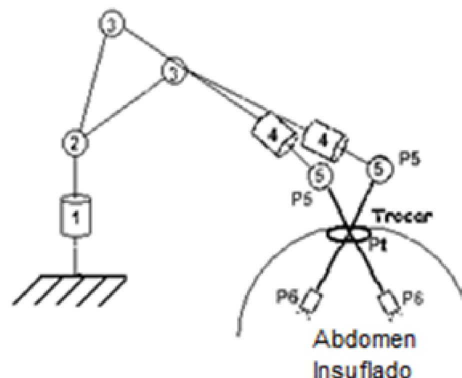


Figura 1.10. Movimiento de las articulaciones alrededor del punto de incisión (trocar) [1].

Observando la figura 1.11, la restricción consiste en expresar el vector $\overline{P5P6}$, como un producto cruz entre los vectores $\overline{P5Pt}$ y $\overline{PtP6}$, es decir que el vector $\overline{P5P6}$ sea colineal con los vectores $\overline{P5Pt}$ y $\overline{PtP6}$, como muestra en la ecuación [5].

$$(Pt - P5)\ X\ (P6 - Pt) = 0 \qquad (1.2)$$

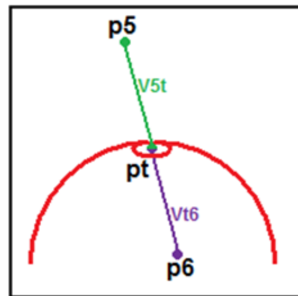


Figura 1.11. Restricción de colinealidad del vector $\overline{P5P6}$ con $\overline{P5Pt}$ y $\overline{PtP6}$ [1].

Una vez planteadas las ecuaciones (Anexo A), se procede a calcular las soluciones para las variables articulares, mediante la combinación del método de Paul y un método numérico. Este último se utilizó debido la dificultad para encontrar las soluciones de ciertas variables articulares del robot. El método numérico consiste en probar una serie de valores para cada variable a partir de unas condiciones iniciales. Cuando el resultado de reemplazar los valores de cada variable en las ecuaciones dadas para la solución del modelo es cercano a cero, (aproximadamente $1e^{-15}$), entonces el proceso se detiene y entrega los valores de cada variable para los cuales se cumplió el mínimo resultado posible.

El método numérico utilizado para el cálculo del MGI fue el algoritmo de *Levenberg Marquardt*. La solución del método depende de las derivadas parciales de la función objetivo para hallar la solución a partir de unas condiciones iniciales [20],[21]. En Matlab® existe una herramienta que facilita este procedimiento desde el comando *f_solve*.

Para el cálculo de los modelos dinámicos se presenta el diseño del cuerpo de HIBOU. Estos modelos permiten representar la relación entre las fuerzas ejercidas por los motores y la posición, velocidad y aceleración de las articulaciones del robot.

1.2.1.3. DISEÑO DEL CUERPO DEL HIBOU.

Con base en unos criterios de diseño se seleccionaron las dimensiones del robot (figura 1.12) en relación con la tabla de parámetros mostrada en una sección anterior, son las siguientes:

$$D3 = 0.3\,m \qquad R4 = 0.4\,m \qquad D6 = 0.3\,m \qquad D7 = 0\,m$$

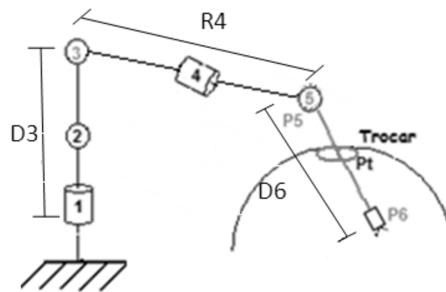


Figura 1.12. Dimensiones del HIBOU [1].

Ahora presentamos la construcción del robot (figura 1.13), que se diseñó en Solid Edge®

Figura 1.13. Robot HIBOU en SolidEdge® [1]

1.2.1.4. MODELOS DINÁMICOS.

Una vez tenido el diseño del robot se calculan los parámetros de base ya que estos permiten eliminar aquellos parámetros que no tienen efecto sobre el modelo y agrupar otros con el fin de simplificar ecuaciones finales del modelo [1].

Para el MDI (Modelo Dinámico Inverso) y el MDD (Modelo Dinámico Directo), es necesario tener los valores de masas e inercias de cada una de las piezas mostradas en la sub-sección anterior. SolidEdge ayuda a calcular esos valores

físicos de los cuerpos que posteriormente son introducidos al software de modelado simbólico de robots SYMORO® [22]. El software proporciona la tabla de parámetros mínimos del robot

Con la ayuda de SYMORO® se obtuvo el MDI, que calcula los pares mecánicos a aplicar en los motores en términos de la posición, velocidad y aceleración de cada articulación (1.3).

$$\boldsymbol{\Gamma} = A(q)\ddot{q} + C(q,\dot{q})\dot{q} + Q(q) \quad (1.3)$$

Dónde:

- $\boldsymbol{\Gamma}$: Pares aplicados a los motores.
- $\boldsymbol{A}$: Matriz de inercias.
- $\boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}})$: Matriz de fuerzas Coriolis y centrifugas.
- $\boldsymbol{Q}(\boldsymbol{q})$: Vector de fuerzas centrifugas.
- $\boldsymbol{q}$: Posiciones articulares.
- $\dot{\boldsymbol{q}}$: Velocidades articulares.
- $\ddot{\boldsymbol{q}}$: Aceleraciones articulares.

Para el cálculo del MDD se despejan las aceleraciones articulares de (1.3) la expresión de este modelo se muestra en la ecuación (1.4)**.** También presentamos los parámetros inerciales logrados con el diseño del robot en la tabla 1.4 [1].

$$\ddot{q} = inv\big(A(q)\big) * \big(\sigma - C(q,\dot{q})\dot{q} - Q(q)\big) \quad (1.4)$$

Parámetro	**Valor**	**Parámetro**	**Valor**
$\boldsymbol{XX2R}$	-0.5720	$\boldsymbol{ZZ4R}$	0.9269
$\boldsymbol{XX3R}$	0.3830	$\boldsymbol{ZZ5R}$	0.1867
$\boldsymbol{XX4R}$	-0.0810	$\boldsymbol{ZZ6R}$	0.4802
$\boldsymbol{XX5R}$	-0.1865	$\boldsymbol{ZZ7}$	0.2400
$\boldsymbol{XX6R}$	2.1000e-005	$\boldsymbol{MX2R}$	2.0531
$\boldsymbol{XX7R}$	-0.2402	$\boldsymbol{MX3}$	1.1439
$\boldsymbol{XZ2R}$	5.6790e-004	$\boldsymbol{MX4}$	0.8348
$\boldsymbol{XZ5R}$	-1.9054e-004	$\boldsymbol{MX5R}$	0.3280
$\boldsymbol{YZ3R}$	3.4341e-005	$\boldsymbol{MX6}$	0.2648
$\boldsymbol{YZ4R}$	-1.7729e-005	$\boldsymbol{MX7}$	0.2648
$\boldsymbol{YZ6R}$	-1.0044e-005	$\boldsymbol{MY3R}$	-0.8065
$\boldsymbol{ZZ1R}$	1.0497	$\boldsymbol{MY4R}$	-0.0011
$\boldsymbol{ZZ2R}$	0.5886	$\boldsymbol{MY6R}$	-5.5061e-004
$\boldsymbol{ZZ3R}$	1.2854		

Tabla 1.2. Parámetros inerciales de base del HIBOU [1].

1.3. AMBIENTES VIRTUALES EN LA CIRUGÍA

Los ambientes virtuales son herramientas para la construcción de interfaces gráficas que interactúan de una forma más amigable con el usuario. Los ambientes virtuales de los simuladores para cirugía laparoscópica son los que se encuentran con mayor presencia en el mercado, así que a continuación hacemos reseña de algunos de ellos y de uno desarrollado en la Universidad del Cauca.

1.3.1. LAP MENTOR

El LAP Mentor es un simulador quirúrgico multi-disciplinario, permite la práctica de uno o varios alumnos. El sistema ofrece oportunidades de capacitación a los cirujanos nuevos y experimentados, desde el perfeccionamiento de habilidades básicas de laparoscopía hasta realizar procedimientos quirúrgicos laparoscópicos completos. El sistema cuenta con siete diferentes procedimientos laparoscopicos y más de 60 escenarios del paciente, incluyendo la cirugía general, ginecología, urología, cirugía bariatrica y cirugía de colon y recto.

El LapMentor permite la simulación de las técnicas quirúrgicas con alta fidelidad para que los estudiantes experimenten una serie de procedimientos laparoscopicos en un ambiente seguro. La figura 1.14 se presenta la interfaz virtual en la que se practica un procedimiento laparoscópico [23].

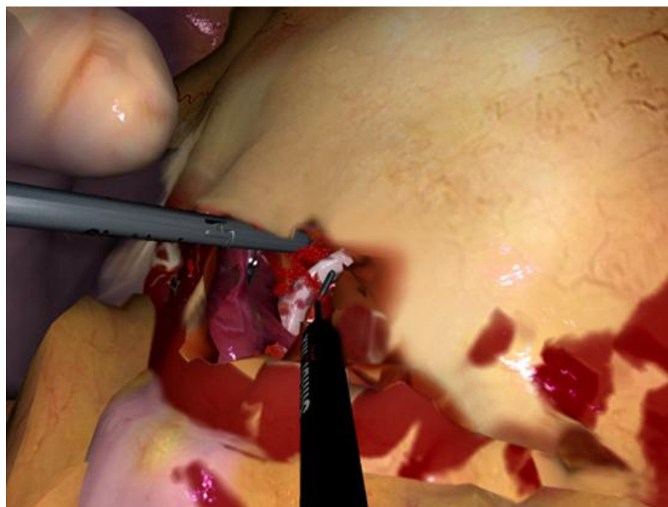


Figura 1.14. Interfaz virtual del LapMentor [23]

1.3.2. VIRTUAL LAPAROSCOPY SIMULATOR (LapSimÂ®)

El entrenamiento de las habilidades básicas incluye la navegación con cámara, la navegación con instrumental, la coordinación, grapado, disección, colocación de clips, sutura y medición de precisión y velocidad. En todos los ejercicios el cirujano debe identificar el objeto propuesto y ubicar los instrumentos, realizando la tarea requerida con la mayor precisión y en el menor tiempo posible; para eso se hace uso de la interfaz virtual vista en la figura 1.15 [24].

Figura 1.15. Interfaz virtual del LapSim [24].

1.3.3. POSICIONAMIENTO DE LAPBOT EN UN AMBIENTE TRIDIMENSIONAL VIRTUAL USANDO INTERFAZ HÁPTICA

Software desarrollado por estudiantes de la Universidad del Cauca para simular el robot quirúrgico para cirugía laparoscopia LapBot, que es posicionado a través de una interfaz háptica en un ambiente tridimensional construido usando el programa de representación gráfica por computador Ogre3D.

Esta aplicación cuenta con el kit para desarrollo de software *Ogre Haptics*, que permite comunicar la interfaz háptica con el motor de renderizado gráfico, haciendo que un objeto virtual creado en Ogre3D siga el movimiento generado por el usuario en el mundo real con la interfaz háptica *Novint Falcon* [25]. La interfaz virtual del LapBot se aprecia en la figura 1.16.

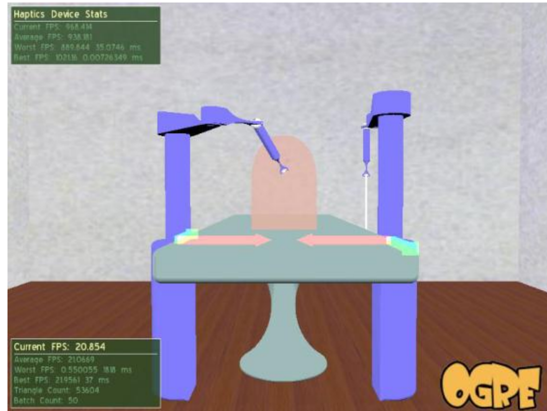


Figura 1.16. Ambiente 3D del LapBot [25].

2. Construcción del ambiente virtual

Se le llama ambiente virtual a una interfaz tridimensional generada por computadora de algún aspecto del mundo real, en el cual el usuario puede interactuar con él. Por ejemplo, un usuario puede realizar acciones dentro de un modelo virtual como son desplazar y mover objetos para experimentar situaciones que se asemejan al mundo real

Existen diferentes tipos de ambientes virtuales, el primero de éstos es llamado ambiente inmersivo. Los ambientes inmersivos son aquellos sistemas donde el usuario se siente dentro del mundo virtual que está explorando. Este tipo de ambientes utiliza diferentes dispositivos o accesorios como pueden ser guantes, trajes especiales, visores o cascos; éstos últimos le permiten al usuario visualizar el mundo a través de ellos. Los cascos son el principal elemento que hacen al usuario sentirse inmerso dentro del mundo. Este tipo de sistemas son usados para aplicaciones de entrenamiento o capacitación [26].

El segundo tipo de ambiente es llamado ambiente no inmersivo. Los ambientes no inmersivos o de escritorio, son aquellos donde el monitor es la ventana hacia el mundo virtual y la interacción es por medio del teclado, micrófono, ratón o joystick. Se emplean para visualizaciones científicas, enseñanza o como medio de entretenimiento y aunque no ofrecen una total inmersión, son una alternativa de bajo costo [26].

El ambiente virtual que se presenta en este proyecto es de tipo no inmersivo, en el cual se posicionará el robot HIBOU, que será manipulado por medio de un joystick. Éste le entrega al sistema tres posiciones cartesianas y sus cambios se ven reflejados en los movimientos del HIBOU, los cuales son una aproximación de cómo se debería mover el robot real.

Como se mencionó antes este desarrollo va de la mano con un proyecto complementario que se encarga de manipular el robot por medio de un dispositivo de mando externo. Este dispositivo también le entrega al sistema tres posiciones cartesianas que obtiene a partir de un acelerómetro incrustado en un casco, el cual llevara puesto el usuario en el momento de la cirugía, pero todo esto será explicado con más detalle en este otro proyecto.

El joystick es la herramienta que permite probar la interfaz virtual del robot además de reducir los inconvenientes que se puedan presentar en el desarrollo de la aplicación antes de introducir el dispositivo de mando externo.

En la construcción del ambiente 3D es necesario un entorno de IDE (Ambiente de Desarrollo Integrado), que permita la codificación en lenguaje C++. También es necesario un motor gráfico que facilite el renderizado de mallas 3D, además de un software CAD (Diseño Asistido por Computadora), que permita un rápido aprendizaje en la construcción de objetos tridimensionales.

2.1. REQUERIMIENTOS DE LA APLICACIÓN

Como se dijo en un capítulo anterior, el robot HIBOU ya había sido construido en un ambiente virtual en un proyecto de tesis anterior, este ambiente fue montado en Virtual Reality que trabaja directamente con Matlab® y Simulink®.

Matlab® es una herramienta de cálculo matemático que consume demasiados recursos del sistema por lo que bajo ella es prácticamente imposible crear una aplicación que se ejecute de forma rápida ante cambios en la consigna. El requerimiento es crear un nuevo ambiente capaz de soportar una interacción con el robot de forma más rápida, además de ser totalmente independiente a los algoritmos utilizados por Matlab, es decir que la herramienta a diseñar debe ser escrita bajo código abierto. Este sistema deberá ser robusto evitando posibles fallas en la ejecución del programa que se puedan presentar.

También según pautas establecidas al inicio del proyecto, además del robot HIBOU, el entorno virtual debe presentar un ambiente con cierta similitud a un quirófano junto con un paciente recostado sobre una mesa de cirugía. Este paciente, debido a que el robot es principalmente utilizado para asistencia en una cirugía laparoscópica, debe presentar en el interior del abdomen los órganos que comúnmente se verían en este tipo de cirugía.

2.2. HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO DE LA APLICACIÓN

Para escoger las herramientas utilizadas en el desarrollo de la aplicación, básicamente se tuvieron en cuenta tres criterios, uno que permita un rápido y fácil aprendizaje; dos, la existencia de un amplio soporte técnico en internet (foros); y tres, que no tengan problemas sobre la plataforma a trabajar (Windows). Los foros hacen que sea posible interactuar con los demás desarrolladores del mundo y

facilita la aclaración de dudas y la solución de problemas que se pueda llegar a tener en la escritura del código o en el renderizado de mallas 3D.

C++ es prácticamente un lenguaje universal, teniendo así, en existencia, una amplia gama de foros en la red. El entorno de desarrollo para aplicaciones Windows con el cual se trabajó es Microsoft Visual Studio® al que se hace referencia a continuación.

2.2.1. MICROSOFT VISUAL STUDIO ® [27]

Microsoft Visual Studio permite crear aplicaciones capaces de ejecutarse de una manera más rápida, además tiene herramientas que facilita el desarrollo de aplicaciones. Existen motores gráficos que trabajan sobre este IDE, que son de libre uso, tal y como lo es OGRE3D (Motor de Renderizado Grafico Orientado a Objetos).

2.2.2. OGRE3D [28]

Ogre3D (*Object-Oriented Graphics Rendering Engine*) es un motor gráfico de fuente abierta orientado a objetos escrito en C++. Tiene licencia GNU Licencia Pública General Menor (LGPL) que permite su uso libre con algunas pequeñas restricciones. Fue diseñado con el objetivo de facilitar el trabajo de los desarrolladores que producen aplicaciones basadas en hardware de aceleración gráfica 3D.

Hay que destacar que OGRE es un componente en un largo sistema de desarrollo. OGRE no es, y nunca fue, pensado como una plataforma para el desarrollo de video juegos, el propósito específico de esta herramienta es llevar objetos a un ambiente virtual. Ogre no contiene en su paquete las librerías que son necesarias para crear video juegos pero estas pueden ser fácilmente integrables a su estructura.

El motor de Ogre3D genera objetos en un ambiente tridimensional a partir de mallas triangulares. Para llevar una malla al ambiente virtual de Ogre es necesario haberla construido en un software de diseño asistido por computadora. Blender y Solid Edge® son los software de diseño utilizados en este trabajo.

2.2.3. BLENDER [29]

Las piezas elaboradas en SolidEdge® son importadas desde Blender, estas mallas son guardadas en archivos con extensión *.mesh* y luego cargadas en Visual Studio, como se puede ver en la figura 2.1, a la izquierda esta una de las piezas del robot antes de ser convertida en malla, a la derecha se observa la misma pieza en forma de malla, con la cual trabaja el motor de Ogre.

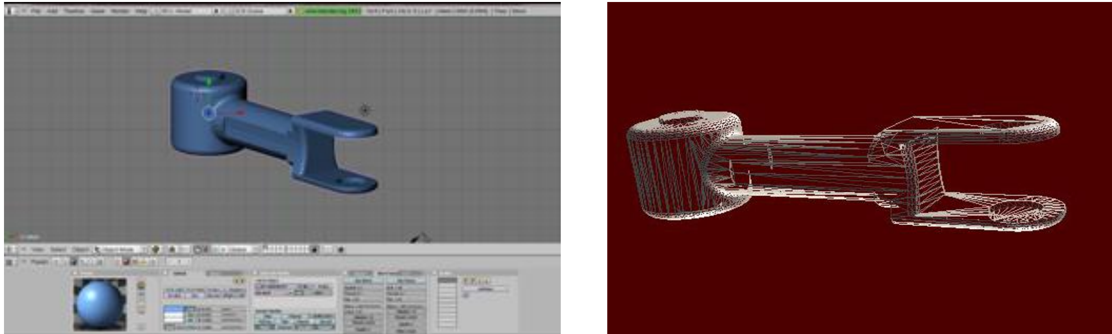


Figura 2.1. Pieza sólida en Blender (Izquierda), conversión a malla (derecha).

2.2.4. SOLID EDGE® [30]

Las piezas construidas para el robot HIBOU fueron diseñadas anteriormente en SolidEdge® así como su ambiente virtual integrado a Virtual Reality. Ya que el propósito de SolidEdge® es modelado de piezas para el diseño mecánico, la construcción de mallas para otro propósito de animación, como lo son cuerpos humanos y otros objetos se hace mucho más complejo. Este programa no es capaz de generar mallas a partir de las piezas creadas, por lo tanto estas son exportadas con extensión *.stl* a Blender, para su posterior renderizado.

2.2.5. MAKE HUMAN [31]

Para este proyecto es necesario colocar un paciente en el ambiente virtual, pero crear un humano en los programas mencionados anteriormente es muy complicado, por lo tanto se hace uso del programa llamado Make Human el cual está hecho específicamente para el modelado de personas en 3D, este programa es *open source* lo que facilita su implementación en este proyecto. Aunque es muy útil esta herramienta, el humano creado desde este programa no está listo para ser puesto en el ambiente virtual por lo que es exportado a Blender donde se le

agregan detalles, como material, color, ropa y orientación para luego ser transformado en malla y renderizado desde Ogre.

Este software permite la exportación a una gran gama de formatos, entre ellos el *.obj*, el cual es compatible con el Blender, en la figura 2.2, se puede observar como se ve el humano creado en Make Human.

Figura 2.2. Interfaz gráfica de usuario de MakeHuman.

Construir un ambiente virtual es bastante complejo por lo cual requiere mucha paciencia en su construcción. Los programas anteriormente mencionados fueron las herramientas que facilitaron ese diseño y los niveles de jerarquía se ven en la figura 2.3.

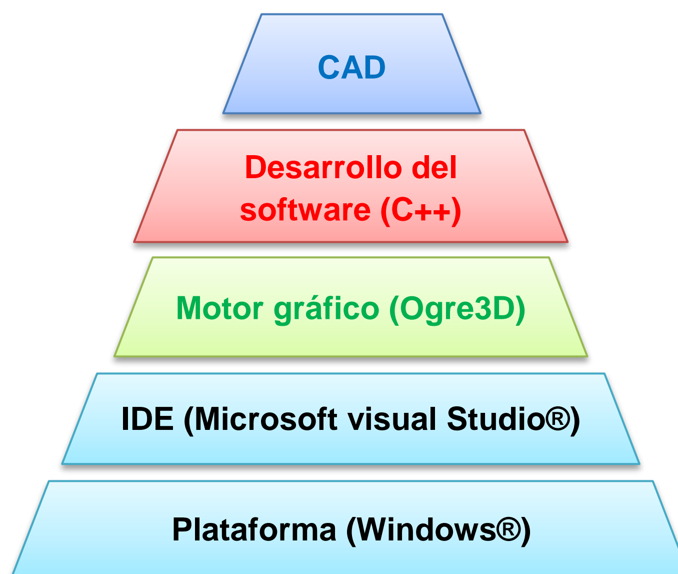


Figura 2.3. Niveles jerárquicos del software a utilizar.

2.3. AMBIENTE VIRTUAL DE OGRE3D

Integrar la librería OgreSDK_vc8 a Microsoft Visual Studio 2005 no es muy complicado utilizando las últimas versiones de los paquetes de Ogre. La versión utilizada para este proyecto es la *v1.7.2* y no requiere de muchos pasos para su instalación, además de que tiene asistentes que facilitan el direccionamiento de las *dll's* necesarias para su compilación. Para más detalles ver anexo B en la sección de instalación de ogre3D para Visual Studio 2005.

Construir una escena con las librerías de Ogre es muy sencillo. Los formatos de malla que recibe son de extensión *.mesh*, y *.material* para poder renderizar el material necesario como lo son colores y efectos a la malla. Para llevar una malla a tal formato, Ogre tiene una interfaz que trabaja junto con Blender para realizar esa conversión, y en Blender se puede agregar el material que se necesita (ver Anexo B).

Es muy importante saber cuáles son los ejes *X, Y* y *Z* con los que trabaja Ogre3D pues son los que orientan las posiciones de cada objeto así como sus ángulos de inclinación requeridos para recrear una animación amigable. Los ejes del ambiente virtual se pueden observar en la figura 2.4, y es importante compararlos con los ejes de Matlab® pues su orientación es diferente.

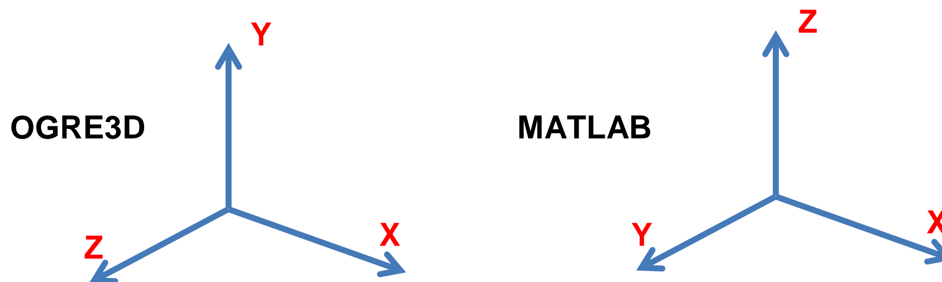


Figura 2.4. Diferencia de los ejes entre Matlab® y Ogre3D.

Como se puede ver el eje Z tiene diferente orientación y para dar solución a ese problema, la arquitectura de Ogre nos permite inclinar las piezas sin necesidad de cambiar su eje de giro, con unas simples líneas de código que más adelante se explicarán.

2.4. CONSTRUCCIÓN DE MALLAS TRIDIMENSIONALES

Esta sección se divide en tres grupos los cuales son el robot HIBOU, el cuarto quirúrgico y el cuerpo humano con órganos en el interior del abdomen (paciente).

2.4.1. Robot HIBOU

Para cada una de las piezas que componen la estructura del robot se utilizaron los archivos SolidEdge® que se diseñaron en el trabajo de grado anterior. El problema que tienen estas mallas es que no giran sobre el eje Z, debido a que en el momento de su construcción no se tomaron en cuenta los ejes de la estructura cinemática, resultando en que en la animación del robot el eje de giro no corresponde al eje Z.

Para evitar estos problemas al momento de la implementación de los modelos se reconstruyeron nuevamente las piezas en SolidEdge®, considerando sus medidas y, teniendo en cuenta la figura 2.5., en la que se pueden ver los ejes X y Z de cada pieza.

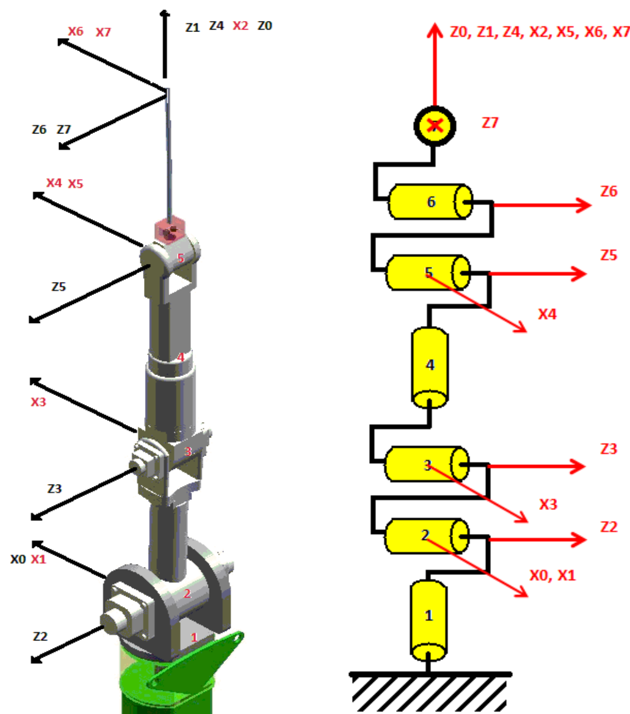


Figura 2.5. Ejes de las articulaciones del robot en su estructura cinemática.

Aunque los ejes para SolidEdge® también son diferentes, la pieza es exportada tal y como se construye, es decir que los mismos ejes de Ogre vendrían siendo los mismos para SolidEdge®. A continuación se presentan cada una de las nuevas piezas tal y como se exportarán a Ogre. Estos archivos son guardados en extension *.stl* ya que Blender no acepta *.par*.

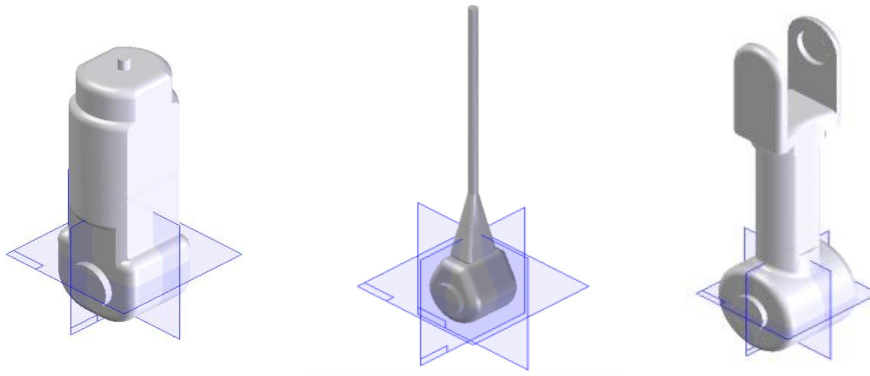


Figura 2.6. Pieza de la articulación 3 (Izq.), pieza de la articulación pasiva 5 (centro) y pieza de la articulación 2 (der.).

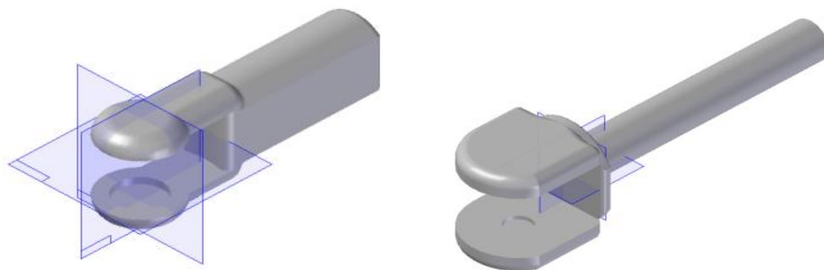


Figura 2.7. Pieza de la articulación pasiva 4 (izq.) y pieza de la articulación 1 (der.).

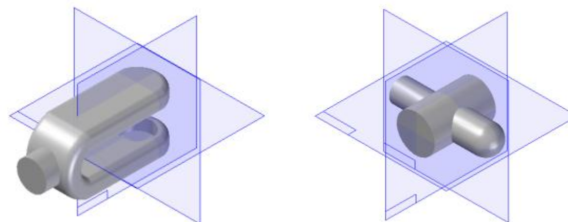


Figura 2.8. Pieza de la articulación 6 (Izq.) y pieza de la articulación 7 (der.).

Una vez hecha la conversión de formatos y agregado un material, es posible ahora montar el robot para la aplicación. La arquitectura de Ogre define que es necesario una entidad que haga referencia a la malla, y un nodo de escena para llevar ese objeto al ambiente virtual, y con este mismo nodo girar la pieza según sea necesario.

El robot HIBOU es construido en escena a partir de una pieza estática, en este caso la base, esta pieza quedará anclada y no se moverá durante la ejecución del programa. El resto de las piezas del robot son piezas móviles las cuales se moverán dependiendo de los datos de entrada que lleguen del mando externo. Junto con la base del robot todo el ambiente virtual está formado por piezas inmóviles que son puestas en escena usando un vector de tamaño tres que le indica al objeto su posición en el mundo virtual, su sentido y orientación, los cuales son modificados a gusto con las siguientes tres líneas de código.

```
nodoDeEscena → pitch( grados en radianes ); // GIRO EN X
nodoDeEscena → yaw  ( grados en radianes ); // GIRO EN Y
nodoDeEscena → roll ( grados en radianes ); // GIRO EN Z
```

Los objetos como la mesa y el cuerpo humano pueden ser girados con estos comandos ya que no son móviles y están anclandos, pero para el robot es diferente y se requiere una solución.

El comando para la animación de las articulaciones es `setOrientation` y permite que un objeto sea girado con respecto a un eje.

Ogre tiene integrada una sub-funcion que define un nodo como padre de otro, llamada `createChildSceneNode`,y permite que un nodo se mueva con respecto a otro. Esta caracteristica es muy semejante a lo que se puede apreciar en Virtual Reality, donde una pieza se declara hija de otra facilitando la simulación del movimiento del robot.

En la figura 2.9 se ha colocado en forma de ilustracion cómo se debe modificar la orientación de una pieza móvil. Los ejes en azul representan el nodo padre, y los ejes en rojo representan el nodo hijo, el cual tiene ligada una malla (pieza del robot). Si se desea modificar la orientación de la pieza se debe cambiar la orientación del nodo padre (azul) y no del nodo hijo, ya que al aplicar el comando `setOrienation`, la pieza es reubicada en la orientación original como se observa en la figura 2.10.

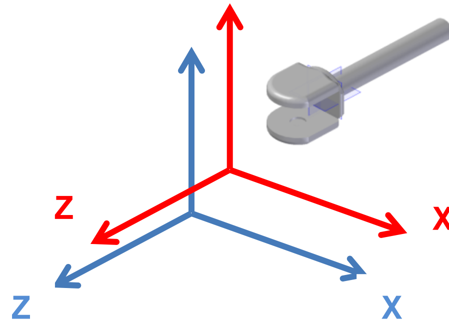


Figura 2.9. Nodos en una pieza móvil.

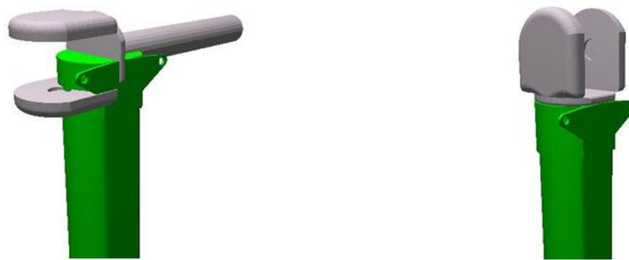


Figura 2.10. Orientación sin nodo padre (Izq.); orientación con nodo padre (der.).

Cada articulación es colocada y girada con un nodo padre ligado a la malla según la estructura cinemática como se aprecia en la figura 2.11.

Figura 2.11. Robot completamente renderizado.

Ahora se hace necesario colocar el robot como aparece en Matlab®. En la figura 2.12 se observa al HIBOU en el ambiente virtual de Ogre3D y la posición de cada una de las articulaciones en cero radianes tal y como aparece en Virtual Reality.

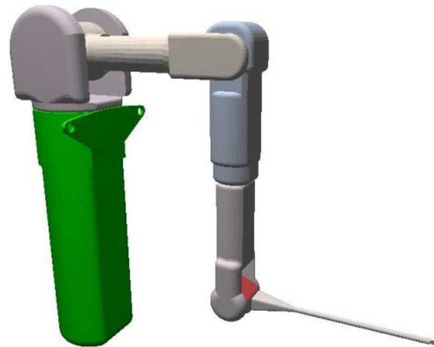


Figura 2.12. Robot HIBOU en Ogre3D con sus articulaciones en cero radianes.

Ver Anexo C para ver el código completo necesario para el renderizado del robot.

2.4.2. Cuarto quirúrgico

Según las especificaciones del proyecto es necesario elaborar un ambiente virtual que se asemeje a un quirófano. Este cuarto quirúrgico junto con la mesa, fueron dibujados en Solid Edge® con medidas reales. El cuarto quirúrgico tiene medidas de 4m de ancho por 4m de largo con una altura de 2m. La mesa tiene 1.80m de largo por 0.7m de ancho y una altura de 0.6m. Como se puede observar en la figura 2.13, este es un ambiente sencillo pero que cumple muy bien con su propósito.

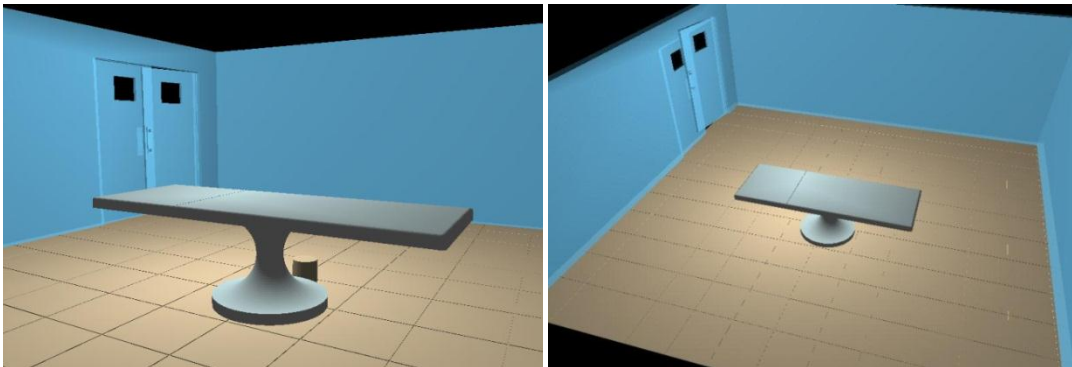


Figura 2.13. Ambiente quirúrgico de la aplicación.

Después de crear los objetos en Solid Edge®, éstos son exportados a Blender, donde se les dan los detalles de color y posteriormente son convertirdos en un archivo de tipo malla. Cada uno de los objetos que se muestra en la figura 2.13, son exportados por separado y ensamblados luego en Visual Studio de la siguiente forma:

```
1. Ogre::Entity *suelo = mSceneMgr-
   >createEntity("suel","suelo.mesh");
2. Ogre::SceneNode *SSuelo = mSceneMgr->getRootSceneNode()-
   >createChildSceneNode("sueloNode",Ogre::Vector3(0,-
   71.7,0));
3. SSuelo->attachObject(suelo);
4. SSuelo->scale(3,3,3);
```

El anterior código es un ejemplo de cómo se cargaría el piso del quirófano en Ogre3D sin nodo padre. En la primera línea se crea una entidad llamada “suelo” donde se carga el archivo *.mesh* del mismo nombre que se creó en Solid Edge® y se importó como malla desde Blender. En la segunda línea se obtiene el nodo raíz y se le asigna un nodo hijo al que se le llama “sueloNode” y se le ubica en la posición deseada. Luego en la línea tres se le víncula la entidad “suelo” al nodo creado en la línea dos y por último el objeto se escaliza de tal forma que se ajuste a los parámetros requeridos.

Los archivos *suelo.mesh, pared.mesh* y mesa.mesh son los nombres de los objetos que hacen parte del cuarto quirúrgico. El código completo necesario para su renderizado se puede apreciar en el Anexo C.

2.4.3. El paciente y el interior de su abdomen

Para lograr un ambiente virtual con mucho realismo se ha puesto en escena un paciente elaborado en Make Human y detallado en Blender, que representa a un hombre de 1.70m de estatura con el abdomen insuflado de gas CO_2 (figura 2.14), debido a que en la cirugía con laparoscópio en necesario separar la pared abdominal de los órganos para poder explorar el interior del paciente sin riesgo a ser lastimado.

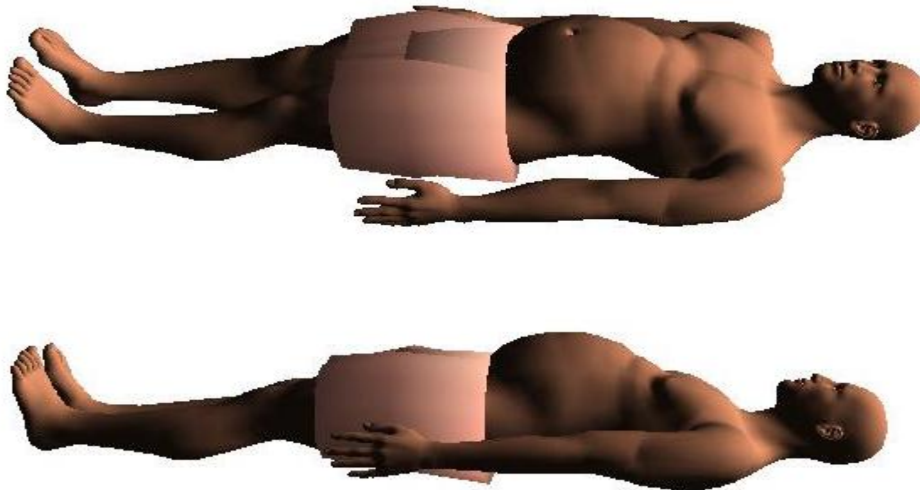


Figura 2.14. Cuerpo humano del ambiente virtual.

Una vez hecho el paciente en Make Human, se exporta como archivo *.obj* a Blender el cual recibe el molde de éste, pero sin textura ni color. Una vez agregados los detalles, es exportado en forma de malla para poder ser utilizado por Ogre, el color y el material son guardados en archivos separados para poder cargarlos desde Visual Studio.

Ahora se realiza la construcción del interior del abdomen. En la cirugía endocavitaria laparoscópica esencialmente se hace observación del interior de la cavidad pélvica-abdominal por lo que se hizo la construcción solamente del estómago, el hígado, la vesícula biliar, el intestino grueso, el intestino delgado y los riñones, como se observa en la figura 2.15.

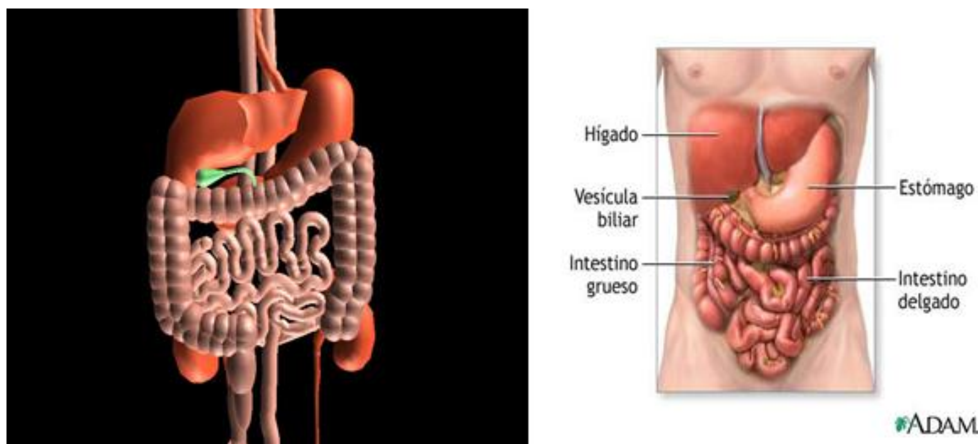


Figura 2.15. Órganos al interior del abdomen del paciente, virtual (Izq.) y real [28] (der.).

En la imagen de la izquierda se encuentran los órganos hechos para el ambiente virtual, y a la derecha se encuentra una imagen de los órganos de la cavidad abdominal de un humano.

Es importante tener una buena visualización de los órganos virtuales pues aunque no es un ambiente inmersivo, disponer de una interfaz realista hace mucho más interactivo manipular la herramienta software. La visualización de estos órganos y el papel que juegan en el ambiente virtual van relacionados con uno de los objetivos del proyecto complementario.

2.5. RESULTADO DE LA CONSTRUCCIÓN DE MALLAS

Se ha añadido una mini pantalla en la parte inferior derecha, en conjunto con el grupo de trabajo del proyecto complementario como medio de integración de los dos trabajos. La mini pantalla es la cámara que da visión de lo que está sucediendo en el interior del abdomen y a continuación se da detalle de toda la interfaz 3D desde diferentes ángulos.

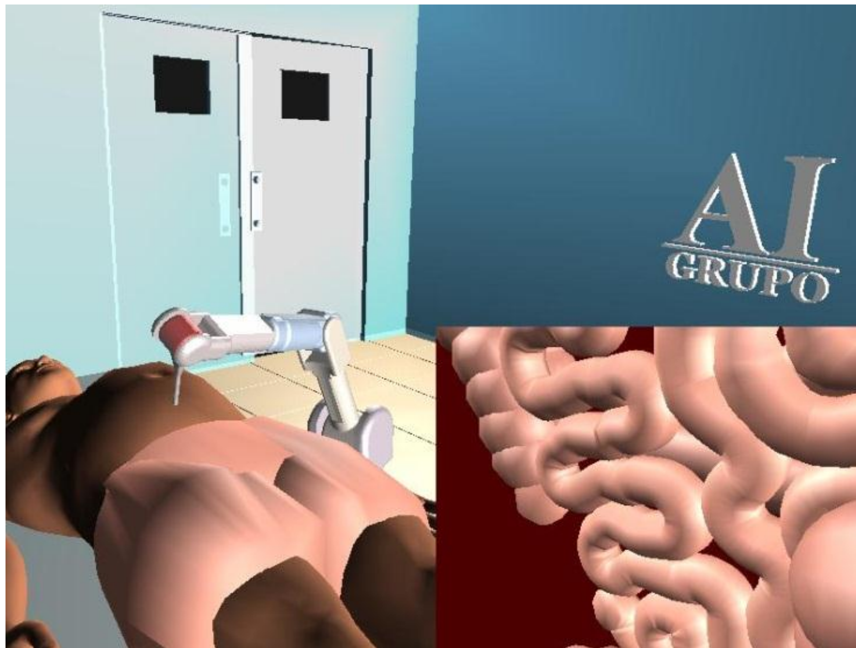


Figura 2.16. Vista lateral derecha del ambiente virtual.

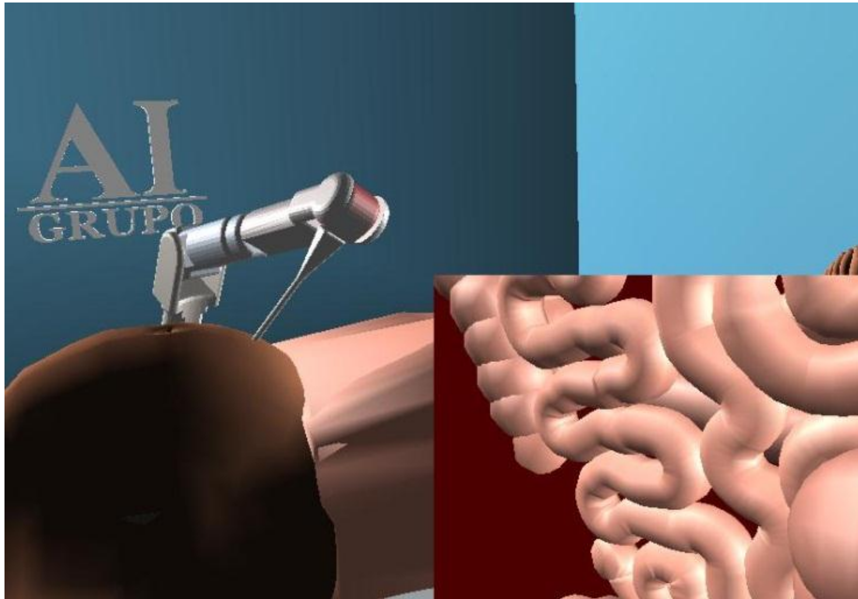


Figura 2.17. Vista lateral izquierda del
ambiente virtual.

Además de las siete articulaciones se ha adicionado una articulación manual para subir y bajar el brazo robótico según sea necesario, como se puede ver en la siguiente figura.

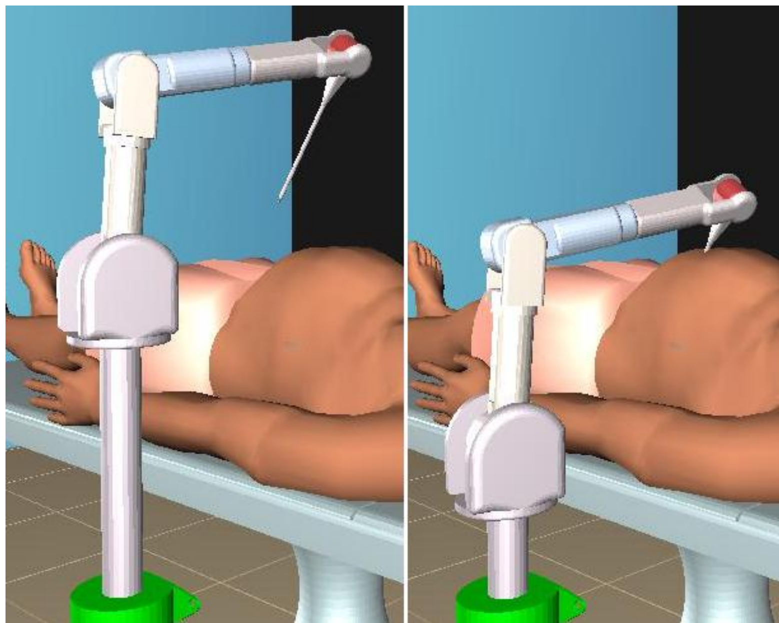


Figura 2.18. Vista de la articulación manual en
dos diferentes posiciones.

Es conveniente aclarar que al subir y bajar el robot se está subiendo y bajando el espacio cartesiano del robot, lo que facilita establecer la coordenada del punto del trocar del robot en la interfaz virtual.

Por otra parte, la distancia de visibilidad de cualquiera de las cámaras se puede configurar, es decir que se puede programar la distancia desde la cual se puede empezar a visualizar algún objeto de la siguiente forma:

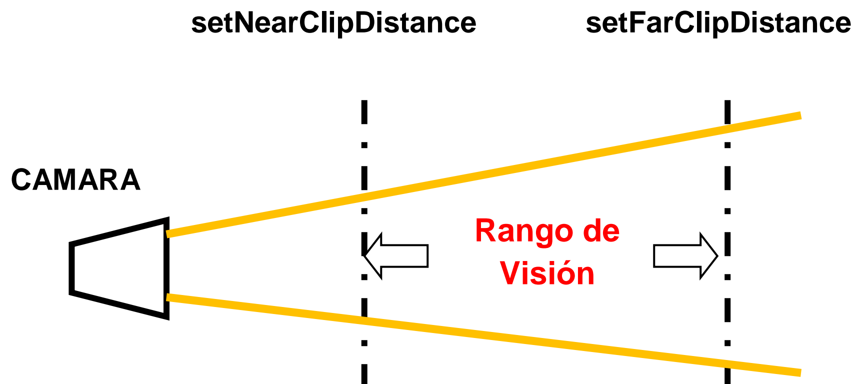


Figura 2.19. Rango de visibilidad de una cámara en Ogre3D.

La función `setNearClipDistance` especifica cuán cerca o lejos puede estar algo antes de dejar de verlo y `setFarClipDistance` describe la distancia en que el motor de renderizado se detendrá cuando un objeto se encuentre más lejos de esa distancia. Es por eso que los objetos tienden a aparecer cuando la cámara de la mini pantalla se acerca a los objetos, creando el efecto de oscuridad en el interior del abdomen.

Para limitar el campo de trabajo del robot en el interior del abdomen se ha construido un anillo semi-visible de diámetro 30 cm (figura 2.20) y se ha colocado junto a los órganos para delimitar ese espacio. En la figura 2.21 se puede observar los órganos, junto con el anillo mencionado, ya colocados dentro del abdomen insuflado del paciente.

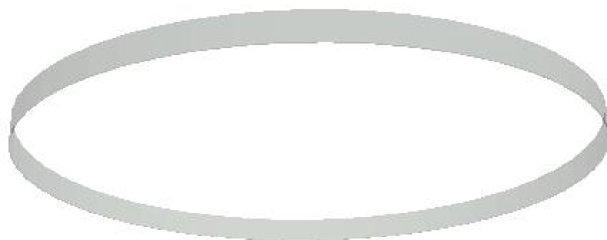


Figura 2.20. Anillo que limita el campo de trabajo

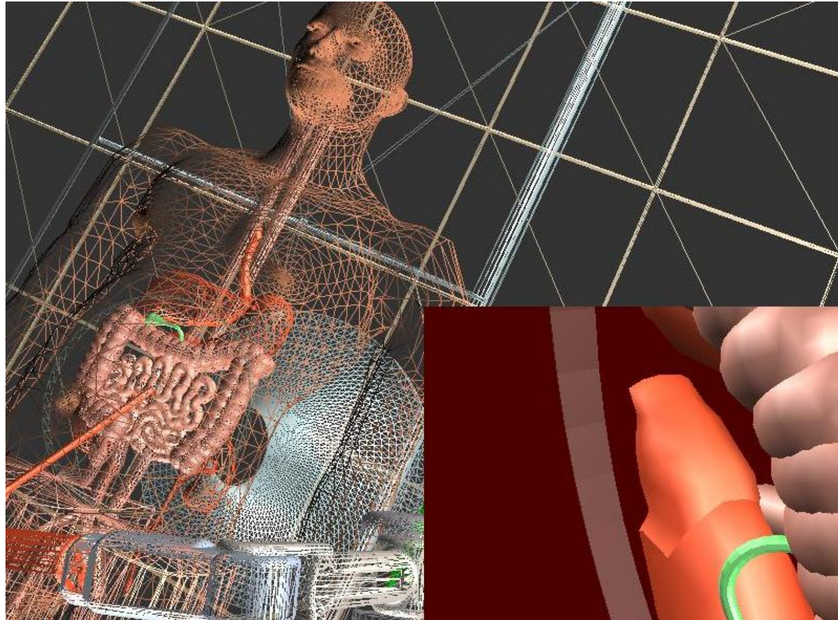


Figura 2.21. Vista del interior del abdomen.

3. Esquema de control a implementar y la solución del MGI

Matlab® presenta un lenguaje de programación propio el cual exhibe muchas características, incluyendo su facilidad de uso e implementación, pero debido a que requiere de muchos recursos de sistema, es imposible construir un robot en un ambiente virtual que trabaje de forma más rápida, por lo que se escogió el lenguaje de programación C++, el cual es capaz de entregar ejecutables que trabajan mucho más ligeros. Gracias a que el código de Matlab® presenta muchas similitudes a la programación en C, se hace fácil trasladarlo a C++ siempre y cuando existan ecuaciones que tengan operaciones básicas existentes en librerías.

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por *Bjarne Stroustrup* [33]. La intención de su creación fue extender el lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En la actualidad es un lenguaje versátil, potente y general ocupando uno de los primeros puestos como herramienta de desarrollo de aplicaciones, además de la gran variedad de librerías libres que se encuentran en la red, como lo son las librerías graficas de Ogre3D, haciendo de este lenguaje una buena opción de desarrollo.

La intención de este capítulo es presentar el esquema de control del sistema del robot en un algoritmo de C++ y la complejidad de implementar las funciones del software Matlab®, en un lenguaje tan básico. El esquema de control que se implementó se puede ver en la figura 3.1.

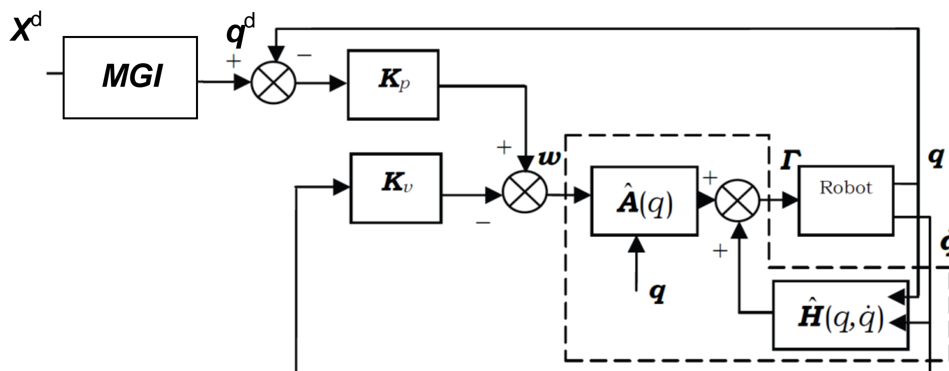


Figura 3.1. Esquema de control CTC [18].

Este es el control por desacoplamiento no lineal CTC (Control por par calculado) o simplemente control dinámico en el que se exige el cálculo del modelo dinámico, y el tipo de control implementado en el espacio articular es siguiendo una posición articular deseada ***(q^d)*** [18].

El vector de posición articular deseada ***(q^d),*** es un vector de siete posiciones que hacen referencia a los siete grados de libertad del robot y para su cálculo se hace necesario el uso del MGI (Modelo Geométrico Inverso), que recibe una posición del espacio cartesiano y entrega la posición articular. Como los modelos dinámicos son necesarios para realizar el control por par calculado también se hace referencia a ellos en la figura 3.2 donde se pueden observar los modelos que intervienen en el lazo de control.

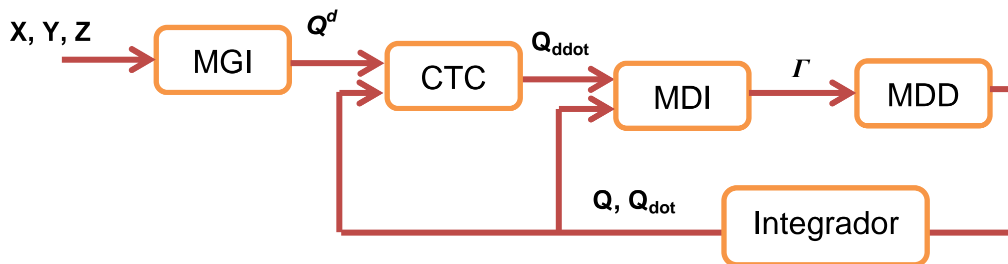


Figura 3.2. Modelos que intervienen en el lazo de control.

En la ecuación (3.1) se presenta la ley de control del CTC, donde $W(t)$ representa la nueva señal de control. La declaración de variables software (tabla 3.1), tiene la siguiente relación con el diagrama de bloques mostrado arriba.

$$QDD = W(t) = Kp(q^d - q) - Kv(\dot{q}) \qquad (3.1)$$

NOMBRE	DIAGRAMA DE BLOQUES	SOFTWARE
Posición deseada	Q^d	Qref
Aceleración	Q_{ddot}	QDD
Velocidad	Q_{dot}	QDot
Pares	Γ	Torq
Condición inicial		Qi

Tabla 3.1. Relación de variables software con el diagrama de bloques

El diagrama en bloques con realimentación de la figura 3.2 puede ser fácilmente expresado en Simulink®, pero aplicarlo a un algoritmo en C++ es un poco más complejo. A continuación se presenta un algoritmo base para la implementación del esquema de control.

ALGORTIMO DE CONTROL

```
1.  while Continuar,

2.      XRef  = LeerJoystick ( );
3.      QRef  = ModeloGeometricoInverso( XRef );

4.      Q     = LeerIntegradorVelocidad( );
5.      QDot  = LeerIntegradorAceleracion( );
6.      QDD   = Kp * ( QRef - Q ) - Kv * QDot

7.      Torq  = ModeloDinamicoInverso(Q, QDot, QDD);

8.      QDD   = ModeloDinamicoDirecto(Torq, Q, Qdot)
9.      IntegradorAceleracion( QDD );
10.     IntegradorVelocidad( QDot );
11. end
```

Ahora se efectúa una explicación línea por línea acerca de este algoritmo:

1. El lazo de control tiene que ir en un ciclo para que se pueda leer una posición cartesiana tras otra.
2. Al ingresar al ciclo se debe obtener una posición cartesiana del mando (joystick o casco).
3. Luego se calculan las posiciones articulares para esa posición cartesiana con ayuda del MGI.
4. Se lee el integrador de velocidad.
5. Se lee el integrador de aceleración.
6. Se ejecuta el control CTC
7. La nueva señal de control es leída por el MDI que almacena los pares de torsión en la variable *Torq*.
8. El MDD recibe las fuerzas (*Torq*) y calcula la aceleración actual del robot.
9. Se integra la aceleración para obtener la velocidad actual.
10. Se integra la velocidad para conocer la posición actual.
11. Cierre del ciclo, que podría decirse que es el cierre del software pues el programa en ejecución tiene que tener activo el lazo de control.

Este algoritmo es una pauta de lo que debe hacer el bucle de actualización que tiene la arquitectura de Ogre3D llamado `frameRenderingQueued`. Esta sub-función es la que se llama antes de que un frame halla terminado su renderizado y antes de que la ventana de renderizado pida que se cambien sus búferes.

En seguida se presenta la conversión de Matlab® a C++ de los modelos MGI, MGD.

3.1. MODELO GEOMETRICO INVERSO EN C++

La dificultad de codificar el MGI en C++ está en encontrar la solución articular para cinco de las siete articulaciones del robot, ya que es muy complicado implementar el método LM (*Levenberg Marquardt)* en un lenguaje de bajo nivel [1].

El método numérico de LM, como se dijo en un capítulo anterior, está basado en las derivadas parciales de un sistema de *n* ecuaciones, siempre y cuando sea posible derivar. El sistema a solucionar es un sistema de 5 ecuaciones con 5 incógnitas (posiciones articulares), junto con unas restricciones necesarias para un funcionamiento deseado del robot. La función objetivo es en donde se tiene definido ese sistema a solucionar y se puede encontrar en el Anexo C.

El valor que regresa la función objetivo se evalúa y se compara a un valor cercano a cero, aproximadamente de 1e-15, y cuando esa condición se apruebe entonces el método se detiene y entrega los valores de cada variable para los cuales se cumplió el mínimo resultado posible. Para llevar esto a C++ es necesario identificar la derivada parcial en ecuaciones de la función anterior, lo cual es algo complicado de solucionar.

La función objetivo tiene restricciones en sus valores para asegurarse de que el método no vaya a dar soluciones que coloquen al robot en una posición indebida. Esas restricciones están relacionadas con las articulaciones *t2* y *t3*, donde se dice que *t2* siempre tiene que estar en un valor que coloque todo el brazo justo hacia arriba, y *t3* junto con *t2* deben posicionar el robot justo por arriba del trocar haciendo referencia al punto *p5* (posición cartesiana de la quinta articulación).

En la figura 3.3., se observa la situación del punto *p5* (círculo rojo) y las posiciones de las articulaciones *t2* y *t3* que evitan que el robot tome posiciones indebidas.

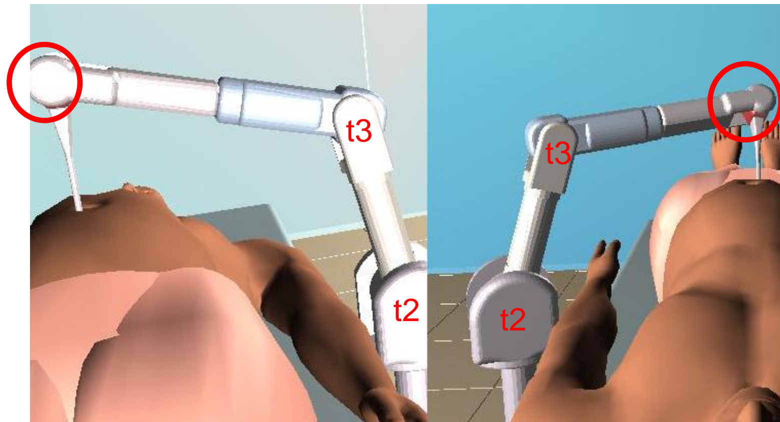


Figura 3.3. Posiciones de las articulaciones t2 y t3.

En el Anexo C se encuentra la función objetivo implementado en C++ con las condiciones de *t2* y *t3.*

Estas restricciones que se agregan al sistema de ecuaciones de quinto orden hacen que derivar se haga complicado y es por eso que encontrar un método numérico que no dependa de derivadas se hace indispensable.

3.1.1. METODO DE NM (NELDER - MEAD)

El método Simplex no lineal (*Nelder - Mead*), es un método heurístico de búsqueda de mínimos de cualquier función N-dimensional [34]. Para ello, a partir de un punto inicial estimativo, el método busca en el hiperespacio paramétrico aquellos valores de éstos que minimizan la función objetivo. Se basa en fundamentos geométricos; a partir de la estimación inicial, y conocidos el número de parámetros a optimizar, N, el algoritmo construye el poliedro más sencillo en ese hiperespacio paramétrico: el poliedro posee N+1 vértices donde se evalúa la función objetivo y se decide qué nuevos valores de los parámetros se ajustarán mejor al objetivo prefijado. Por ejemplo para ajustar dos parámetros, el algoritmo construye un triángulo en cuyos vértices se evalúa la función objetivo y según sus valores el poliedro se va moviendo y deformando para buscar el óptimo.

Este método por ser heurístico, se basa en consideraciones geométricas y no requiere el uso de derivadas de la función objetivo siendo de gran eficacia para ajustar gran número de parámetros [34]. El no requerir de derivadas hace que la función objetivo sea menos complicada de solucionar siendo el método a aplicar en el desarrollo del MGI.

Para comparar los resultados que se tiene con el método de LM (*Levenberg Marquardt*) y el método de NM (*Nelder-Mead*) se hace uso de una función "banana" para analizar su comportamiento. En el demo de optimización que existe en Matlab®, *"Minimization of the Banana Function",* se puede apreciar el progreso de cada método en la búsqueda de una solución (Figura 3.4 y Figura 3.5).

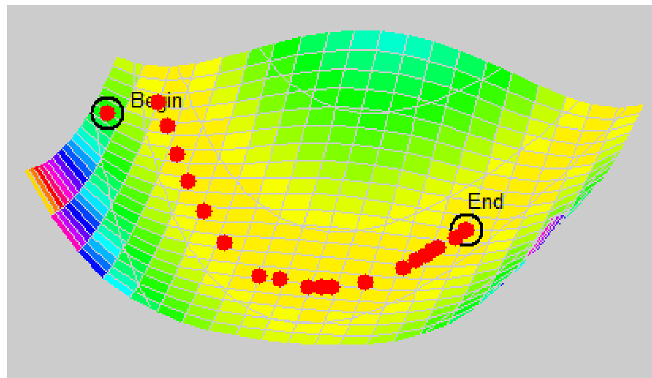


Figura 3.4. Optimización de la función Banana con *Levenberg Marquardt*

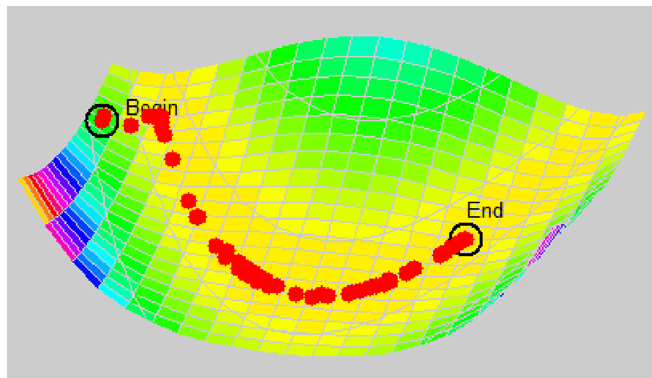


Figura 3.5. Optimización de la función Banana con *Nelder Mead*

El número de iteraciones está representado por los puntos rojos en las figuras, los cuales son menores en el LM que en el NM, pero esto no significa que uno sea mejor que otro. Como se puede observar, el método de NM presenta muy buenos resultados, además, como ya se mencionó, este método no requiere de las

derivadas parciales de las ecuaciones presentes en el MGI, por lo que es mucho más sencillo su implementación en C++.

Para realizar la implementación del algoritmo de minimización de NM se hace uso de la librería *ASA047.h* de la Universidad Estatal de Florida, Estados Unidos, diseñada por *Jhon Burkardt* y distribuida bajo licencia *GNU LGPL* que permite la copia, modificación y uso del código fuente [36].

En el caso del modelo geométrico inverso del robot, el mínimo de la función a solucionar se encuentra en la suma de los mínimos cuadrados de cada de una de las ecuaciones allí planteadas. A continuación hablaremos de la función objetivo implementado en C++ y presentamos el prototipo de la función objetivo.

```
double f_solve(double t[5], double x2, double y2, double z2)
```

El vector `t[]` que recibe la función `f_solve` son las condiciones iniciales estimadas para el método. Las variables de tipo double `x2, y2` y `z2` son las coordenadas cartesianas y hacen parte de la unica modificacion que se le hizo a la librería. La funcion `nelmin` (ver Anexo C), llama a la funcion `f_solve` en los bucles iterativos, por lo tanto ha estas dos funciones se les tiene que agregar como entrada la coordenada cartesiana para la cual se requiere una solución.

En el Anexo C se puede encontrar el archivo cabecera (asa047.h) y el código fuente (asa047.cpp) que se implementó en Visual Studio 2005.

3.1.2. MGI CON EL METODO DE NELDER-MEAD

Una vez hecha la modificación se hace la codificación y prueba del MGI con el método de NM, y para ello se hace necesaria una comparación de resultados con el método de LM ante una consigna circular. Cabe destacar que lo que se desea es saber si NM es capaz de comportarse de forma semejante al LM, ya que, como fue demostrado en el trabajo de grado anterior, los modelos diseñados si respetan el paso por el trocar y la utilización de un método o de otro solo difiere en su exactitud y el tiempo de cálculo.

Ambos métodos realizan un circulo con centro en $X = 0.4$ m, $Y = 0$ m y $Z = 0.13$ m, con un radio de 7 cm. El trocar que debe respetar el robot se encuentra en $X = 0.4$ m, $Y = 0$ m y $Z = 0.23$ m. Se mide el tiempo que demoran en realizar el círculo, así como el número de veces que es llamada la función objetivo.

	Nelder Mead	Levenberg Marquardt
Tiempo [segundos]	4.886	349.454
Llamadas a la función	1'124,723	312,095

Tabla 3.2. Comparación de los métodos numéricos.

Como se puede apreciar el método NM es mucho más rápido que el método de LM aunque el número de llamadas a la función es muy superior. El número de veces que se llama a la función objetivo indica que tan preciso es un método que otro aunque para hallar ese error que se obtuvo, ante una consigna circular, es necesario hacer la utilización del MGD para dicho análisis.

3.2. MODELO GEOMETRICO DIRECTO EN C++

Ya habiendo obtenido el modelo geométrico inverso, se procedió a trasladar el MGD a C++, el cual consta de una serie de ecuaciones que convierten los 7 estados de las articulaciones del robot en la posición cartesiana del efector final así como su orientación. Con esto se puede verificar el error obtenido por el algoritmo utilizado.

Las ecuaciones de este modelo tienen funciones muy básicas encontradas en la librería *math.h* y no presenta problema realizar su conversión.

En la figura 3.6 se observa un diagrama de bloques, en lazo abierto, sencillo de implementar en C++, en el que solo se necesita de un ciclo simple para introducir la consigna circular. Ya introducida la consigna se guardan los resultados en un archivo con extensión *.m* para ser graficados en Matlab® (figura 3.7 y figura 3.8).

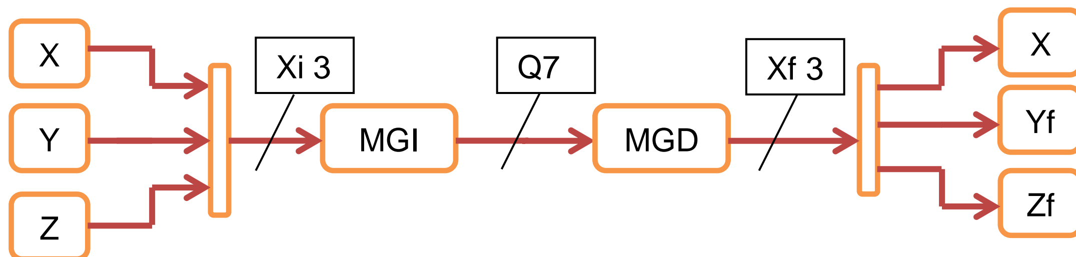


Figura 3.6. Esquema de prueba del MGI.

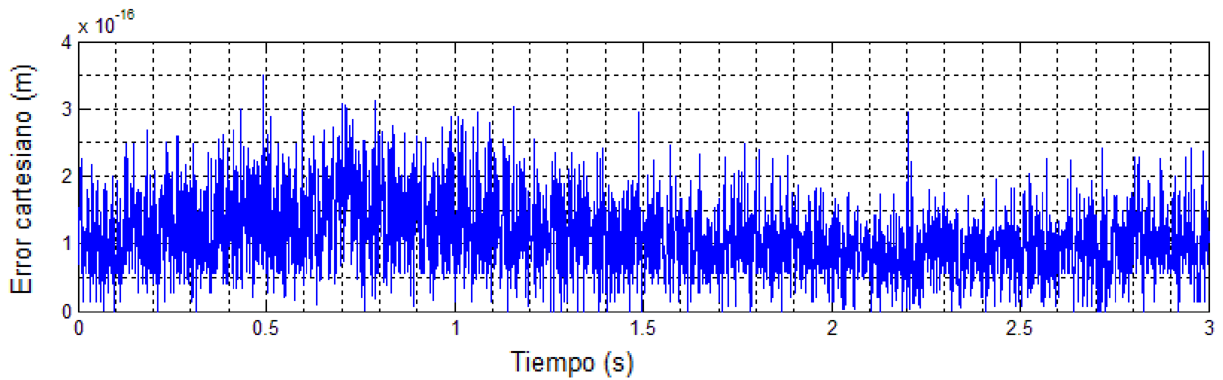


Figura 3.7. Error del LM ante una consigna circular.

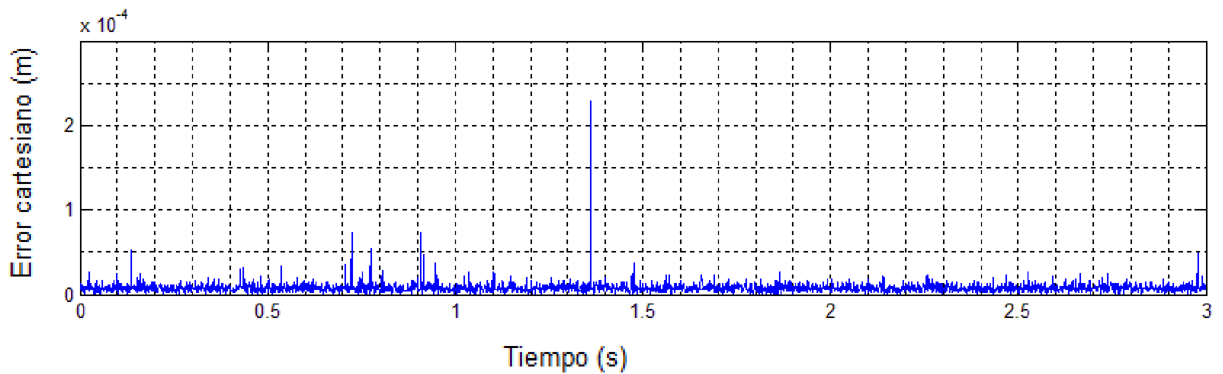


Figura 3.8. Error del NM ante una consigna circular.

El error obtenido con el método de NM es de 1e-4 mientras que con LM es de 1e-16, y se debe a que el LM converge mucho mejor, que el NM. Como se dijo en una sección anterior el LM necesita de las derivadas parciales de la función objetivo para dar una solución, por lo que el número de llamadas a la función tiende a reducirse y el método converge más rápido. Como se pudo ver en la tabla 3.2 el tiempo en el que el LM demora en realizar una consigna circular sugiere que el cálculo de la derivada parcial de la función objetivo es complicado y debe realizarse cada vez que se ejecute el método de LM, aunque el número de ejecuciones del LM sea mucho menor al NM. El LM da una solución más precisa pero hace que el MGI sea mucho más lento de solucionar. Se procede a ver el comportamiento de los modelos con el lazo de control.

4. Modelos dinámicos y el lazo de control

A continuación se explicará en qué consistió la solución de los modelos dinámicos y el lazo de control implementado en C++ bajo Visual Studio, respetando de cierta forma la arquitectura que tiene Ogre para no tener problemas en la ejecución de la aplicación.

4.1. MODELOS DINÁMICOS EN C++

A pesar de que solo con el MGI implementado se tiene el movimiento deseado del robot, es necesario implementar los modelos dinámicos ya que son una apreciación más detallada y real del robot. Hacer conversión de los modelos dinámicos desde Matlab® a C++ es sencillo ya que la mayoría del código son solo fórmulas que hay que traducir al pie de la letra, con unas pocas modificaciones. Implementar el control CTC con los modelos dinámicos y lograr una ejecución correcta, es algo mucho más complejo.

La arquitectura de Matlab® facilita el uso de variables sin ser declaradas, facilitando el ahorro de espacio así como el uso de una variable nueva cuando se antoje. En C++ es necesario declarar las variables que vamos a utilizar para que se les reserve un espacio de memoria a cada de una ellas, y es aquí donde viene el complicado y tedioso traslado de los modelos dinámicos ya que cada una de esas variables ahí nombradas debe ser declarada.

Luego de haber declarado cada una de esas variables utilizadas, también se añaden los parámetros inerciales a cada modelo. Con un poco de paciencia y con algún programa que facilite el editado de código como lo es Notepad++, de uso libre, se hace una exitosa conversión del MDD y el MDI, ahora lo que resta es cerrar el lazo de control. A continuación se presenta la forma en que las sub-funciones de los modelos MDD y MDI, recibirán y devolverán valores.

```
void MDI_model (double *t, double *QP, double *QDP, double *GAM);

void MDD_model (double *t, double *QP, double *QDP, double *GAM);
```

MDI_model recibe los vectores de posición, velocidad y aceleración mediante los punteros de tipo *double* `*t`, `*QP`, `*QDP`, y el resultado es redirigido al vector `Torq` con el puntero `*GAM`. Así mismo se ejecuta *MDD_model,* recibe `*t`, `*QP`, `*GAM` y el resultado es redirigido al vector `QDD` con el puntero `*QDP`.

`QDD` es el vector de las aceleraciones de cada una de las articulaciones del robot, y se necesita implementar un doble integrador que nos entregue los valores de velocidad y posición articular. Es necesario hacer referencia a las condiciones iniciales del robot, pues una incorrecta comprensión podría causar problemas en la ejecución del lazo de control. *Qi*, en el programa, son las posiciones articulares iniciales del robot que dependen de la posición cartesiana del trocar y la posición inicial cartesiana del efector final.

En al archivo `ModelosHibou.cpp` se encuentra implementada la función objetivo en C++ (ver Anexo C), y es ahí donde se encuentra la posición del trocar con las variables `TX`, `TY` y `TZ`. Para modificar la posicion del efector final, en el archivo `Hibou_Joystick.cpp`, que es el cuerpo de la arquitectura de Ogre, se ha declarado una funcion `inicializar()`, que tiene como objetivo declarar los valores iniciales de las articulaciones junto con ayuda del MGI para inicializar el vector Qi (ver Anexo C)

Cuando el robot se encuentra en el punto `Qi` las velocidades y aceleración se encuentran en cero. Es necesario asegurarse de que se empiece con velocidad y aceleración cero pues si no es así el robot se desestabiliza, y eso no puede llegar a ocurrir.

Para la solución de la ecuación diferencial de segundo orden que representa al MDD (4.1), se utilizó el método de *Runge Kutta* de cuarto orden que se aplica calculando un valor a la vez del MDD. La ventaja de este método con respecto a una integración clásica (método explícito de Euler) es la estabilidad numérica y la disminución del error de integración.

$$\ddot{q} = f(\, q \,, \dot{q}, \Gamma,\, f_e) \qquad (4.1)$$

4.1.1. METODO DE RUNGE KUTTA DE CUARTO ORDEN

Es un método iterativo de la familia de los métodos de *Runge Kutta* comúnmente llamado RK4 o simplemente método de *Runge Kutta*, y fue desarrollado para la solución de ecuaciones diferenciales ordinarias sujetas a condiciones iniciales [37].

El problema de valor inicial está definido como:

$$y' = f(x, y) \qquad y(x_0) = y_0 \qquad (4.2)$$

La fórmula con un RK4 para la solución de este problema está dada por:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) * h \qquad (4.3)$$

Dónde:

$$k_1 = f(x_i, y_i) \qquad (4.4)$$

$$k_2 = f\left(x_i + \tfrac{1}{2}h,\ y_i + \tfrac{1}{2}k_1 h\right) \qquad (4.5)$$

$$k_3 = f\left(x_i + \tfrac{1}{2}h,\ y_i + \tfrac{1}{2}k_2 h\right) \qquad (4.6)$$

$$k_4 = f(x_i + h,\ y_i + k_3 h) \qquad (4.7)$$

Como el MDD es una ecuación diferencial de segundo orden, las ecuaciones para la su solución, con un RK4, se presentan el siguiente cuadro:

$\frac{dq}{dt} = Qdot$	$\frac{d\dot{q}}{dt} = MDD(Q, Qdot, Torq)$
$k_1 = H * Qdot$	$l_1 = H * MDD(Q, Qdot, Torq)$
$k_2 = H * (Qdot + l_1/2)$	$l_2 = H * MDD(Q + k_1/2, Qdot + l_1/2, Torq)$
$k_3 = H * (Qdot + l_2/2)$	$l_3 = H * MDD(Q + k_2/2, Qdot + l_2/2, Torq)$
$k_4 = H * (Qdot + l_3)$	$l_4 = H * MDD(Q + k_3, Qdot + l_3, Torq)$
$Q = Q + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$	$Qdot = Qdot + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4)$

Tabla 4.1. Ecuaciones para la solución del MDD con RK4.

Con las siguientes condiciones iniciales de posición y velocidad:

$$Q = Qi \qquad Qdot = 0$$

Los pares (*Torq*), son anteriormente calculados por el *MDI_model* y la secuencia que se debe seguir para el cálculo de las constantes, junto con las nuevas posiciones y velocidades, es la siguiente:

1. Cálculo de los pares: $\boldsymbol{Torq} = \boldsymbol{MDI}(\boldsymbol{q}, \dot{\boldsymbol{q}})$
2. Cálculo de $\boldsymbol{k_1}$.
3. Cálculo de $\boldsymbol{l_1}$.
4. Cálculo de $\boldsymbol{k_2}$
5. Cálculo de $\boldsymbol{l_2}$
6. Cálculo de $\boldsymbol{k_3}$
7. Cálculo de $\boldsymbol{l_3}$
8. Cálculo de $\boldsymbol{k_4}$
9. Cálculo de $\boldsymbol{Q} = \boldsymbol{Q} + \frac{\boldsymbol{1}}{\boldsymbol{6}}\,(\boldsymbol{k_1} + \boldsymbol{2k_2} + \boldsymbol{2k_3} + \boldsymbol{k_4})$.
10. Cálculo de $\boldsymbol{Qdot} = \boldsymbol{Qdot} + \frac{\boldsymbol{1}}{\boldsymbol{6}}\,(\boldsymbol{l_1} + \boldsymbol{2l_2} + \boldsymbol{2l_3} + \boldsymbol{l_4})$.

4.1.2. IMPLEMENTACION DEL RK4 EN C++

Ya refiriéndonos al programa a implementar, el integrador debe ser capaz de manejar vectores, es decir que cada una de las constantes es un vector de 7 espacios relacionados con los siete grados de libertad del robot. Para ello se definen dos sub-funciones que harán la suma y el producto de vectores, `pp` y `sumarPorHK2`.

```
1. pp(double H, double *V, double *K)
2. {
3.          for(int i = 0;  i<7; i++)
4.          {
5.                 K[i] = V[i]*H;
6.          }
7. }
```

La funcion `pp()` recibe, en la primera linea, un vector en el puntero **V* y un valor *H,* que es el tiempo de muestreo. De la tercera a la sexta línea se realiza el producto escalar del vector recibido con el tiempo de muestreo, guardando el resultado en el puntero **K.*

```
1. sumarPorHK2( double *V,double *Vant, double *K, double cons)
2. {
3.        for(int i = 0; i<7; i++)
4.        {
```

```
5.                      V[i] = Vant[i] + cons * K[i];
6.          }
7.  }
```

La función `sumarPorHK2()` es la encargada de sumar los vectores de posición y velocidad actuales. En la primera línea, **V* es el vector resultado de esta operación, **Vant* es el vector anterior, **K* es el vector de una de las constantes calculadas y *cons* es un valor que depende de la operación que se esté realizando.

Ahora se realizará una breve explicación de las primeras líneas del código para realizar una doble integración con RK4. Se llama a la función del MDI para que calcule los pares de torsión.

```
MDI_model(Qi,QDot,QDD,Torq);
```

Una vez obtenido los pares de las articulaciones se hace el cálculo de la primera constante *K1* (4.8) que es la primera constante necesaria para calcular la nueva posicion articular.

$$k_1 = H * Qdot \qquad (4.8)$$

```
pp(H,QDot,k1);
```

Como lo que se está integrando es el MDD como tal, y no la salida de éste, para calcular las nuevas velocidades articulares se hace la llamada al *MDD_model* para el cálculo de las aceleraciones, para luego calcular *l1* (4.9) que es la primera constante necesaria para cálculo de la nueva velocidad articular.

$$l_1 = H * MDD(Q, Qdot, Torq) \qquad (4.9)$$

```
MDD_model(Q,QDot,QDD,Torq);
pp(H,QDD,l1);
```

Para la siguiente constante se hace indispensable realizar la suma entre vectores y es por eso que se ha declarado un vector auxiliar `QDots` encargado de almacenar la adición entre los vectores `QDot` y `l1`, donde `l1` adquiere un producto escalar por una constante. Para el cálculo de *K2* (4.10) se presentan las líneas de código mostradas más abajo.

$$k_2 = H * (Qdot + l_1/2) \qquad (4.10)$$

```
sumarPorHK2(QDots,QDot,l1,1/2);
pp(H,QDots,k2);
```

La ventaja de declarar variables auxiliares es que se pueden almacenar valores que posteriormente pueden ser utilizados, tal y como se puede apreciar en la ecuación (4.11) en donde se calcula el valor del vector `l2`. El vector `Qs` almacena la adición entre los vectores `Q` y `k1`, donde `k1` adquiere un producto escalar por una constante.

$$l_2 = H * MDD(Q + k_1/2, Qdot + l_1/2, Torq) \qquad (4.11)$$

```
sumarPorHK2(Qs,Q,k1,1/2);
MDD_model(Qs,QDots,QDD,Torq);
pp(H,QDD,l2);
```

El código completo de la implementación del RK4 en C++ con el modelo dinámico directo se puede apreciar en el Anexo C.

4.2. LOOPCONTROL

Se declaró una función llamada `LoopControl` la cual recibe 3 variables reales *X1, Y1 y Z1* que representan las coordenadas cartesianas de entrada provenientes del joystick. Dentro de esta función se definieron las respectivas posiciones, velocidades y aceleraciones articulares, vectores de tipo *double* de tamaño siete. Para guardar valores anteriores de velocidad y posición se declararon los vectores de tipo *double* `Qrefp[]` y `QDotp[]`, con el objetivo de calcular la derivada de las articulaciones pasivas.

Las variables son definidas de carácter estático dentro de la función debido a que esta es llamada cada *frame*, y si se inicializaran de manera normal cada vez que se entrara a esta función los valores se reiniciarían y la dinámica del robot no tendría sentido. La ventaja de las variables estáticas en la función del lazo de control es que los valores anteriores quedan almacenados, esto sin necesidad de declararlas de forma global, logrando así un código más legible y fácil de interpretar. Aquí también son definidos las constantes de velocidad, de posición y el tiempo de muestreo.

Para cada *frame* se reciben las posiciones cartesianas y se calcula la posición articular llamando a la función del MGI.

```
MGI_model(X1,Y1,Z1,Qref);
```

Luego se calcula la nueva señal de control a partir del CTC siguiendo una posición deseada. *Kp* y *Kv* son las constantes de posición y velocidad ya calculadas en el espacio de trabajo de Matlab®.

```
QDD[0] = kp1*(Qref[0] - Q[0] ) - kv1*QDot[0];
QDD[1] = kp2*(Qref[1] - Q[1] ) - kv2*QDot[1];
QDD[2] = kp3*(Qref[2] - Q[2] ) - kv3*QDot[2];
QDD[5] = kp6*(Qref[5] - Q[5] ) - kv6*QDot[5];
QDD[6] = kp7*(Qref[6] - Q[6] ) - kv7*QDot[6];
```

Los valores en radianes de cada una de las articulaciones es almacenada en las variables globales de tipo Ogre llamadas "*g1, g2,…, g7*", luego de hallar las aceleraciones se actualizan las posiciones de las articulaciones.

```
g1=Q[0];
g2=Q[1];
g3=Q[2];
g4=Qref[3];
g5=Qref[4];
g6=Q[5];
g7=Q[6];
```

`Qref[3]` y `Qref[4]`, son los valores de las articulaciones a las que no se les realiza control ya que son pasivas, pero para completar los vectores de posición y velocidad que necesitan los modelos dinámicos, se hace indispensable realizar su derivada.

```
QDot[3]= (Qref[3] - Qrefp[3])/H;
QDot[4]= (Qref[4] - Qrefp[4])/H;
QDD[3]= (QDot[3] - QDotp[3])/H;
QDD[4]= (QDot[4] - QDotp[4])/H;
```

Luego de haber completado los vectores de posición y velocidad ya se puede ejecutar la funcion del MDI para calcular los pares de torción. De aquí en adelante es tarea del integrador calcular las posiciones y velocidades actuales del robot, en donde se utiliza un ciclo para realizar los calculos con las constantes del RK4 como se ve en las siguientes lineas.

```
for(i = 0; i<7; i++)
{
	Q[i]    = Q[i]  + (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])/6;
	QDot[i] = QDot[i] + (l1[i] + 2*l2[i] + 2*l3[i] + l4[i])/6;
}
```

Q y *Qdot* son los vectores de tipo estático que realimentan a todo el lazo de control.

Una vez implementados los modelos junto con el integrador de ecuaciones diferenciales de segundo orden se hace preciso hacer prueba del lazo de control con la interfaz virtual.

4.3. COMPORTAMIENTO DEL LAZO DE CONTROL

Como resultado de las diversas observaciones realizadas cuando se interactuaba con el entorno, se realizarán diferentes movimientos para visualizar el comportamiento del robot HIBOU ante cambios de consigna.

4.3.1. CONSIGNA CIRCULAR

En la primera prueba se evaluó el lazo de control ante una entrada de movimiento circular, este círculo con centro en X = 0.4 m, Y = 0 m y Z = 0.13 m, con un radio 7 cm, y una posición de trocar X = 0.4 m, Y = 0 m y Z = 0.23 m. Se presenta en forma de un vector de 3000 puntos cartesianos, los cuales son introducidos al lazo de control, se obtienen los datos de las articulaciones para cada punto y se evalúan en el MGD para hallar la posición real del robot en el espacio cartesiano.

En la figura 4.1, se puede apreciar que el error obtenido está en el orden de 8e-4. La primera oscilación que se presenta se le atribuye a las inercias de los motores ya que no pueden realizar cambios bruscos, y la segunda oscilación es atribuida a la imprecisión del método de NM. Ambos cambios generan una oscilación que rápidamente es corregido por el controlador.

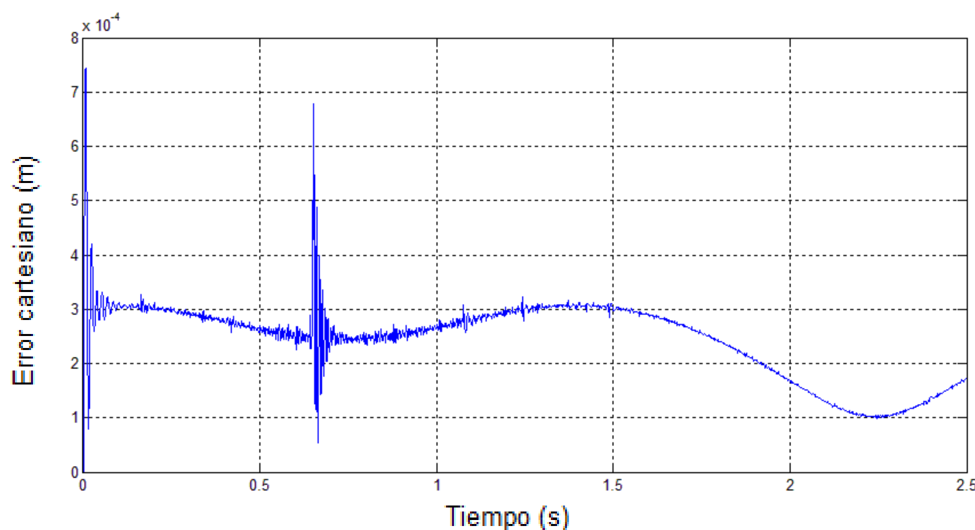


Figura 4.1. Error ante consigna circular en C++.

Comparando estos resultados con los obtenidos por el lazo de control desarrollado en Simulink®, como se muestra en la figura 4.2.

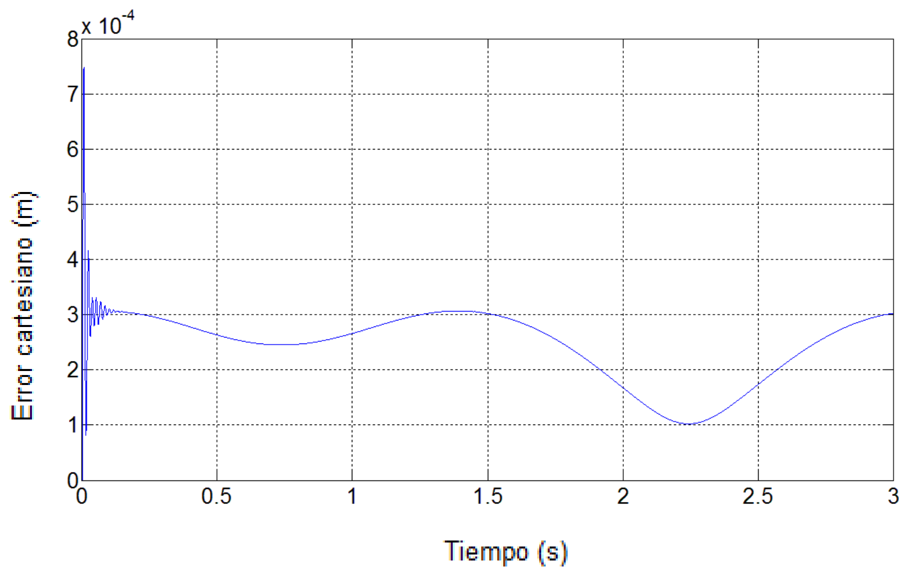


Figura 4.2. Error ante consigna circular en Simulink®.

El comportamiento tan similar de las dos ejecuciones se debe a que el error de seguimiento en el lazo de control, es de alrededor de 1e-3 (figura 4.3), y toda la precisión que otorga el método de LM se pierde por dicho error.

La precisión de cada método queda dependiendo de ese error de seguimiento por lo que el método de NM vendría siendo aceptable para la ejecución del software manipulador.

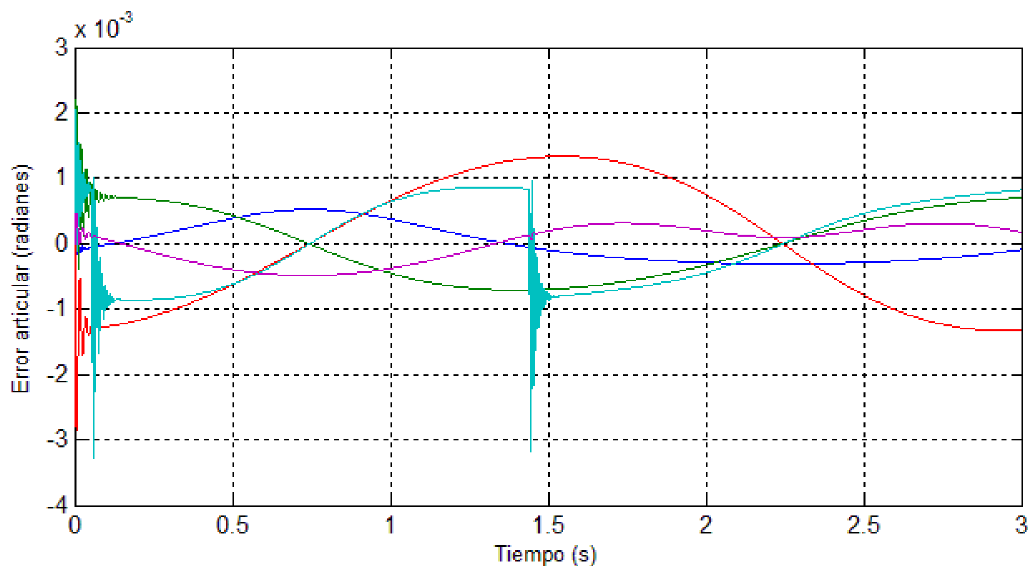


Figura 4.3. Error de seguimiento articular del lazo de control en Simulink®.

Ahora para probar el comportamiento del robot ante cambios de dirección de manera más interactiva, se ha puesto en funcionamiento un joystick, utilizado las librerías OIS (sistema de entrada orientada a objetos) de Ogre para que sean reconocidos los dispositivos de este tipo (ver Anexo B). Este es un dispositivo básico que le entrega al sistema tres consignas de entrada que hacen referencia a los tres valores en el plano cartesiano: X y Y son obtenidos con el movimiento de la palanca del dispositivo, los valores en Z son obtenidos con dos botones del mismo Joystick.

Los datos que entrega este dispositivo se ven reflejados en el movimiento del robot, por lo que se realizaron pruebas observando el comportamiento del HIBOU ante distintas consignas de entrada.

4.3.2. PRIMERA CONSIGNA GENERADA POR EL JOYSTICK

La segunda prueba consiste en darle al robot una consigna de movimiento recto sin cambio de dirección con el joystick y constante (sin detenerse). Se obtuvieron los datos de salida del MGD, que son las posiciones cartesianas finales y se compararon con los valores de entrada, el error es calculado hallando la diferencia al cuadrado de cada punto entre la entrada y la salida y sumándolos entre sí, como se muestra a continuación:

```
Error = abs (sqrt ((Xf - Xi).^2 + (Yf - Yi).^2)+(Zf - Zi).^2);
```

Como se puede ver en la figura 4.4, el error que se presenta es menor a 8e-4. Durante el primer tramo el robot se encuentra quieto, por lo que el error es aproximadamente cero hasta el momento en el que se realiza un cambio en la entrada con el joystick. El primer pico que se presenta es debido a que el robot intenta llegar a la posición deseada lo más rápido posible llevando a la aceleración a su máximo valor, una vez llega a este punto, debido a la inercia se sobrepasa y comienza un movimiento oscilatorio muy común en este tipo de sistemas que tiende a estabilizarse rápidamente con un error casi constante.

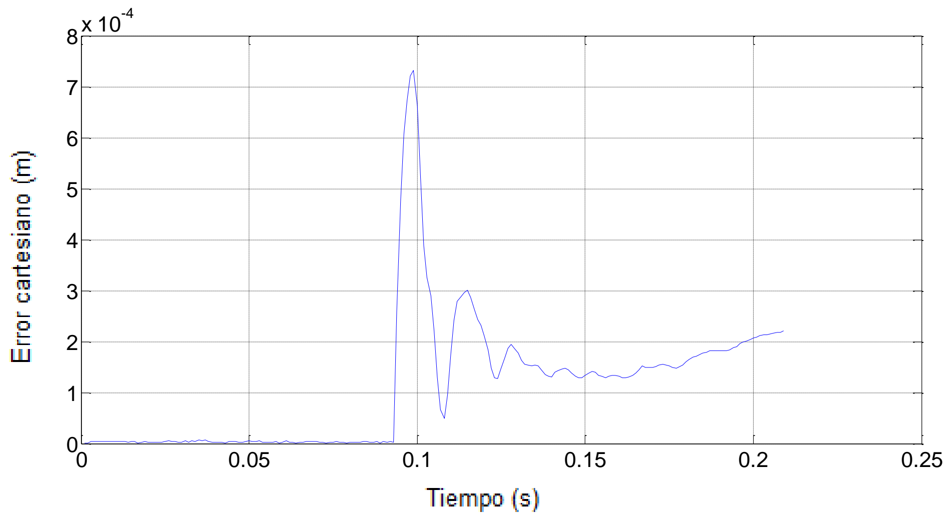


Figura 4.4. Error ante una entrada recta constante del Joystick.

4.3.3. SEGUNDA CONSIGNA GENERADA POR EL JOYSTICK

La tercera prueba consiste en realizar movimientos con el joystick por tramos, realizando un movimiento por cada eje y analizando cada tramo, en esta ocasión los movimientos ya no serán continuos como lo han sido en las pruebas anteriores para poder observar la oscilación que presenta el robot al detenerse.

Para esto primero se realizara un movimiento en *X*, se esperará a que se estabilice, procediendo a realizar un movimiento en *Y*, siguiendo los mismos pasos hasta realizar un movimiento en *Z*. Los resultados se pueden observar en la figura 4.5.

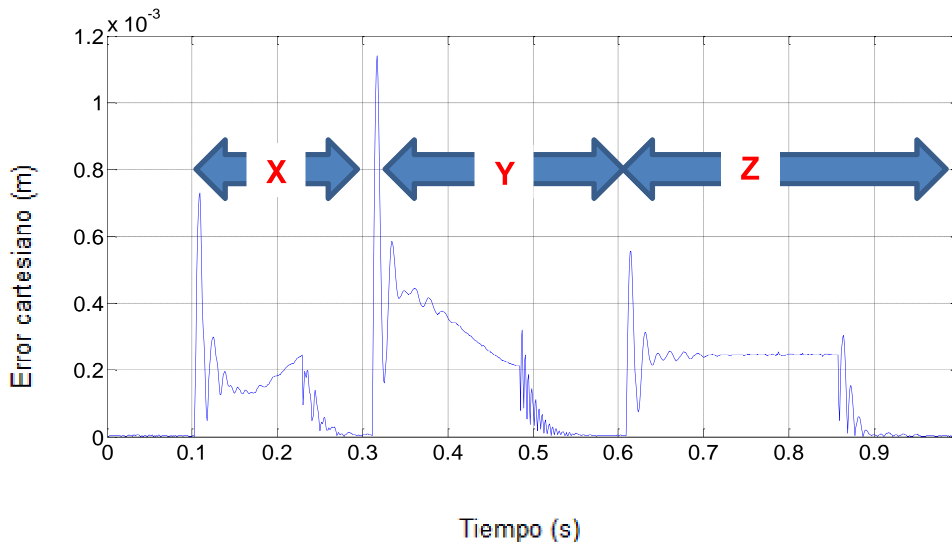


Figura 4.5. Error ante una consigna con cambios de dirección, deteniendo el robot.

Analizando la gráfica anterior se puede observar que ante cada cambio de movimiento se presenta una oscilación en el robot que luego tiende a estabilizarse, una vez termina de hacer el movimiento presenta otra oscilación pero mucho menor a la inicial la cual se estabiliza en cero, debido a que la fuerza para empezar el movimiento es mucho mayor que para detenerlo. El movimiento en *Z* presenta un error más estable debido a que su movimiento no depende de la palanca del joystick, sino que es controlado por medio de 2 botones alternos, uno para que aumente el valor de *Z* y otro para que disminuya, los cuales presentan mucho menos ruido ya que son 2 simples estados.

Los movimientos del Joystick son muy estables, en el sentido que no se esperan valores no deseados por ruido a la entrada del dispositivo, así que si necesitamos dejar de enviar datos solo colocamos la palanca en el centro colocando el robot en reposo, como se pudo observar en las figuras anteriores. A continuación cambiamos de dispositivo de prueba y habilitamos el dispositivo auxiliar.

4.3.4. INTEGRACION DEL DISPOSITIVO AUXILIAR

La arquitectura de Ogre permite definir eventos que van de la mano con otros dispositivos. Para el dispositivo de mando auxiliar se ha declarado la función `processUnbufferedInput`, habilitada con la tecla A y desactivando el mando con el joystick.

```
bool Hibou_Joystick::processUnbufferedInput(const
Ogre::FrameEvent& evt)
{


}
```

Esta función es colocada en el `frameRenderingQueued`, actualizando antes de cada frame. Generar consignas uniformes con este dispositivo tiene mayor grado de dificultad a comparación del joystick, asi que los movimientos a probar son el líneal con cambio de dirección, y en formas curvas.

4.3.4.1. PRIMERA CONSIGNA DEL DISPOSITIVO AUXILIAR

El sensor es manipulado de tal forma que el robot comienza a moverse en direcciones diferentes generando una consiga en forma de curvas. El movimiento del sensor solo genera dos señales las cuales van a modificar los ejes *X* y *Y* en la consigna del robot, es decir el eje *Z* no es manipulado.

La trayectoria generada es guardada en un archivo *.m* y es graficada con la ayuda de Matlab (figura 4.6).

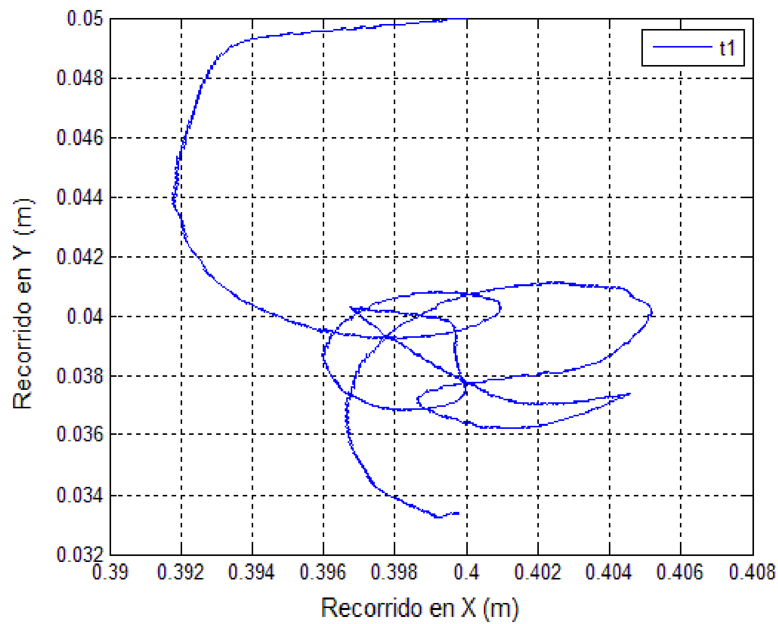


Figura 4.6. Consigna con movimientos curvos del casco.

Generar movimientos curvos tiene gran dificultad, porque el sensor tiene que siempre que estar en movimiento, si no fuera así se entregaría una señal en línea recta. Ahora presentamos el error cartesiano en la figura 4.7.

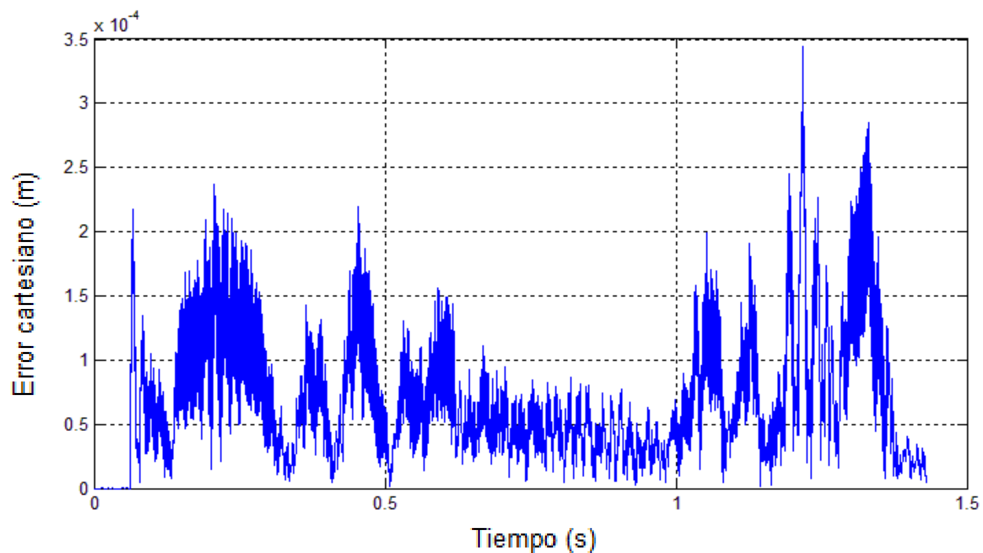


Figura 4.7. Error para la primera consigna del casco.

Las variaciones de la señal mostrada en la figura anterior son atribuidas a la resolución de los valores entregados por el dispositivo de mando externo, de 0 a 255, pues no es tan amplia como lo son las del joystick, cuyos valores van de 32767 a -32767 para cada eje. Aunque la resolución es baja el comportamiento del robot ante su movimiento es bueno presentando un error de 1e-4 metros.

4.3.4.2. SEGUNDA CONSIGNA DEL DISPOSITIVO DE MANDO AUXILIAR.

Para que la consigna entregada al robot sea en una sola orientación es necesario que el dispositivo se encuentra levemente inclinado hacia una dirección, y para cambiar de sentido, se debe hacer suavemente pues los movimientos bruscos producen una elevación en la aceleración y por ende también una elevación en los pares de torsión de los motores, generando un movimiento amortiguado con picos altos.

Los movimientos del robot con el sensor son de baja sensibilidad para evitar que el robot presente esas oscilaciones mencionadas. A continuación se presenta la consigna para movimientos lineales y cambios suaves de dirección (figura 4.8).

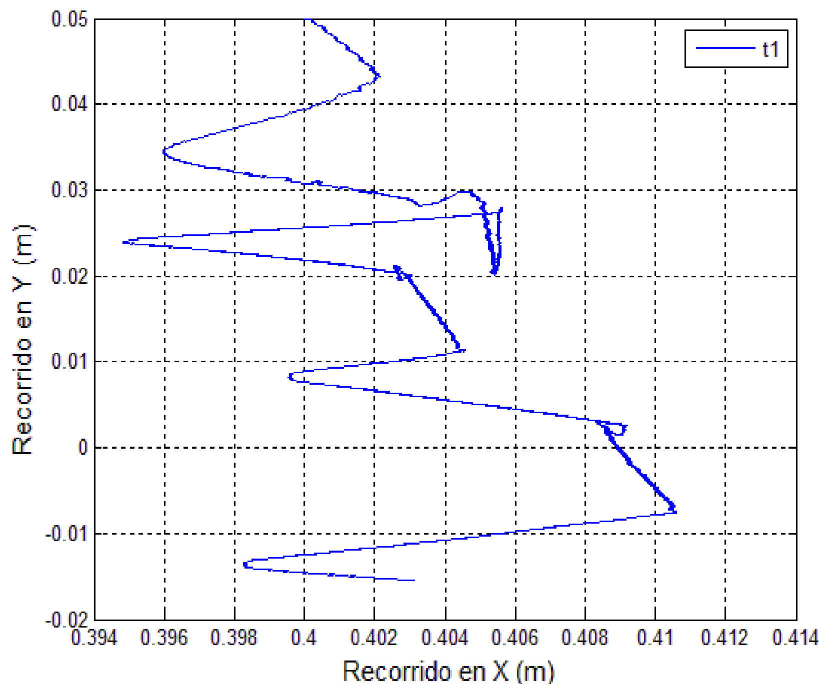


Figura 4.8. Consigna lineal con el casco.

El error para esta trayectoria ante movimientos del sensor se muestra a en la figura 4.9.

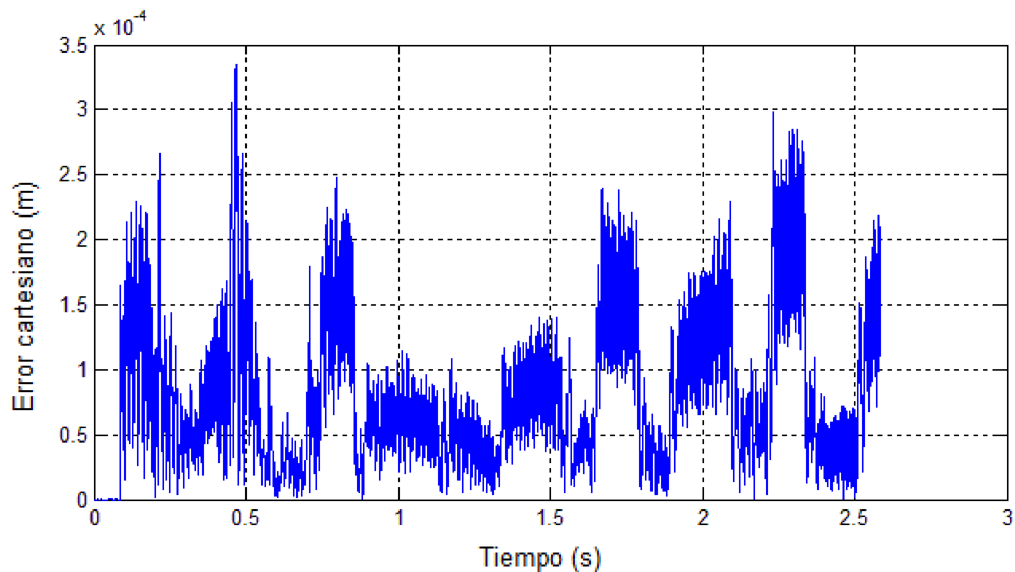


Figura 4.9. Error para la segunda consigna del casco.

4.4. COMPORTAMIENTOS A TENER EN CUENTA EN LA EJECUCIÓN DE LA APLICACIÓN

R es la tecla con la que se habilita el movimiento automático del robot con la trayectoria circular, pero esta función solo puede ser habilitada cuando la aplicación inicia, es decir que si se ha iniciado el movimiento con la tecla E, que habilita los dispositivos de entrada con joystick o con casco, y luego se intenta realizar una trayectoria circular, el robot no va a ser capaz de realizar dicha función porque se desestabilizaría. Cuando el robot intenta ir de cierto punto a un punto muy lejano, el controlador lanza una señal muy grande que hace que se desestabilice el robot.

Cuando en el espacio multidimensional de la función objetivo se encuentran soluciones al sistema que generan comportamientos no adecuados en el robot, lo cual genera pequeños saltos del órgano terminal, se ha colocado una condición que restringe esos valores evitando que sean llevados al lazo de control. La aplicación genera una pequeña señal sonora que le indica al usuario cuando se encuentran esas soluciones no adecuadas para el esquema de control.

La aplicación reconoce al joystick cuando inicia, es decir que cuando no esté conectado al ejecutar el software se genera error. Pero una vez abierta la aplicación; desconectar el joystick no genera problema.

5. Aspectos importantes a tener en cuenta

5.1. EN EL LAZO DE CONTROL

En la ejecución de la aplicación existen ciertas posiciones en las que el robot presenta saltos, que son corregidos por el controlador, pero que a su vez son un problema. Estos saltos son como el presentado en la figura 4.1 del cuarto capítulo en el que se le aplica al algoritmo de control una consigna circular.

Se propone entonces encontrar la forma adecuada de darle solución al lazo de control evitando los posibles lazos algebraicos como el visto en los modelos dinámicos y el método de integración (figura 6.1).

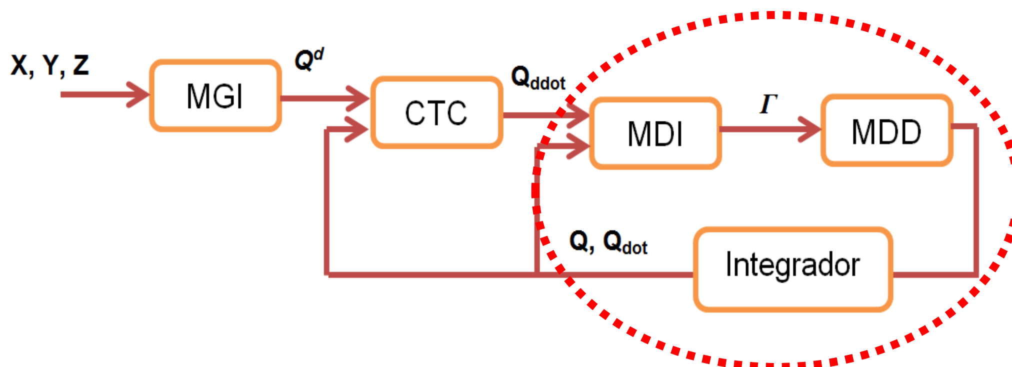


Figura 5.1. Lazo de control del algoritmo implementado

Un ejemplo de donde se puede presentar un lazo algebraico es el presentado en círculo rojo en la figura 6.1. El modelo dinámico directo, como se dijo en un capítulo anterior, es una ecuación diferencial de segundo que depende de lo que halla en su salida, y es ahí donde existe un lazo algebraico. Una posible solución se encuentra en expresar el modelo de orden dos, el cual se compone de n ecuaciones diferenciales, como uno equivalente de 2n ecuaciones diferenciales de primer orden, donde la única entrada es el par.

5.2. EN EL TIEMPO DE EJECUCIÓN DE LA APLICACIÓN

Lo que se diseño fue un software capaz de responder más rápido ante una consigna generada por un dispositivo de mando como lo es el joystick o el dispositivo auxiliar (proyecto complementario), pero no se comporta en tiempo real ya que no se está asegurando que el tiempo de muestreo sea el adecuado ante cada *frame* renderizado.

No obstante hay que tener en cuenta de que Windows no es sistema operativo capaz de trabajar en tiempo real debido a las múltiples tareas que siempre se encuentra ejecutando, y eso también afectaría el tiempo de muestreo.

El tiempo de muestreo para la aplicación es de un milisegundo, es decir que es de paso fijo. Cada vez que sea renderizado un *frame* no es posible asegurarse, en la aplicación, que el tiempo de muestreo sea de un milisegundo pues todo depende de cuánto demore en ejecutarse el lazo de control, el modelo geométrico inverso (con el método de Nelder Mead), además también depende en el estado que se encuentre el sistema operativo.

6. Publicaciones en el marco de esta tesis

6.1. Publicación en evento

F. Ordoñez, C. Tosse, E. Lasso, C. Muñoz, A. Vivas. “Ambiente Virtual y control a distancia de un robot para cirugía laparoscópica”. XIV CONVENCION DE INGENIERIA ELECTRICA. Santa Clara. Cuba. 2011.

6.2. Publicación en revista

C. Muñoz, E. Lasso, C. Tosse, F. Ordoñez y A. Vivas. “Sistema virtual para el posicionamiento del robot porta endoscopio HIBOU a través de un casco”, RUTIC, Vol. 1, No. 1, pp. 59-65, 2012.

7. Conclusiones

En este proyecto complementario se desarrolló un software, con una interfaz virtual, del robot porta endoscopio HIBOU para cirugía laparoscópica, y se implementó en C++ su dinámica haciendo uso de los modelos ya diseñados. Se implementó una interfaz dentro del software capaz de integrarse con el dispositivo de mando externo diseñado por el otro proyecto complementario

Se utilizó el método de Nelder - Mead como remplazo del método Levenberg Marquardt, y se comprobó que dicho método tiene la capacidad de comportarse de forma semejante al LM haciendo uso del lazo de control. Para la solución del método de integración del MDD se utilizó el método de Runge Kutta de cuarto orden utilizando las fórmulas para una ecuación diferencial de segundo orden.

Se concluye que utilizar un método numérico que necesite derivadas parciales para solucionar un sistema de ecuaciones tan complicado como el presentado en este trabajo, no hubiera permitido la ejecución de manera más rápida que se requería en la aplicación y que el método de Nelder Mead junto con método de Runge Kutta, ofrecen un comportamiento muy bueno frente ya anteriormente diseñado

La gran mayoría de código utilizado en este proyecto corre bajo licencia GNU LGPL, como es la librería *asa047* la cual fue de gran ayuda para la implementación del método del Nelder Mead con el MGI, y el motor gráfico de Ogre3D.

El lazo de control implementado se ejecuta de forma ordenada con una fácil interpretación de su código logrando una gran interacción con los dispositivos de mando. La ejecución en un tiempo más rápido de la aplicación ofrece una mayor visibilidad de cómo se comportan los modelos dinámicos y su función en la simulación entregando datos que pueden ser de utilidad para determinar el manejo adecuado del robot en una cirugía real. Con esto se concluye que los movimientos de este robot deben ser suaves, sin movimientos bruscos que lo hagan oscilar de manera inadecuada, lo cual está en estrecha relación con los requerimientos quirúrgicos.

Futuros trabajos construirán un primer prototipo del robot HIBOU, utilizando esta interfaz virtual para su manipulación, abordando los problemas que se puedan

presentar en la construcción, en la adecuación de señales y la selección de materiales para mejorar la dinámica del mismo.

Referencias Bibliográficas

[1] C. Méndez y V. Torres, “Diseño y simulación en 3D de un robot porta endoscopio para cirugía laparoscópica”, Trabajo de grado de pregrado, Universidad del Cauca, Colombia, 2010.

[2] J. Ma y P. Berkelman, “Control Software design of a compact laparoscopic surgical robot system”, IEEE/RSJ International Conference on Intelligent robots and systems, Beijing, October 9, 2006.

[3] R. Gonzales, “Aplicaciones de robótica en medicina”, Revista Labor Académico, Vol. 1, No. 2, pp. 4-11. Diciembre 2005.

[4] Observatorio de Prospectiva Tecnológica Industrial y Federación Española de Empresas de Tecnología Sanitaria, “El Futuro de la Cirugía Mínimamente Invasiva”, Federación Española de Empresas de Tecnología sanitaria, 2004. [Online]. Disponible: http://www.fenin.es/pdf/prospectiva_cmi.pdf. Consultado: Octubre de 2011.

[5] S. Salinas, “Modelado y Simulación en 3D y Control de un Robot para Cirugía Laparoscópica”, Tesis de maestría, Universidad del Cauca, Colombia, 2009.

[6] S. Schwartz y J. Hunter, “Principios de Cirugía”, 7ªEd. México: McGrawHill Interamericana, 1999.

[7] M. Meinero, G. Melotti y Ph. Mouret, “Cirugía Laparoscópica”. Buenos Aires: Editorial Médica Panamericana, 1996.

[8] M. Krisha, “Procedimientos Laparoscopicos Asistidos por Robot”, *Health education library for people,* 2011. [Online]. Disponible en: http://healthlibrary.epnet.com/print.aspx?token=5344349d-8fbc-446e-8ae5-03a924025f8c&chunkiid=177901. Consultado: Octubre de 2011.

[9] Universidad Politécnica de Catalunya, Facultad de informática de Barcelona, Robótica Médica, Disponible en: http://www.fib.upc.edu/retro-informatica/avui/salut.html, Consultado: Octubre de 2011.

[10] INTUITIVE SURGICAL. www.intuitivesurgical.com. Consultado: Enero 2012.

[11] H. Villavicencio, "Cirugía laparoscópica avanzada robótica Da Vinci: Origen, aplicación clínica actual en Urología y su comparación con la cirugía abierta y laparoscópica", Vol. 30, No. 1, pp. 1-12. Octubre 2009.

[12] A. Córdoba y G. Ballantyne, "Sistemas Quirúrgicos Robóticos y Tele robóticos para cirugía abdominal". Revista de Gastroenterología del Perú, vol. 23, 2003.

[13] A. Barrientos, "Robótica en medicina", Universidad Politécnica de Madrid, 2008

[14] MEDSYS. www.medsys.be. Consultado: Enero 2012.

[15] S. Kommu, P. Rimington, C. Anderson, y A. Rane "Initial experience with the EndoAssist camera-holding robot in laparoscopic urological surgery", *Journal of Robotic Surgery*, vol.1, pp.133-137, 2007.

[16] G. Morel Y P. Poignet, "Kinematics, position and force control issue in minimally invasive surgery", *Conferencia Internacional de Imagen Médica Computarizada e Intervención asistida por Computador*, Francia 2004.

[17] W. Khalil, and E. Dombre, "Modeling, Identification and Control of Robots", London: Kogan Page Science, 2002.

[18] A. Vivas, "Diseño y Control de Robots Industriales: Teoría y Práctica", Buenos Aires: Elaleph, 2010.

[19] R. Paul, "Robot Manipulators: Mathematics, Programming and Control", Cambrigde: MIT Press, 1982.

[20] K. Levenberg, "A Method for the Solution of Certain Non-linear Problems in Least Squares", Quarterly of Applied Mathematics, Vol. 2, pp. 164–168, Jul. 1944.

[21] D. Marquardt, “An Algorithm for the Least-Squares Estimation of Nonlinear Parameters”, SIAM Journal of Applied Mathematics, Vol. 11, pp. 431–441, Jun.1963.

[22] W. Khalil y D. Creusot, “Symoro+: A System for the Symbolic Modelling of Robots”, Robotica, Vol. 15, pp. 153-161, 1997.

[23] SIMBIONIX: LapMentor http://www.simbionix.com/LAP_Mentor.html. Consultado: Abril de 2011.

[24] THE UNIVERSITY OF MISSISSIPPI MEDICLA CENTER. “Virtual laparocopy (LapSimÂ®)”. http://surgery.umc.edu/facultystaff/learning/SSC/virtual.html. Consultado: Noviembre de 2011.

[25] S. Pusil y K. Zuñiga, “posicionamiento de LapBot en un ambiente tridimensional virtual usando interfaz háptica”, Tesis de grado, Universidad del Cauca, Colombia, 2011.

[26] A. Murgate, “Desarrollo de un ambiente 3D para simular problemas de robótica”, Reporte Técnico PII-02-04-09, Universidad Politécnica de Puebla, México, 2009.

[27] Microsoft Visual Studio. msdn.microsoft.com/es-es/vstudio. Consultado: diciembre 2011.

[28] Ogre3D. www.ogre3d.org. Consultado: Enero 2012.

[29] Blender. www.blender.org. Consultado: Enero 2012.

[30] Solid Edge. www.medis3d.com/solid-edge/web. Consultado: diciembre 2011.

[31] Make Human. www.makehuman.org: Consultado: Enero 2012.

[32] A.D.A.M. www.adam.com: Consultado: Enero 2012.

[33] B. Stroustrup. “El lenguaje de programación en C++”. Madrid: Addison Wesley, 1998.

[34] J. Nelder and R. Mead, “A simplex method for function minimation”, Computer Journal, Vol. 7, pp. 308-313, 1965.

[35] J. Martínez, “Optimización y ajuste de parámetros mediante el método simplex (Nelder-Mead)”, 1ª Reunión de usuarios de Ecosimpro, Madrid, España, 2011.

[36] The Florida State University. www.sc.fsu.edu/~ jburkardt. Consultado: diciembre 2011.

[37] U. Ascher y L. Ruth, “Computer methods for ordinary differential equations”, 1ra Ed, Philadelphia: Siam, 1988.

Anexo A. Ecuaciones de los Modelos del robot HIBOU

A.1. ECUACIONES PARA EL CÁLCULO DEL MGD

Las ecuaciones para el vector de posición *P*, son las siguientes:

$$\mathrm{Px} = 3/10((\mathrm{C1C2C3} - \mathrm{C1S2S3})\mathrm{C4} + \mathrm{S1S4})\mathrm{C5} - 3/10(\mathrm{C1S23})\,\mathrm{S5} + 2/5(\mathrm{C1S23}) + 3/10(\mathrm{C1C2}) \qquad (\mathrm{A.2})$$

$$Py = 3\,/\,10((S1C2C3 - S1S2S3)C4 - C1S4)C5 - 3\,/\,10(S1S23)S5 + 2/5(S1S23) + 3/10(S1C2) \qquad (\mathrm{A.3})$$

$$Pz = 3/10(S23)C4C5 - 3/10(S2S3 - C2C3)S5 + 2\,/\,5\,(S2S3) - 2\,/\,5(C2C3) + 3\,/\,10(S2) \qquad (\mathrm{A.4})$$

Dónde:

$$Ci = \cos(\theta j)$$
$$Si = \mathrm{sen}(\theta j)$$
$$Cij = \cos(\theta i + \theta j)$$
$$Sij = sen(\theta i + \theta j)$$

Las ecuaciones que representan la matriz de orientación $^{0}\boldsymbol{A}_{7}$, se puede ver a continuación.

$$s_x = ((((C1C2C3 - C1S2S3)C4 + S1S4)C5 - (C1S23)S5)C6$$
$$+ \left(-\left((C1C2C3 - C1S2S3)C4 \ + S1S4\right)S5 - (C1S23)C5\right)S6)C7$$
$$- (-(C1C2C3 - C1S2S3)S4$$
$$+ \ S1C4)S7 \quad (A.6)$$

$$s_y = \ ((((S1C2C3 - S1S2S3)C4 + C1S4)C5 - (S1S23)S5)C6$$
$$+ \left(-\left((S1C2C3 - S1S2S3)C4 \ + C1S4\right)S5 - (S1S23)C5\right)S6)C7$$
$$- (-(S1C2C3 - S1S2S3)S4$$
$$+ \ C1C4)S7 \quad (A.7)$$

$$s_z$$
$$= \ ((S23C4C5 - (S2S3 - C2C3)S5)C6 + (-S23C45 - \ (S2S3 - C2C3)C5)S6)C7$$
$$+ S23S4S7 \quad (A.8)$$

$$n_x = \ -((((C1C2C3 - C1S2S3)C4 + S1S4)C5 - (C1S23)S5)C6$$
$$+ \left(-\left((C1C2C3 - C1S2S3)C4 + S1S4)S5 - (C1S23)C5\right)S6\right)S7$$
$$- (-(C1C2C3$$
$$- C1S2S3)S4S1C4)C7 \quad (A.9)$$

$$n_Y = \ -(\left(\left((S1C2C3 - S1S2S3)C4 - C1S4\right)C5 - (S1S23)S5\right) C6$$
$$+ \left(-\left((S1C2C3 - S1S2S3)C4 - C1S4S5 - (S1S23)C5\right)S6\right)S7$$
$$- (-(S1C2C3 - S1S2S3)S4$$
$$- C1C4)C7 \quad (A.10)$$

$$n_Z$$
$$= -\left(\left((S23)C4C5 - (S2S3 - C2C3)S5\right)C6\right.$$
$$\left.+ (-(S23)C4S5 - (S2S3 - C2C3)C5)S6\right) S7$$
$$+ (S23)S4C7 \quad (A.11)$$

$$a_x = \ -\left(\left((C1C2C3 - C1S2S3)C4 + S1S4\right)C5 - (C1S23)S5\right) S6$$
$$+ \left(-\left((C1C2C3 - C1S2S3)C4 + S1S4\right)S5\right.$$
$$\left.- (C1S23)C5\right)C6 \quad (A.12)$$

$$a_Y = \ -\left(\left((S1C2C3 - S1S2S3)C4 - C1S4\right)C5 - (S1S23)S5\right) S6$$
$$+ \left(-\left((S1C2C3 - S1S2S3)C4 - C1S4\right)S5\right.$$
$$\left.- (S1S23)C5\right)C6 \quad (A.13)$$

$a_Z =$
$-((S23)C4C5 - (S2S3 - C2C3)S5)S6 +$
$(-(S23)C4C5 - (S2S3 - C2C3)C5)C6 \qquad (A.14)$

A.2. ECUACIONES PARA EL CÁLCULO DEL MGI

Para θ4 y θ5, las dos primeras filas de la matriz de 3x3, obtenida de la ecuación

$$(Pt - P5) X (P6 - Pt) = 0$$

$$\begin{aligned}(1,1) = \; & tyD6C4C5S2C3 - R4C2^2C3^2D6C5C4S1 + R4C2C3D6C5C1S4 \\ & - R4S2^2S3^2D6C5C4S1 - R4S2S3D6C5C1S4 + tzD6C5C4S3S1S2 \\ & - tzD6C5C4C3S1C2 + tzD6S5C3S1S2 + tzD6S5S3S1C2 \\ & - D3S2D6C5C1S4 + tzD6C5C1S4 - S1R4C2^2S3^2D6C4C5 \\ & - S1D3C2^2D6C4C5S3 - S1D3C2^2D6S5C3 + tyD6C4C5C2S3 \\ & + tyD6S5C2C3 - tyD6S5S2S3 - S1R4S2^2C3^2D6C4C5 \\ & - D3S2^2D6C5C4S3S1 - D3S2^2D6S5C3S1\end{aligned}$$

$(2,1) =$
$D3S2^2D6S5C3C1 - D3S2D6C5S1S4 + tzD6C5S1S4 + D3S2^2D6C5C4S3C1 +$
$R4C2^2C3^2D6C5C4C1 + R4C2C3D6C5S1S4 + R4S2^2S3^2D6C5C4C1 -$
$R4S2S3D6C5S1S4 + tzD6C5C4C3C1C2 - tzD6C5C4S3C1S2 - tzD6S5S3C1C2 -$
$tzD6S5C3C1S2 + R4C3^2C1S2^2D6C4C5C + R4S3^2C1C2^2D6C4C5 +$
$C1D3C2^2D6C4C5S3 + C1D3C2^2D6S5C3 - txD6C4C5S2C3 - txD6C4C5C2S3 -$
$txD6S5C2C3 + txD6S5S2S3$

Para θ1, θ2 y θ3, las tres filas del vector de posición *p6.*

p6(1,1) = $D6C5C4C3C1C2 - D6C5C4S3C1S2 + D6C5S1S4 - D6S5S3C1C2 -$
$D6S5C3C1S2 + R4S3C1C2 + R4C3C1S2 + D3C1C2$

p6(2,1) = $-D6C5C4S3S1S2 + D6C5C4C3S1C2 - D6C5C1S4 - D6S5C3S1S2 -$
$D6S5S3S1C2 + R4C3S1S2 + R4S3S1C2 + D3S1C2$

p6(3,1) = $D6C4C5S2C3 + D6C4C5C2S3 + D6S5C2C3 - D6S5S2S3 - R4C2C3 +$
$R4S2S3 + S2D3$

Anexo B. Tutoriales

B.1. INSTALACIÓN DE OGRE3D PARA VISUAL STUDIO

Para instalar el motor gráfico de Ogre3D v1.7.2, se sigue de los siguientes pasos.

- **Instalar Visual Studio 2005:** Actualizar al *Service Pack 1* para *Microsoft® Visual Studio® 2005 Standard*, Professional y *Team Editions*, y luego Instalar el service pack 1 para Windows Vista encontrados en los siguientes enlaces.

  - ✓ http://www.microsoft.com/downloads/es-es/details.aspx?FamilyID=bb4a75ab-e2d4-4c96-b39d-37baf6b5b1dc
  - ✓ http://www.microsoft.com/downloads/es-es/details.aspx?familyid=90e2942d-3ad1-4873-a2ee-4acc0aace5b6&displaylang=es

  Tener en cuenta que ambas versiones deben estar en el idioma del Visual Studio.

- **Descargar el paquete:** Dirigirse a la página oficial de Ogre3D. Estando ahí ir al enlace *download* y luego a *SDK (Software Development Kit)*, ahí se encuentran todas las versiones de Ogre incluyendo la versión más actual (v1.7.3). Descargar la v1.7.2 para Visual C++ 2005 de 32 bits.

- **Descomprimir el paquete en una dirección que no sea muy difícil de encontrar en el equipo**, es recomendable que se ubique en el DISCO C.

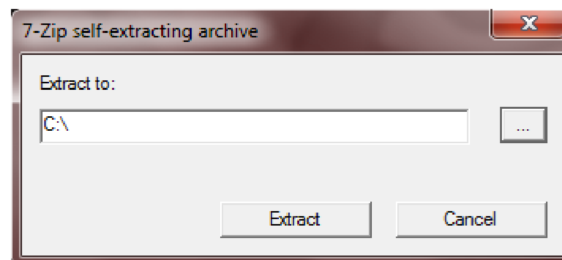


Figura B.1. Ventana para extraer el paquete de Ogre

- **Descargar el asistente de instalación de Ogre:** Las versiones más actualizadas de Ogre tienen asistentes que direccionan Visual Studio hacia los dll´s necesarios para la compilación, ya sea en *Debug* o en *Release.* A continuación se presenta el link de descarga.

    ✓ http://code.google.com/p/ogreappwizards/

  Escoger el archivo Ogre_VC8_AppWizard_1.7.2_2.exe.

- **Ejecutar el asistente de instalación:**

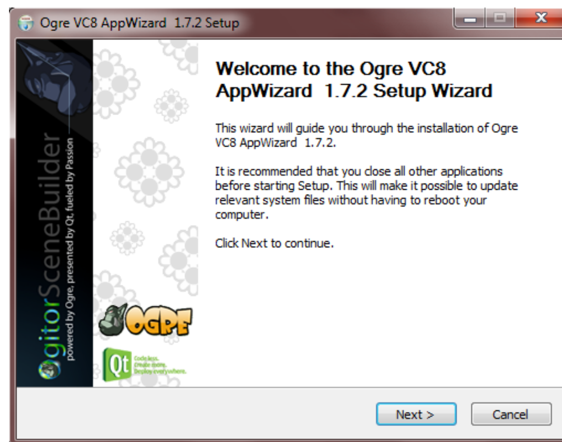


Figura B.2. Ventana de la instalación del *AppWizard*.

  El *wizard* configura las propiedades para que se adicionen los archivos de inclusión y de biblioteca alrededor de una variable de entorno, además de que crea una plantilla para iniciar de forma fácil con Ogre.

- **Variable de entorno:** Para agregar la variable, se abre una ventana de comandos y se escribe el comando *setx* y luego la dirección de la carpeta donde se extrajo el paquete de Ogre, como lo vemos a continuación.

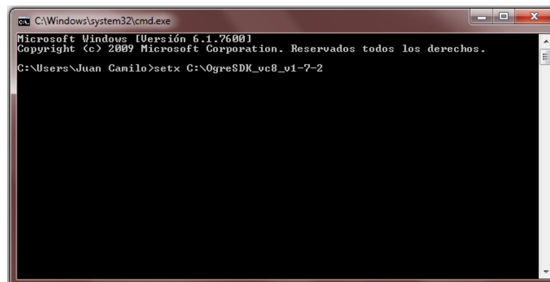


Figura B.3. Comandos para agregar la variable de entorno.

- Instalar las librerías DirectX (agosto 2009) del siguiente enlace e instalar:

  ✓ http://www.microsoft.com/download/en/details.aspx?id=6883

- **Crear un nuevo proyecto en Visual Studio:** Nos dirigimos a Archivo > Nuevo > Proyecto.

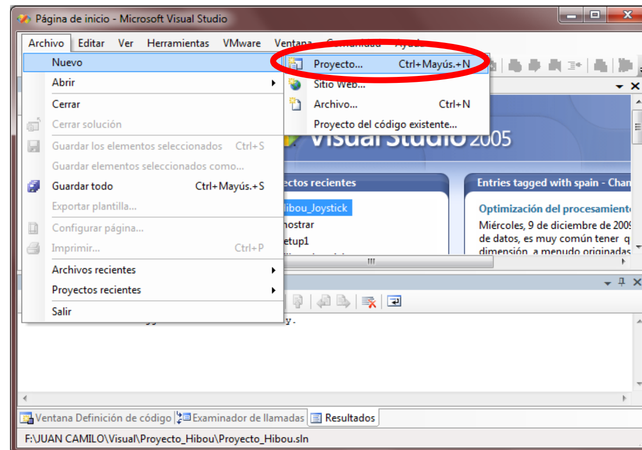


Figura B.4. Abrir nuevo proyecto.

Seleccionamos el tipo de proyecto *Visual C++,* la plantilla *OGRE Application* y luego le damos un nombre a nuestro proyecto. Aceptamos y finalizamos el asistente de aplicaciones.

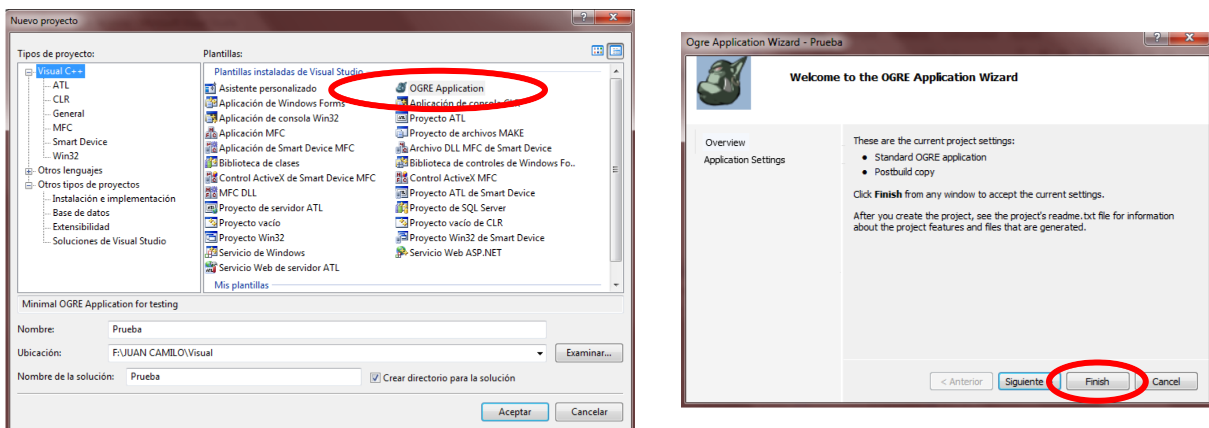


Figura B.5. Seleccionar tipo de proyecto (Izq.), Asistente de aplicaciones Ogre (der.)

Visual Studio automáticamente abre una plantilla sobre la cual se puede probar si en verdad se configuró de la forma correcta el motor gráfico, además de seguir trabajando cobre la misma plantilla.

Esta configuración tiene un pequeño problema que encontramos al instalar esta versión de Ogre3D y que se explicará y corregirá a continuación.

En la carpeta *OgreSDK_vc8_v1-7-2* se encuentra un carpeta con el nombre *boost_1_42*. El *AppWizzard* se encuentra configurado para re direccionarse a una carpeta con una versión de *boost* más reciente, haciendo referencia a la *boost_1_44.* Para solucionar este inconveniente hacemos lo siguiente.

Vamos al explorador de soluciones, damos click derecho sobre el nombre del proyecto y seleccionamos "propiedades"**.**

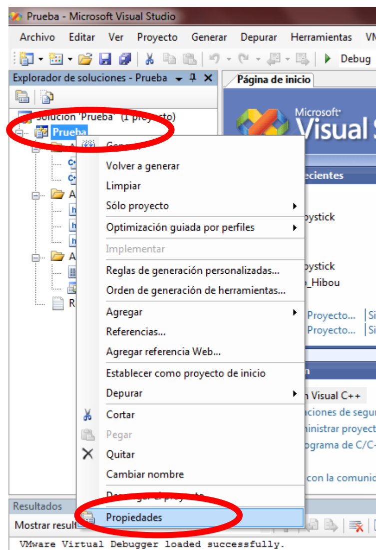


Figura B.6. Explorador de soluciones

Se abre la ventana de "páginas de propiedades"**.** Es importante tener en cuenta cómo vamos a correr el proyecto si en *Debug* o en *Release.* Para el proyecto se compilara en *Release,* así que vamos a Administrador de Configuración y cambiamos a *Release.* Luego vamos al árbol de "propiedades de configuración" > "C/C++" > "General" > "Directorios de inclusión adicionales" y cambiamos la siguiente línea:

$(OGRE_HOME)\boost_1_44

Por:

$(OGRE_HOME)\boost_1_42

Hacemos lo mismo en “Vinculador” > “General” > “Directorios de bibliotecas adicionales”

$(OGRE_HOME)\boost_1_44\lib

Por:

$(OGRE_HOME)\boost_1_42\lib

Ya hecho esto generamos la solucion y buscamos en la carpeta de Ogre la carpeta bin y luego release, donde se encuentra el ejecutable de este archivo. Ejecutamos la aplicación que inicia con la siguiente ventana.

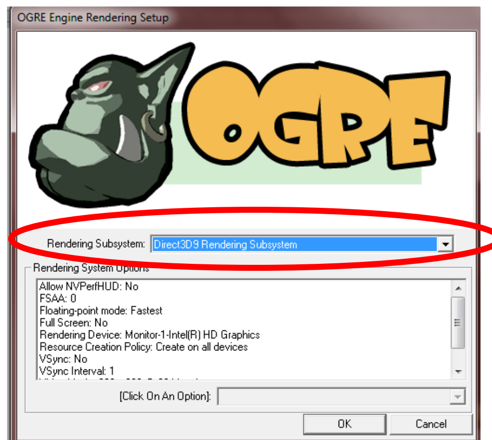


Figura B.7. Ventana de inicio de aplicación de Ogre

Seleccionamos el *rendering subsystem* y luego *OK.*

Figura B.8. Aplicación prueba de Ogre3D.

B.2. INSTALACIÓN DE BLENDER Y SU UTILIZACIÓN

- **DESCARGA DE ARCHIVOS E INSTALACIÓN**

Blender es software libre y se puede obtener del siguiente enlace de descarga:

✓ *http://www.blender.org/download/get-blender/*

Además de éste se necesita la versión apropiada de Python, que es el lenguaje sobre el cual Blender es ejecutado. En ese mismo link de descarga se puede verificar la versión compatible con el Blender y luego dirigirse al siguiente enlace para realizar su descarga.

✓ *http://python.org/download/*

Ogre3D tiene un script que trabaja junto a Blender que se puede descargar de la siguiente dirección.

✓ *http://www.xullum.net/lefthand/downloads/temp/BlenderExport.zip*

Una vez que ya se tengan los archivos de Blender, Python, realizamos su instalación. Ya hecho esto descomprimimos el archivo *.zip* llamado BlenderExport y copiamos esos archivos en la siguiente dirección.

✓ *C:\Users\<name>\AppData\Roaming\BlenderFoundation\Blender\.blender\sripts*

- **TRABAJANDO CON EL SCRIPT OGREMESHES**

Antes de exportar una malla se necesita importarla, es decir traerla a Blender. Para ello nos dirigimos a "*File*" > "*Import*", y buscamos la extensión de nuestro interés.

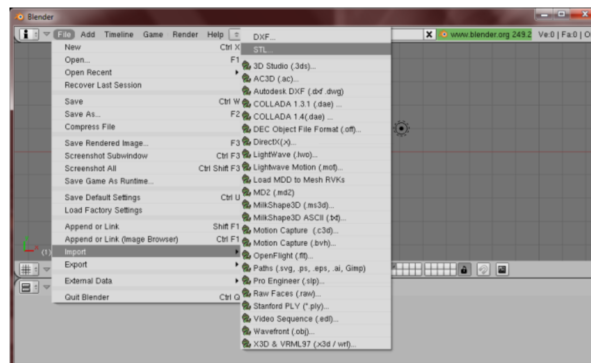


Figura B.9. Interfaz de Blender para importar un archivo.

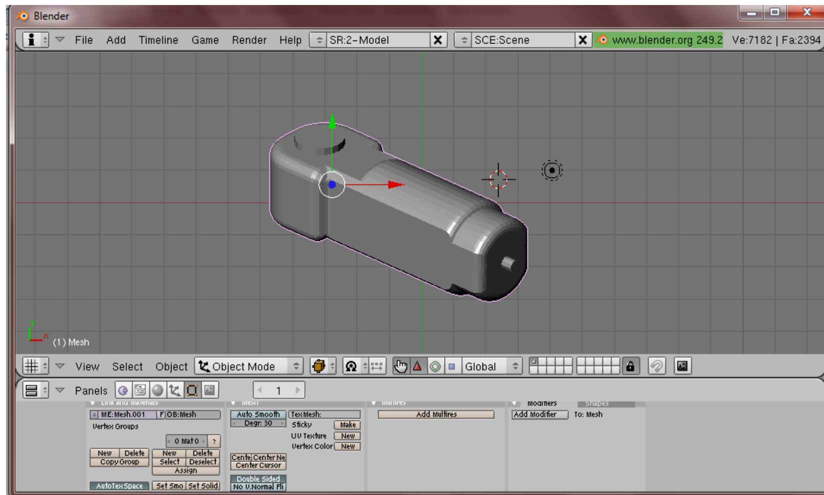


Figura B.10. Pieza importada a Blender.

Ahora podemos agregarle un material. Presionamos F5 y se activan las propiedades de materiales para la malla.

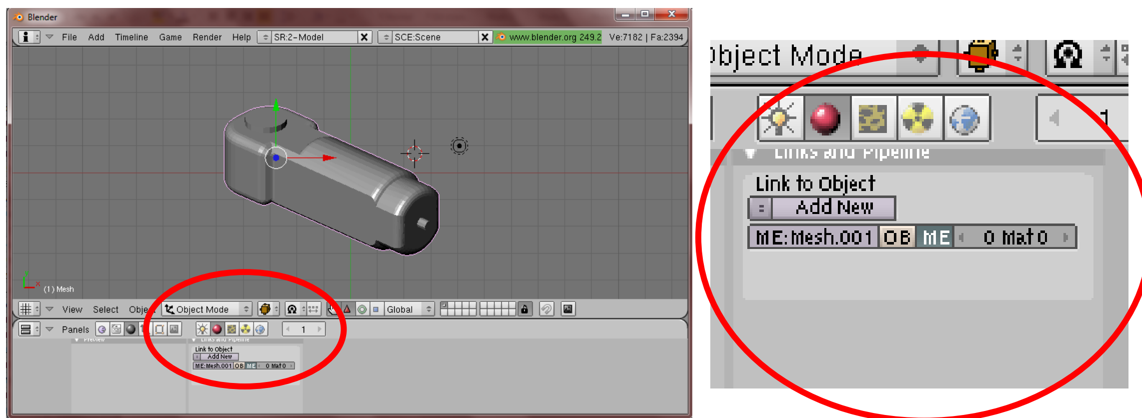


Figura B.11. Propiedades de los materiales.

Cambiamos el nombre *Mesh.001* por el nombre (en este caso “3”) de la malla y damos click en *Add New.*

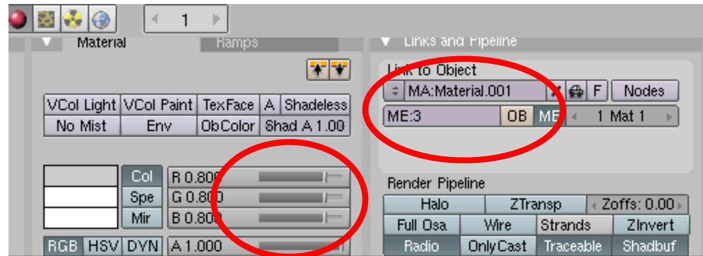


Figura B.12. Panel de colores y efectos de malla.

Cambiamos el nombre del material por el mismo nombre de la malla y modificamos las barras de color (RGB) para dar una tonalidad a la malla. Luego de esto nos preparamos para exportar la malla, asi que vamos a “*File*” > “*Export*” > “*OGRE Meshes*”.

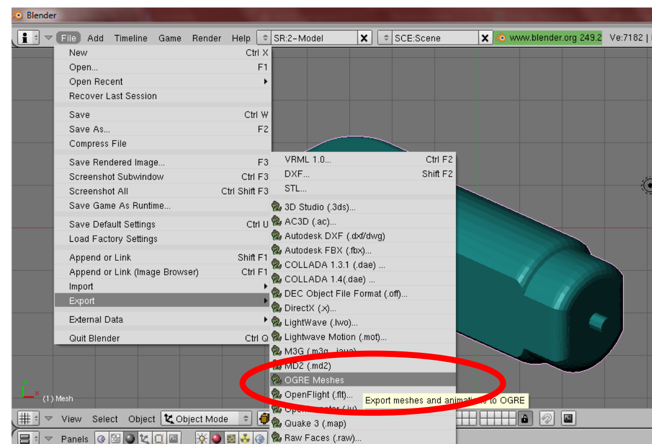


Figura B.13. Ingresando al script de OGRE Meshes

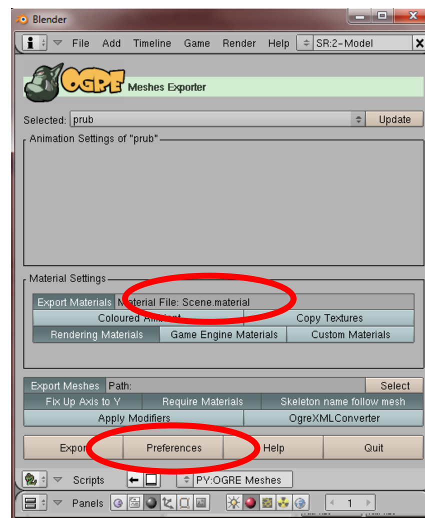


Figura B.14. Interfaz de OGRE Meshes.

El círculo rojo presenta el nombre del material de salida *Scene.material* lo cambiamos al nombre de nuestro archivo (*3.material)*. Ahora como último paso de configuración seleccionamos la casilla *Preferences.*

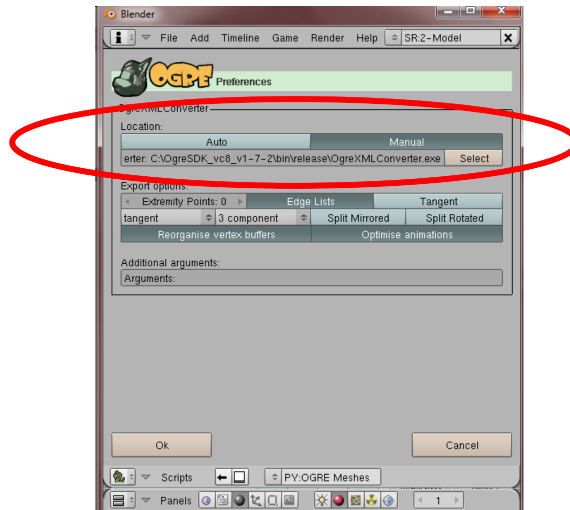


Figura B.15. Preferencias de OGRE Meshes.

Activamos el botón manual y seleccionamos el ejecutable *OgreXMLConverter.exe* que se encuentra en la carpeta *bin\release* de Ogre y le damos *OK*.

Buscamos una carpeta para exportar los archivos y presionamos el botón *Export*. De esta forma ya tenemos el *.mesh* y el *.material* que son guardados en la carpeta de Ogre de la siguiente forma:

- ✓ El *.mesh* en la carpeta: …\media\models de Ogre.
- ✓ El *.material* en la carpeta: …\media\materials\scripts de Ogre.

A continuación se observa el ambiente virtual junto con la pieza renderizada.

Figura B.16. Pieza renderizada en Ogre3D.

B.3. COMANDOS FUNCIONALES EN EL SOFTWARE

Para ofrecer una mayor comodidad en la interacción de la interfaz virtual con el software, se han configurado las siguientes teclas:

TECLAS	FUNCION
Q	Subir el robot
W	Bajar el robot
A	Habilitar dispositivo auxiliar para mover el robot
D	Habilita el movimiento de la cámara del robot con el casco
N	Calcula el MGD en "MGD_posActual_JOYST.txt"
U	Subir cámara del ambiente
R	Habilita el movimiento automático con la consigna circular
E	Habilita el movimiento del robot con el joystick
Z	Habilita el movimiento del robot en el eje Z pero con el casco
U	Subir cámara del ambiente
J	Bajar cámara del ambiente
K	Ver los polígonos del ambiente
L	Ver los polígonos de la mini pantalla
UP	Cámara del ambiente en el eje -Z
DOWN	Cámara del ambiente en el eje +Z
RIGHT	Cámara del ambiente en el eje +X
LEFT	Cámara del ambiente en el eje –X

Tabla B.1. Comandos con el teclado.

Para girar la cámara del ambiente se hace uso del ratón presionando los dos botones, izquierdo y derecho del dispositivo y moviéndolo según sea necesario. Un movimiento en X del ratón genera un movimiento en Y de la cámara ante el ambiente, y un movimiento del ratón en Y produce un movimiento en X de la cámara.

B.4. Habilitando el joystick en Ogre3D

Para poder implementar el joystick en Ogre3D se tiene que seguir unos cuantos pasos, primero que todo se debe llamar a la librería de *OISJoyStck.h* la cual

contiene todas las funciones y la estructura para su implementación, si se está programando orientado a objetos, como se realizó en este proyecto, se debe declarar como función pública el *listener* del *JoyStick* que se encargará de leer todos las variables provenientes de éste.

```
public OIS::JoyStickListener
```

Una vez hecho esto se deben declarar las funciones que se van a utilizar, hay que tener cuidado en este punto ya que solo se podrán definir en el archivo de cabecera las funciones que estén en el *OISJoyStick.h* declaradas como funciones virtuales puras. Estas funciones se pueden reconocer ya que tienen un **"= 0;"** al final de ésta.

```
virtual bool buttonPressed(const JoyStickEvent &arg,int button)=0;

virtual bool buttonReleased(const JoyStickEvent &arg,int butto)=0;

virtual bool axisMoved(const JoyStickEvent &arg, int axis)=0;
```

De esta manera estas tres funciones ya podrán ser heredadas al archivo principal y ser utilizadas de manera normal. La primera función se llamará cada vez que un botón especificado dentro de la función sea presionado, la segunda función se llamará en caso contrario cada vez que el mismo botón sea liberado, y por último la última función será llamada en el momento en que la palanca del joystick sea movida en cualquier dirección.

Para lograr que lo anterior funcione se debe definir una variable de tipo *OISJoyStick* de esta manera.

```
OIS::JoyStick* mJoystick;
```

Una vez declarado lo anterior en el archivo de cabecera se deben configurar sus parámetros en el código fuente. En el `createFrameListener` se debe inicializar el OIS (*Output Input Systems*). Este punto sirve para inicializar los distintos tipos de mandos que se le puedan conectar al programa como los son el teclado, el mouse y el joystick.

```
Ogre::LogManager::getSingletonPtr()->logMessage("* Initializing OIS *");
OIS::ParamList pl;
size_t windowHnd = 0;
std::ostringstream windowHndStr;
Window->getCustomAttribute("WINDOW", &windowHnd);
windowHndStr << windowHnd;
pl.insert(std::make_pair(std::string("WINDOW"), windowHndStr.str()));
mInputManager = OIS::InputManager::createInputSystem( pl );
```

```
//inicializando las variables  con buffer
mJoystick = static_cast<OIS::JoyStick*>(mInputManager->createInputObject(
OIS::OISJoyStick, true ));
```

El `createFrameListener` es una función la cual solo se llama una vez por lo que debe incializarse la variable global declarada en el archivo de cabecera.

```
mJoystick->setEventCallback(this);
```

Ahora cualquier cambio de estado de tipo *JoyStick* se le asignara a *mJoyStick*. Para terminar de configurar el *JoyStick* se tienen que capturar cada cambio en el mismo, los cuales deben ser leídos cada *frame*, por lo que dentro del `frameRenderingQueued` *se* le ordena a *mJoyStick* que lea los cambios de estado durante el último *Frame*.

```
mJoystick->capture();
```

Ya terminada la configuración del *JoyStick*, se puede empezar a realizar las acciones que desee dependiendo de las entradas del mismo. Por ejemplo como se mostró en la declaración de las funciones en la página anterior, tenemos dos variables "*axis*" y "*button*", *axis* puede tomar uno de 2 valores 0 o 1 indicando si está realizando movimientos en X o Y, por lo tanto podemos usar algo como esto.

```
Ogre::int32 x = evt.state.mAxes[axis].abs;
```

Hemos definido un entero X de 32 bits de tipo Ogre al cual se le asignan los valores del cambio en el movimiento de la palanca que pueden ir de -32767 hasta 32767 dependiendo del valor de axis, así podríamos generar un *switch* como este.

```
switch(axis)
{
     case 0:
          V1yJoys = x;
          break;
     case 1:
          V1xJoys = x;
          break;
}
```

Para el caso de la variable *button*, ésta devuelve un valor entero dependiendo de la cantidad de botones que posea el Joystick, así dependiendo de qué botón se esté presionando se puede realizar una acción distinta, igualmente para cada vez que se deje de presionar el botón.

Anexo C. CODIGO UTILIZADO

C.1. CODIGO NECESARIO PARA EL RENDERIZADO DEL ROBOT

El siguiente código se encuentra ubicado en la función `createScene` de la arquitectura de Ogre.

```
Ogre::SceneNode *np_0 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("ejebase");
np_0->yaw(Ogre::Degree(0));

Ogre::Entity *entbase = mSceneMgr->createEntity( "base", "base.mesh" );
Ogre::SceneNode *nodobase = np_0->createChildSceneNode( "RobotNode",Ogre::Vector3(0,-
20,0));
nodobase->attachObject( entbase );
nodobase->scale(2.7,2.7,2.7);

Ogre::SceneNode *np_1 = nodobase->createChildSceneNode(
"RobotNode15",Ogre::Vector3(0,0,0));
np_1->pitch(Ogre::Degree(-90));
np_1->roll(Ogre::Degree(0));

Ogre::Entity *entart1 = mSceneMgr->createEntity( "rotoide1", "1.mesh" );
nodoart1 = np_1->createChildSceneNode( "RobotNode2",Ogre::Vector3(0,0,0));
nodoart1->attachObject( entart1);

Ogre::SceneNode *np_2 = nodoart1->createChildSceneNode(
"RobotNode16",Ogre::Vector3(0,0,3.6));
np_2->pitch(Ogre::Degree(90));
np_2->roll(Ogre::Degree(-90));

Ogre::Entity *entbr1 = mSceneMgr->createEntity( "br1", "2.mesh" );
nodobr1 = np_2->createChildSceneNode( "RobotNode3",Ogre::Vector3(0,0,0));
nodobr1->attachObject( entbr1);

Ogre::SceneNode *np_3 = nodobr1->createChildSceneNode(
"RobotNode34",Ogre::Vector3(0,11.9,0));
np_3->roll(Ogre::Degree(-90));

Ogre::Entity *entbr = mSceneMgr->createEntity( "br", "3.mesh" );
nodobr = np_3->createChildSceneNode( "RobotNode4",Ogre::Vector3(0,0,0));
nodobr->attachObject( entbr);

Ogre::SceneNode *np_4 = nodobr->createChildSceneNode(
"RobotNode17",Ogre::Vector3(0,0,0));
np_4->pitch(Ogre::Degree(-90));

Ogre::Entity *entpasiva1 = mSceneMgr->createEntity( "primera_pasiva", "4.mesh" );
nodopasiva1 = np_4->createChildSceneNode( "RobotNode5",Ogre::Vector3(0,0,15.2));
nodopasiva1->attachObject( entpasiva1);

Ogre::SceneNode *np_5 = nodopasiva1->createChildSceneNode(
"RobotNode18",Ogre::Vector3(0,0,0));
np_5->pitch(Ogre::Degree(90));
np_5->roll(Ogre::Degree(90));

Ogre::Entity *entpasiva2 = mSceneMgr->createEntity( "segunda_pasiva", "5.mesh" );
nodopasiva2 = np_5->createChildSceneNode( "RobotNode6",Ogre::Vector3(0,0,0));
```

```
nodopasiva2->attachObject( entpasiva2);

Ogre::Entity *entcam2 = mSceneMgr->createEntity( "portaendoscopio", "port.mesh" );
nodocam2 = nodopasiva2->createChildSceneNode( "RobotNode7",Ogre::Vector3(0,0,0));
nodocam2->attachObject( entcam2);

Ogre::SceneNode *np_6 = nodocam2->createChildSceneNode(
"RobotNode19",Ogre::Vector3(0,0,0));
np_6->pitch(Ogre::Degree(90));
np_6->roll(Ogre::Degree(90));
```

En el codigo anterior se puede identificar la funcion de los nodos padre con el sufijo ***np*** los cuales son utilizados para la orientacion de cada pieza.

C.2. CODIGO PARA EL RENDERIZADO DEL CUARTO QUIRÚRGICO

El siguiente debe colocarse en el función `createScene` de la arquitectura de Ogre.

```
Ogre::Entity *suelo = mSceneMgr->createEntity("suel","suelo.mesh");
Ogre::SceneNode *SSuelo = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("sueloNode",Ogre::Vector3(0,-71.7,0));
SSuelo->attachObject(suelo);
SSuelo->scale(3,3,3);

Ogre::Entity *pared = mSceneMgr->createEntity("pared","pared.mesh");
Ogre::SceneNode *ppared = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("parednodo",Ogre::Vector3(0,-71.7,0));
ppared->attachObject(pared);
ppared->scale(3,3,3);

Ogre::Entity *mesa = mSceneMgr->createEntity("mesa","mesa.mesh");
Ogre::SceneNode *MMesa = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("mesaNode",Ogre::Vector3(9,0,0));
MMesa->attachObject(mesa);
MMesa->scale(3,3,3);//*/
```

C.3. FUNCION OBJETIVO EN MATLAB

Esta es la función obetivo (llamada *f_solveDOS* en Matlab®) a la que se le aplica el método de LM con el comando *f_solve*

```
function out=f_solveDOS(X)
global  R4 D3 D6 tx ty tz x y z

t1=X(1);
t2=X(2);
t3=X(3);
t4=X(4);
t5=X(5);

if (t2<0)
    t2=t2+pi;
```

```
elseif (pi<t2)
    t2=t2-pi;
end
%**************************
%*********t1***************
% ecuacion de la condicion de longitud V5t-Vt6
        p5= [cos(t1)*(R4*sin(t2 + t3) + D3*cos(t2));
             sin(t1)*(R4*sin(t2 + t3) + D3*cos(t2));
                      D3*sin(t2) - R4*cos(t2 + t3)];

        pz=p5(3);

while(pz<tz)
        t2=t2+0.01;
        t3=t3+0.01;
        p5= [cos(t1)*(R4*sin(t2 + t3) + D3*cos(t2));
             sin(t1)*(R4*sin(t2 + t3) + D3*cos(t2));
                      D3*sin(t2) - R4*cos(t2 + t3)];

        pz=p5(3);
end
tr = [tx; ty; tz];

p6=[(((1/2*cos(-t3+t1-
t2)+1/2*cos(t3+t1+t2))*cos(t4)+sin(t1)*sin(t4))*cos(t5)-(-1/2*sin(-
t3+t1-t2)+1/2*sin(t3+t1+t2))*sin(t5))*D6-(1/2*sin(-t3+t1-t2)-
1/2*sin(t3+t1+t2))*R4+1/2*D3*cos(t1-t2)+1/2*D3*cos(t1+t2)
    (((1/2*sin(t3+t1+t2)+1/2*sin(-t3+t1-t2))*cos(t4)-
cos(t1)*sin(t4))*cos(t5)-(-1/2*cos(t3+t1+t2)+1/2*cos(-t3+t1-
t2))*sin(t5))*D6-(1/2*cos(t3+t1+t2)-1/2*cos(-t3+t1-
t2))*R4+1/2*D3*sin(t1+t2)+1/2*D3*sin(t1-t2)

(sin(t2+t3)*cos(t4)*cos(t5)+cos(t2+t3)*sin(t5))*D6-
cos(t2+t3)*R4+sin(t2)*D3];
V5t = tr - p5;
Vt6 = p6 - tr;

eq1=p6(1)-x;
eq2=p6(2)-y;
eq3=p6(3)-z;

cruz=cross(V5t,Vt6);
eq4=cruz(1);
eq5=cruz(2);

out=[eq1;eq2;eq3;eq4;eq5];
```

C.4. FUNCIÓN OBJETIVO EN C++

```
double f_solve(double t[5], double x2, double y2, double z2)
{

      // se reciben los valores iniciales y se devuelven 5 valores
resultantes en forma de un vector puntero
        int i;
        // definicion de las variables
        double Fk,t1,t2,t3,t4,t5;
        double p5a, p5b, p5c, p5[3];
        double p6a, p6b, p6c, p6[3];
        double tr[3],V5t[3],Vt6[3],Vt[3],resultado[5];

       /*//TROCAR arriba del hombligo
```

```
        double TX = 0.45;
        double TY = -0.1;
        double TZ = 0.18;//*/

       //TROCAR cerca a la cintura
        double TX = 0.45;
        double TY = 0.07;
        double TZ = 0.18;//*/

/*//TROCAR en el hombligo
        double TX = 0.45;
        double TY = 0;
        double TZ = 0.23;//*/


  double X,Y,Z;

  double pz;
  X = x2;
  Y = y2;
  Z = z2;

  t1=t[0];
  t2=t[1];
  t3=t[2];
  t4=t[3];
  t5=t[4];

  for(i=0;i<=4;i++)
  {
      resultado[i]=0; //limpieza de la variable
  }


  if (t2<0)
      t2=t2+PI;
  else
  {
      if (PI<t2)
      t2=t2-PI;
  }//*/
//      %*************************
//      %*********t1**************
//      % ecuacion de la condicion de longitud V5t-Vt6
  p5a= (cos(t1)*(R4*sin(t2 + t3) + D3*cos(t2)));
  p5b= (sin(t1)*(R4*sin(t2 + t3) + D3*cos(t2)));
  p5c= (D3*sin(t2) - R4*cos(t2 + t3));

  pz=p5c;

  while(pz<TZ)
  {
          t2=t2+0.01;
          t3=t3+0.01;
          p5a= (cos(t1)*(R4*sin(t2 + t3) + D3*cos(t2)));
          p5b= (sin(t1)*(R4*sin(t2 + t3) + D3*cos(t2)));
          p5c= (D3*sin(t2) - R4*cos(t2 + t3));


          pz=p5c;
  }
```

```
        p5[0]=p5a;
        p5[1]=p5b;
        p5[2]=p5c;

        tr[0]=TX;
        tr[1]=TY;
        tr[2]=TZ;

        p6a=(((0.5*cos(-t3+t1-
t2)+0.5*cos(t3+t1+t2))*cos(t4)+sin(t1)*sin(t4))*cos(t5)-(-0.5*sin(-t3+t1-
t2)+0.5*sin(t3+t1+t2))*sin(t5))*D6-(0.5*sin(-t3+t1-t2)-
0.5*sin(t3+t1+t2))*R4+0.5*D3*cos(t1-t2)+0.5*D3*cos(t1+t2);

        p6b=(((0.5*sin(t3+t1+t2)+0.5*sin(-t3+t1-t2))*cos(t4)-
cos(t1)*sin(t4))*cos(t5)-(-0.5*cos(t3+t1+t2)+0.5*cos(-t3+t1-t2))*sin(t5))*D6-
(0.5*cos(t3+t1+t2)-0.5*cos(-t3+t1-t2))*R4+0.5*D3*sin(t1+t2)+0.5*D3*sin(t1-t2);
        p6c=(sin(t2+t3)*cos(t4)*cos(t5)+cos(t2+t3)*sin(t5))*D6-
cos(t2+t3)*R4+sin(t2)*D3;


        p6[0]=p6a;
        p6[1]=p6b;
        p6[2]=p6c;

        for(i=0;i<=2;i++)
        {
                      V5t[i]=tr[i]-p5[i];
        }

        for(i=0;i<=2;i++)
        {
                      Vt6[i]=p6[i]-tr[i];
        }

//resultado de las primeras variables
        resultado[0]=p6a-X;
        resultado[1]=p6b-Y;
        resultado[2]=p6c-Z;


        Vt[0]=(V5t[1]*Vt6[2])-(V5t[2]*Vt6[1]);
        Vt[1]=-((V5t[0]*Vt6[2])-(V5t[2]*Vt6[0]));
        Vt[2]=(V5t[0]*Vt6[1])-(V5t[1]*Vt6[0]);
//resultado de las ultimas variables
        resultado[3]=Vt[0];
        resultado[4]=Vt[1];


Fk=(resultado[0]*resultado[0])+(resultado[1]*resultado[1])+(resultado[2]*resultad
o[2])+(resultado[3]*resultado[3])+(resultado[4]*resultado[4]);
       Fk=sqrt(Fk);
       return Fk;


}
```

C.5. ARCHIVO CABECERA Y CUERPO DEL NELDER-MEAD

Archivo cabecera del método de Nelder-Mead (asa047.h)

void nelmin (double fn (double xi[], double x, double y, double z), int n, double start[], double xmin[], double *ynewlo, double reqmin, double step[], int konvge, int kcount,   int *icount, int *numres, int *ifault , double COR[]);

void timestamp (void);

Archivo codigo fuente del metodo de Nelder-Mead (asa047.cpp)

```
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <ctime>
# include <cmath>

using namespace std;

# include "asa047.H"


void nelmin (double fn(double x[], double x1, double y1, double z1), int n,
double start[], double xmin[],  double *ynewlo, double reqmin, double
step[], int konvge, int kcount,     int *icount, int *numres, int
*ifault,double COR[3])
{
            double ccoeff = 0.5;
            double del;
            double dn;
            double dnn;
            double ecoeff = 2.0;
            double eps = 0.001;
            int i;
            int ihi;
            int ilo;
            int j;
            int jcount;
            int l;
            int nn;
            double *p;
            double *p2star;
            double *pbar;
            double *pstar;
            double rcoeff = 1.0;
            double rq;
            double x;
            double *y;
            double y2star;
            double ylo;
            double ystar;
            double z;
//
//  Check the input parameters.
//
            if ( reqmin <= 0.0 )
            {
                 *ifault = 1;
                 return;
            }
```

```
  if ( n < 1 )
  {
       *ifault = 1;
       return;
  }

  if ( konvge < 1 )
  {
       *ifault = 1;
       return;
  }

  p = new double[n*(n+1)];
  pstar = new double[n];
  p2star = new double[n];
  pbar = new double[n];
  y = new double[n+1];

  *icount = 0;
  *numres = 0;

  jcount = konvge;
  dn = ( double ) ( n );
  nn = n + 1;
  dnn = ( double ) ( nn );
  del = 1.0;
  rq = reqmin * dn;
//
//  Initial or restarted loop.
//
  for ( ; ; )
  {
       for ( i = 0; i < n; i++ )
       {
         p[i+n*n] = start[i];
       }
       y[n] = fn ( start,COR[0],COR[1],COR[2]);
       *icount = *icount + 1;

       for ( j = 0; j < n; j++ )
       {
         x = start[j];
         start[j] = start[j] + step[j] * del;
         for ( i = 0; i < n; i++ )
         {
              p[i+j*n] = start[i];
         }
         y[j] = fn ( start,COR[0],COR[1],COR[2]);
         *icount = *icount + 1;
         start[j] = x;
       }
//
//  The simplex construction is complete.
//
//  Find highest and lowest Y values.  YNEWLO = Y(IHI) indicates
//  the vertex of the simplex to be replaced.
//
```

```
                ylo = y[0];
                ilo = 0;

                for ( i = 1; i < nn; i++ )
                {
                  if ( y[i] < ylo )
                  {
                       ylo = y[i];
                       ilo = i;
                  }
                }
        //
        //  Inner loop.
        //
                for ( ; ; )
                {
                  if ( kcount <= *icount )
                  {
                       break;
                  }
                  *ynewlo = y[0];
                  ihi = 0;

                  for ( i = 1; i < nn; i++ )
                  {
                       if ( *ynewlo < y[i] )
                       {
                         *ynewlo = y[i];
                         ihi = i;
                       }
                  }
        //
        //  Calculate PBAR, the centroid of the simplex vertices
        //  excepting the vertex with Y value YNEWLO.
        //
                  for ( i = 0; i < n; i++ )
                  {
                       z = 0.0;
                       for ( j = 0; j < nn; j++ )
                       {
                         z = z + p[i+j*n];
                       }
                       z = z - p[i+ihi*n];
                       pbar[i] = z / dn;
                  }
        //
        //  Reflection through the centroid.
        //
                  for ( i = 0; i < n; i++ )
                  {
                       pstar[i]  =  pbar[i]  +  rcoeff  *  (  pbar[i]  -
p[i+ihi*n] );
                  }
                  ystar = fn ( pstar,COR[0],COR[1],COR[2]);
                  *icount = *icount + 1;
        //
        //  Successful reflection, so extension.
        //
```

```
                    if ( ystar < ylo )
                    {
                         for ( i = 0; i < n; i++ )
                         {
                           p2star[i]  =  pbar[i]  +  ecoeff  *  (  pstar[i]  -
pbar[i] );
                         }
                         y2star = fn ( p2star,COR[0],COR[1],COR[2]);
                         *icount = *icount + 1;
          //
          //  Check extension.
          //
                         if ( ystar < y2star )
                         {
                           for ( i = 0; i < n; i++ )
                           {
                                p[i+ihi*n] = pstar[i];
                           }
                           y[ihi] = ystar;
                         }
          //
          //  Retain extension or contraction.
          //
                         else
                         {
                           for ( i = 0; i < n; i++ )
                           {
                                p[i+ihi*n] = p2star[i];
                           }
                           y[ihi] = y2star;
                         }
                    }
          //
          //  No extension.
          //
                    else
                    {
                         l = 0;
                         for ( i = 0; i < nn; i++ )
                         {
                           if ( ystar < y[i] )
                           {
                                l = l + 1;
                           }
                         }

                         if ( 1 < l )
                         {
                           for ( i = 0; i < n; i++ )
                           {
                                p[i+ihi*n] = pstar[i];
                           }
                           y[ihi] = ystar;
                         }
          //
          //  Contraction on the Y(IHI) side of the centroid.
          //
                         else if ( l == 0 )
```

```
                        {
                          for ( i = 0; i < n; i++ )
                          {
                               p2star[i] = pbar[i] + ccoeff * ( p[i+ihi*n]
- pbar[i] );
                          }
                          y2star = fn ( p2star,COR[0],COR[1],COR[2]);
                          *icount = *icount + 1;
          //
          //  Contract the whole simplex.
          //
                          if ( y[ihi] < y2star )
                          {
                               for ( j = 0; j < nn; j++ )
                               {
                                 for ( i = 0; i < n; i++ )
                                 {
                                      p[i+j*n] = ( p[i+j*n] + p[i+ilo*n] ) *
0.5;

                                      xmin[i] = p[i+j*n];
                                 }
                                 y[j] = fn ( xmin,COR[0],COR[1],COR[2]);
                                 *icount = *icount + 1;
                               }
                               ylo = y[0];
                               ilo = 0;

                               for ( i = 1; i < nn; i++ )
                               {
                                 if ( y[i] < ylo )
                                 {
                                      ylo = y[i];
                                      ilo = i;
                                 }
                               }
                               continue;
                          }
          //
          //  Retain contraction.
          //
                          else
                          {
                               for ( i = 0; i < n; i++ )
                               {
                                 p[i+ihi*n] = p2star[i];
                               }
                               y[ihi] = y2star;
                          }
                        }
          //
          //  Contraction on the reflection side of the centroid.
          //
                        else if ( l == 1 )
                        {
                          for ( i = 0; i < n; i++ )
                          {
                               p2star[i] = pbar[i] + ccoeff * ( pstar[i] -
pbar[i] );
```

```
                }
                y2star = fn ( p2star,COR[0],COR[1],COR[2]);
                *icount = *icount + 1;
//
//  Retain reflection?
//
                if ( y2star <= ystar )
                {
                     for ( i = 0; i < n; i++ )
                     {
                       p[i+ihi*n] = p2star[i];
                     }
                     y[ihi] = y2star;
                }
                else
                {
                     for ( i = 0; i < n; i++ )
                     {
                       p[i+ihi*n] = pstar[i];
                     }
                     y[ihi] = ystar;
                }
              }
         }
//
//  Check if YLO improved.
//
         if ( y[ihi] < ylo )
         {
              ylo = y[ihi];
              ilo = ihi;
         }
         jcount = jcount - 1;

         if ( 0 < jcount )
         {
              continue;
         }
//
//  Check to see if minimum reached.
//
         if ( *icount <= kcount )
         {
              jcount = konvge;

              z = 0.0;
              for ( i = 0; i < nn; i++ )
              {
                z = z + y[i];
              }
              x = z / dnn;

              z = 0.0;
              for ( i = 0; i < nn; i++ )
              {
                z = z + pow ( y[i] - x, 2 );
              }
```

```
          if ( z <= rq )
          {
            break;
          }
        }
      }
//
//  Factorial tests to check that YNEWLO is a local minimum.
//
      for ( i = 0; i < n; i++ )
      {
        xmin[i] = p[i+ilo*n];
      }
      *ynewlo = y[ilo];

      if ( kcount < *icount )
      {
        *ifault = 2;
        break;
      }

      *ifault = 0;

      for ( i = 0; i < n; i++ )
      {
        del = step[i] * eps;
        xmin[i] = xmin[i] + del;
        z = fn ( xmin,COR[0],COR[1],COR[2]);
        *icount = *icount + 1;
        if ( z < *ynewlo )
        {
             *ifault = 2;
             break;
        }
        xmin[i] = xmin[i] - del - del;
        z = fn ( xmin,COR[0],COR[1],COR[2]);
        *icount = *icount + 1;
        if ( z < *ynewlo )
        {
             *ifault = 2;
             break;
        }
        xmin[i] = xmin[i] + del;
      }

      if ( *ifault == 0 )
      {
        break;
      }
//
//  Restart the procedure.
//
      for ( i = 0; i < n; i++ )
      {
        start[i] = xmin[i];
      }
      del = eps;
      *numres = *numres + 1;
```

```
            }
            delete [] p;
            delete [] pstar;
            delete [] p2star;
            delete [] pbar;
            delete [] y;

            return;
}

void timestamp ( void )
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );

  len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

  cout << time_buffer << "\n";

  return;
# undef TIME_SIZE
}
```

C.6. FUNCION PARA INICIALIZAR EL ROBOT

A continuación se presenta la función que se encarga de posicionar el efector final en el espacio cartesiano. Los variables `cons1`, `cons2` y `cons3` son los vectores de la consigna circular. Inicialmente el robot se encuentra preparado para realizar una trayectoria circular, pero facilmente pueden modificarse en el caso de que solo se vaya a utilizar algun dispositivo de entrada como Joystick o el casco.

```
void Hibou_Joystick::inicializar()
{
              datanum = 0;
              datanum2 = 0;

              ALTURA = 7.08;

               Xmouse = cons1[2];
               Ymouse = cons2[2];
               Zmouse = cons3[2];

               MGI_model(Xmouse,Ymouse,Zmouse,Qi);

               double q1 = Qi[0];
               double q2 = Qi[1];
               double q3 = Qi[2];
               double q4 = Qi[3];
               double q5 = Qi[4];
```

```
            double q6 = Qi[5];
            double q7 = Qi[6];

        g1 = q1;g2 = q2; g3 = q3; g4 = q4;g5 = q5;g6 = q6;g7 = q7;
}
```

C.7. CODIGO COMPLETO DEL RK4

```
MDI_model(Q,QDot,QDD,Torq);

pp(H,QDot,k1);

MDD_model(Q,QDot,QDD,Torq);
pp(H,QDD,l1);

sumarPorHK2(QDots,QDot,l1,1/2);
pp(H,QDots,k2);

sumarPorHK2(Qs,Q,k1,1/2);
MDD_model(Qs,QDots,QDD,Torq);
pp(H,QDD,l2);

sumarPorHK2(QDots,QDot,l2,1);
pp(H,QDots,k3);

sumarPorHK2(Qs,Q,k2,1);
MDD_model(Qs,QDots,QDD,Torq);
pp(H,QDD,l3);

sumarPorHK2(QDots,QDot,l3,1);
pp(H,QDots,k4);

sumarPorHK2(Qs,Q,k3,1);
MDD_model(Qs,QDots,QDD,Torq);
pp(H,QDD,l4);

QDotp[3] = QDot[3];
Qrefp[3] = Qref[3];

QDotp[4] = QDot[4];
Qrefp[4] = Qref[4];

for(i = 0; i<7; i++)
{
             Q[i]    = Q[i]  + (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])/6;
            QDot[i] = QDot[i] + (l1[i] + 2*l2[i] + 2*l3[i] + l4[i])/6;
}
```