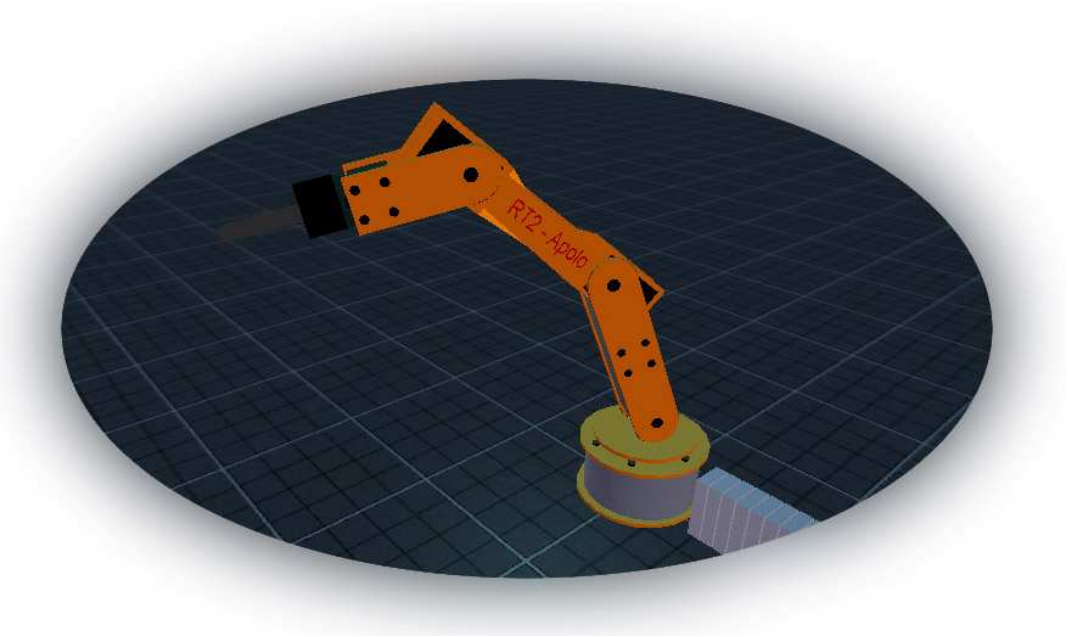
Tesis Final Robótica



Sistema de Control, Simulación y Programación RT2-Apolo.

Índice:

1) Introducción:

Desarrollo de Sistema Embedded de Control para el robot RT2-Apolo:

En vistas de la tarea a desarrollar se planteó como objetivo desarrollar un entorno de programación sencillo y amigable para realizar el control del brazo articulado de 6 grados de libertad. Para lograr mayor capacidad de prestaciones se incorporó un Simulador 3D para realizar la prueba de las secuencias de programación antes de ser enviadas por puerto serie al módulo FPGA-VHDL que controla el hard del robot.
La resolución de los cálculos de cinemática inversa y directa se realiza a través de este sistema de control, así como también la generación de trayectorias, ya sean sincrónicas, asincrónicas o sincronismo total punto a punto. El sistema a su vez, realiza el control de límites y errores, evitando que le sea enviada una posición o movimiento inválido que puedan afectar el sistema físico vinculado. Se ha hecho mucho hincapié y énfasis en este punto, ya que es una tarea vital del sistema de control garantizar la seguridad del mismo.
Para su desarrollo se analizaron diversas librerías y entornos de simulación existentes en el mercado. Se optó por desarrollar la aplicación mediante el uso del Framework Microsoft Robotics Studio debido a su facilidad para ser acoplado a un desarrollo en .Net (C#) y poder hacer uso de entorno de simulación 3D que cuenta con librerías físicas AGEIA para dotar de realismo, por ejemplo, una simulación de pick and place, donde el brazo tiene una masa determinada y un torque máximo a realizar. El producto un peso específico y coeficientes de rozamiento entre las tenazas y el objeto. Esto se resuelve de manera sencilla mediante la utilización del entorno AGEIA.

Microsoft Robotics:

Es una plataforma de desarrollo para la creación de aplicaciones para robots de forma sencilla, utilizando las tecnologías de .Net. Es aplicable a un gran número de dispositivos, siendo extensible y escalable, dada su orientación a servicios y la posibilidad de controlar tanto desde un aplicativo de escritorio como uno web. Su arquitectura posee una infraestructura de servicios y concurrencia, todo con una interface unificada. Posee además un entorno virtual para pruebas y un lenguaje visual propio para el desarrollo. Cuenta desde su misma instalación, con una gran variedad de sensores (webcam, sonar ,etc.) y actuadores (motor, actuador ,etc.), pero a través de terceros o desarrollos propios, pueden incorporarse infinidad de nuevos módulos para la creación de nuestro robot.

Componentes del Microsoft Robotics Developer Studio:

Microsoft Robotics Developer Studio SDK consiste en un numero determinado de componentes. El CCR (Concurrency and Coordination Runtime) y el DSS (Decentralized Software Services) manejan el entorno de ejecución de las aplicaciones. Ambos manejan las librerías, por lo tanto los servicios creados con el MRDS que se ejecuten bajo su entorno de ejecución operaran con código gestionado, posibilitando un marco de seguridad mayor. Luego podemos encontrar el VSE (Visual Simulation Enviroment) que es un poderoso simulador 3D que cuenta con simulación física, ideal para probar nuevos bocetos o prototipos de algoritmos de ejecución. Otra herramienta del paquete es el VPL (Visual Programming Language) que es un entorno de programación grafica que puede ser usado para la implementacion de servicios.

Concurrency and Coordination Runtime (CCR)

El CCR es una librería gestionada que provee las clases y los métodos para manejar la concurrencia, coordinación y el manejo de errores. CCR nos permite escribir segmentos de código que operen independientemente. Los comunica entre si, cuando sea necesario, mediante un sistema de pase de mensajes. Cuando un mensaje es recibido, este es depositado en una cola, llamada port, hasta que sea procesada por el receptor.

Cada segmento de código puede ejecutarse de manera concurrente y asincrónicamente sin necesidad de sincronizarlos puesto que para ese fin se encuentran las colas de mensajes. Sin embargo, cuando es necesario esperar hasta que 2 o más operaciones hayan sido completadas, la librería CCR provee las construcciones necesarias.

El CCR permite determinar que segmento de código es actualmente ejecutado mediante la utilización de dispatchers. El número total de threads se asocia al número de procesadores independientes del sistema y los segmentos de código comienzan a ejecutarse cuando se libera un threads y puede realizar otra tarea.

Uno de los elementos en la programación asincrónica que suele traer confusión es el manejo de errores. Es posible usar rutinas de manejo de excepciones para distinguir errores aislados en un único segmento de código, pero el dato que causo el error fue pasado al segmento a través de un mensaje que llego de otro segmento de ejecución por lo tanto el manejador de excepciones del error no tiene manera de saber donde se encuentra la operación original que provoco el error o donde el error debe ser reportado.

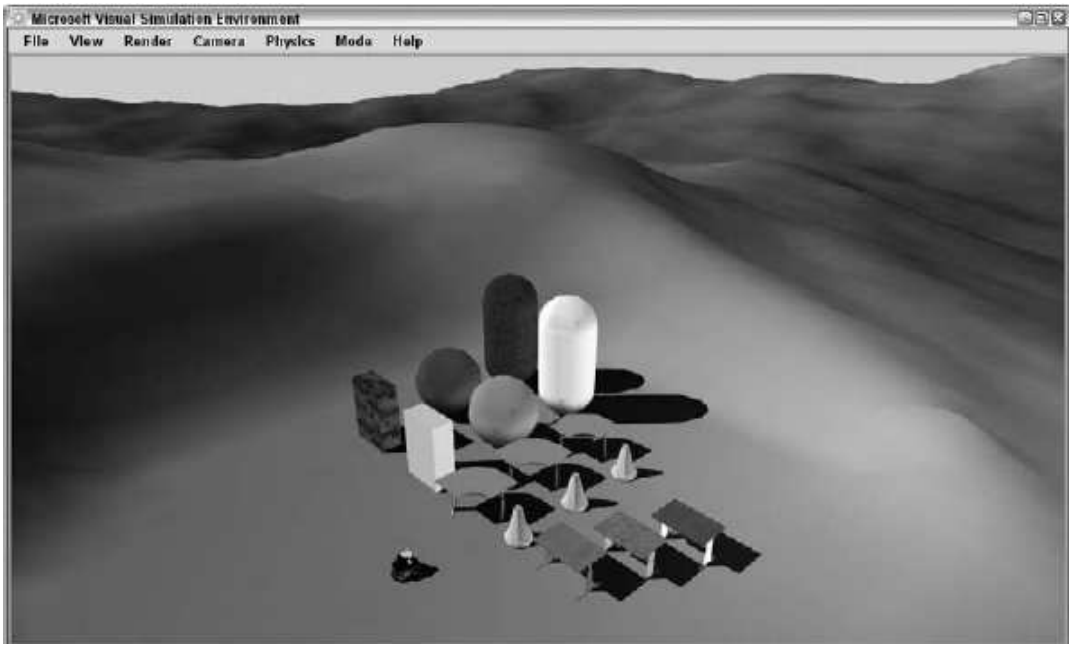
Para resolver esta dificultad, la CCR incorporo un mecanismo denominado "causalidad". Una causalidad mantiene una referencia a un puerto que se utiliza para reportar errores. Cuando se pasan mensajes entre procesos, a estos mensajes se les asocia una causalidad que sigue al mensaje durante su recorrido, si en algún momento, se produce un error, el mensaje se redirige al puerto de errores con su correspondiente causalidad. De esta manera los errores son enviados de vuelta al segmento de código que inicio la operación, a la raíz del problema.

Decentralized Software Services (DSS)

La CCR permite que los segmentos de código pasen mensajes y corran en paralelo en un mismo procesador mediante la utilización y administración de threads. La librería DSS extiende este concepto a varios procesadores e inclusive entre varias maquinas. Una aplicación desarrollada con DSS consiste en múltiples e independientes servicios que se ejecutan en paralelo. Cada servicio tiene un estado asociado y ciertos tipos de mensajes que recibe llamados operaciones. Cuando un servicio recibe un mensaje, este cambia su estado, procesa y luego manda mensajes adicionales y notificaciones a otros servicios.
Para averiguar el estado de un servicio basta con enviarle un mensaje de GET o mediante la utilización del sistema embebido web del MRSD. Los servicios pueden subscribirse para ser notificados de los cambios o eventos que ocurran en otros servicios que forman parte del mismo sistema. También se pueden asociar servicios de manera que solo puedan pasarse información entre ellos.

Entorno de simulación visual:

El entorno de simulación 3D trae incorporada librerías de simulación física para facilitar la prueba de prototipos.



El entorno de simulación de MSRS está compuesto de varios módulos:
Simulation Engine Service: responsable del progreso del tiempo en el motor físico de la simulación.

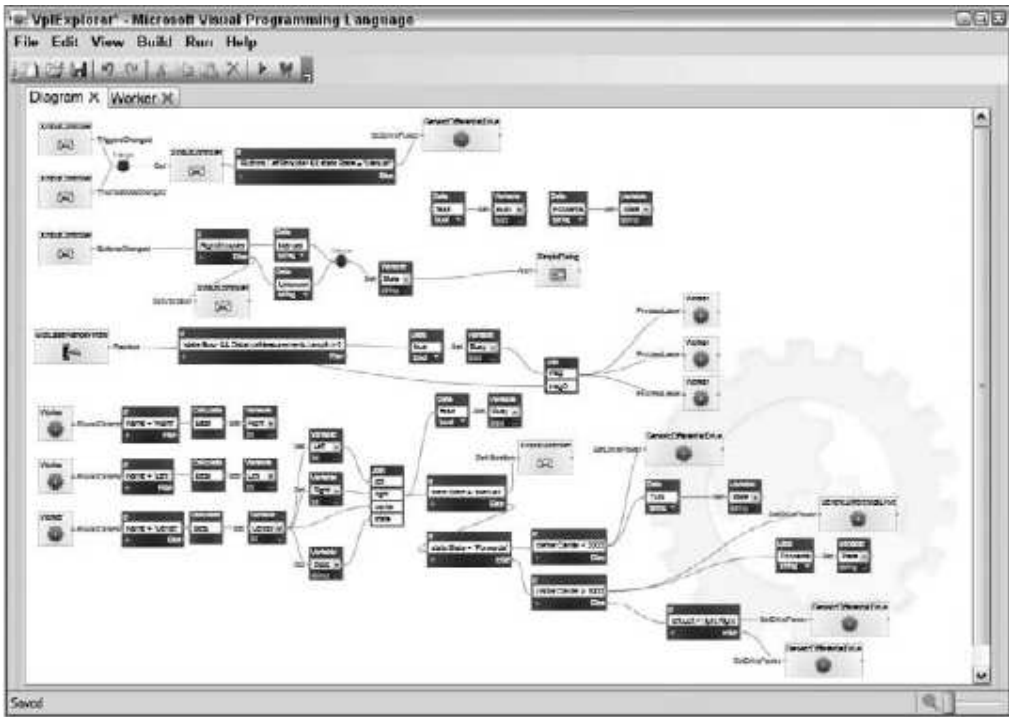
Managed Physics Engine Wrapper: abstrae al programador del nivel bajo del motor físico.

Native Physics Engine Library: permite la aceleración del hardware a través de AGEIA PhysX Technology.

Entities: representa el hardware y los objetos físicos en el mundo simulado. Un gran número de entidades están definidas en MSRS y permiten al usuario crear rápidamente un entorno de simulación.

Lenguaje de programación visual (VPL)

Una aplicación desarrollada con MRDS consiste en la ejecución de uno o varios servicios. Algunos de estos son servicios de bajo nivel que funcionan de interface directa con el hardware del robot o del prototipo. Otros servicios pueden ser de alto nivel, por ejemplo: servicios de simulación que están conectados directamente al motor de simulación. Usualmente uno o más servicios de alto nivel controlan el comportamiento de uno o más robots. Estos servicios de alto nivel usualmente se conectan con otros servicios de más bajo nivel y se los llama "servicios de orquesta" o de coordinación. Es decir, se encargan de gestionar la utilización de otros servicios de mas bajo nivel creados previamente. Estos servicios pueden ser programados mediante el VPL de manera grafica, vinculando servicios y condiciones previamente creadas.



Cada bloque representa un servicio, un cálculo, una condición o un diagrama anidado y son llamados "actividades". Las líneas que conectan los bloques representan el flujo de mensajes entre los procesos. Una vez conectados los bloques, el proyecto se puede

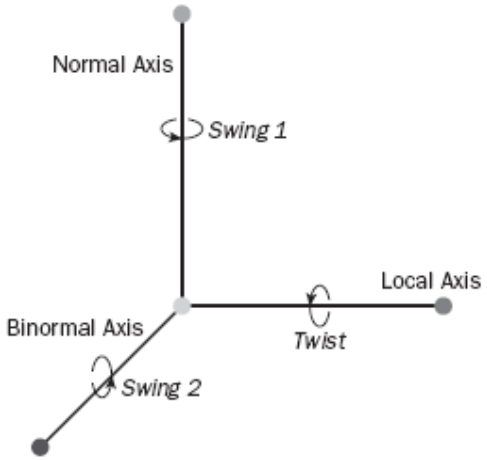
ejecutar y subsanar los errores que aparezcan. Además, se podrá generar código del proyecto realizado. Este código se genera en C # y puede servir como punto de partida para crear aplicaciones más complejas.

2- Realización del modelo físico del robot para ser simulado:

El concepto radica en la creación de entidades concretas denominadas Joints que vinculan 2 segmentos entre sí. A estos objetos que permiten el movimiento se les debe especificar la posición en la que aparecerán en el plano, así como también sus grados de libertad y límites de movimiento para poder crear así las distintas partes que constituyen el brazo articulado.

La Clase Joint

Un Joint es un objeto que se utiliza para unir entre si 2 entidades físicas. Tienen hasta 6 grados de libertad (3 angulares y 3 lineares), con lo cual pueden configurarse de varias maneras según la necesidad del problema. El Joint mas simple es el de revolución, que sólo tiene 1 grado de libertad angular habilitado y se comporta de manera similar a una bisagra. Un Joint de un solo grado de libertad lineal se comporta como un resorte de suspensión, solo se le permite el movimiento en una sola dirección.
La clase Joint se define en el espacio de Microsoft.Robotics.PhysicalModel, que esta implementada en la DLL RoboticsCommon. Es importante destacar que la clase Joint esta disponible para ser utilizada por todos los servicios, no tan solo los servicios de simulación. El método más importante de esta clase es el JointProperties que es donde se especifica el comportamiento del Joint cuando se inserta en el simulador físico.
El sistema de coordinadas empleado se rige bajo la regla de la mano derecha, es decir que las rotaciones positivas alrededor de cada eje son en la dirección de los dedos cuando tenemos el pulgar apoyado sobre el eje.



Propiedades de los Joints:

Las propiedades definen como se comporta el Joint y como se lo vincula a otras entidades. Un Joint debe tener una referencia a una instancia de la clase

JointAngularProperties que habilita 1 o más grados de libertad angular, así como también una referencia a un objeto de la clase JointLinearProperties que habilita los grados de libertad lineales. Los vínculos con otras entidades se deben explicitar en el campo de EntityJointConnector que es un array de 2 valores.

Clase JointAngularProperties:

Miembro	Descripción
TwistMode	Define el modo para el Twist DOF, si esta bloqueado, limitado o libre. Si esta bloqueado, el Joint no tiene permitido moverse alrededor del eje local. Si esta limitado, la rotación del Joint alrededor del eje local se ve determinada por los límites máximos y mínimos especificados en las variables UpperTwistLimit y LowerTwistLimit. Si está libre, el Joint puede moverse sin limitaciones alrededor del eje local.
Swing1Mode	Define el modo para el Swing1 DOF. Si es limitado se rige bajo el límite explicitado en Swing1Limit
Swing2Mode	Define el modo para el Swing2 DOF. Si es limitado se rige bajo el límite explicitado en Swing2Limit
UpperTwistLimit LowerTwistLimit	JointLimitProperties, limitan el movimiento del Joint alrededor del eje local.
Swing1Limit	JointLimitProperties, limita el movimiento del Joint alrededor del eje normal.
Swing2Limit	JointLimitProperties, limitan el movimiento del Joint alrededor del eje binormal.
TwistDrive	JointDriveProperties que define como se desplaza alrededor del eje local
SwingDrive	JointDriveProperties que define como se desplaza alrededor del eje swing
SlerpDrive	JointDriveProperties. Interpolación esférica lineal que define una manera especial de desplazar los 3 DOF del joint
GearRatio	Si este miembro es distinto de cero, la velocidad angular de la segunda entidad se traslada a la primera multiplicada por el GearRatio. Es el factor de conversión.
DriveTargetOrientation	La orientación inicial del Joint
DriveTargetVelocity	La velocidad angular inicial del Joint

Clase JointLimitProperties:

Miembro	Descripción
LimitThreshold	Especifica los límites, ya sean angulares o lineales, del Joint. Los límites angulares se expresan en radianes
Restitution	Variable del tipo flota que especifica el nivel de balanceo del Joint cuando se mueve cerca de los valores lìmites

Spring	Define un límite tolerable. Si el SpringCoefficient es distinto de cero, el Joint puede desplazarse momentáneamente por encima de los límites especificados para luego volver al rango acordado. La variable DamperCoefficient especifica cuánto oscila el Joint cuando se lo está regresando hacia lo determinado por el LimitThreshold

Clase JointDriveProperties:

Miembro	Descripción
Mode	Especifica el modo de desplazamiento del DOF. Tiene 2 modos. Position: el Joint se desplaza a una posición Velocity: El Joint es llevado a una determinada velocidad angular.
Spring	Especifican los coeficientes utilizados cuando se debe regresar a una posición determinada el Joint.
ForceLimit	Especifica el máximo torque o fuerza que un Joint puede realizar para moverse a una determinada posición

Clase JointLinearProperties:

Miembro	Descripción
DriveTargetPosition, DriveTargetVelocity	La posición destino y velocidad para el grado de libertad lineal del Joint.
XMotionMode, YMotionMode, ZMotionMode	Los modos para cada grado de libertad. Bloqueado, limitado o libre. Si esta limitado, el movimiento del Joint sobre cada eje está limitado por el miembro MotionLimit. XMotionMode especifica el movimiento sobre el eje local. YMotionMode sobre el eje normal y ZMotionMode sobre el eje binormal.
XDrive, YDrive, ZDrive	Las características de desplazamiento del Joint a lo largo de los respectivos ejes.
MotionLimit	Un único miembro JointLimitProperties controla los 3 grados de libertad. La variable LimitThreshold del método MotionLimit especifica el máximo desplazamiento posible. Por ejemplo, si los 3 DOF lineales son limitados, este miembro define el radio de una esfera que contiene el movimiento de la primer entidad respecto de la segunda.

Clase EntityJointConnector:

Los Joints y las entidades se conectan en un punto y una orientación determinada especificada por un objeto de esta clase.

Miembro	Descripción
JointAxis	Vector que define el eje local. Generalmente es (1,0,0) pero puede ser cualquier vector.
JointNormal	Vector que define el eje normal. Generalmente es (0,1,0) pero puede ser cualquier vector que sea perpendicular al vector denominado theJointAxis. El eje binormal surge de la

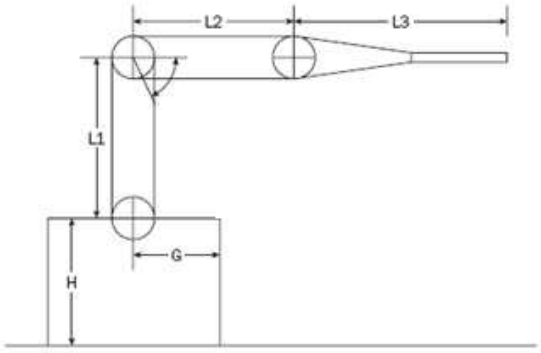
	realización interna del producto cruz entre el eje normal y el local
JointConnectPoint	Punto donde el Joint se conecta a la entidad. Se expresa en coordenadas relativas a la entidad, no en globales. Este punto no debe encontrarse en la superficie de la entidad, puede ser interno o externo.
Entity	Referencia a la entidad con la que está conectado.
EntityName	Nombre de la entidad referida por el campo Entity

Clase JointProperties:

Los Joints y las entidades se conectan en un punto y una orientación determinada especificada por un objeto de esta clase.

Miembro	Descripción
Connectors	Es un array de 2 objetos EntityJointConnectors que especifican el punto de conexión y la orientación para ambas entidades.
Angular	Referencia a un objeto JointAngularProperties que especifica las propiedades angulares del Joint. Si la referencia es nula, todos los grados de libertad angulares se consideran bloqueados.
Linear	Referencia a un objeto JointLinearProperties que especifica las propiedades lineales del Joint. Si la referencia es nula, todos los grados de libertad lineales se consideran bloqueados.
EnableCollisions	Variable del tipo bool que especifica si se desea chequear colisiones entre las entidades conectadas por este Joint
MaximumForce, MaximunTorque	Máximo torque o fuerza que puede ser aplicado al Joint. Si este límite se excede, el Joint se anula y deja de funcionar. Con 0 se garantiza un torque infinito.
Name	Nombre del Joint, debe ser único
Projection	El motor físico provee una forma para corregir errores de los Joints, proyectándolo a una configuración válida.

Modelo Físico de la entidad del brazo articulado de 6 DOF



Para la correcta simulación del modelo físico es necesario definir las dimensiones del brazo articulado. Estas se encuentran en el archivo RT2-Apolo.cs contenidas en la definición de la clase RT2-Articulado. Esta entidad se crea con el método SingleShapeEntity, que es la base del brazo. Cada uno de los otros segmentos es una entidad independiente que se vincula a otro segmento como hijo de esta. Esto facilita utilizar el constructor ParentJoin que vincula las entidades entre si.

```
static float PulgadasToMetros(float pulgadas) { return (float)(pulgadas * 0.0254); }
// Atributos fisicos del brazo
static float L1 = PulgadasToMetros (4.75f);
static float L2 = PulgadasToMetros (4.75f);
static float Grip = PulgadasToMetros (2.5f);
static float L3 = PulgadasToMetros (5.75f) - Grip;
static float L4 = 0.03f;
static float H = PulgadasToMetros (3f);
static float G = PulgadasToMetros (2f);
static float L1Radio = PulgadasToMetros (0.7f);
static float L2Radio = PulgadasToMetros (0.7f);
static float L3Radio = PulgadasToMetros (0.7f);
static float GripRadio = PulgadasToMetros (0.2f);
```

Además de las dimensiones, definimos los joints que utilizaremos para vincular los segmentos entre sí. La siguiente clase define las características de cada Joint, así como también algunas propiedades relativas al tiempo de ejecución, tales como la posición y velocidad inicial.

```
// Descripcion de los Joints
class JointDesc
{
	public string Name;
	public float Min; // Angulo minimo permitido
	public float Max; // Angulo maximo permitido
	public JoinFisico Joint; // Vinculo Fisico
	public JoinFisico Joint2; // Auxiliar para la tenaza
	public float Destino; // Posicion destino
	public float Actual; // Posicion actual
	public float Velocidad;
	public JointDesc(string name, float min, float max)
	{
		Name = name; Min = min; Max = max;
		Joint = null;
		Joint2 = null;
		Actual = Destino = 0;
		Velocidad = 30;
	}
	// Retorna True si la posicion destino se encuentra dentro de los limites del Joint
	public bool ValidDestino(float Destino)
	{
		return ((Destino > = Min) & & (Destino < = Max));
	}
	// Retorna True si todavia no se alcanzo la posicion destino
	public bool EnMovimiento(float epsilon)
	{
		if (Joint == null) return false;
		return (Math.Abs(Destino - Actual) > epsilon);
	}
	// Actualiza la posicion actual en base a la velocidad especificada
	public void UpdateActual(double time)
	{
		float delta = (float)(time * Velocidad);
		if (Destino > Actual)
			Actual = Math.Min(Actual + delta, Destino);
		else
			Actual = Math.Max(Actual - delta, Destino);
	} }
```

Las variables de inicialización de cada Joint son el nombre y los límites máximos y mínimos que se permiten en su recorrido. Por ejemplo, para el caso de las tenazas, se usan Joints Lineales y las variables min y max representan distancias de apertura.

Los Joints utilizados para el RT2-Apolo se definen en un array de descriptores de la siguiente manera:

```
// Array de Joints de RT2-Apolo
JointDesc[] _joints = new JointDesc[]
{
        new JointDesc("Base", -180, 180),
        new JointDesc("L1", -90, 90),
        new JointDesc("L2", -65, 115),
        new JointDesc("L3", -90, 90),
        new JointDesc("L4", -90, 90),
        new JointDesc("Tenaza", 0, PulgadasToMetros (2))
};
```

Luego inicializamos la base del robot mediante el siguiente código:

```
// Forma Fisica de la base del RT2
float baseAltura = H - L1Radius - 0.001f;
State.Name = name;
State.Pose.Position = position;
State.Pose.Orientation = new Quaternion(0, 0, 0, 1);
State.Assets.Mesh = "RT2_Base.obj";
MeshTranslation = new Vector3(0, 0.026f, 0);
// Construimos la base
BoxShape = new BoxShape(new BoxShapeProperties(
"Base",
150, // masa del cuerpo
new Pose(new Vector3(0, baseHeight / 2, 0), new Quaternion(0, 0, 0, 1)),
new Vector3(G * 2, baseHeight, G * 2)));
```

En este punto es importante notar que el motor físico AGEIA necesita que le pasen como parámetros además de las dimensiones, la masa del objeto creado (150 kg). La pose inicial para la base esta determinada en su altura/2 por encima del punto de origen, que es la referencia para todas las entidades que agregamos al simulador

Para inicializar los 3 segmentos que constituyen el brazo articulado se crean distintas SingleShapeEntities que los contienen. Luego se realizan las conexiones mediante el método ParentJoin. Creamos una entidad llamada L0Entidad que es de forma esférica de radio igual al ancho del 1er segmento del brazo que estará representado por L1Entidad. El ParentJoin de esta entidad controla la rotación de la base y se crea de la siguiente manera.

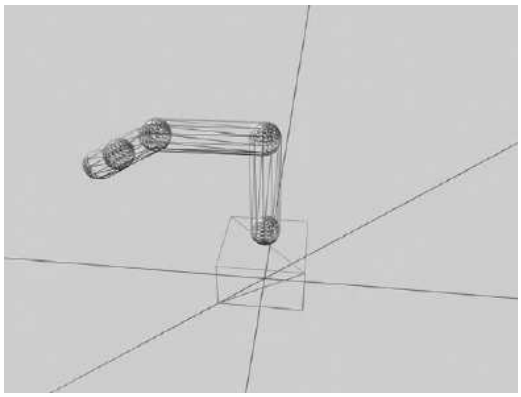
```
// Construye y posiciona L0. Base del RT2
SphereShape L0Esfera = new SphereShape(new SphereShapeProperties(
        "L0Esfera", 50, new Pose(new Vector3(0, 0, 0), new Quaternion(0, 0, 0, 1)),
        L1Radio));
SingleShapeSegmentEntity L0Entidad =
        new SingleShapeSegmentEntity(L0Esfera, posicion + new Vector3(0, H, 0));
        L0Entity.State.Pose.Orientation = new Quaternion(0, 0, 0, 1);
        L0Entity.State.Name = name + "_L0";
        L0Entity.State.Assets.Mesh = "RT2_L0.obj";
        L0Entity.MeshTranslation = new Vector3(0, -0.02f, 0);
JointAngularProperties L0Angular = new JointAngularProperties();
L0Angular.Swing1Mode = JointDOFMode.Free;
L0Angular.SwingDrive = new JointDriveProperties(JointDriveMode.Position,
new SpringProperties(50000000, 1000, 0), 100000000);
EntityJointConnector[] L0Connectors = new EntityJointConnector[2]
{
        new EntityJointConnector(L0Entidad,
        new Vector3(0,1,0), new Vector3(1,0,0), new Vector3(0, 0, 0)),
        new EntityJointConnector(this,
        new Vector3(0,1,0), new Vector3(1,0,0), new Vector3(0, H, 0))
};
L0Entidad.CustomJoint = new Joint();
L0Entidad.CustomJoint.State = new JointProperties(L0Angular, L0Connectors);
L0Entidad.CustomJoint.State.Name = "BaseJoint";
this.InsertEntityGlobal(L0Entidad);
```

L0Entidad esta formada por una simple esfera. Su centro está posicionado de manera tal de coincidir con su centro de rotación. En la definición del Joint se especifica el método Swing1 por lo tanto rotará libremente sobre el eje Y.
Luego conectamos el siguiente segmento encima de la base creada:

```
// Construimos el primer segmento adjunto a la base
CapsuleShape L1Capsula = new CapsuleShape(new CapsuleShapeProperties(
                "L1Capsula",
                2,
                new Pose(new Vector3(0, 0, 0), new Quaternion(0, 0, 0, 1)),
                L1Radius,
                L1));
SingleShapeSegmentEntity L1Entidad =
                new SingleShapeSegmentEntity(L1Capsula, position + new Vector3(0, H, 0));
L1Entidad.State.Pose.Orientation = new Quaternion(0, 0, 0, 1);
L1Entidad.State.Name = name + "_L1";
L1Entidad.State.Assets.Mesh = "RT2_L1.obj";
JointAngularProperties L1Angular = new JointAngularProperties();
L1Angular.TwistMode = JointDOFMode.Free;
L1Angular.TwistDrive = new JointDriveProperties(JointDriveMode.Position,
                new SpringProperties(50000000, 1000, 0), 100000000);
EntityJointConnector[] L1Connectors = new EntityJointConnector[2]
{
        new EntityJointConnector(L1Entidad,
        new Vector3(0,1,0), new Vector3(0,0,1), new Vector3(0, -L1/2, 0)),
        new EntityJointConnector(L0Entity,
        new Vector3(0,1,0), new Vector3(0,0,1), new Vector3(0, 0, 0))
};
L1Entidad.CustomJoint = new Joint();
L1Entidad.CustomJoint.State = new JointProperties(L1Angular, L1Connectors);
L1Entidad.CustomJoint.State.Name = "Hombro|-80|80|";
L0Entity.InsertEntityGlobal(L1Entidad);
```

Esta parte del código para colocar el segmento L1 es muy similar al mismo utilizado para construir la base excepto que en este caso no se utiliza una esfera sino que se modela mediante una capsula. El resto de los segmentos del brazo articulado se

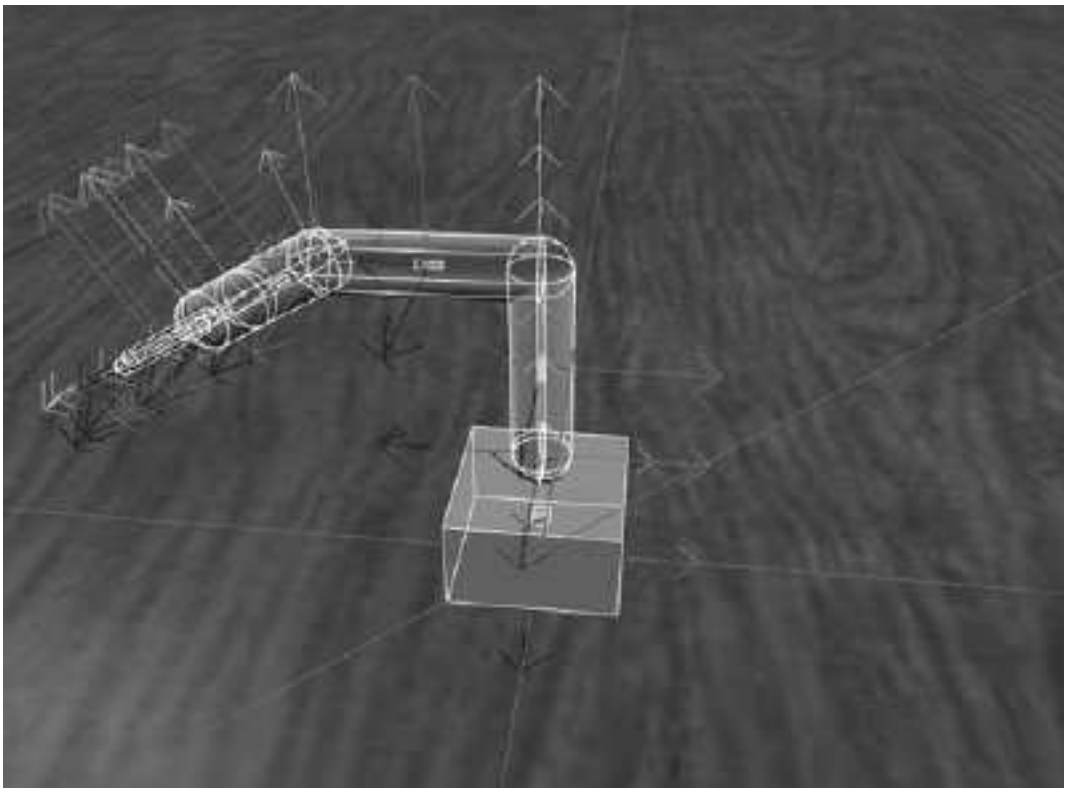
construye mediante un código similar y con lo expuesto hasta ahora uno obtiene un modelo similar al siguiente:



Resta modelar las tenazas, la punta de aplicación que se realiza como 2 capsulas que se mueven juntas usando Joints lineales. Aunque cada parte de la tenaza se vincula a L4Entidad (controla el ultimo segmento del brazo) mediante un Joint propio, la tenaza se maneja como si fuera un solo Joint. El siguiente código crea las tenazas del RT2.

```
// Crea y posiciona  TenazaIzq
CapsuleShape TenazaIzqCapsula = new CapsuleShape(new CapsuleShapeProperties(
                        “TenazaIzqCapsula”,
                        1f,
                        new Pose(new Vector3(0, 0, 0), new Quaternion(0, 0, 0, 1)),
                        GripRadio,
                        Grip));
TenazaIzqCapsule.State.DiffuseColor = new Vector4(0, 0, 0, 1);
TenazaIzqEntidad = new SingleShapeSegmentEntity(
                        TenazaIzqCapsule, position + new Vector3(0, H, 0));
// Posicion cerca de su valor final una vez que este conectado
TenazaIzqEntidad.Position = new xna.Vector3(-0.24f, 0.19f, 0.01f);
TenazaIzqEntidad.Rotation = new xna.Vector3(179.94f, -176.91f, 89.67f);
TenazaIzqEntidad.State.Name = name + “_TenazaIzq”;
// Uso un Joint Lineal para la tenaza
JointLinearProperties TenazaIzqLinear = new JointLinearProperties();
TenazaIzqLinear.XMotionMode = JointDOFMode.Free;
TenazaIzqLinear.XDrive = new JointDriveProperties(JointDriveMode.Position,
                        new SpringProperties(5000000000, 1000, 0), 100000000);
EntityJointConnector[] TenazaIzqConnectors = new EntityJointConnector[2]
{
                        new EntityJointConnector(TenazaIzqEntidad,
                        new Vector3(1,0,0), new Vector3(0,0,1), new Vector3(0, -Grip/2, 0)),
                        new EntityJointConnector(L4Entidad,
                        new Vector3(1,0,0), new Vector3(0,0,1), new Vector3(0, L4/2, GripRadio))
};
TenazaIzqEntidad.CustomJoint = new Joint();
TenazaIzqEntidad.CustomJoint.State =
new JointProperties(TenazaIzqLinear, TenazaIzqConnectors);
TenazaIzqEntidad.CustomJoint.State.Name = “TenazaIzqJoint|-0.0254|0|”;
L4Entity.InsertEntityGlobal(TenazaIzqEntidad);
```

Con la punta de aplicación ya agregadas finalmente obtenemos el modelo físico del RT2-Apolo:



Finalmente agregamos una cámara justo por encima de L4Entidad para tener una vista completa del robot en el entorno de simulación:

```
// Agregamos una camara sobre el RT2
AttachedCameraEntity gripCam = new AttachedCameraEntity();
gripCam.State.Name = “RT2 Cam”;
// movemos la camara sobre el L4
gripCam.State.Pose = new Pose(new Vector3(0.05f, -0.01f, 0),
        Quaternion.FromAxisAngle(0, 1, 0, (float)(Math.PI / 2)) *
        Quaternion.FromAxisAngle(1, 0, 0, (float)(Math.PI / 3)));
// ajustamos el plano
gripCam.Near = 0.01f;
// La camara se situa con posiciones relativas a L4
InsertEntityGlobal
L4Entity.InsertEntity(gripCam);
```

La posición y la orientación de la cámara se fijan con posiciones relativas a L4Entidad. En la variable gripCam.Near se ajusta el plano cercano a 1 cm, esto es importante porque todos los objetos que se encuentren a menos de 1 cm de la cámara no se mostrarán en pantalla.

3 - Movimiento del RT2

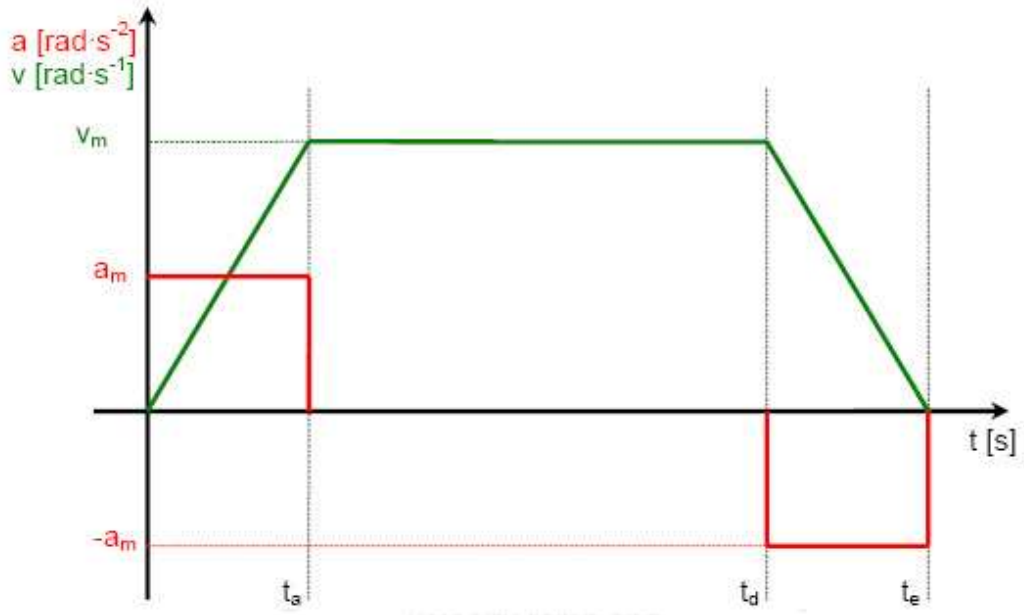
Marco Teórico:

Notación:

t_a .= Tiempo de aceleración
$\widetilde{t}_a$.= Tiempo de aceleración del intervalo
$t_{a,\max}$.= Tiempo de aceleración máximo de todos los ejes
t_d .= Tiempo de inicio de desaceleración
$\widetilde{t}_d$.= Tiempo de inicio de desaceleración del intervalo
$t_{d,\max}$.= Tiempo de desaceleración máximo de todos los ejes
t_e .= Tiempo total del movimiento, de finalización
$\widetilde{t}_e$.= Tiempo de finalización de un intervalo
$t_{e,\max}$.= Tiempo total máximo de todos los ejes
a_m .= Aceleración máxima
$\widetilde{a}_m$.= Aceleración máxima predeterminada
v_m .= Velocidad de rampa máxima
$\widetilde{v}_m$.= Velocidad de rampa máxima predeterminada
$\widetilde{x}_e$.= Distancia predeterminada de movimiento
H .= Período de tiempo de sample
N .= Cantidad de ejes
n .= Paso de tiempo discreto
$X_i[n]$.= Tiempo Discreto

Rampa General del movimiento PTP:

La rampa del movimiento punto a punto (PTP) cuenta con 3 etapas, aceleración, constante y desaceleración. La etapa de aceleración arranca en el tiempo $t = 0$ y termina en el tiempo t_a, la etapa de velocidad constante sigue a continuación de la etapa de aceleración hasta llegar al tiempo t_d, donde comienza la tercer y última etapa, desaceleración. Finalizando el movimiento en el tiempo t_e.
Mientras transcurre la etapa de aceleración, la velocidad se incrementa desde 0 hasta alcanzar v_m. Tanto en $t = 0$ como en t_e, la velocidad debe ser nula. Debido a que $|a_m| = |-a_m|$, las duraciones de las etapas de aceleración y desaceleración, serán idénticas.

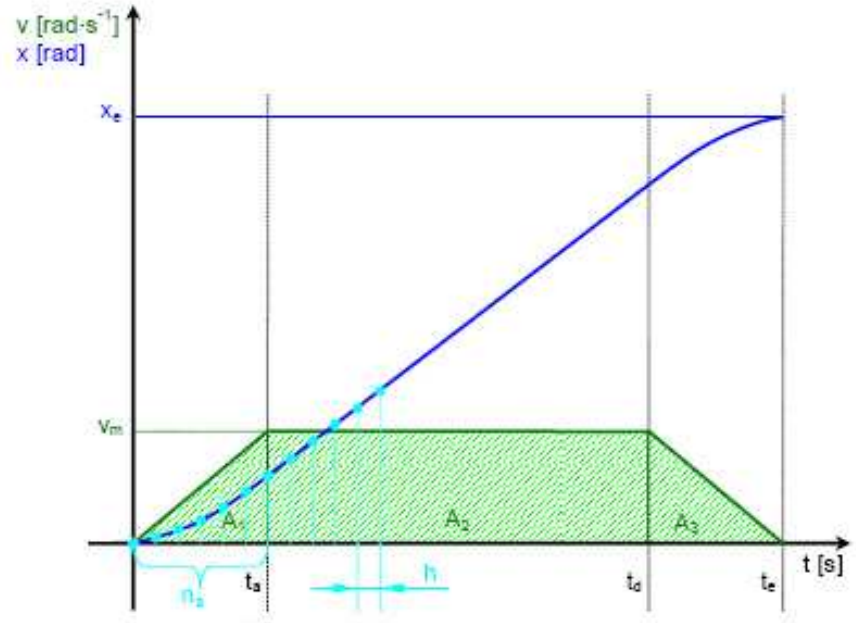


Presumiento una aceleración constante a_m, la velocidad alcanza su máxim nivel en v_m en el lapso de tiempo t_a:

$$t_a = \frac{v_m}{a_m}$$

Como mencionamos anteriormente, los tiempos de aceleración y desaceleración son igualmente largos, por lo tanto la etapa de desaceleración comienza en:

$$t_d = t_e - t_a$$



El área de la figura sombreada corresponde a:

$$A_1 = \frac{v_m}{2} \cdot t_a \qquad A_2 = v_m \cdot (t_a - t_d) \qquad A_1 = \frac{v_m}{2} \cdot (t_a - t_d)$$

$$x_e = A_1 + A_2 + A_3 = \frac{v_m}{2} \cdot (t_a + t_d + t_e)$$

Debido a la simetría $t_d = t_e - t_a$ podemos establecer que:

$$x_e = \frac{v_m}{2} \cdot (t_e - t_a)$$

Despejando el tiempo de duración total del movimiento:

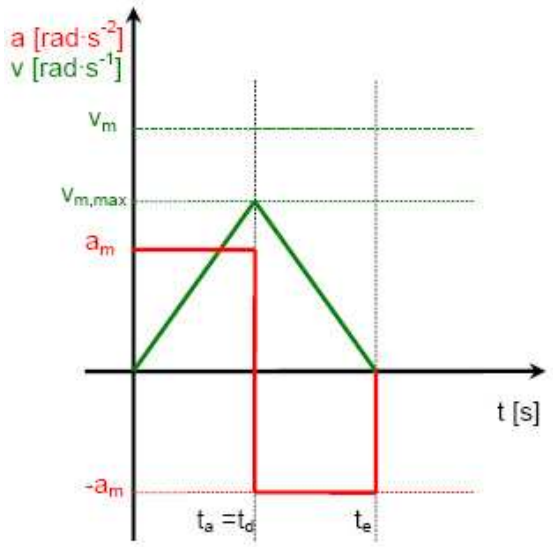
$$t_e = \frac{x_e}{v_m} \cdot + t_a = \frac{x_e}{v_m} + \frac{v_m}{a_m}$$

Ahora ya estamos en condiciones de calcular la velocidad en cualquier intervalo de tiempo:

$$x(t) = \begin{cases} \frac{1}{2} a_m \cdot t^2, & 0 \le t < t_a \\ v_m \cdot t - \frac{1}{2} \cdot \frac{v_m^2}{a_m}, & t_a \le t < t_d \\ v_m \cdot (t_e - t_a) - \frac{a_m}{2} \cdot (t_e - t)^2, & t_d \le t \le t_e \end{cases}$$

Como último, debemos considerar el caso en el que la distancia x_e no es suficientemente grande para permitir que la velocidad crezca hasta v_m. Por lo tanto la rampa de velocidad se convierte en un triángulo. La etapa de aceleración dura la mitad de la duración total del movimiento debido a que los tiempos de aceleración y desaceleración son idénticos. Aplicando esta condicion $t_e = 2.t_a$, obtenemos la siguiente relación:

$$x_e = \cdot t_a \cdot v_m$$



Este análisis nos lleva a la siguiente relación:

$$v_{m,\max} = \sqrt{a_m \cdot x_e}$$

El caso del triángulo no se aplica si me excedo en la velocidad máxima permitida para el Joint, en tal caso:

$$v_{m,\lim itada} = \min(v_m, v_{m,\max})$$

Discretización del tiempo:

Para terminar el servicio generador de trayectorias es necesario determinar los puntos por los que pasará el brazo articulado a medida que realiza el movimiento. El servicio generador de trayectorias deberá mandar periodicamente los valores con los movimientos al simulador y al servicio de conexión con el hard para realizar el correcto movimiento, en un tiempo h (tiempo de sample predeterminado). Como el movimiento punto a punto total esta dividido en n_e waypoints, se obtiene el tiempo discreto $t = n.h$, entonces podemos calcular el tiempo discreto del movimiento:

$$X[n] = x(t)\,|_{t \to n \cdot h,\ n \in N^+}$$

La cantidad de pasos discretos n_a, n_d y n_e multiplicados por el período de tiempo h, deben coincidir con t_a, t_d y t_e

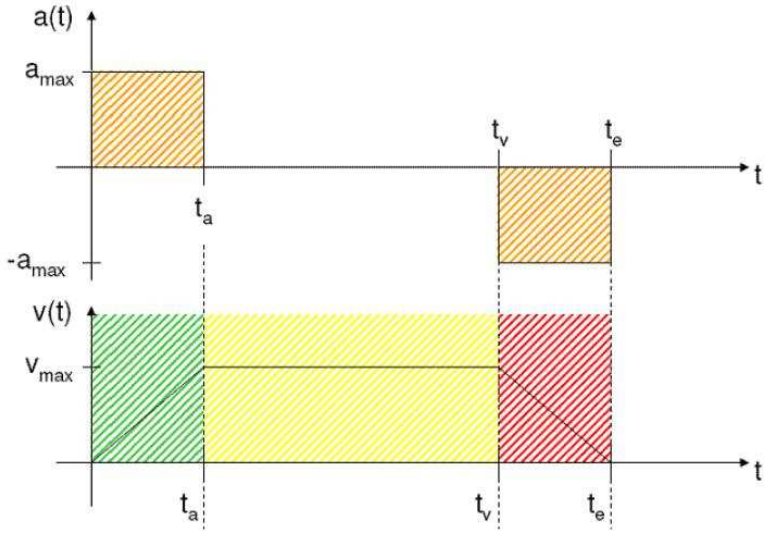
$$t_a = n_a \cdot h, \quad n_a \in N$$
$$t_d = n_d \cdot h, \quad n_d \in N^+$$
$$t_e = n_e \cdot h, \quad n_e \in N^+$$

De esta manera podemos determinar los waypoints totales del generador de trayectorias:

$$X[n]=\begin{cases}\frac{1}{2}a_m\cdot h^2\cdot n^2, & 1\le n\le n_a\\ v_m\cdot h\cdot n-\frac{1}{2}\cdot\frac{v_m^2}{a_m}, & n_a<n\le n_e-n_a\\ v_m\cdot(t_e-t_a)-\frac{a_m}{2}\cdot(t_e-h\cdot n)^2, & n_e-n_a<n\le n_e\end{cases}$$

Movimiento de un Eje:

Para alcanzar la posición destino, el motor del eje determinado necesita acelerar a cierta velocidad para luego desacelerar antes de alcanzar el objetivo. Para planear estos tiempos correctamente se debe calcular el perfil de velocidades para cada eje. La siguiente figura muestra el perfil de velocidades de un eje – la curva superior muestra la aceleración de entrada del motor y la inferior muestra la velocidad resultante.



Área Verde: Fase de aceleración
Área amarilla: fase de velocidad constante
Área roja: fase de desaceleración.

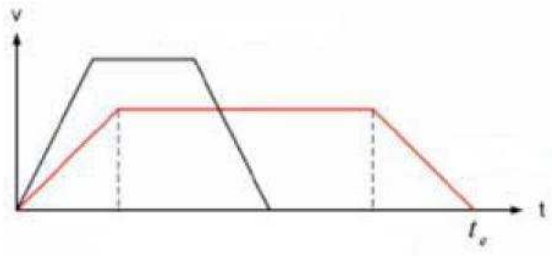
El perfil de velocidad se usa para mandar valores precisos a los motores que controlan los ejes a determinados intervalos de tiempo.

Movimiento de múltiples ejes:

Para alcanzar la posición final del brazo articulado, generalmente varios ejes van a tener que moverse. Para cada uno de los ejes involucrados se calcula su respectivo perfil de velocidad y para el movimiento en conjunto estos perfiles pueden ser coordinados de 3 maneras diferentes.

Movimiento Asincrónico punto a punto (Async PTP):

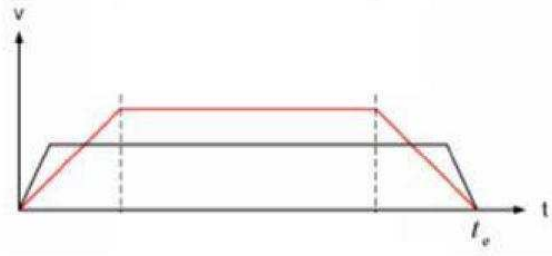
En este tipo de movimiento, cada motor se mueve tan rápido como le sea posible hasta alcanzar el ángulo determinado. Debido a esto, algunos ejes alcanzaran la posición objetivo más rápido que otros. La posición final se considera alcanzada cuando cada uno de los ejes intervinientes llego a su valor objetivo. El perfil de velocidades de 2 ejes, por ejemplo, tiene la siguiente forma;



En la figura se observa que los perfiles alcanzan distintas alturas, por lo tanto se considera que los motores que manejan los distintos ejes pueden tener capacidades diferentes.

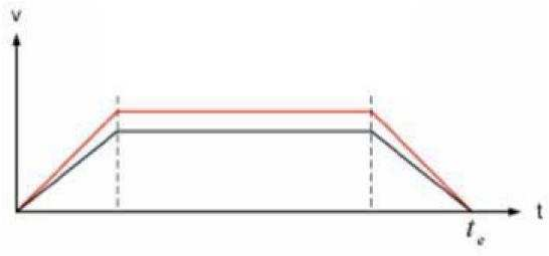
Movimiento Sincrónico punto a punto (Sync PTP):

Debido a que la posición destino no se alcanza hasta que cada uno de los motores de los ejes intervinientes no haya alcanzado su ángulo destino, alguno de los ejes será el que tarde y consuma más tiempo en llegar. A este eje se lo denomina “eje líder”. Para minimizar el desgaste y garantizar su vida útil, todos los motores, exceptuando el correspondiente al eje líder, deberán desplazarse a una velocidad inferior a la máxima permitida. De esta manera el tiempo necesario para completar el recorrido se encuentra sincronizado. El perfil de movimiento para el eje mas lento se recalcula, de esta manera todos los ejes llegan a destino al mismo tiempo.



Movimiento Sincrónico punto a punto Full (Full Sync PTP):

El desgaste de los motores se previene aún más si no sólo coordinamos el tiempo de llegada al objetivo, sino también sincronizamos los tiempos de aceleración y desaceleración de los mismos. De manera que durante un instante determinado todos los ejes se encuentran en acelerando, luego terminan todos a la vez y pasan a una velocidad constante propia de cada una hasta que todos comienzan a desacelerar.



4 - Programación de los distintos generadores de trayectorias

Movimiento Asincrónico (Método asyncPTP)

Los ejes están todos independientes entre sí, se calculan los waypoints $X_i[n]$ para cada eje $i=1...6$. El cálculo se basa en los valores pasados como parámetros $\hat{v}_{m,i}$, $\hat{a}_{m,i}$ y $\hat{x}_{e,i}$.

Primero determinamos la amplitud máxima para la rampa de velocidad. Debemos verificar que la distancia $\hat{x}_{e,i}$ sea suficiente para permitir que la velocidad suba hasta $\hat{v}_{m,i}$, o no. Si no se alcanza corremos la velocidad de forma triángulo, es decir, acelero hasta que de repente comienzo a desacelerar. Debemos limitar $\hat{v}_{m,i}$ hasta llegar a $\hat{v}_{m,\max,i}$. Luego evaluamos la siguiente ecuación:

$$\hat{v}_{m,i}=\min(\hat{v}_{m,i},\sqrt{\hat{x}_{ei}\cdot\hat{a}_{mi}})$$

```
if (_estado.MaxJointVel > Math.Sqrt(Math.Abs(cambios[i])*_estado.MaxJointAcel))
            velocidad[i] = (float) Math.Sqrt(Math.Abs(cambios[i]) *
            _estado.MaxJointAcel);
else
            velocidad[i] = _estado.MaxJointVel;
```

Luego se calcula el tiempo de aceleración $t_{a,i}$, el tiempo de inicio de desaceleración $t_{d,i}$ y el tiempo total de movimiento $t_{e,i}$.

Tiempo de aceleración $$t_{a,i}=\frac{v_{m,i}}{\hat{a}_{m,i}}$$

Tiempo total movimiento $$t_{e,i} = \frac{\hat{x}_{e,i}}{v_{m,i}} + t_{a,i}$$

Tiempo inicio desaceleración $$t_{d,i} = t_{e,i} - t_{a,i}$$

Código:

```
        ta[i]= velocidad / _estado.MaxJointAcel;
        te[i]= (Math.Abs(cambios[i])/ velcoidad[i])+ ta[i];
        td[i]= te[i] - ta[i];
```

Luego debemos determinar los waypoints, es decir los puntos por los que pasará el brazo articulado en su recorrido. El largo de la fase de aceleración $t_{a,i}$, y todo el proceso de movimiento $t_{e,i}$ son divididos en pequeños instantes de tiempos (timeslice) con un período h.

```
        timeslice_e[i]= (int)Math.Round(te[i] / SAMPLERATE * 1000f);
        timeslice_a[i]= (int)Math.Round(ta[i] / SAMPLERATE * 1000f);
```

Finalmente calculamos todos los waypoints y queda generada la trayectoria:

$$X_i[n_i] = \begin{cases} \frac{1}{2}\hat{a}_{m,i} \cdot h^2 \cdot n_i^2, & 1 \le n_i \le n_{a,i} \\ v_{m,i} \cdot h \cdot n_i - \frac{1}{2} \cdot \frac{v_{m,i}^2}{\hat{a}_{m,i}}, & n_{a,i} < n_i \le n_{e,i} - n_{a,i} \\ v_{m,i} \cdot (t_{e,i} - t_{a,i}) - \frac{\hat{a}_{m,i}}{2} \cdot (t_{e,i} - h \cdot n_i)^2, & n_{e,i} - n_{a,i} < n_i \le n_{e,i} \end{cases}$$

```
        if(j <= timeslices_a[i])
            ActualAnguloCambio = 1f /2f * _estado.MaxJointAcel *
            tiempo * tiempo;
        else if ( j<=timeslices_e[if] -timeslices_a[i])
            ActualAnguloCambio= (velocidad[i] * tiempo) - (1f/3f *
            velocidad[i]*velocidad[i] / estado_MaxJointAcel);
        else if(j <=timeslices_e[i])
            ActualAnguloCambio = (velocidad[i] *(te[i]-ta[i])) -
            estado.MaxJointAcel / 2f *
                (te[i] - tiempo) * (te[i] - tiempo) ;
```

Movimiento Sincrónico PTP (Método SyncPTP):

En general el calculo para el movimiento sincrónico PTP es igual al asincrónico con la única diferencia de que debo elegir las distintas velocidades de cada motor $\hat{v}_{m,i}$ de manera tal que la duración de cada movimiento individual coincida con $t_{e,\max}$. Esto es:

$$t_{e,1} = t_{e,2} = t_{e,3} = t_{e,4} = t_{e,5} = t_{e,6} = t_{e,\max}$$

Desarrollo:
Primero calculamos el tiempo total máximo, al igual que el AsynPTP, calculamos el tiempo de desplazamiento de cada uno de los motores $t_{e,i}$ para los 6 ejes del robot.

$$\tilde{v}_{m,i} = \min(\tilde{v}_{m,i}, \sqrt{\tilde{x}_{e,i} \cdot \tilde{a}_{m,i}})$$

$$\tilde{t}_{a,i} = \frac{\tilde{v}_{m,i}}{\hat{a}_{m,i}}$$

$$\tilde{t}_{e,i} = \frac{\hat{x}_{e,i}}{\tilde{v}_{m,i}} + \tilde{t}_{a,i}$$

El máximo tiempo de desplazamiento será $t_{e,\max}$:

$$t_{e,\max} = \max(\tilde{t}_{e,1}, \tilde{t}_{e,2}, \tilde{t}_{e,3}, \tilde{t}_{e,4}, \tilde{t}_{e,5}, \tilde{t}_{e,6})$$

```
for( int i=0 ; i< _numjoin; int++)
{
      if( cambios[i] == 0 ) continue;

      if(_estado.MaxJointVel > Math.Sqrt(Math.Abs(cambios[i]) *
      estado_MaxJointAcel))
                  velocidad[i] =
                  (float)Math.Sqrt(Math.Abs(cambios[i]) *
                  estado_MaxJointAcel;
      else
            velocidad[i] = estado.MaxJointVel;

      ta[i] = velocidad[i] / estado.MaxJointAcel;
      te = (Math.Abs(cambios[i] / velocidad[i] + ta[i];
      td[i]= te - ta[i];
      if(te > temax) temax = te;

}
```

Luego calculamos las velocidades $\hat{v}_{m,i}$ imponiendo que cada eje debe terminar su movimiento exactamente en un tiempo $t_{e,\max}$

$$t_{e,\max} = \frac{\hat{x}_{e,i}}{v_{m,i}} + \frac{v_{m,i}}{\hat{a}_{m,i}}$$

Resolvemos la ecuación cuadrática para $\hat{v}_{m,i}$. De los 2 resultados matemáticamente posibles, la condición $t_{e,\max} \cdot 2 \le t_{e,\max}$ anula la mayor, quedando una única solución.

$$v_{m,i} = \frac{t_{e,\max} \cdot \hat{a}_{m,i}}{2} - \sqrt{\frac{t_{e,\max}^2 \cdot \hat{a}_{e,\max}^2}{4} - \hat{a}_{m,i} \cdot \hat{x}_{e,i}}$$

```
velocidad[i]=(float) ((estado.MaxJointAcel * temax /2)
                - (Math.Sqrt((estado.MaxJointAcel *
                estado.MaxJointAcel * temax * temax /4) -
                (Math.Abs(cambios[i] * estado_MaxJointAcel)))));
```

Debido a que ya tenemos el tiempo total de desplazamiento $t_{e,\max}$ solamente resta calcular los tiempos de aceleración $t_{a,i}$ y de desaceleración $t_{d,i}$ para cada eje del sistema. Al igual que el asyncPTP obtenemos los valores de $t_{a,i}$ y de $t_{d,i}$ de las siguientes ecuaciones:

$$t_{a,i} = \frac{v_{m,i}}{\hat{a}_{m,i}}$$

$$t_{d,i} = t_{e,i} - t_{a,i}$$

```
        ta[i]= velocidad / _estado.MaxJointAcel;
        td[i]= te[i] - ta[i];
```

Para finalizar, resta calcular los waypoints para la generación de la trayectoria, que esto se realiza de manera idéntica a lo expuesto en el método asincrónico.

Movimiento Sincrónico Full (Método FullSyncPTP):

En adición al movimiento sincrónico PTP, en este método también debemos sincronizar los tiempos de aceleración $t_{a,i}$ y el inicio de los tiempos de desaceleración $t_{d,i}$ para todos los ejes.

Desarrollo:

Al igual que en el método sincrónico debemos calcular $t_{e,\max}$. De la misma manera calculamos $t_{a,\max}$ y $t_{d,\max}$.

$$\tilde{v}_{m,i} = \min(\tilde{v}_{m,i}, \sqrt{\tilde{x}_{e,i} \cdot \tilde{a}_{m,i}})$$

$$\tilde{t}_{a,i} = \frac{\tilde{v}_{m,i}}{\hat{a}_{m,i}}$$

$$\tilde{t}_{d,i} = \tilde{t}_{e,i} - \tilde{t}_{a,i}$$

$$\tilde{t}_{e,i} = \frac{\hat{x}_{e,i}}{\tilde{v}_{m,i}} + \tilde{t}_{a,i}$$

$$t_{e,\max} = \max(\tilde{t}_{e,1}, \tilde{t}_{e,2}, \tilde{t}_{e,3}, \tilde{t}_{e,4}, \tilde{t}_{e,5}, \tilde{t}_{e,6})$$

$$t_{a,\max} = \max(\tilde{t}_{a,1}, \tilde{t}_{a,2}, \tilde{t}_{a,3}, \tilde{t}_{a,4}, \tilde{t}_{a,5}, \tilde{t}_{a,6})$$

$$t_{d,\max} = \max(\tilde{t}_{d,1}, \tilde{t}_{d,2}, \tilde{t}_{d,3}, \tilde{t}_{d,4}, \tilde{t}_{d,5}, \tilde{t}_{d,6})$$

```
for( int i=0 ; i< _numjoin; int++)
        {
            if( cambios[i] == 0 ) continue;

            if(_estado.MaxJointVel > Math.Sqrt(Math.Abs(cambios[i])
            * estado_MaxJointAcel))
                velocidad[i] = (float)Math.Sqrt(Math.Abs(cambios[i])
                * estado_MaxJointAcel;
            else
                velocidad[i] = estado.MaxJointVel;

            ta = velocidad[i] / estado.MaxJointAcel;
            te = (Math.Abs(cambios[i] / velocidad[i] + ta;
            td = te - ta;
            if(te > temax) temax = te;
            if(td > tdmax) tdmax = td;
            if(ta > tamax) tamax = ta;
        }
```

Luego procedemos a sincronizar los timpos de aceleración $t_{a,i}$, los tiempoes de desaceleración $t_{d,i}$ y el tiempo total de movimiento $t_{e,i}$.

$$t_{e,1} = t_{e,2} = \ldots = t_{e,\max}$$
$$t_{d,1} = t_{d,2} = \ldots = t_{d,\max}$$
$$t_{a,1} = t_{a,2} = \ldots = t_{a,\max}$$

Las velocidades máximas y las aceleraciones se computan individualmente para cada eje:

$$v_{m,i} = \frac{\hat{x}_{e,i}}{t_{d,\max}}$$
$$a_{m,i} = \frac{v_{m,i}}{t_{a,\max}}$$

```
for( int i=0 ; i< _numjoin; int++)
{
      velocidad[i]=(Math.Abs(cambios[i]) / tdmax;
      aceleracion[i]=velocidad[i]/ tamax
}
```

Para finalizar, los waypoints se calculan como lo expuesto en los puntos anteriores. Todos los métodos los calculan con el mismo método.

5 - Implementación de funciones de movimiento de bajo nivel:

MoveTo del Brazo Articulado - RT2:

Se creó el método MoveTo que puede ser utilizado para mover el brazo a una posición específica. Este método toma los siguientes parámetros:

Parámetro	Unidad	Descripción
baseVal	Grados	Angulo de rotación del base Joint
L1Val	Grados	Angulo Pivot del Joint L1
L2Val	Grados	Angulo Pivot del Joint L2
L3Val	Grados	Angulo Pivot del Joint L3
L4Val	Grados	Angulo Pivot del Joint L4
TenazaVal	Metros	Distancia de apertura de la tenaza
Tiempo	Segundos	Tiempo para completar el movimiento

Se utiliza una variable del tipo bool llamada _moveToActivo que indica cuando una operación de movimiento aún sigue ejecutándose. Si se hace una llamada al método MoveTo mientras se encuentra realizando una operación de movimiento, se postea un

mensaje de excepción al puerto de consola. Cada parámetro que se pasa a la función es chequeado respecto de los límites específicos correspondientes a la descripción del Joint. Si se pasan parámetros inválidos se postea un mensaje de excepción en el puerto de consola para notificar el error. El método retorna un objeto SuccessFailurePort una vez que ha sido iniciado.

```
SuccessFailurePort PuertoRespuesta = new SuccessFailurePort();
if (_moveToActivo)
{
        PuertoRespuesta.Post(new Exception("MoveTo previo aún ejecutándose."));
        return PuertoRespuesta;
}
// chequea los límites, si son invalidos -> error
if(!_joints[0].ValidTarget(baseVal))
{
        esponsePort.Post(new Exception(
        _joints[0].Name + "Valor no válido en el Joint:: " + baseVal.ToString()));
        return PuertoRespuesta;
}
```

Luego de que todos los parámetros han sido validados, las variable Destino de cada Joint se setean con los correspondientes valores. Se calcula la variable velocidad de cada Joint basándose en la distancia entre la posición actual y el destino. Cada Joint recibe su propia velocidad para lograr que todas las partes del brazo articulado lleguen al destino al mismo tiempo.

```
// setae las variables destino de cada joint
_joints[0].Target = baseVal;
_joints[1].Target = L1Val;
_joints[2].Target = L2Val;
_joints[3].Target = L3Val;
_joints[4].Target = L4Val;
_joints[5].Target = TenazaVal;
// calcula la velocidad de cada joint para cumplir con un tiempo determinado
for(int i=0; i < 6; i++)
_joints[i].Speed = Math.Abs(_joints[i].Target - _joints[i].Current) / tiempo;
```

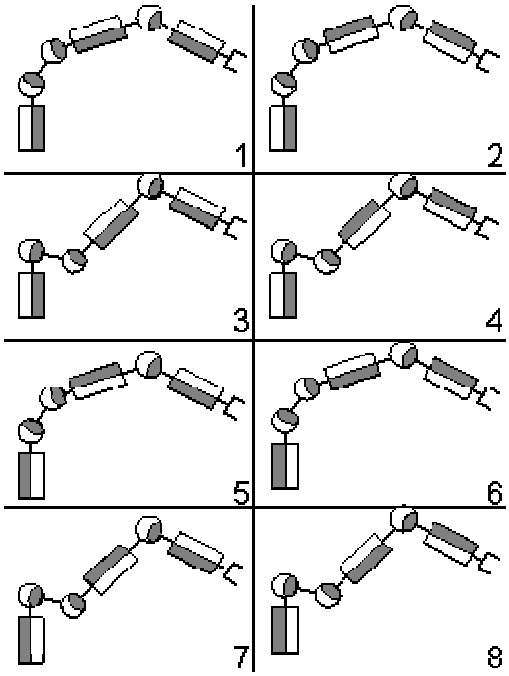
Una vez seteados los valores destino de cada Joint, el método MoveTo setea el flag _moveToActivo a verdadero y retorna el puerto de respuesta utilizado. Es necesario aclarar que este método setea los valores pero el Joints no se moverá hasta que se llame el método update que idealmente se realiza unas 60 veces por segundo. Cada vez que éste método es llamado el Joint se desplazará en pantalla.

```
// Actualiza los joints si es necesario
if (_moveToActivo)
{
	bool done = true;
	// Chequea cada Joint y actualize si es necesario
	if (_joints[0].NeedToMove(_epsilon))
	{
		done = false;
		Vector3 normal = _joints[0].Joint.State.Connectors[0].JointNormal;
		_joints[0].UpdateCurrent(_prevTime);
		_joints[0].Joint.SetAngularDriveOrientation(
			Quaternion.FromAxisAngle(
				normal.X, normal.Y, normal.Z,
				DegreesToRadians(_joints[0].Current)));
	}
```

6 -Funciones de movimiento de alto nivel:

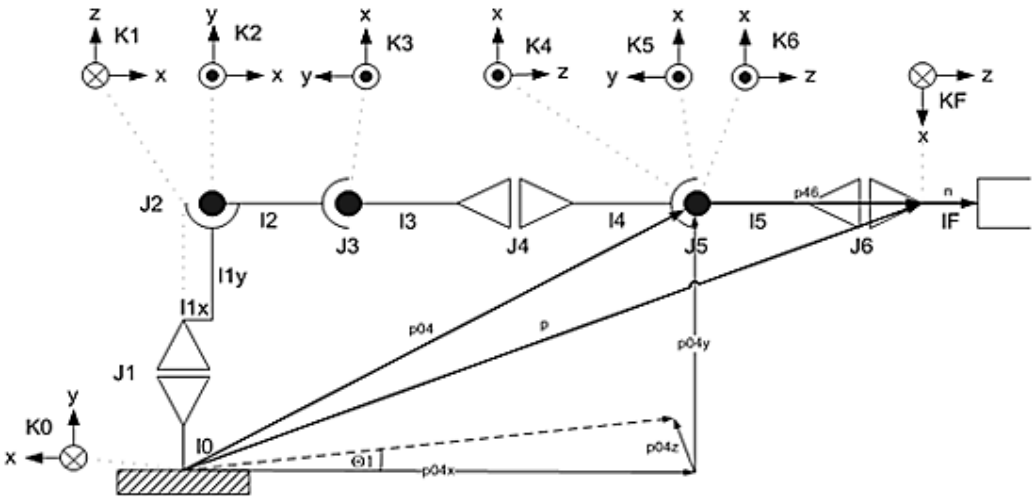
Cinemática Inversa:

La cinemática se ocupa de la descripción del movimiento sin tener en cuenta sus causas. El objetivo de la cinemática inversa consiste en encontrar el gesto que deben adoptar las diferentes articulaciones para que el final del sistema articulado llegue a una posición concreta. Uno de los inconvenientes radica en la multiplicad de soluciones que pueden ser encontradas. Por ejemplo, para una misma coordenada cartesiana requerida, en la figura se muestran distintas formas de llegar al resultado:



En nuestro caso implementamos la cinemática inversa a través de una aproximación geométrica.

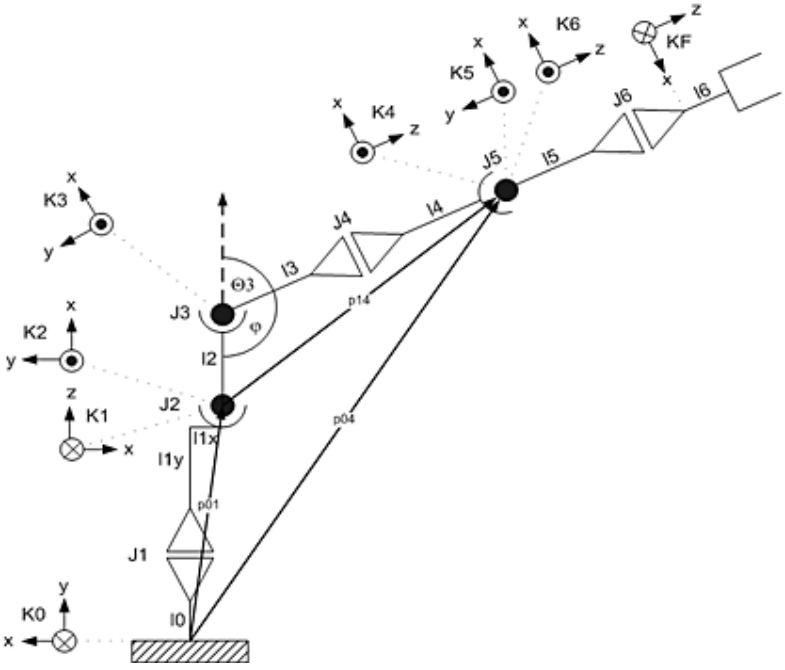
Ángulo Joint 1:



Este ángulo Θ_1 puede ser calculado mediante la proyección del marco K4 en el plano x-z del marco de referencia K0, como se puede observar en la figura. En nuestro modelo físico del brazo articulado tenemos definidos 3 Joints que tienen modo de rotación Twist (Joints 1, 4 y 6), para simplificar los cálculos, asumimos que estos Joints Twist, no cambian la posición cartesiana del brazo pero si su orientación.
De esta configuración resultan 2 soluciones posibles:

$$\Theta_{1,1} = \Theta_{1,2} = \Theta_{1,3} = \Theta_{1,4} = arctan2\left(-p_{04,z}, p_{04,x}\right)$$
$$\Theta_{1,5} = \Theta_{1,6} = \Theta_{1,7} = \Theta_{1,8} = arctan2\left(-p_{04,z}, p_{04,x}\right) + \pi$$

Ángulo de Joint 3:



Sabiendo $\vec{p}_{14}$ obtenemos el ángulo φ que conforma el ángulo del Joint Θ_3.
Nuevamente consideramos que los Joint Twist no cambian el marco de referencia cartesiano, sólo la orientación del mismo. Calculamos $\vec{p}_{14}$ como:

$$\vec{p}_{14} = \vec{p}_{04} - \vec{p}_{01}$$

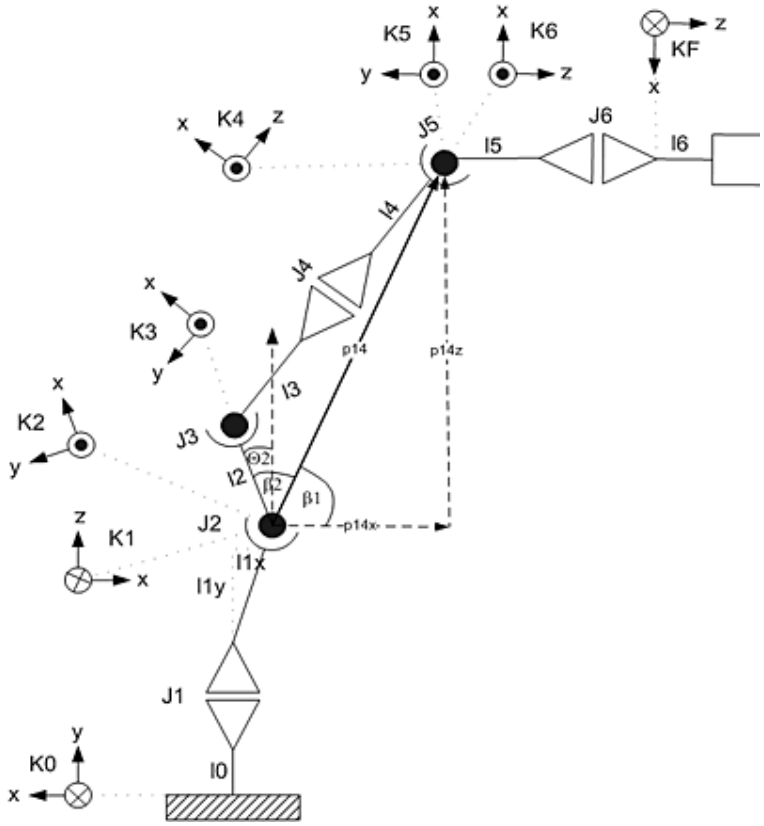
Y aplicando la regla trigonométrica del coseno:

$$\varphi = arccos\left(\frac{l_2^2 + (l_3 + l_4)^2 - |\vec{p}_{14}|^2}{2 \cdot l_2 \cdot (l_3 + l_4)} \right)$$

Finalmente obtenemos las demás soluciones válidas del problema:

$$\Theta_{3,1} = \Theta_{3,2} = \pi - \varphi$$
$$\Theta_{3,3} = \Theta_{3,4} = \pi + \varphi$$
$$\Theta_{3,5} = \Theta_{3,6} = -(\pi - \varphi)$$
$$\Theta_{3,7} = \Theta_{3,8} = -(\pi + \varphi)$$

Ángulo de Joint 2:



Mediante el mismo procedimiento obtenemos:

$$\beta_1 = \arctan\left(\vec{p}_{14,x}^{(1)}, \vec{p}_{14,z}^{(1)}\right)$$

$$\beta_2 = \arccos\left(\frac{l_2^2 + \left|\vec{p}_{14}\right|^2 - (l_3 + l_4)^2}{2 \cdot l_2 \cdot \left|\vec{p}_{14}\right|}\right)$$

Luego las demás soluciones válidas:

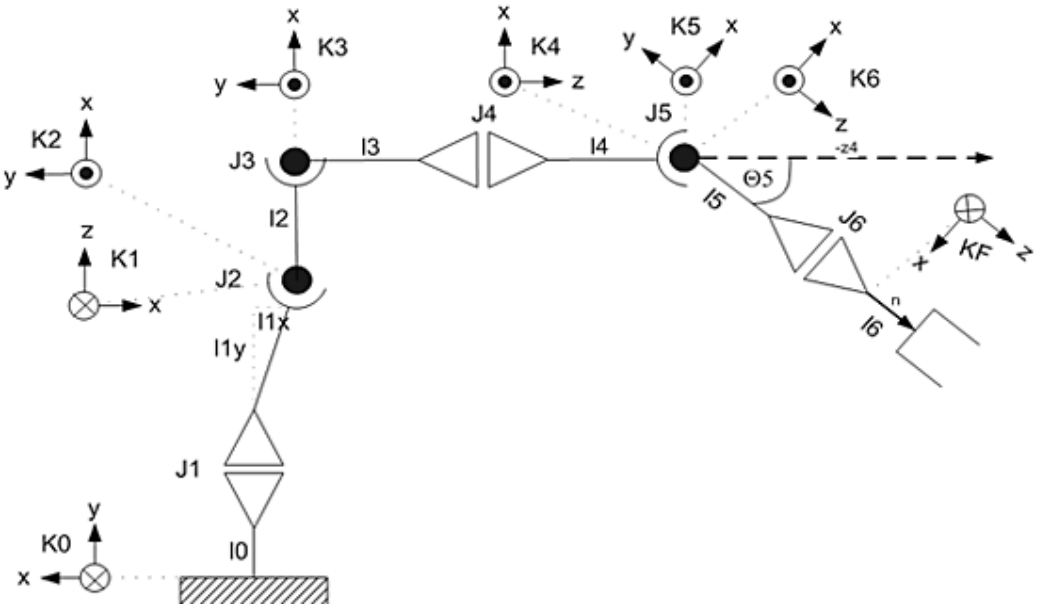
$$\Theta_{2,1} = \Theta_{2,2} = -(\beta_1 + \beta_2)$$

$$\Theta_{2,3} = \Theta_{2,4} = -(\beta_1 - \beta_2)$$

$$\Theta_{2,5} = \Theta_{2,6} = \beta_1 + \beta_2$$

$$\Theta_{2,7} = \Theta_{2,8} = \beta_1 - \beta$$

Ángulo Joint 5:



La aproximación geométrica de este Joint es la siguiente:

$$\Theta_{5,1} = \Theta_{5,5} = \arccos\left(\vec{z}_{4,A} \cdot \vec{n}\right)$$
$$\Theta_{5,2} = \Theta_{5,6} = -\arccos\left(\vec{z}_{4,A} \cdot \vec{n}\right)$$
$$\Theta_{5,3} = \Theta_{5,7} = \arccos\left(\vec{z}_{4,B} \cdot \vec{n}\right)$$
$$\Theta_{5,4} = \Theta_{5,8} = -\arccos\left(\vec{z}_{4,B} \cdot \vec{n}\right)$$

Ángulos Joints 4 y 6:

Estos Joints se encargan de definir la orientación. La rotación total puede ser expresada de la siguiente manera:

$${}^{0}\mathbf{R}_{7} = {}^{0}\mathbf{R}_{4} \cdot {}^{4}\mathbf{R}_{7} \quad \Rightarrow \quad {}^{4}\mathbf{R}_{7} = \left({}^{0}\mathbf{R}_{4}\right)^{-1} \cdot {}^{0}\mathbf{R}_{7} = {}^{4}\mathbf{R}_{0} \cdot {}^{0}\mathbf{R}_{7}$$

Debido a que la matriz R es ortogonal, su inversa puede ser calculada mediante su transpuesta.

$$\left({}^{0}\mathbf{R}_{4}\right)^{T} \cdot {}^{0}\mathbf{R}_{4} = \mathbf{I} \quad \Rightarrow \quad {}^{4}\mathbf{R}_{0} = \left({}^{0}\mathbf{R}_{4}\right)^{T} = \left({}^{0}\mathbf{R}_{4}\right)^{-1}$$

La matriz R_7 esta compuesta por 3 rotaciones secuenciales alrededor de Z-Y-Z

$${}^{4}\mathbf{R}_{7} = \text{Rot}_{z}(\Theta_{4}) \cdot \text{Rot}_{y}(\Theta_{5}) \cdot \text{Rot}_{z}(\Theta_{6})$$

Por lo tanto obtenemos:

$${}^{4}\mathbf{R}_{7} = \left({}^{0}\mathbf{R}_{4}\right)^{T} \cdot {}^{0}\mathbf{R}_{7}$$

$$= \begin{pmatrix} c\Theta_4 \cdot c\Theta_5 \cdot c\Theta_6 - s\Theta_4 \cdot s\Theta_6 & -c\Theta_4 \cdot c\Theta_5 \cdot s\Theta_6 - s\Theta_4 \cdot c\Theta_6 & c\Theta_4 \cdot s\Theta_5 \\ s\Theta_4 \cdot c\Theta_5 \cdot c\Theta_6 + c\Theta_4 \cdot s\Theta_6 & -s\Theta_4 \cdot c\Theta_5 \cdot s\Theta_6 - c\Theta_4 \cdot c\Theta_6 & s\Theta_4 \cdot s\Theta_5 \\ -s\Theta_5 \cdot c\Theta_6 & s\Theta_5 \cdot s\Theta_6 & c\Theta_5 \end{pmatrix}$$

De esta manera tenemos 2 matrices que contienen todas las soluciones posibles:

$$ {}^{4}\mathbf{R}_{7,A} = \left({}^{0}\mathbf{R}_{4,A}\right)^{T} \cdot {}^{0}\mathbf{R}_{7} $$

$$ {}^{4}\mathbf{R}_{7,B} = \left({}^{0}\mathbf{R}_{4,B}\right)^{T} \cdot {}^{0}\mathbf{R}_{7} $$

Sabiendo la relación de la tangente de un ángulo respecto de sus senos y cosenos, calculamos:

$$ \Theta_4 = \operatorname{atan}2\left(\pm r_{23}, \pm r_{13}\right) $$

Ahora estamos en condiciones de calcular todas sus soluciones:

$$ \Theta_{4,1} = \Theta_{4,6} = \operatorname{atan}2\left(r_{23,A}, r_{13,A}\right) $$

$$ \Theta_{4,2} = \Theta_{4,5} = \operatorname{atan}2\left(r_{23,A}, r_{13,A}\right) + \pi $$

$$ \Theta_{4,3} = \Theta_{4,8} = \operatorname{atan}2\left(r_{23,B}, r_{13,B}\right) $$

$$ \Theta_{4,4} = \Theta_{4,7} = \operatorname{atan}2\left(r_{23,B}, r_{13,B}\right) + \pi $$

Implementación en Soft de la Cinemática Inversa:

Se implemento un método llamado MoveToPosicion que toma los parámetros de la siguiente tabla, calcula las posiciones de los Joints y luego llama al método MoveTo para ejecutar el movimiento mediante la cinemática inversa.

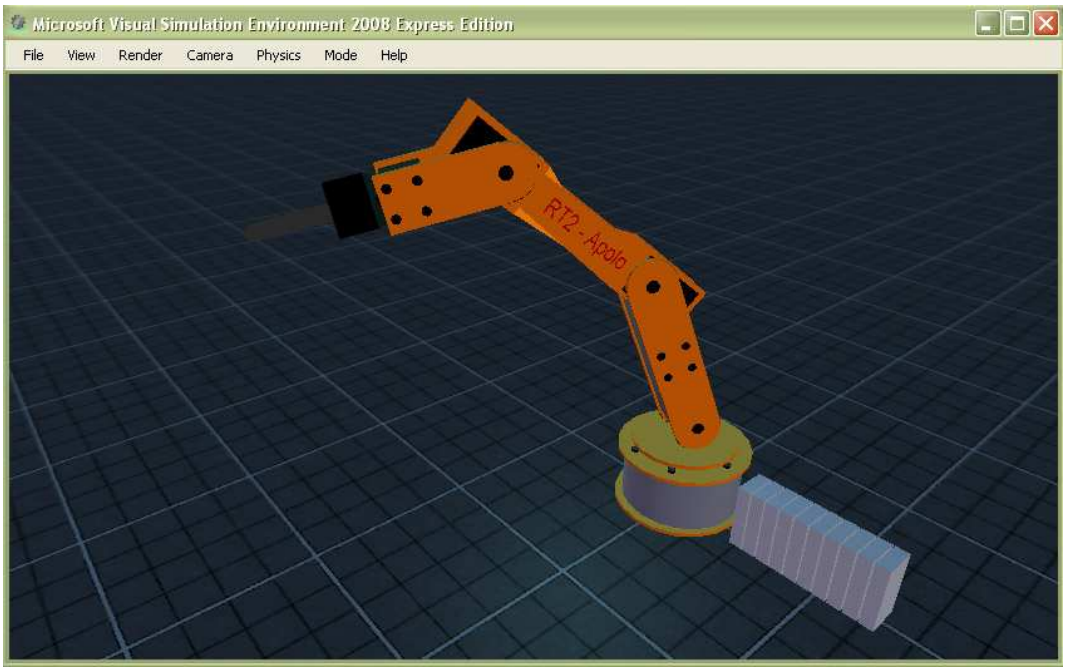
Parámetro	Unidad	Descripción
X	Metros	Posición eje X
Y	Metros	Posición eje Y
Z	Metros	Posición eje Z
P	Grados	Angulo de llegada de la tenaza
W	Grados	Angulo de rotación de la tenaza
Tenaza	Metros	Distancia de apertura de la tenaza
Tiempo	Segundos	Tiempo para completar el movimiento

```
public SuccessFailurePort MoveToPosicion(
            float x,
            float y,
            float z,
            float p,
            float w,
            float grip,
            float tiempo)
{

float r = (float)Math.Sqrt(x * x + z * z); // distancia horizontal al objetivo
float baseAngle = (float)Math.Atan2(-z, -x); // Angulo respecto del objetivo
float pRad = GradosARadianes(p);
float rb = (float)((r - L3 * Math.Cos(pRad)) / (2 * L1));
float yb = (float)((y - H - L3 * Math.Sin(pRad)) / (2 * L1));
float q = (float)(Math.Sqrt(1 / (rb * rb + yb * yb) - 1));
float p1 = (float)(Math.Atan2(yb + q * rb, rb - q * yb));
float p2 = (float)(Math.Atan2(yb - q * rb, rb + q * yb));
float L1 = p1 - GradosARadianes (90); // Angulo de L1
float L2 = p2 - shoulder; // Angulo de L2
float L3 = pRad - p2; // Angulo de L3
// Posiciono el brazo con los valores calculados
return _l6Arm.MoveTo(
            RadiansToDegrees(baseAngle),
            GradosARadianes (L1),
            GradosARadianes (L2),
            GradosARadianes (L3),
            w,
            grip,
            tiempo);
}
```

6- Resultados Finales del Sistema de Control y Simulación:

Vista Del Motor de Simulación Físico terminada:

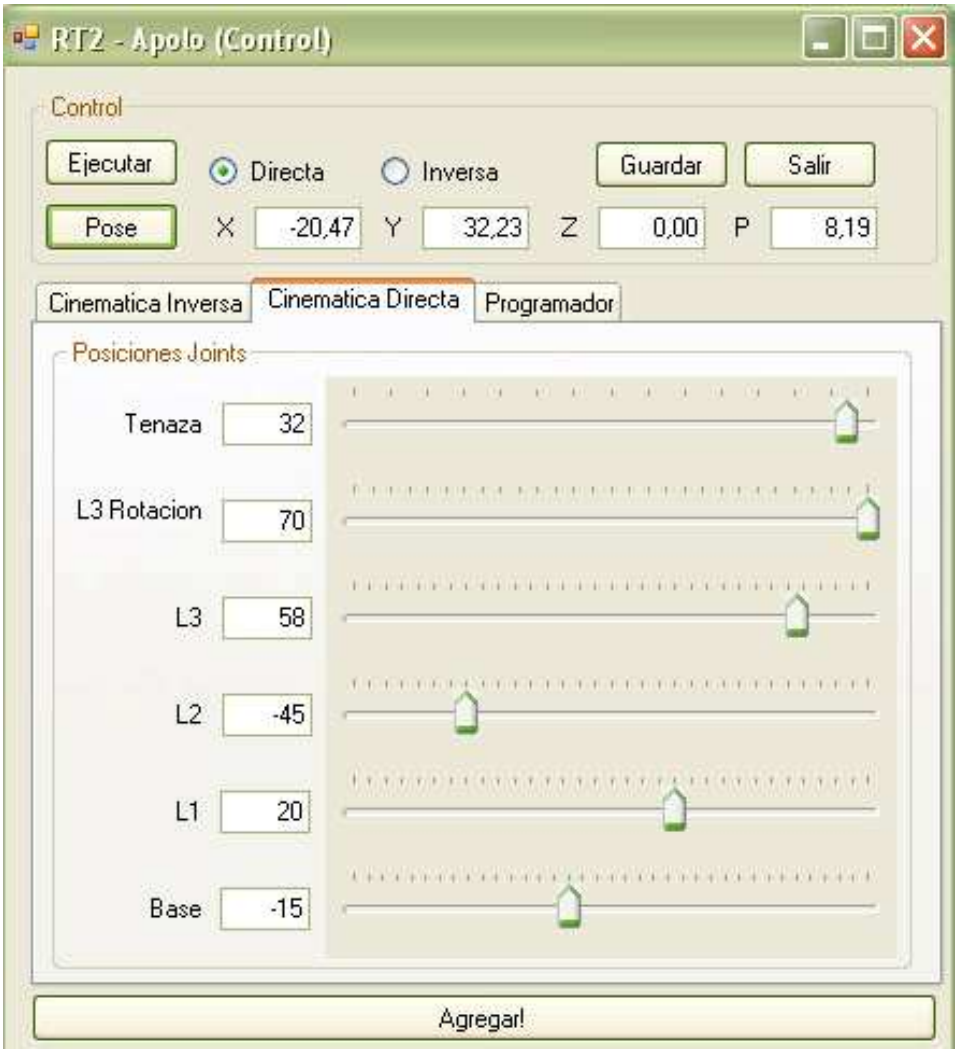


Como se puede observar a las entidades y Joints creados mediante la programación expuesta se le cargo un modelo 3D realizado con el programa BlenderStudio para lograr un resultado estético superior a lo previamente analizado.

Consola de Movimientos del Sistema:

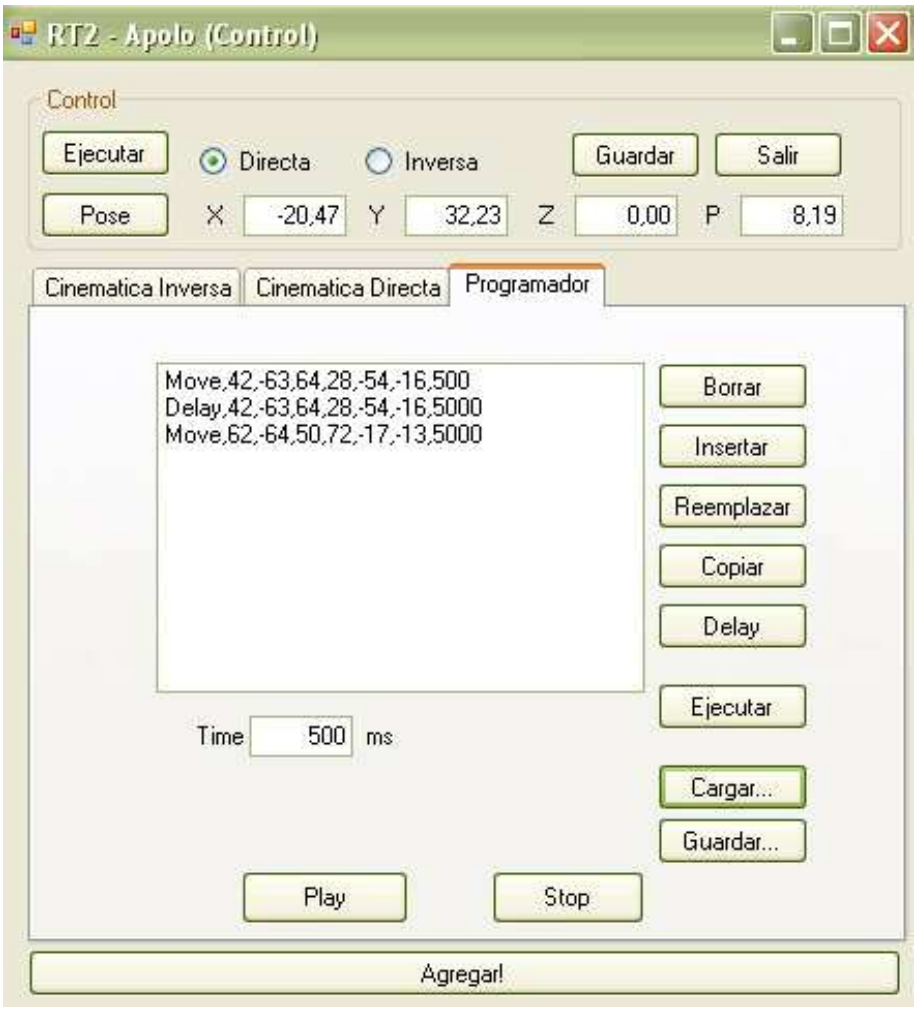
Esta Consola nos permite realizar movimientos aleatorios mediante el servicio de cinemática inversa y se le han cargado series de movimientos preestablecidos para realizar una demostración de las capacidades del sistema desarrollado. Hay 3 Secuencias, 2 son de Pick&Place y uno para "estacionar" el brazo articulado.
Los Pick&Place al ejecutarse hacen que el brazo articulado agarre las cajas que estan dispuestas a su alrededor y las lleven a otro sector, de 2 maneras diferentes. El modo de "estacionar" simplemente apaga el robot pero antes lo lleva a una posición de descanso predeterminada.

Consola de Control del Sistema:

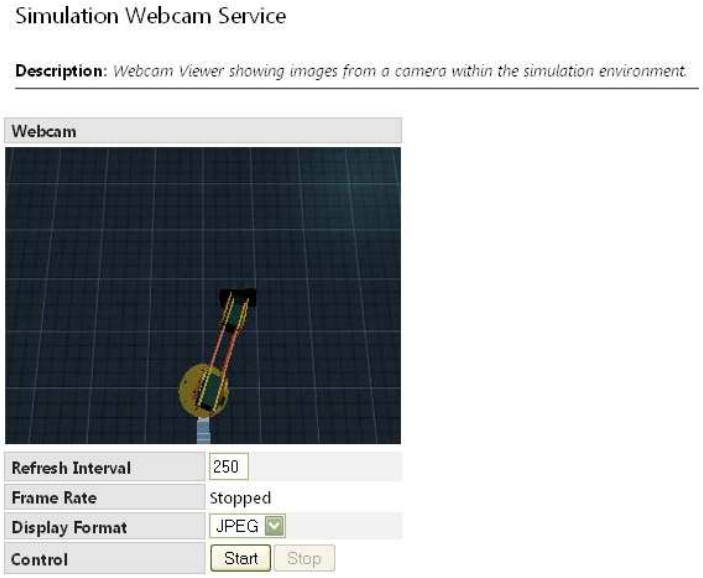


Esta aplicación es una evolución de la consola de Debug vista anteriormente y permite el control cinemático inverso, directo, movimientos independientes de cada uno de los joints y articulaciones del brazo. También cuenta con la opción de grabar una secuencia de movimientos, guardarla en un archivo para poder ser cargada y ejecutada cuando sea necesario. Se incluye el módulo de comunicación serie que vincula este sistema con el módulo FPGA-VHDL desarrollado para el RT2-Apolo. Este sistema le envía las coordenadas de cada uno de los 6 motores y el FPGA se encarga de realizar la ejecución en el hardware.

Consola de Programación del Sistema:



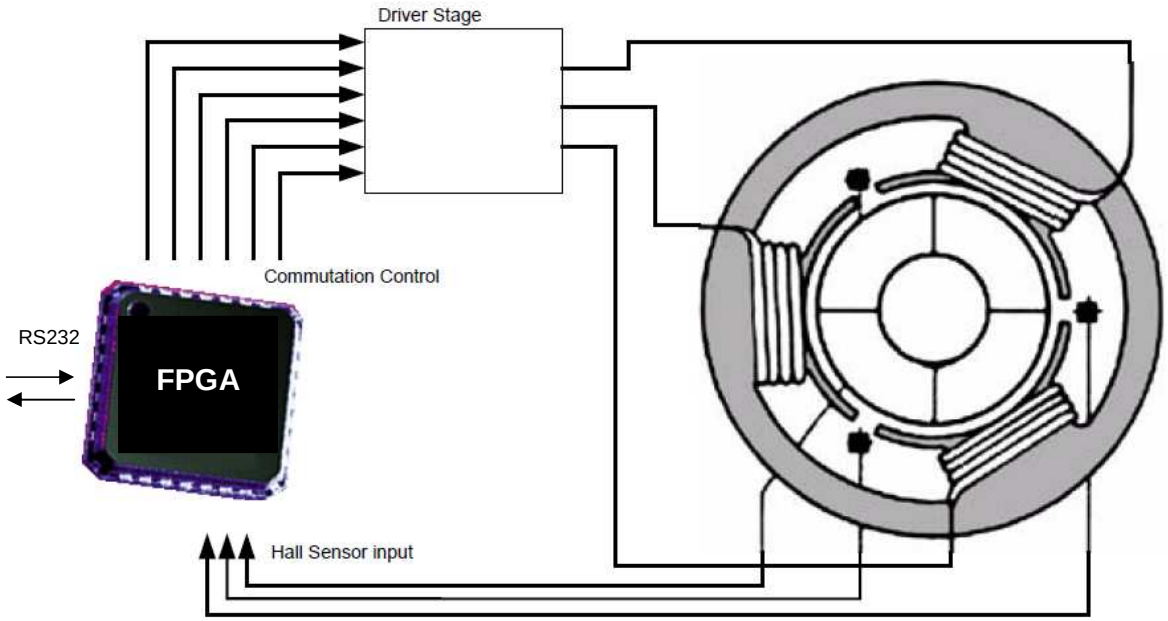
Simulación de WebCam para ser transmitida por Internet:



Mediante el uso de los servicios genéricos que trae el Microsoft Robotics Studio, es muy sencillo acoplarle un servicio de WebCam para ser conectado a Internet. En la imagen se puede observar la vista superior del RT2-Apolo, simulando un control remoto del brazo articulado

8- Implementación en un FPGA del control de los motores del robot:

Diagrama conceptual



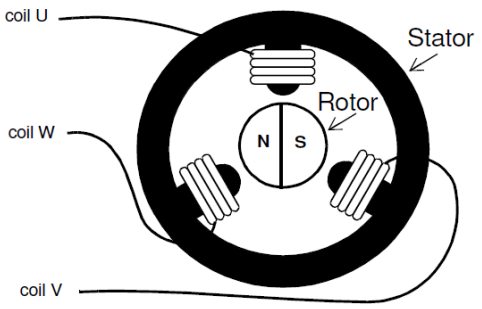
Introducción

La idea de la implementación, es embeber en un FPGA toda la lógica de control de todos los motores del robot.
Para esto, dentro del mismo contaremos con:

- Unidad de transmición asincrónica RS232.
- Microprocesador PicoBlaze para comunicación con protocolo propietario con la PC.
- Unidades de control de giro y velocidad de cada uno de los motores.

Además, se necesitará una etapa de potencia para cada unidad de control de giro, para conectar directamente al motor.

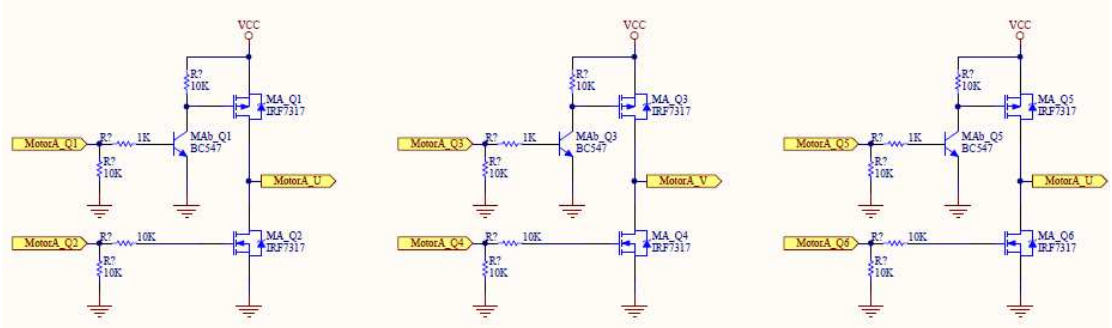
Motor de continua sin escobillas



1 pair of poles

Este es un diagrama del tipo de motor que utilizaremos. Para conocer la posición angular del rotor, se incluyen tres sensores hall, obteniendo una resolución de giro de 60°, aceptable para la decisión de activación de las diferentes bobinas.

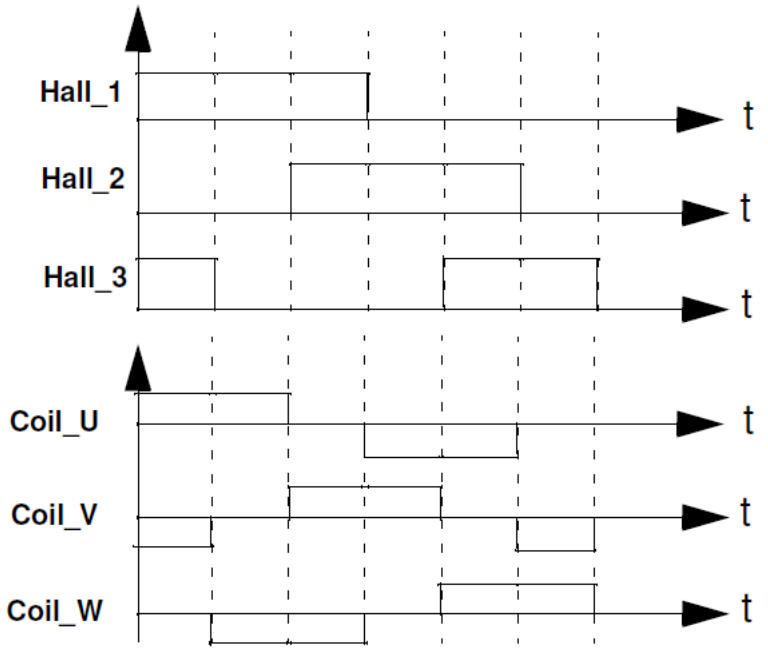
Etapa de potencia



De acuerdo al par de transistores que se activen, se logrará que el motor gire en un sentido u otro.
Los transistores Q1, Q3 y Q5 se activarán siempre en forma continua. En cambio, los transistores Q2, Q4 y Q6 serán modulados en su compuerta en PWM, permitiendo así ajustar digitalmente la velocidad de giro de cada motor.
Las resistencias y transistores bipolares agregados son para adaptar los niveles de salida del FPGA a los de trabajo de los transistores.

Sentido de giro del motor

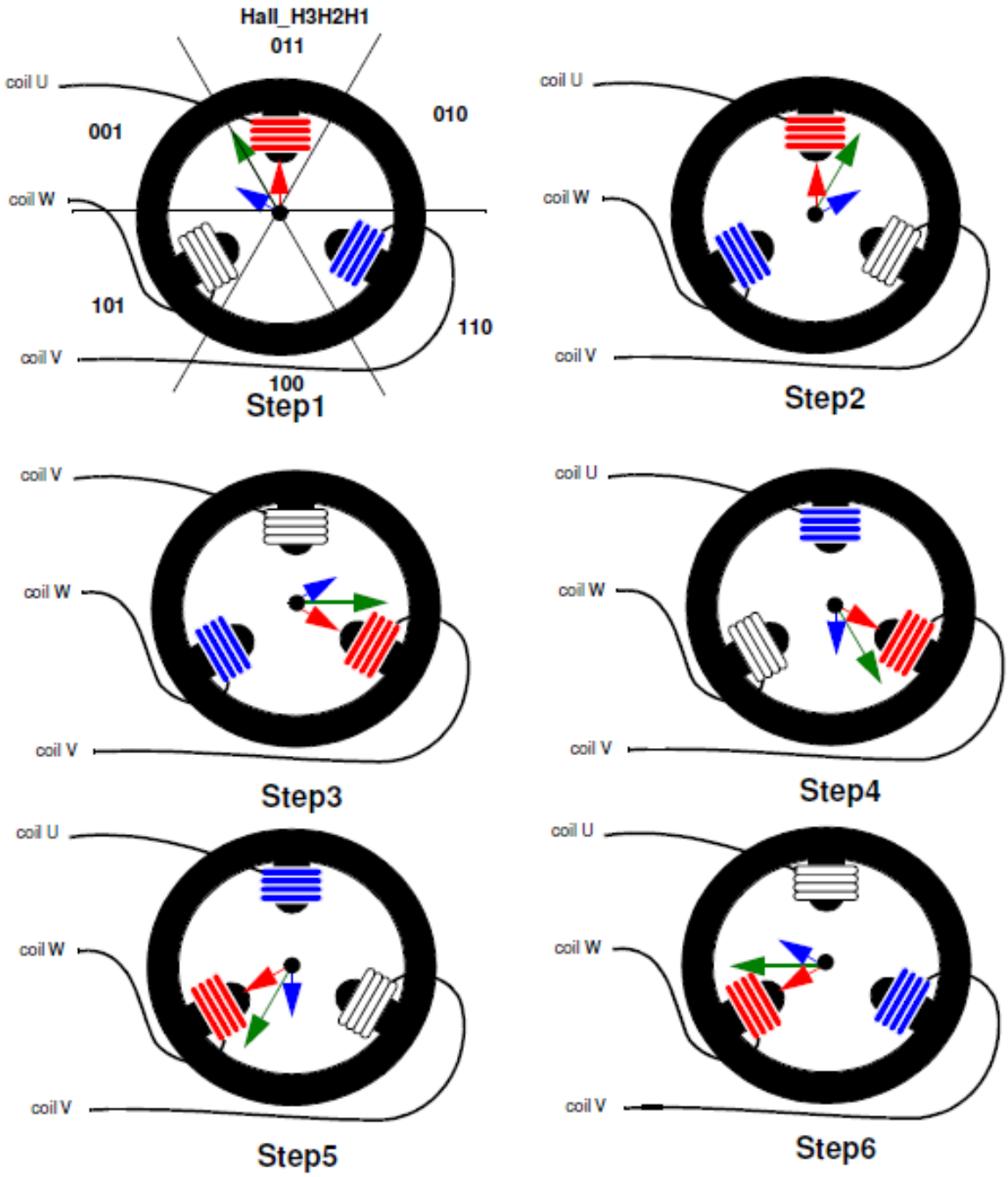
Como se mencionó, de acuerdo al estado de los sensores hall, se decide que bobinas alimentar para obtener un sentido de giro deseado.
En el siguiente diagrama puede visualizarse la secuencia de activación de bobinas, para lograr un giro horario.



Hall Sensors Value (H3 H2 H1)	Phase	Switches
101	U-V	Q1 ; Q4
001	U-W	Q1 ; Q6
011	V-W	Q3 ; Q6
010	V-U	Q3 ; Q2
110	W-U	Q5 ; Q2
100	W-V	Q5 ; Q4

La tabla simplemente evidencia más claramente que par de transistores hace falta activar para encender cada bobina con el sentido de circulación de corriente que se necesita.

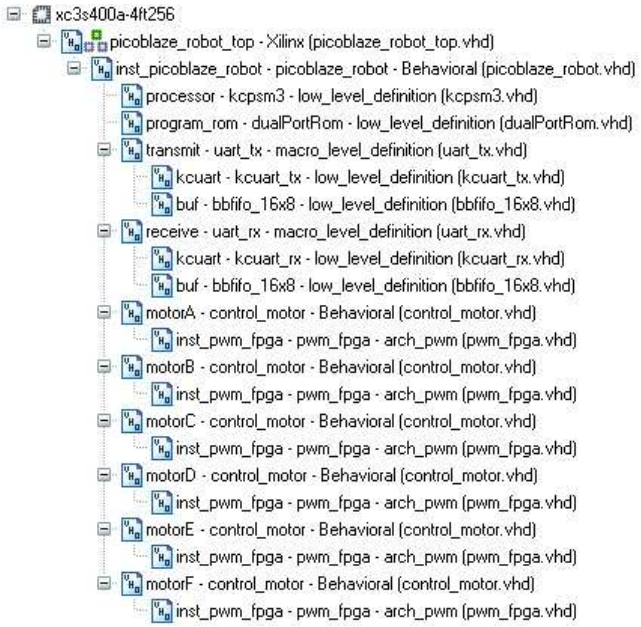
Esquematización de un giro completo



Implementación en VHDL

Los archivos fuentes en vhdl, se adjuntan en formato digital. La simulacion e implementacion se realizo sobre la plataforma ISE, de Xilinx Inc. Para un FPGA de la lineaSpartan 3, modelo XC3S400A-4.

La estructura de la implementación puede apreciarse en el siguiente árbol de jerarquías



Se utilizo una libreria de un pwm para obtener una modulación por pulsos a partir un valor de 8 bits. Dicha librería se encuentra en el archivo “pwm_fpga.vhd”.

Se instancia un componente de esta libreria en el archivo “control_motor.vhd”.
Esta entidad es la que se ocupa de decidir que transistor activar en función del estado de tres sensores hall y el valor deseado para el pwm.
Basicamente, según que sensores estén activados, se activa un par de transistores: uno de los dos esta siempre activado y el otro modulado según el pwm.
Cuando los sensores hall cambian de posición, el par de transistores activados cambia, siendo esto ciclico para lograr asi la implementacion del giro del motor a la velocidad deseada.
Todo esto se hace respetando la tabla de verdad anterior.

Para la simulacion de la entidad, se escribio un test bench de nombre “control_motor_tb.vhd”. En este text bench se instancian seis componentes de “control_motor”, para así simular el funcionamiento conjunto de los seis motores de nuestro robot.
Este banco de prueba simula una senal de clock de 10 Mhz (ya que se opto por una implementación sincronica de la solucion), tres senales de sensores hall que van cambiando a una velocidad de 400 us (la misma para los seis motores, que si bien esto no es cierto simplifica la simulación) y un puerto de 8 bits con el que se setea el valor

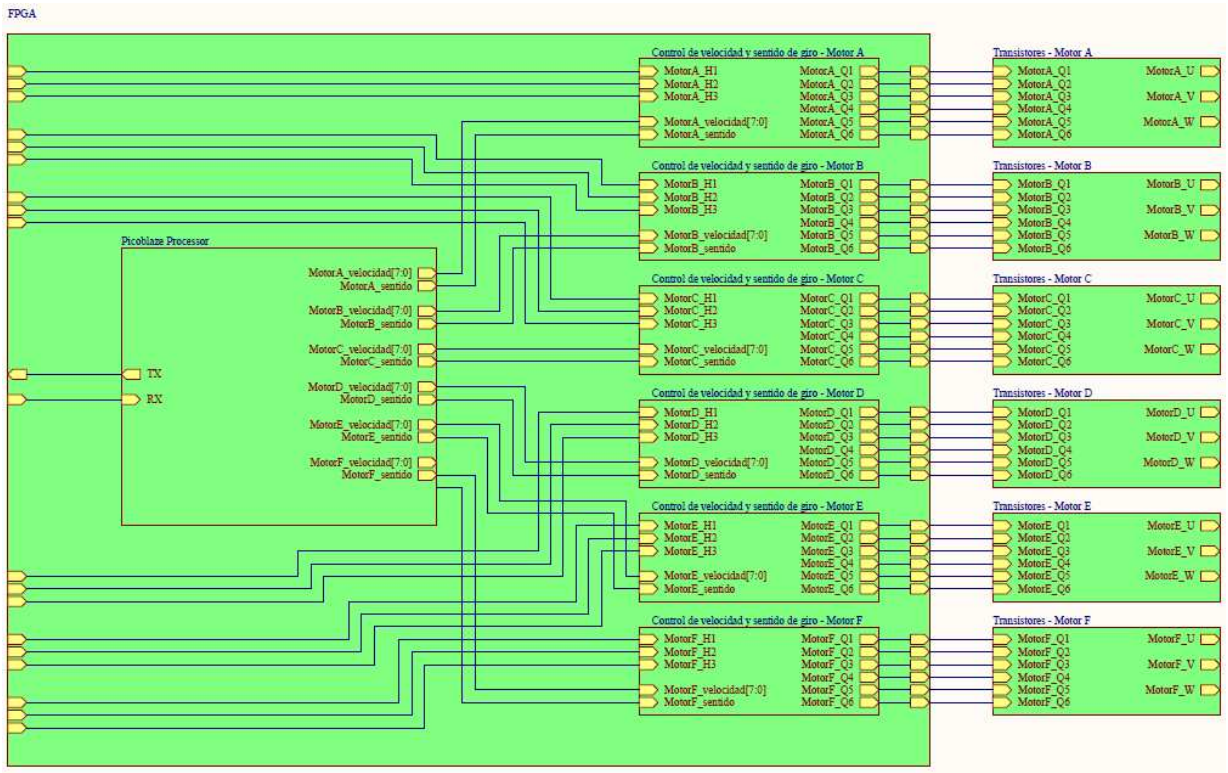
del pwm (para ajustar la velocidad de los seis motores por igual, que nuevamente no es cierto, pero simplifica la simulación).

Para la implementación se creo una nueva entidad que instancia (al igual que el test bench) seis etapas controladote de sentido de giro y velocidad de los motores ("control_motor.vhd") y además se instancia un microprocesador PicoBlaze, una rom con el programa del procesador, una UART de transmición y una UART de recepción.

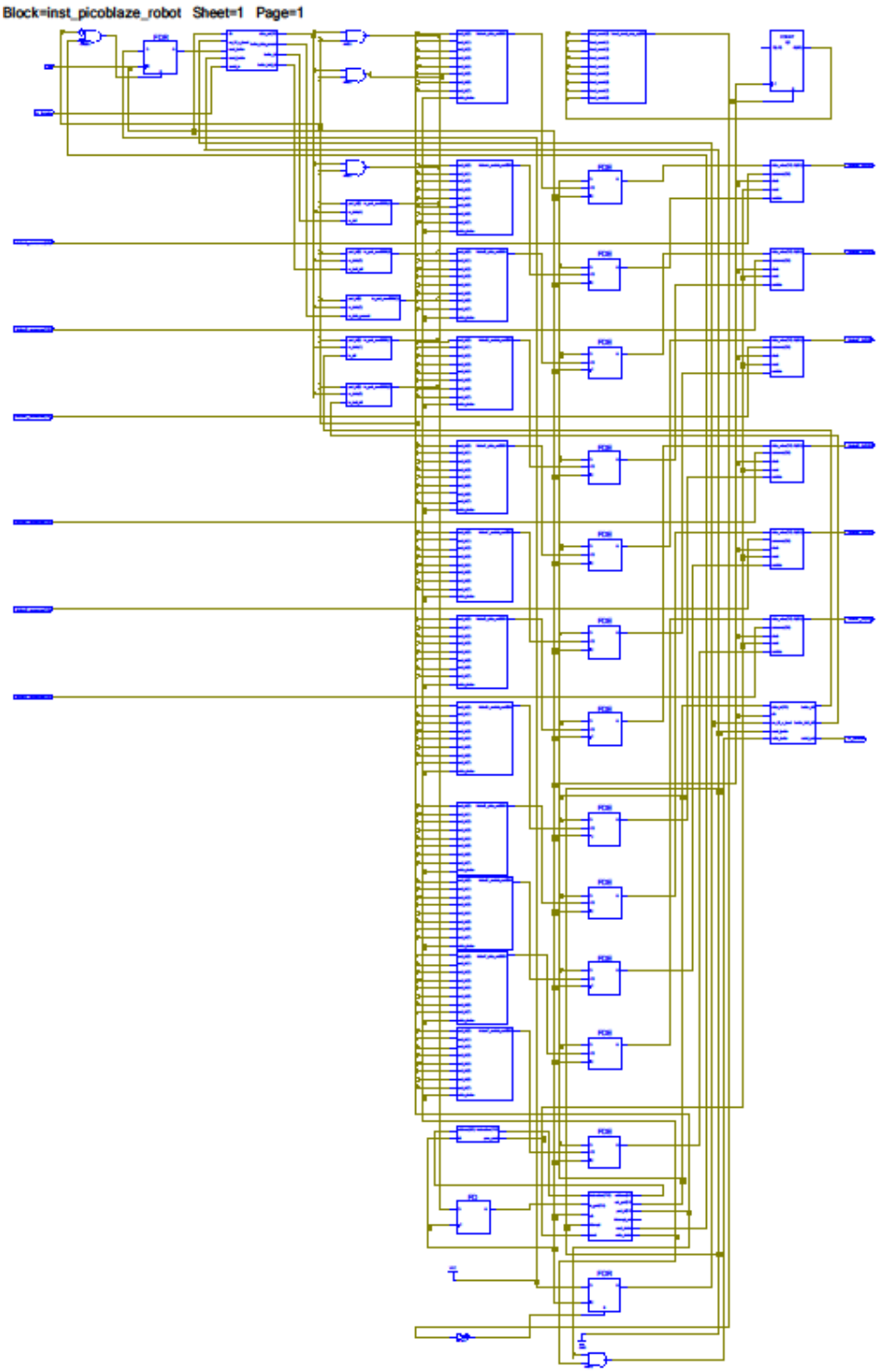
Toda esta implementación permite embeber en un único chip todo lo necesario para el control de los seis motores y su comunicación con la PC para configurar el sistema en tiempo real.

Por ultimo se escribio un top module de la solución con el nombre para poder bajar la configuracion a la FPGA y verificar su correcto funcionamiento práctico.

Diagrama en bloques de la implementación en VHDL



Esquemático interno de la implementación en el FPGA



Información de la implementación

Tesis FPGA Project Status (08/10/2009 - 10:20:18)			
Project File:	Tesis FPGA.ise	**Current State:**	Placed and Routed
Module Name:	picoblaze_robot_top	**• Errors:**	No Errors
Target Device:	xc3s400a-4ft256	**• Warnings:**	70 Warnings (69 new, 0 filtered)
Product Version:	ISE 10.1.03 - WebPACK	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

Tesis FPGA Partition Summary [-]
No partition information was found.

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	327	7,168	4%	
Number of 4 input LUTs	674	7,168	9%	
Logic Distribution				
Number of occupied Slices	424	3,584	11%	
Number of Slices containing only related logic	424	424	100%	
Number of Slices containing unrelated logic	0	424	0%	
Total Number of 4 input LUTs	696	7,168	9%	
Number used as logic	562			
Number used as a route-thru	22			
Number used for Dual Port RAMs	16			
Number used for 32x1 RAMs	52			
Number used as Shift registers	44			
Number of bonded IOBs				
Number of bonded	57	195	29%	
IOB Flip Flops	12			
Number of BUFGMUXs	2	24	8%	
Number of BSCANs	1	1	100%	
Number of BSCAN_SPARTAN3As	1	1	100%	
Number of RAMB16BWEs	1	20	5%	

Tiempos máximos

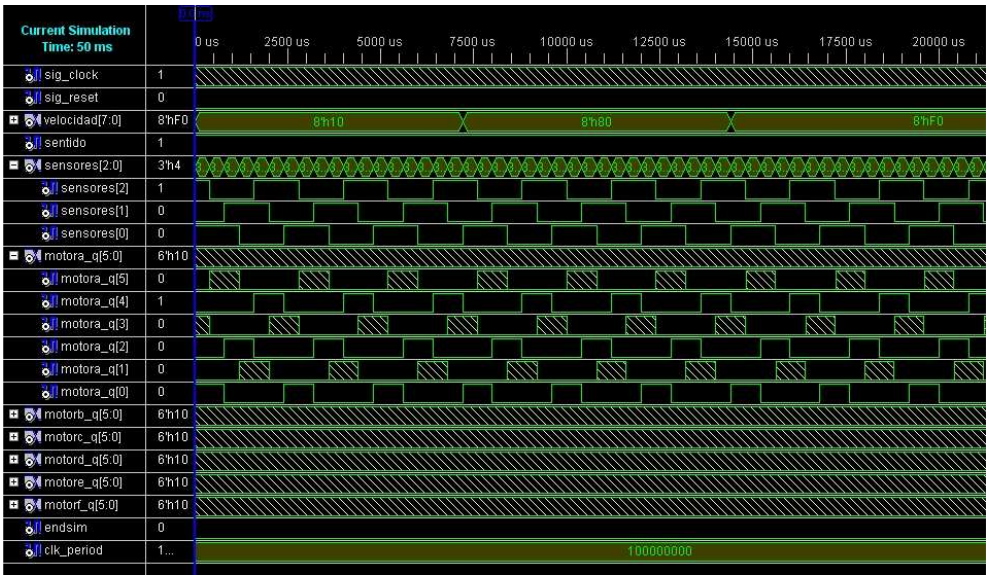
Pudiendo apreciarse que para esta FPGA Spartan 3, puede implementarse esta solucion con un clock maximo de 97,409 Mhz.

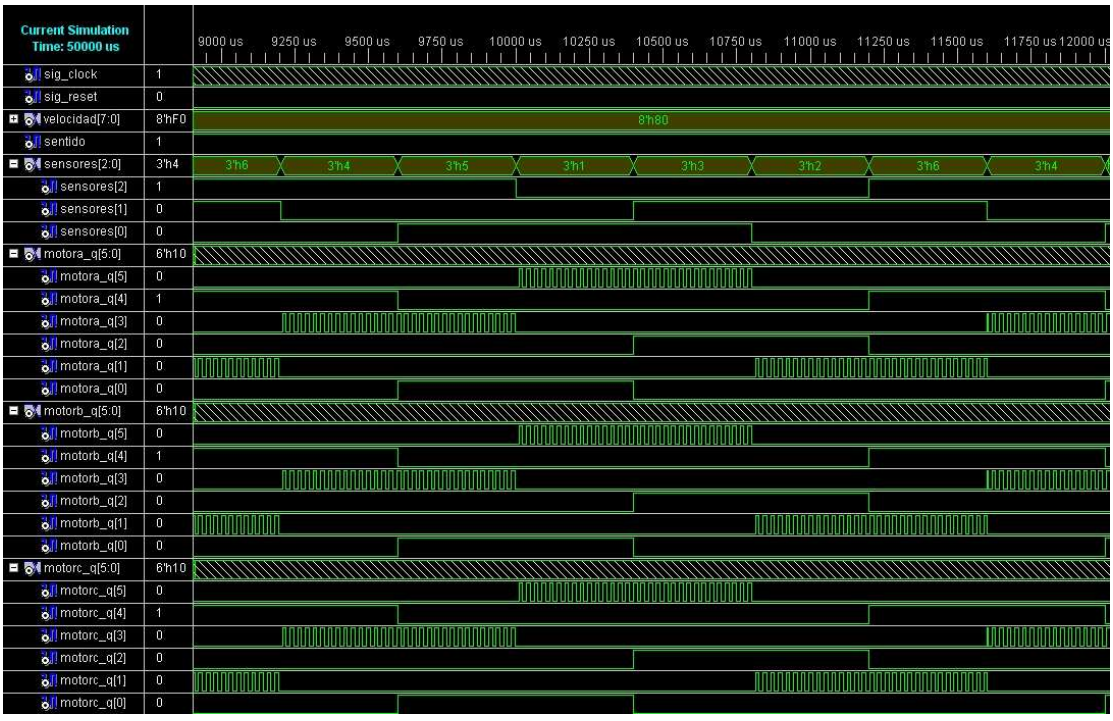
```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 10.266ns (Maximum Frequency: 97.409MHz)
   Minimum input arrival time before clock: 3.627ns
   Maximum output required time after clock: 5.531ns
   Maximum combinational path delay: No path found
```

Resultados de la simulación

Puede observarse el estado de los seis transistores de cada motor, y como los mismos respetan la tabla de verdad de acuerdo al sentido de giro indicado.
Se ve como tres de los transistores siguen a los sensores hall, mientras que los otros tres también siguen a los sensores, pero con la modulacion pwm.





8- Conclusiones:

Se ha desarrollado un Sistema de Control y Simulación del robot propuesto RT2-Apolo mediante la aplicación del Framework de trabajo Microsoft Robotics. Este sistema se encarga de gestionar los movimientos y los cálculos correspondientes a las cinemática directa e inversa, así como también generar la trayectoria necesaria para cumplir con el recorrido en el tiempo dispuesto. Como Modo de programación se ha optado por una interfaz gráfica mas amigable en vez de la realización de un parser y compilador, aprovechando las ventajas de contar con un entorno visual.
El simulador desarrollado nos permite realizar tareas de prueba y error sin la necesidad de conectar directamente el equipo, de manera de poder probar secuencias de movimiento y algoritmos de generación de trayectorias sin poner en peligro los bienes físicos. La conexión con el hardware exterior esta desarrollada mediante el servicio genérico de puerto serie que transmite al módulo FPGA-VHDL desarrollado especialmente para este proyecto, que lee los datos que envía este sistema y se encarga de controlar las velocidades de los 6 motores dispuestos para tal fin.

9-Bibliografia

Fundamentos de la robótica – Barrientos, Peín y Aracil – McGraw Hill 2001
Open-Source Robotics and Process Control Cookbook, Lewin, Edwards – Elsevier 2005
Real Time and Embedded Computing Systems and Applications – Cheng – Springer
Programming Microsoft Robotics - Sara Morgan – McGrawHill 2007