DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA ABIERTA Y DE BAJO COSTE PARA BRAZOS ROBOT EN SERIE

Autor
JUAN CALVO MORA
(cjuanmora@gmail.com)

Director: RANKO ZOTOVIC STANISIC
(rzotovic@isa.upv.es)

TRABAJO FIN DE MÁSTER
DE AUTOMÁTICA E INFORMÁTICA INDUSTRIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Valencia, 24 de octubre de 2020.

Agradecimientos

Un trabajo de investigación nunca sería posible sin el compromiso de un grupo de personas que, ofreciendo su aportación personal, ya sea académica como sentimental, se preocupen por guiar, animar o motivar al autor.

He tenido el enorme gusto de cursar el máster de Automática e Informática Industrial en la Universitat Politècnica de València (UPV), donde he encontrado la oportunidad, tanto de fortalecer mis ya previos conocimientos, como de adquirir otros de nuevos. He requerido del conjunto de todas estas competencias para la finalización de este Trabajo Fin de Máster, el cual ha estado ha estado tutorizado por el Dr. Ranko Zotovic, quien me ha ayudado sobre todo a organizar y movilizar todas mis capacidades para dicha realización. Así mismo, me encantaría hacer una especial mención al Dr. Juan José Serrano Martín por haber resuelto todas las posibles dudas que se me han ido presentando.

Tanto la familia como los amigos han jugado, en general, un papel realmente esencial en el progreso de este proyecto. Quisiera destacar un tipo de aportación que suele pasar desapercibida: la motivacional y sentimental. Mi familia me ha apoyado permanente e incondicionalmente, una actitud que me ha permitido superar gran parte de los obstáculos que tanto caracterizan la elaboración de un trabajo de suma importancia.

Resumen

Este proyecto se centra en el diseño e implementación de una arquitectura abierta y de bajo coste para brazos robot en serie. La funcionalidad de este algoritmo es que, dado un hardware específico, el usuario pueda utilizar la librería donde pueda especificar por ejemplo matriz de inercias, masas de cada eslabón …; para posteriormente dar solución al robot que se quiera controlar.

Para dar solución a este proyecto se describirán los principales bloques que conforman el diseño integro de este trabajo. Por lo tanto, se explicarán las distintas características que se han estudiado para la resolución e implementación del algoritmo. Donde mostrará las propiedades cinemáticas y dinámicas que conforman el robot y los distintos controladores aplicados.

El algoritmo desarrollado se implementa en la placa de desarrollo *Discovery* STM322f407 de STMicroelectronics.

Palabras clave: Newton-Euler · Trayectorias · STM32

Síntesi

Aquest projecte s'interessa pel disseny i implementació d'una arquitectura oberta i de baix cost per a braços robòtics en sèrie. L'objectiu d'aquest algoritme és que, tenint en compte un hardware específic, l'usuari puga utilitzar la llibreria per què puga especificar, per exemple, la matriu d'inèrcies, la massa de cada eslavó (entre d'altres), aplicables posteriorment al robot que es desitge utilitzar.

Es descriuran els principals components en què s'organitza el disseny del treball. Seguint doncs aquest propòsit, s'explicaran les diferents característiques que s'han estudiat per la resolució i implementació de l'algoritme, pel qual es detallaran les propietats cinemàtiques i dinàmiques constitutives del robot, així com també els varis controladors aplicats.

L'algoritme desenvolupat s'implementa amb la placa de desenvolupament Discovery STM322f407 de STMicroelectronics.

Paraules clau: Newton-Euler · Trajectòries · STM32

Abstract

This project tackles the design and implementation of an open and rentable architecture for serial robotic arms. The aim of the algorithm is, according to a specific hardware, to allow users to use a library, for which anyone will be able to determine a particular inertia matrix, or even, for instance, the mass related to a specific link. All these features will then be applied to our particular robot.

The main components of this concrete design will be rigorously detailed, touching on such characteristics as cinematic and dynamic ones, but also the specifics of the implemented controllers.

The STMicroelectronics' Discovery STM322f407 development board has been the selected one for our actual algorithm.

Keywords: Newton-Euler · Trajectory · STM32

Contenido

Tabla de Ilustraciones

1. Introducción y Justificación

Este proyecto sigue la normativa del Trabajo Fin de Máster. La razón por la que se ha decidido ampliar los conocimientos en este ámbito está principalmente relacionada con mis motivaciones personales. La innovación tecnológica es, y ha sido siempre, mi elección por excelencia.

Para este trabajo en concreto, se ha optado por la implementación de una librería que sirva de base para introducir las características del robot para cualquier persona con conocimientos previos de C++. Se busca que las funciones y características presentes en esta puedan ser utilizadas para ejecutar un robot en serie de N dimensiones. La preferencia por los robots en serie se debe a que son los más utilizados en el dominio industrial, cuyos usos se extienden desde el pulido hasta la soldadura por punto.

En segundo lugar, la voluntad de desarrollar una arquitectura de bajo coste, como una alternativa a los robots comerciales existentes, ha sido también un punto a considerar en la elección del tema.

La información se organiza en un prólogo, que servirá como una introducción básica al mundo de la robótica; los objetivos; el cuerpo del proyecto (en el que se expondrán, de forma detallada, cada una de las características con las que poder definir el robot) y, finalmente, en una conclusión, la cual permitirá de evaluar si los objetivos planteados inicialmente se han cumplido, o no, y en qué medida.

2. Objetivos

El objetivo principal de este proyecto es el diseño e implementación de una arquitectura abierta y de bajo coste para brazos robot en serie, concretamente:

1. El estudio y la implementación sobre la cinemática de los robots:

   a. Cinemática directa e inversa.
   b. Jacobiana directa e inversa.

2. El estudio y la implementación de un algoritmo para los comportamientos dinámicos:

   a. Algoritmo de Newton Euler.
   b. Cálculo de pares de gravedad.

3. El estudio y la implementación del movimiento de los robots:

   a. Interpoladores Spline.
   b. Interpolador SCurve.
   c. Interpolador trapezoidal.

4. El estudio y la implementación de los distintos controles dinámicos.

5. Estudio y la implementación de comandos de movimiento:

   a. Movimiento del robot siguiendo una trayectoria lineal (MoveL).
   b. Movimiento del robot mediante un movimiento de ejes (MoveJ).

6. Implementación de distintos protocolos de comunicación para el uso de los posibles sensores, control de motores, etc.

7. La implementación de una arquitectura software genérica que permita desarrollar y rediseñar futuros proyectos.

Con este propósito, se programará las distintas clases mediante C++ y se hará uso de la tarjeta de desarrollo *Discovery*.

3. Matrices homogéneas y vectores

Un punto viene definido en un espacio tridimensional por su posición, pero en el caso de un sólido rígido, además de la posición, también viene determinado por la orientación. Por lo tanto, en el caso de un robot se tiene que determinar la posición y la orientación del extremo del robot o *End Effector*[1]. Si, por ejemplo, se dispone de un sistema robótico que tenga que realizar un *pick and place*[2] de una pieza u objeto, se habrá de determinar la posición y la orientación del extremo del robot para recoger el objeto y para hacer lo mismo, posteriormente, a la hora de dejar la pieza en el destino final.

En un espacio cartesiano tridimensional la posición viene determinada por tres coordenadas cartesianas (x, y, z) y la orientación por tres grados de libertad o tres componentes linealmente independientes.

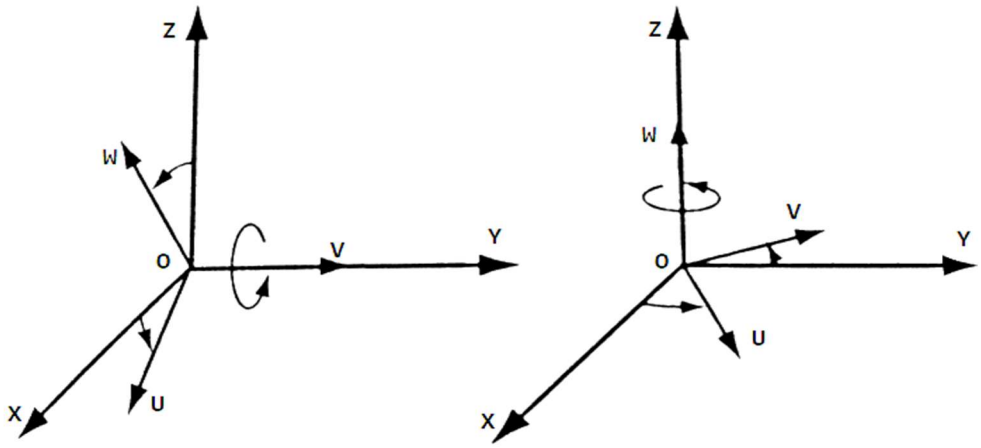

Ilustración 1. Rotación con respecto a OX del sistema de coordenadas XYZ.

Las matrices algébricas juegan un papel muy importante en el desarrollo de sistemas robóticos. Las matrices de rotación en la robótica permiten describir las

[1] Se trata de un dispositivo ubicado en el final de un brazo robot, diseñado para desempeñar una tarea concreta. En función de la morfología del terminal, será específico para realizar una tarea u otra, como por ejemplo en la manipulación de objetos, el ensamblaje, la soldadura, etc.
[2] Se refiere al hecho de que un robot sea capaz de recoger una pieza y colocarla en una nueva ubicación. Este proceso es ampliamente utilizado, ya que acelera los procesos de producción, lo que contribuye a un augmento de la tasa de producción. Una de las herramientas más utilizadas en el extremo del robot para este proceso son las ventosas (a partir del efecto Venturi).

orientaciones de un eje de coordenadas con respecto a otro, debiéndose asimismo a la comodidad en el uso del álgebra matricial.

Si se analiza la *Ilustración 1* se observa que se ha girado o rotado en X un ángulo α con respecto al sistema de coordenadas XYZ. Por lo tanto, si esta rotación se traduce a una matriz de tres por tres, se obtiene la siguiente ecuación:

$$Rot(X,\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sen\alpha \\ 0 & sen\alpha & cos\alpha \end{bmatrix} \quad (1)$$

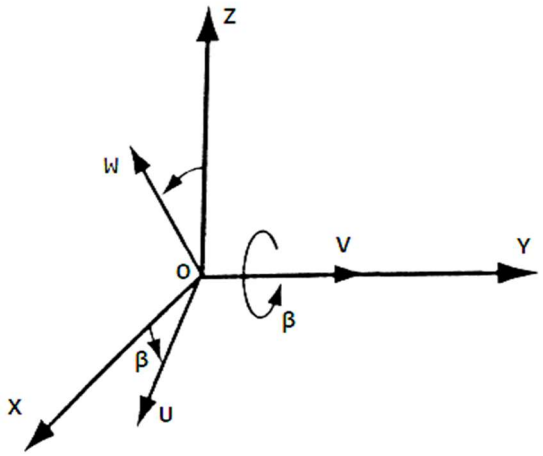

Ilustración 2. Rotación con respecto a OY del sistema de coordenadas XYZ.

De igual forma, en la *Ilustración 2* se muestra la rotación en el eje Y con un ángulo β con respecto al sistema de coordenadas XYZ.

$$Rot(Y,\beta) = \begin{bmatrix} cos\beta & 0 & sen\beta \\ 0 & 1 & 0 \\ -sen\beta & 0 & cos\beta \end{bmatrix} \quad (2)$$

Finalmente, la matriz de rotación para una rotación en el eje Z con un ángulo γ con respecto al sistema de coordenadas será:

$$Rot(Z,\gamma) = \begin{bmatrix} cos\gamma & -sen\gamma & 0 \\ sen\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3)$$

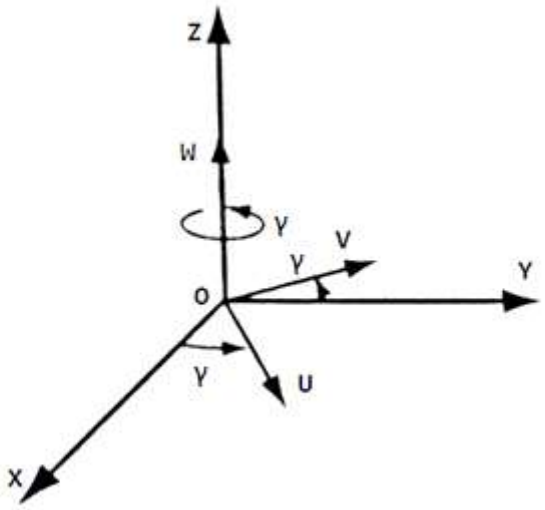

Ilustración 3. Rotación con respecto a OZ del sistema de coordenadas XYZ.

Este proyecto se desarrollará mediante el uso de matrices, en vez de cuaterniones[3].

Para representar de forma matricial la posición y orientación de un objeto se hace uso de las matrices homogéneas. Las matrices de transformación homogénea se tratan de matrices 4x4 que representan la transformación de un vector de coordenadas de un sistema a otro. Por lo tanto, estas matrices se pueden identificar por cuatro submatrices: la primera es una matriz de rotación 3x3 (R), un vector de posición p, una submatriz f que representa la perspectiva, y finalmente, una submatriz w que representa el escalado global.

$$T = \begin{bmatrix} R_{3x3} & p_{3x1} \\ f_{1x3} & w \end{bmatrix} \qquad (4)$$

Las matrices son muy importantes en las aplicaciones robóticas, pues permiten el cambio de un sistema de coordenadas a otro (a partir del uso de las matrices homogéneas). Es por esta razón que en este proyecto se ha programado una clase *matrix* para el desarrollo de los posteriores apartados. Además, también se ha programado una clase

[3] Es una notación matemática que sirve para representar orientaciones y rotaciones de objetos en tres dimensiones

Vector, cuyo uso facilitará que el código sea más claro y sencillo a la hora de depurar el programa.

```
template<class T>
class Vector {
    private:
        int dimension;
        static int objectCount;
        T *n;
    public:
        Vector();
        Vector(int dimension);
        ~Vector();
    int getDimension();
    void PrintResult();
    Vector<T> crossProduct(Vector<T> const& VectorB);


    Vector& operator=(const Vector &other );

    Vector& operator=(T param[]);

    Vector operator+(Vector<T> other);
    Vector operator-(Vector<T>& other);
    T operator*(Vector<T>& other);
    Vector operator*(T factor);
    T& operator()(int dim);
};
```

El uso de la clase *matrix* permite la creación de objetos de tipo matriz de nxm dimensiones, así como el uso de las operaciones que las conforman.

```
template<class T>
class Matrix{
protected:
    int rows;
    int columns;
    static int objectCount;
    T **n;

public:
    Matrix();
    Matrix(int rows,int columns);
    ~Matrix();

    void PrintResult();
    void Reconfigurate(int rows,int columns);
    T Determinant();
    Matrix<T> Cofactor();
    Matrix<T> Inverse();
    Matrix<T> PseudoInverse();
    Matrix<T> Transpose();
```

```
    Matrix& operator=(const Matrix &other );
    Matrix& operator=(const T param[][2]);
    Matrix& operator=(const T param[][3]);
    Matrix& operator=(const T param[][4]);
    Matrix operator+(const Matrix &other);
    Matrix operator*(const Matrix& other);
    Vector<T> operator*(Vector<T>& other);
    T& operator()(int row, int col);

};
```

4. Ángulos de Euler

Los ángulos de Euler permiten describir la orientación de un sólido rígido con respecto a un sistema fijo de coordenadas; de esta manera, se pueden definir mediante una composición de rotaciones, ya que las tres rotaciones elementales sobre el eje de coordenadas XYZ permiten alcanzar cualquier orientación.

La descripción de la orientación del sistema de coordenadas OX'''Y'''Z''' con respecto al sistema OXYZ se puede detallar mediante los tres ángulos de Euler: φ, θ, ψ; que representan los valores de giro a realizar sobre los tres ejes, de manera que girando el sistema OXYZ con esos valores se obtenga el sistema OX'''Y'''Z'''.

Existen doce posibles secuencias para la rotación, siendo:

Los números 1,2,3 representan los ángulos correspondientes a la primera (φ), segunda (θ) y tercera (ψ) rotación.

Las letras X,Y,Z representan las matrices de rotación de los ejes X,Y o Z.

Ángulos de Euler $X_1Z_2X_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OX, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OZ', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OX'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_2 & -c_3 \cdot s_2 & s_2 \cdot s_3 \\ c_1 & c_1 \cdot c_2 \cdot c_3 - s_1 \cdot s_3 & -c_3 \cdot s_1 - c_1 \cdot c_2 \cdot s_1 \\ s_1 & c_1 \cdot s_3 + c_2 \cdot c_3 \cdot s_1 & c_1 \cdot c_3 - c_2 \cdot s_1 \cdot s_3 \end{bmatrix} \quad (5)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerXZX(float thetaX1,float thet
aZ2,float thetaX3){
    Clone(RotXInner(thetaX1)*RotZInner(thetaZ2)*RotXInner(thetaX3));
    return *this;
}
```

Ángulos de Euler $X_1Y_2X_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OX, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OY', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OX'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_2 & s_2 \cdot s_3 & c_3 \cdot s_2 \\ s_1 \cdot s_2 & c_1 \cdot c_3 - c_2 \cdot s_1 \cdot s_3 & -c_1 \cdot s_3 - c_2 \cdot c_3 \cdot s_1 \\ -c_1 \cdot s_2 & c_3 \cdot s_1 + c_1 \cdot c_2 \cdot s_3 & c_1 \cdot c_2 \cdot c_3 - s_1 \cdot s_3 \end{bmatrix} \quad (6)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerXYX(float thetaX1,float thet
aY2,float thetaX3){
    Clone(RotZInner(thetaX1)*RotXInner(thetaY2)*RotZInner(thetaX3));
    return *this;
}
```

Ángulos de Euler $Y_1X_2Y_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OY, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OX', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OY'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_3 - c_2 \cdot s_1 \cdot s_3 & s_1 \cdot s_2 & c_1 \cdot s_3 + c_2 \cdot c_3 \cdot s_1 \\ s_2 \cdot s_3 & c_2 & -c_3 \cdot s_2 \\ -c_3 \cdot s_1 - c_1 \cdot c_2 \cdot s_3 & c_1 \cdot s2 & c_1 \cdot c_2 \cdot c_3 - s_1 \cdot s_3 \end{bmatrix} \quad (7)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerYXY(float thetaY1,float thet
aX2,float thetaY3){
    Clone (RotZInner(thetaY1)*RotXInner(thetaX2)*RotZInner(thetaY3));
    return *this;
}
```

Ángulos de Euler $Y_1Z_2Y_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OY, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OZ', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OY'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_2 \cdot c_3 - s_1 \cdot s_3 & -c_1 \cdot s_2 & c_3 \cdot s_1 + c_1 \cdot c_2 \cdot s_3 \\ c_3 \cdot s_2 & c_2 & s_2 \cdot s_3 \\ -c_1 \cdot s_3 - c_2 \cdot c_3 \cdot s_1 & s_1 \cdot s_2 & c_1 \cdot c_3 - c_2 \cdot s_1 \cdot s_3 \end{bmatrix} \quad (8)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerYZY(float thetaY1,float thet
aZ2,float thetaY3){
    Clone (RotZInner(thetaY1)*RotXInner(thetaZ2)*RotZInner(thetaY3));
    return *this;
}
```

Ángulos de Euler $Z_1Y_2Z_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OZ, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OY', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OZ'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_2 \cdot c_3 - s_1 \cdot s_3 & -c_3 \cdot s_1 - c_1 \cdot c_2 \cdot s_3 & c_1 \cdot s_2 \\ c_1 \cdot s_3 + c_2 \cdot c_3 \cdot s_1 & c_1 \cdot c_3 - c_2 \cdot s_1 \cdot s_3 & s_1 \cdot s_2 \\ -c_2 \cdot s_2 & s_2 \cdot s_3 & c_2 \end{bmatrix} \quad (9)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerZYZ(float thetaZ1,float thet
aY2,float thetaZ3){
    Clone (RotZInner(thetaZ1)*RotXInner(thetaY2)*RotZInner(thetaZ3));
    return *this;
}
```

Ángulos de Euler $Z_1X_2Z_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OZ, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OX', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OZ'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_3 - c_2 \cdot s_1 \cdot s_3 & -c_1 \cdot s_3 - c_2 \cdot c_3 \cdot s_1 & s_1 \cdot s_2 \\ c_3 \cdot s_1 + c_1 \cdot c_2 \cdot s_3 & c_1 \cdot c_2 \cdot c_3 - s_1 \cdot s_3 & -c_1 \cdot s_2 \\ s_2 \cdot s_3 & c_3 \cdot s_2 & c_2 \end{bmatrix} \quad (10)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerZXZ(float thetaZ1,float thet
aX2,float thetaZ3){
    Clone (RotZInner(thetaZ1)*RotXInner(thetaX2)*RotZInner(thetaZ3));
    return *this;
}
```

Ángulos de Tait-Bryan $X_1Z_2Y_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OX, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OZ', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OY'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_2 \cdot c_3 & -s_2 & c_2 \cdot s_3 \\ s_1 \cdot s_3 + c_1 \cdot c_3 \cdot s_2 & c_1 \cdot c_2 & c_1 \cdot s_2 \cdot s_3 - c_3 \cdot s_1 \\ c_3 \cdot s_1 \cdot s_2 - c_1 \cdot s_3 & c_2 \cdot s_1 & c_1 \cdot c_3 + s_1 \cdot s_2 \cdot s_3 \end{bmatrix} \quad (11)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerXZY(float thetaX1,float thet
aZ2,float thetaY3){
    Clone (RotZInner(thetaX1)*RotXInner(thetaZ2)*RotZInner(thetaY3));
    return *this;
}
```

Ángulos de Tait-Bryan $X_1Y_2Z_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OX, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OY', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OZ'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_2 \cdot c_3 & -c_2 \cdot s_3 & s_2 \\ c_1 \cdot s_3 + c_3 \cdot s_1 \cdot s_2 & c_1 \cdot c_3 - s_1 \cdot s_2 \cdot s_3 & -c_2 \cdot s_1 \\ s_1 \cdot s_3 - c_1 \cdot c_3 \cdot s_2 & c_3 \cdot s_1 + c_1 \cdot s_2 \cdot s_3 & c_1 \cdot c_2 \end{bmatrix} \quad (12)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerXYZ(float thetaX1,float thet
aY2,float thetaZ3){
    Clone (RotZInner(thetaX1)*RotXInner(thetaY2)*RotZInner(thetaZ3));
    return *this;
}
```

Ángulos de Tait-Bryan $Y_1X_2Z_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OY, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OX', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OZ'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_3 + s_1 \cdot s_2 \cdot s_3 & c_3 \cdot s_1 \cdot s_2 - c_1 \cdot s_3 & c_2 \cdot s_1 \\ c_2 \cdot s_3 & c_2 \cdot c_3 & -s_2 \\ c_1 \cdot s_2 \cdot s_3 - c_3 \cdot s_1 & c_1 \cdot c_3 \cdot s_2 + s_1 \cdot s_3 & c_1 \cdot c_2 \end{bmatrix} \quad (13)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerYXZ(float thetaY1,float thet
aX2,float thetaZ3){
    Clone (RotZInner(thetaY1)*RotXInner(thetaX2)*RotZInner(thetaZ3));
    return *this;
}
```

Ángulos de Tait-Bryan $Y_1Z_2X_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OY, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OZ', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OX'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_2 & s_1 \cdot s_3 - c_1 \cdot c_3 \cdot s_2 & c_3 \cdot s_1 + c_1 \cdot s_2 \cdot s_3 \\ s_2 & c_2 \cdot c_3 & -c_2 \cdot s_3 \\ -c_2 \cdot s_1 & c_1 \cdot s_3 + c_3 \cdot s_1 \cdot s_2 & c_1 \cdot c_3 - s_1 \cdot s_2 \cdot s_3 \end{bmatrix} \quad (14)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerYZX(float thetaY1,float thet
aZ2,float thetaX3){
    Clone (RotZInner(thetaY1)*RotXInner(thetaZ2)*RotZInner(thetaX3));
}
```

Ángulos de Tait-Bryan $Z_1Y_2X_3$:

Giro del sistema XYZ un ángulo φ con respecto al eje OZ, convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z' un ángulo θ con respecto al eje OY', convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'' un ángulo ψ con respecto al eje OX'', convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_2 & c_1 \cdot s_2 \cdot s_3 - c_3 \cdot s_1 & s_1 \cdot s_3 + c_1 \cdot c_3 \cdot s_2 \\ c_2 \cdot s_1 & c_1 \cdot c_3 + s_1 \cdot s_2 \cdot s_3 & c_3 \cdot s_1 \cdot s_2 - c_1 \cdot s_3 \\ -s_2 & c_2 \cdot s_3 & c_2 \cdot c_3 \end{bmatrix} \quad (15)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerZYX(float thetaZ1,float thet
aY2,float thetaX3){
    Clone (RotZInner(thetaZ1)*RotXInner(thetaY2)*RotZInner(thetaX3));
    return *this;
}
```

Otro método equivalente al anterior, también muy utilizado en robótica, son los giros sobre los ejes fijos, denominados *Yaw, Pitch and Roll*; consiste en una configuración comúnmente utilizada entre las que se aplican sobre los ejes del sistema fijo. De esta manera, la orientación viene definida por:

1. Giro del sistema XYZ, un ángulo φ con respecto al eje OX (*Yaw*).
2. Giro del sistema XYZ, un ángulo θ con respecto al eje OY (*Pitch*).
3. Giro del sistema XYZ, un ángulo ψ con respecto al eje OZ (*Roll*).

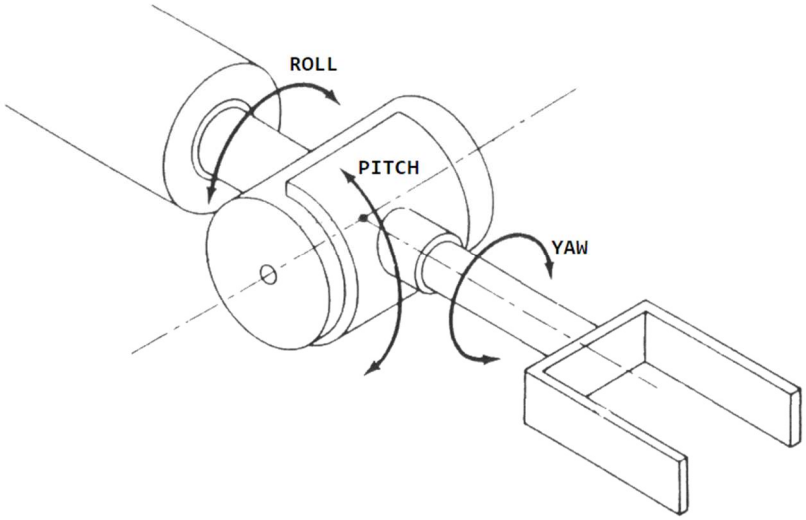

Ilustración 4. Representación de un terminal de robot indicando el Roll, Pitch y Yaw.

Esta opción es ampliamente utilizada para posicionar la orientación de la herramienta de un robot. De este modo, para garantizar un uso más sencillo de la librería implementada para el usuario, se ha desarrollado una clase llamada *End_Effector,* dentro de la cual el usuario podrá introducir todas las características que definan dicha parte del robot (posición: x, y, z; orientación: *Roll*, *Pitch*, *Yaw*).

```
class End_Effector {
public:
    //Position
    float x;
    float y;
    float z;

    //Orientation
    float roll;
    float pitch;
    float yaw;

    End_Effector();
};
```

Ángulos de Tait-Bryan $Z_1X_2Y_3$:

Giro del sistema XYZ, un ángulo φ con respecto al eje OZ convirtiéndose en OX'Y'Z'.

Giro del sistema OX'Y'Z', un ángulo θ con respecto al eje OX' convirtiéndose en OX''Y''Z''.

Giro del sistema OX''Y''Z'', un ángulo ψ con respecto al eje OY'' convirtiéndose en OX'''Y'''Z'''.

$$Rot = \begin{bmatrix} c_1 \cdot c_3 - s_1 \cdot s_2 \cdot s_3 & -c_2 \cdot s_1 & c_1 \cdot s_3 + c_3 \cdot s_1 \cdot s_2 \\ c_3 \cdot s_1 + c_1 \cdot s_2 \cdot s_3 & c_1 \cdot c_2 & s_1 \cdot s_3 - c_1 \cdot c_3 \cdot s_2 \\ -c_2 \cdot s_3 & s_2 & c_2 \cdot c_3 \end{bmatrix} \quad (16)$$

Cuya función es:

```
Matrix<float>& TransformationMatrix::EulerZXY(float thetaZ1,float thet
aX2,float thetaY3){
    Clone (RotZInner(thetaZ1)*RotXInner(thetaX2)*RotZInner(thetaY3));
    return *this;
}
```

5. Cinemática

La cinemática aplicada a los robots describe un movimiento de los cuerpos en un mecanismo robótico sin consideras las fuerzas. Así, la cinemática es un tema muy importante en el diseño del movimiento, en el análisis y en el control y diseño de los robots.

Para la resolución de la cinemática de un brazo robot, existen dos cuestiones a considerar: el problema de cinemática directa, el cual consiste en determinar la posición y orientación final del robot a partir de las posiciones en todas las articulaciones, y el problema de cinemática inversa, que correspondería al inverso de lo anterior, es decir, el cual consiste en definir las posiciones de todas las articulaciones, dada la posición y orientación final del robot.

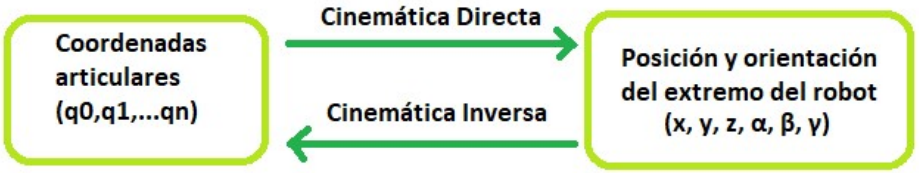

Ilustración 5. Diagrama de la relación entre las cinemáticas directa e inversa.

5.1. Cinemática Directa

La cinemática directa encuentra la posición final relativa a la base, dadas las posiciones de todas las articulaciones y los parámetros de enlace entre cada articulación.

La resolución del problema de cinemática directa calcula la transformación entre una referencia entre el punto final y el punto fijo en la base, por ejemplo, entre la herramienta y la estación.

Un robot de N grados de libertad está formado por N eslabones que están unidos con N articulaciones. Por lo tanto, para cada eslabón se le puede asociar una matriz de transformación homogénea (${}^{i-1}T_i$), la cual relaciona la articulación anterior (i-1) con la

posterior (i). Esta matriz de transformación presenta la orientación y la posición relativa para los dos eslabones consecutivos del robot.

En la siguiente ecuación se muestra la matriz de transformación homogénea para un robot de cuatro grados de libertad, en la que se relaciona el eslabón cero con el eslabón cuatro.

$$ {}^{0}T_{4} = {}^{0}T_{1} \cdot {}^{1}T_{2} \cdot {}^{2}T_{3} \cdot {}^{3}T_{4} \qquad (17) $$

Para poder encontrar las matrices homogéneas, se hace uso de la representación de Denavit-Hartenberg (D-H). Las matrices homogéneas de cada eslabón se pueden definir gracias a cuatro transformaciones básicas:

1- Rotación del eje z_{i-1}, un ángulo θ_i.
2- Translación a lo largo de z_{i-1}, una distancia d_i.
3- Translación a lo largo de x_i, una distancia a_i.
4- Rotación alrededor del eje x_i, un ángulo α_i.

Como el producto de las matrices no es conmutativo, en la siguiente ecuación se muestra las transformaciones que se deben realizar:

$$ {}^{i-1}T_{i} = Rot(z,\theta_i) \cdot T(0,0,d_i) \cdot T(a_i,0,0) \cdot Rot(x,\alpha_i) \qquad (18) $$

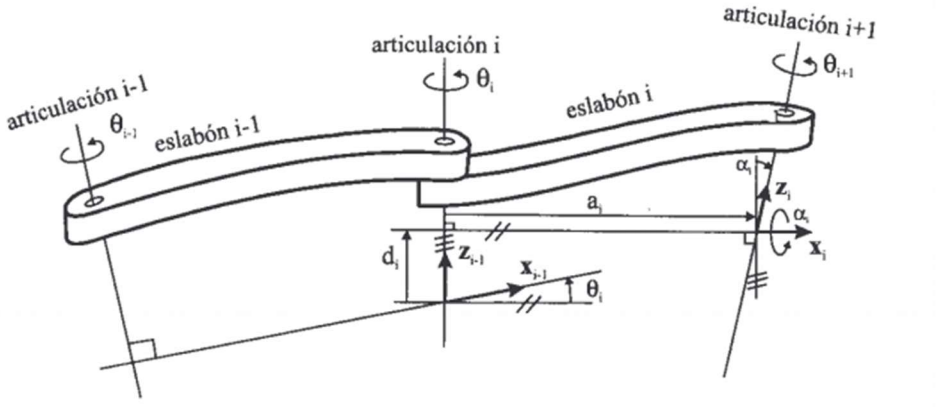

Ilustración 6. Parámetros D-H para un eslabón. [Fuente: Barrientos]

Se puede obtener así la matriz de transformación homogénea, capaz de relacionar los eslabones anterior y posterior:

$$
{}^{i-1}T_i = \begin{bmatrix} C\theta_i & -S\theta_i & 0 & 0 \\ S\theta_i & C\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot
$$

$$
\cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\alpha_i & -S\alpha_i & 0 \\ 0 & S\alpha_i & C\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =
$$

$$
= \begin{bmatrix} C\theta_i & -S\theta_i C\alpha_i & S\theta_i S\alpha_i & a_i C\theta_i \\ S\theta_i & C\theta_i C\alpha_i & -C\theta_i S\alpha_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (19)
$$

Los cuatro parámetros que definen el problema de Denavit-Hartenberg permiten desarrollar el algoritmo para la resolución del problema cinemático directo [1]:

DH1. Numerar los eslabones comenzando por 1 (primer eslabón móvil de la cadena) y acabando por n (último eslabón móvil). Se denominará eslabón 0 a la base fija del robot.

DH2. Numerar cada articulación comenzando por 1 (la correspondiente al primer grado de libertad) y acabando por n.

DH3. Localizar el eje Ji de cada articulación. Si esta es rotativa, el eje será su propio eje de giro; si es prismática, será el eje a lo largo del cual se producirá el desplazamiento.

DH4. Para i de 0 a n-1, situar el eje zi, sobre el eje de la articulación i+1. Es decir, z0 va sobre el eje de la 1ª articulación, z1 va sobre el eje de la 2ª, etc.

DH5. Situar el origen O0 del sistema de la base (S0) en cualquier punto del eje z0. Los ejes x0 e y0 se situarán como se desee (siempre formando un sistema de referencia con z0).

DH6. Para i de 1 a n-1, situar el origen Oi del sistema de referencia (Si) asociado al eslabón i en la intersección del eje zi con la línea perpendicular común a zi-1 y zi.

Si ambos ejes se cortasen, se situaría Oi en el punto de corte; si fuesen paralelos, Oi se situaría sobre zi en cualquier lugar.

DH7. Situar xi en la línea perpendicular común a zi-1 y zi. Cada xi va en la dirección de zi-1 hacia zi.

DH8. Situar yi de modo que forme un sistema de referencia dextrógiro (regla de la mano derecha) con xi y zi.

DH9. Sistema del extremo del robot (el n-ésimo sistema XYZ):

Se coloca en el extremo del robot (herramienta).

Eje zn paralelo a zn-1 y ejes xn e yn en cualquier dirección válida.

DH10. Obtener θi como el ángulo de giro en torno a zi-1 para que xi-1 sea paralelo a xi (de xi-1 a xi siguiendo la regla de la mano derecha; parámetro variable en articulaciones giratorias).

DH11. Obtener di como la distancia medida a lo largo de zi-1 para que xi-1 y xi queden alineados (parámetro variable en articulaciones prismáticas).

DH12. Obtener ai como la distancia medida a lo largo del nuevo xi-1, (es decir, xi) desde zi-1 hasta zi.

DH13. Obtener αi como el ángulo medido en torno al nuevo del nuevo xi-1 (es decir, xi) desde zi-1 hasta zi.

DH14. Obtener las matrices de transformación ${}^{i-1}T_i$.

DH15. Obtener la matriz que relaciona el sistema de la base de coordenadas con el extremo del robot.

Si se observa la matriz homogénea ${}^{0}T_4$ de la *Ecuación 17*, se puede definir la orientación y la posición del extremo a la base.

En este proyecto se ha implementado en C++ un método llamado *ForwardKinematics*, el cual, en función de los grados de libertad del robot, calcula la tabla de Denavit-Hartenberg. A partir de eso, se obtienen las matrices de transformación homogéneas de cada eslabón. Dichos parámetros se guardan en la clase Robot, que almacena toda la información del robot.

```
void Robot::ForwardKinematics(const float thetaOffset[]){
```

```
    float STheta,CTheta,SAlpha,CAlpha,theta,d;
    for (int i=0; i<_RobotDOF;i++) {
        if ( _Linker[i].Type == Link::REVOLUTION) {
            theta = thetaOffset[i];
            d =  _Linker[i].d;
        }else{
            d =  thetaOffset[i];
            theta =  _Linker[i].theta;
        }
        STheta = sinf(theta);
        CTheta = cosf(theta);
        SAlpha = sinf(  _Linker[i].alpha);
        CAlpha = cosf(  _Linker[i].alpha);
        if( (CAlpha>-0.000001F)&&(CAlpha<0.000001F)){
            CAlpha = 0;
        }
        if( (SAlpha>-0.000001F)&&(SAlpha<0.000001F)){
            SAlpha = 0;
        }
        if( (STheta>-0.000001F)&&(STheta<0.000001F)){
            STheta = 0;
        }
        if( (CTheta>-0.000001F)&&(CTheta<0.000001F)){
            CTheta = 0;
        }
        for (int x = 0; x < 4; x++) {
            for (int y = 0; y < 4; y++) {
                if (x == 0 && y == 0) {
                     _Linker[i] .A0 (x,y)  = CTheta;
                } else if (x == 0 && y == 1) {
                     _Linker[i] .A0  (x,y)  = -1 * CAlpha * STheta;
                } else if (x == 0 && y == 2) {
                     _Linker[i] .A0  (x,y)  = SAlpha * STheta;
                } else if (x == 0 && y == 3) {
                     _Linker[i] .A0  (x,y)  =  _Linker[i] .a * CTheta;
                } else if (x == 1 && y == 0) {
                     _Linker[i] .A0  (x,y)  = STheta;
                } else if (x == 1 && y == 1) {
                     _Linker[i] .A0  (x,y)  = CAlpha * CTheta;
                } else if (x == 1 && y == 2) {
                     _Linker[i] .A0  (x,y)  = -1 * SAlpha * CTheta;
                } else if (x == 1 && y == 3) {
                     _Linker[i] .A0  (x,y)  =  _Linker[i] .a * STheta;
                } else if (x == 2 && y == 1) {
                     _Linker[i] .A0  (x,y)  = SAlpha;
                } else if (x == 2 && y == 2) {
                     _Linker[i] .A0  (x,y)  = CAlpha;
                } else if (x == 2 && y == 3) {
                     _Linker[i] .A0  (x,y)  = d;
                } else if (x == 3 && y == 3) {
                     _Linker[i] .A0  (x,y)  = 1;
                }
                else{
                     _Linker[i] .A0  (x,y) =0;
                }
                if( ( _Linker[i] .A0  (x,y) >-
0.000001F)&&( _Linker[i] .A0  (x,y) )<0.000001F){
                     _Linker[i] .A0  (x,y) =0;
```

```
                        }

                    }
                }
            }
        }
}
```

5.2. Cinemática Inversa

La cinemática inversa es la encargada de obtener la posición de cada una de las articulaciones que definen el robot, a partir de la posición y orientación final del robot.

El problema cinemático directo es posible desarrollarlo a partir de la utilización de matrices de transformación homogénea; en cambio, en el problema cinemático inverso la obtención de las ecuaciones dependerá de la configuración del robot. Por esta razón, resulta óptimo el hecho de encontrar una solución cerrada.

Posteriormente, se tiene que encontrar una relación matemática entre la posición y orientación, para que se pueda obtener así la posición de cada articulación, tal y como se muestra en la siguiente ecuación:

$$qi = f_i(x, y, z, \phi, \theta, \psi)$$
$$i = 1 \dots n.\ D\acute{o}nde\ n = DOF \qquad (20)$$

Esta resolución presenta varias soluciones; en el caso de un robot de dos grados de libertad podremos encontrar dos posibles soluciones: codo hacia arriba y codo hacia abajo. Dependiendo de la configuración del robot y de los límites físicos nos interesará una configuración u otra.

Ilustración 7. A la izquierda se muestra la configuración del robot BCN3D MOVEO con una configuración de codo arriba y a la derecha se observa la configuración codo hacia abajo.

Como nos interesamos en el diseño e implementación de una arquitectura abierta para cualquier brazo robot, en el método de *InverseKinematics* el usuario deberá introducir la cinemática inversa para un determinado robot.

Para hacer uso de dicho método, el usuario creará un objeto de la clase Robot y, posteriormente, llamará al método *InverseKinematics*, pasando un objeto de la clase *End_Effector* y su respectiva configuración. Esta función devolverá un vector de N dimensiones, donde se devolverán las coordenadas articulares.

```
Vector<float> Robot::InverseKinematics(End_Effector* EFF_q0, int conf){
    /*USER SPECIFICATIONS*/
    Vector<float> result (_RobotDOF);
    float Xf = EFF_q0->x;
    float Yf = EFF_q0->y;
    float fi = EFF_q0->pitch;
    float theta = EFF_q0->roll;

    float L1 = _Linker[0].ReturnLenght();
    float L2 = _Linker[1].ReturnLenght();
    float L3 = _Linker[2].ReturnLenght();
    float L4 = _Linker[3].ReturnLenght();

    //Wrist points
    float Cfi = cosf(fi);
    float Sfi = sinf(fi);
    if ((Cfi > -0.000001F) && (Cfi < 0.000001F)) {
        Cfi = 0;
    }
    if ((Sfi > -0.000001F) && (Sfi < 0.000001F)) {
        Sfi = 0;
    }
    float Xm = Xf - L4 * Cfi;
    float Ym = Yf - L4 * Sfi;


    //q4
    result(3) = theta;

    //q2
    float C2 = ((powf(Xm, 2.0F)) + (powf(Ym, 2.0F)) - (powf(L1, 2.0F)) - (po
wf(L2, 2.0F))) / (2 * L1 * L2);
    float S2 = sqrtf(1 - powf(C2, 2.0F));
    float q21 = atan2f(S2, C2);
    float q22 = atan2f(-S2, C2);
    result (1) = q21;

    //q1
    float alpha = atan2f((L2 * sinf(result(1))), (L1 + L2 * cosf(result(1)))
);
    float beta_1 = atan2f(Ym, Xm);
    float beta_2 = atan2f(Ym, -Xm);

    float q1;
```

```
    if (conf == 0) {
        //Elbow up
        q1 = beta_1 - alpha;
    } else {
        //Elbow down
        q1 = beta_2 - alpha;
    }
    result (0)= q1;
    //q3
    result (2) = fi-result(0)-result(1);
    return result;
}
```

6. Jacobiana

La diferenciación con respecto al tiempo de las ecuaciones cinemáticas de posición produce una serie de ecuaciones, de la forma en la que se muestra a continuación:

$$v_N = J(q) \cdot \dot{q} \qquad (21)$$

Donde vn es la velocidad del punto final del robot, J(q) es una matriz de 6·N dimensiones llamada matriz Jacobiana y $\dot{q}$ es el vector de velocidades de cada articulación.

Se puede afirmar que la matriz Jacobiana de un robot relaciona el vector de velocidades de cada articulación con otro vector de velocidades en otro espacio.

Podemos establecer una relación entre las velocidades de las articulaciones con las velocidades lineales y angulares del extremo del robot; en este caso, el parámetro que relaciona ambos espacios se llama matriz Jacobiana geométrica. En ella se nos presentan dos casos:

- Jacobiana geométrica Directa: Dadas las velocidades de las articulaciones, esta permite hallar las velocidades lineales y angulares del extremo del robot.

- Jacobiana geométrica Inversa: Hace posible conocer las velocidades de las articulaciones a partir de las velocidades lineales y angulares del extremo del robot. Se afirma que la inversa de la Jacobiana geométrica directa equivale a la Jacobiana geométrica inversa.

Igualmente, se puede establecer una relación entre las velocidades de las articulaciones y las velocidades de la localización del extremo del robot. Para pasar de un espacio vectorial al otro se emplea la matriz Jacobiana analítica.

6.1. Jacobiana geométrica

La Jacobiana geométrica relaciona las velocidades de las articulaciones con las velocidades lineales (v) y angulares (w) del extremo del robot. De acuerdo con eso, se puede obtener la siguiente expresión:

$$\begin{bmatrix} v_X \\ v_Y \\ v_Z \\ w_X \\ w_Y \\ w_Z \end{bmatrix} = J \cdot \begin{bmatrix} \dot{q}_1 \\ . \\ . \\ . \\ . \\ \dot{q}_N \end{bmatrix} \quad (22)$$

Las velocidades lineales del extremo del robot vendrán dadas por las derivadas de las coordenadas del extremo del robot con respecto al tiempo.

$$v_X = \frac{dp_X}{dt} = \dot{p}_X \quad (23)$$

$$v_Y = \frac{dp_Y}{dt} = \dot{p}_Y \quad (24)$$

$$v_Z = \frac{dp_Z}{dt} = \dot{p}_Z \quad (25)$$

Si se analiza la expresión anterior, las velocidades lineales del extremo del robot equivalen a hacer la derivada con respecto al tiempo de cada matriz de transformación T.

Existen distintas soluciones para determinar de forma numérica la matriz Jacobiana a partir de las matrices de transformación. De este modo, se puede determinar la matriz geométrica de un robot a partir de la siguiente expresión, la cual nos permitirá definir el modelo cinemático del robot:

$$T = \begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (26)$$

El procedimiento utilizado en la programación en C++ de la función *Jacobian* se ha basado en la propagación de las velocidades. Este algoritmo obtiene las columnas de la matriz Jacobiana geométrica que relaciona las velocidades de cada articulación con las velocidades lineales y angulares del punto final del robot, a partir de las matrices homogéneas (T).

$$^{i-1}T_i = \begin{bmatrix} ^{i-1}n_i & ^{i-1}o_i & ^{i-1}a_i & ^{i-1}p_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (27)$$

$$^{0}T_i = \begin{bmatrix} ^{0}n_i & ^{0}o_i & ^{0}a_i & ^{0}p_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (28)$$

Se puede observar que la matriz $^{i-1}T_i$ contiene la información de los vectores directores y del origen del sistema $\{S_i\}$ en la base $\{S_{i-1}\}$. Posteriormente, la matriz $^{0}T_i$ contiene la información de los vectores del sistema en $\{S_i\}$ en la base $\{S_0\}$; (i es solidario a cada eslabón).

También se necesitará conocer el vector unitario $^{0}z_i$, que se obtendrá a partir de los tres primeros elementos de la tercera columna de la matriz homogénea de cada eslabón.

$$^{0}z_i = {}^{0}T_i(1{:}3,3) \quad (29)$$

Se denominará $^{0}p_i$ al vector que va desde el origen del sistema $\{S_i\}$ hasta el extremo del robot.

Como la cuarta columna de la matriz de transformación $^{0}T_n$ contiene la posición en el extremo del robot y la cuarta columna de la matriz de transformación $^{0}T_i$ contiene la posición del origen del sistema $\{S_i\}$ en el sistema $\{S_0\}$, se conseguirá $^{i}p_n$ a partir de la diferencia de $^{0}T_n$ y $^{0}T_i$. Por lo tanto, se obtendrá la siguiente ecuación:

$$^{i}p_n = {}^{0}T_n(1{:}3,4) - {}^{0}T_i(1{:}3,4) \quad (30)$$

Una vez se han obtenido los vectores ${}^0z_i$ y ${}^0p_i$, la matriz Jacobiana se puede expresar como una matriz de 6xn dimensiones, en la que n es el número de grados de libertad expresado en columnas, tal y como se observa en la siguiente ecuación.

$$J = [J_1|J_2|\dots|J_n] \tag{31}$$

Donde J se obtiene a partir de:

$$J_i = \begin{cases} \begin{bmatrix} {}^0z_{i-1} - {}^{i-1}z_n \\ {}^0z_{i-1} \end{bmatrix} & Cuando\ el\ eslabón\ i\ es\ de\ rotación \\ \begin{bmatrix} {}^0z_{i-1} \\ 0 \end{bmatrix} & Cuando\ el\ eslabón\ i\ es\ de\ translación \end{cases} \tag{32}$$

6.2. Jacobiana pseudoinversa

En el caso que se desee hacer la inversa de la Jacobiana, cuando el número de ejes del robot sea inferior a la dimensión del espacio de la tarea, se utilizará la pseudoinversa.

Para precisar la pseudoinversa de la matriz Jacobiana, si esta tiene más filas que columnas, primeramente, habrá que determinar si las columnas son linealmente independientes o no. Si son linealmente independientes se utilizará la siguiente expresión:

$$J^+ = (J^T \cdot J)^{-1} \cdot J^T \tag{33}$$

En caso contrario, si la matriz Jacobiana presenta más columnas que filas, y las filas son linealmente independientes, la pseudoinversa será igual a la siguiente fórmula:

$$J^+ = J^T \cdot (J \cdot J^T)^{-1} \tag{34}$$

Se ha programado el método *PseudoInverse*, perteneciente a la clase *Matrix*, como se especifica a continuación:

```
Matrix<T>  Matrix<T>::PseudoInverse(){
    Matrix<T> trans(this->columns, this->rows);
    Matrix<T> A(this->rows, this->columns);
    A = *this;
    trans = Transpose();
    Matrix<T> result = Matrix<T>(trans.rows, trans.columns);
    if((trans*A).Determinant()!=0){
```

```
        result = ((trans * A).Inverse())*trans;
    }else if((A*trans).Determinant()!=0){
        result = trans *((A*trans).Inverse());
    }else{
        printf("Error Matrix PseudoInverse\n");
    }
    return result;

}
```

7. Dinámica

La dinámica abarca la relación entre las fuerzas que actúan sobre un cuerpo y el movimiento que estas proporcionan, es decir, consiste en la correspondencia entre las actuaciones y las fuerzas de contacto. La dinámica es una parte fundamental en el diseño mecánico, el control y la simulación de brazos robots. De esta manera, un modelo dinámico de un robot relaciona el movimiento del robot con las fuerzas implicadas.

Existen varios algoritmos capaces de realizar la respectiva computación: la dinámica inversa, la dinámica hacia adelante y la matriz de inercia. En este proyecto se ha utilizado el algoritmo computacional de Newton-Euler de manera eficiente para realizar cada uno de estos cálculos en un modelo dinámico de un robot.

Existe otro método de resolución a la dinámica mediante la formulación de Lagrangiana, pero, a diferencia del método elegido, este último consiste en un algoritmo con un alto costo computacional.

7.1. Algoritmo Newton Euler

Las ecuaciones de Newton-Euler describen la combinación de rotaciones y translaciones dinámicas de un cuerpo rígido.

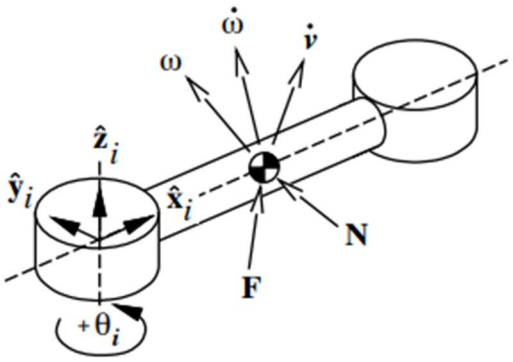

Ilustración 8. Representación de la dinámica en un eslabón.

La obtención del modelo dinámico de un robot se hace a partir del planteamiento del equilibrio de fuerzas de la segunda Ley de Newton, y viene dada también por la cantidad de movimiento, según la Ley de Euler[4].

$$\sum F_i = m_i \cdot \dot{v}_i \qquad (35)$$

$$\sum M_i = I \cdot \dot{\omega}_i + \omega_i \, x \, (I \cdot \omega_i) \qquad (36)$$

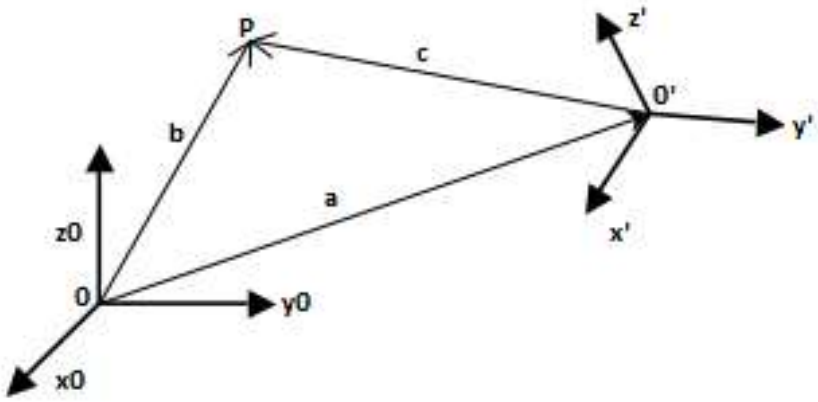

Ilustración 9. Sistema de coordenadas con translación y rotación.

Si analizamos la *Ilustración 9*, en la cual se muestra el sistema de coordenadas *O'*, se observa la presencia de una rotación y una translación en el espacio con respecto al sistema de coordenadas *O*, en el que *a* es el vector que describe el movimiento de del origen *O*; el punto *P* se describe a partir del vector *c* respecto al sistema *O'*. La descripción del punto *P* con respecto a la base del sistema de coordenadas viene dado por la siguiente ecuación:

$$b = a + c \qquad (37)$$

[4] Ley de Euler: La ecuación de Euler es una ecuación diferencial de primer orden que describe la rotación de un cuerpo rígido.

Seguidamente, si esta se deriva con respecto al tiempo, se obtiene la siguiente ecuación:

$$v = \frac{db}{dt} = \frac{da}{dt} + \frac{dc}{dt} = v_a + v_c \qquad (38)$$

Donde v_a es la velocidad del punto *P* respecto el origen de coordenadas *O'* y v_c es la velocidad del origen *O*.

Si se analiza que el punto *P* se desplaza y gira con respecto al sistema de coordenadas *O'*, se puede reformular la *Ecuación 38* de la siguiente forma:

$$v = \frac{db}{dt} = \frac{da}{dt} + \frac{dc}{dt} = \frac{da}{dt} + \left(\frac{dc}{dt} + \omega x c\right) \qquad (39)$$

Se puede apreciar que el desplazamiento del punto *P* se asocia a la velocidad angular, y a la velocidad lineal del punto con respecto al origen *O'*. De igual modo, se pueden escribir la aceleración del sistema estudiado.

$$a = \frac{d^2a}{dt^2} + \frac{d^2c}{dt^2} = a_a + a_c \qquad (40)$$

$$a = \frac{d^2a}{dt^2} + \frac{d^2c}{dt^2} + 2 \cdot \omega x \frac{dc}{dt} + \omega x (w x c) + \frac{d\omega}{dt} x c \qquad (41)$$

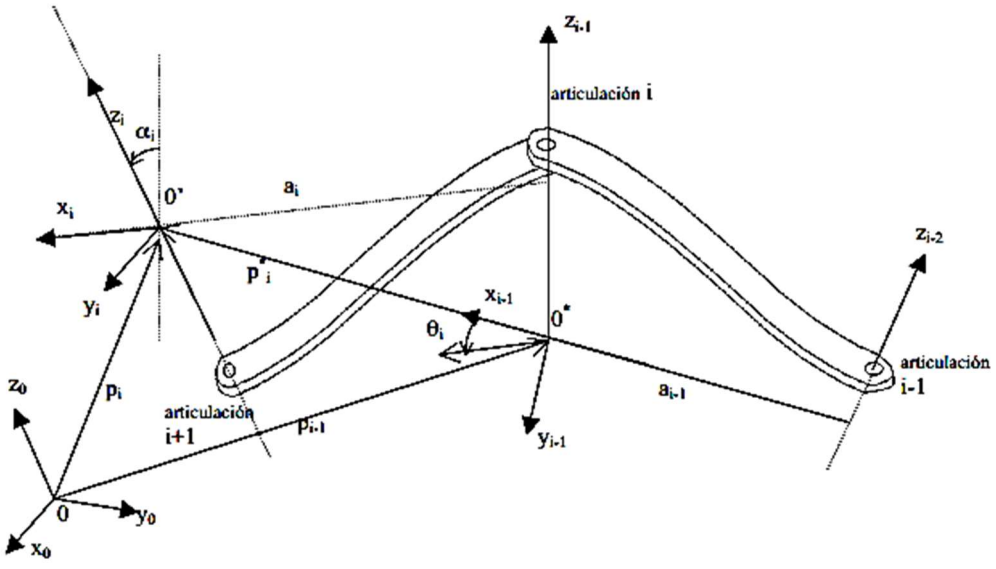

Ilustración 10. Sistema de coordenadas en los distintos eslabones que conforman el robot. [2]

Si la ecuación anterior se extrapola para los distintos eslabones que conforman el robot, las ecuaciones cinemáticas se podrán describir según:

$$v_i = \frac{dp_i^*}{dt} + \omega_{i-1} x p_i^* + v_{i-1} \quad (42)$$

$$\omega_i = \omega_{i-1} + \omega_i^* \quad (43)$$

Se observa que la velocidad angular del eslabón i viene dada por la velocidad angular del eslabón anterior y por la velocidad angular relativa del eslabón del sistema de coordenadas de *O'*. En la próxima ecuación se indicará cómo obtener la aceleración lineal del sistema de coordenadas de la articulación:

$$\dot{v}_i = \frac{d^2 p_i^*}{dt^2} + \dot{\omega}_{i-1} x p_i^* + 2 \cdot \omega_{i-1} x \frac{dp_i^*}{dt} + \omega_{i-1} x (\omega_{i-1} x p_i^*) + \dot{v}_{i-1} \quad (44)$$

$$\dot{\omega}_i = \dot{\omega}_{i-1} + \dot{\omega}_i^* \quad (45)$$

La aceleración angular del sistema de coordenadas (xi, y_i, z_i), con respecto a (x_{i-1}, y_{i-1}, z_{i-1}), se puede formular de la siguiente manera:

$$\dot{\omega}_i^* = \frac{d\dot{\omega}_i}{dt} + \dot{\omega}_{i-1} x \omega_i^* \quad (46)$$

Y, de esta forma:

$$\dot{\omega}_i = \dot{\omega}_{i-1} + \frac{d\omega_i^*}{dt} + \omega_{i-1} x \omega_i^* \quad (47)$$

A partir de las ecuaciones descritas precedentemente, y haciendo uso de las matrices homogéneas, se puede determinar el modelo cinemático, según si la configuración del eslabón es de rotación o translación [1]:

N-E1. Se asigna a cada eslabón un sistema de referencia, de acuerdo con las normas de D-H.

N-E2. Se establece las condiciones iniciales $\{S_0\}$, suponiendo que la base del robot está fija y no está en movimiento:

Velocidad angular:

$$^{0}\omega_0 = [0,0,0]^T \quad (48)$$

Aceleración angular:

$$^{0}\dot{\omega}_0 = [0,0,0]^T \quad (49)$$

Velocidad lineal:

$$^{0}v_0 = [0,0,0]^T \quad (50)$$

Aceleración lineal:

$$^{0}\dot{v}_0 = -[g_{x0}, g_{y0}, g_{z0}]^T \quad (51)$$

Según la configuración del robot, se elige una configuración u otra del vector de gravedad.

Ilustración 11. Distintas posiciones de un robot. [Fuente: ABB]

Tal y como se observa en la *Ilustración 11*, en función de la aplicación del robot, se podrán presentar distintas posiciones, así como también diferentes orientaciones de la base.

El vector $^{i}s_i$ representa las coordenadas del centro de masas del eslabón i con respecto al sistema $\{S_i\}$.

El vector $^{i}p_i$ que unirá el origen $\{S_{i-1}\}$ con $\{S_i\}$, para el que:

$$S_i = [a_i, d_i \cdot sen(\alpha_i), d_i \cdot \cos(\alpha_i)$$

(52)

N-E3. Se obtienen las matrices de rotación ($^{i-1}R_i$) de cada eslabón y sus respectivas inversas ($^{i}R_{i-1}$).

N-E4. Se obtienen las velocidades angulares de cada eslabón, dependiendo de la configuración de este.

$$^{i}\omega_i = \begin{cases} {}^{i}R_{i-1} \cdot \left({}^{i-1}\omega_{i-1} + z_0 \cdot \dot{q}_i \right) & Si\ el\ eslabón\ i\ es\ de\ rotación \\ {}^{i}R_{i-1} \cdot {}^{i-1}\omega_{i-1} & Si\ el\ eslabón\ i\ es\ de\ translación \end{cases}$$

(53)

N-E5. Se obtienen las aceleraciones angulares de cada eslabón, en función de la configuración de este.

$$^{i}\dot{\omega}_i = \\ = \begin{cases} {}^{i}R_{i-1} \cdot \left({}^{i-1}\dot{\omega}_{i-1} + z_0 \cdot \ddot{q}_i \right) + {}^{i-1}\omega_{i-1}\ x\ z_0 \cdot \dot{q}_i & Si\ el\ eslabón\ i\ es\ de\ rotación \\ {}^{i}R_{i-1} \cdot {}^{i-1}\omega_{i-1} & Si\ el\ eslabón\ i\ es\ de\ translación \end{cases}$$

(54)

N-E6. Se obtienen las aceleraciones lineales del sistema i.

$$^{i}\dot{v}_i = \\ = \begin{cases} {}^{i}\dot{\omega}_i x\ {}^{i}p_i + {}^{i}\omega_i x\left({}^{i}\omega_i x\ {}^{i}p_i \right)\ {}^{i}R_{i-1} \cdot {}^{i-1}\dot{v}_{i-1} \\ Si\ el\ eslabón\ i\ es\ de\ rotación \\ {}^{i}R_{i-1} \cdot \left(z_0 \cdot \ddot{q}_i + {}^{i-1}\dot{v}_{i-1} \right) + {}^{i}\omega_i x\ {}^{i}p_i\ + 2 \cdot {}^{i}\omega_i x\ {}^{i}R_{i-1} \cdot z_0 \cdot \dot{q}_i + \\ + {}^{i}\omega_i\ x\left({}^{i}\omega_i x\ {}^{i}p_i \right) \\ Si\ el\ eslabón\ i\ es\ de\ translación \end{cases}$$

(55)

N-E7. Se obtiene la aceleración lineal del centro de gravedad del eslabón i.

$$^{i}a_i = {}^{i}\dot{\omega}_i x\ {}^{i}s_i + {}^{i}\omega_i x\left({}^{i}\omega_i x\ {}^{i}s_i \right) + {}^{i}\dot{v}_i$$

(56)

Una vez formuladas las ecuaciones que describen el modelo cinemático de cada articulación, se aplica el principio de Alembert[5] (para el cual no se considera el rozamiento viscoso[6] de las articulaciones). Por lo tanto, si se reformulan las ecuaciones 35 y 36 se obtiene:

$$F_i = \frac{d(m_i \cdot {}^i v_i)}{dt} = {}^i I_i \cdot {}^i \dot{\omega}_i + {}^i \omega_i x ({}^i I_i \cdot {}^i \omega_i) \quad (57)$$

$$M_i = \frac{d({}^i I_i \cdot {}^i \omega_i)}{dt} = {}^i I_i \cdot {}^i \dot{\omega}_i + {}^i \omega_i x ({}^i I_i \cdot {}^i \omega_i) \quad (58)$$

donde:

F_i: fuerza total externa ejercida sobre el eslabón i y el centro de masas.

${}^iI_i$: es la matriz de inercia del eslabón i.

M_i: es el momento total ejercido sobre el eslabón i y el centro de masas.

[5] Principio que establece que cualquier sistema de fuerzas está en equilibrio con la suma de las fuerzas externas y las fuerzas generadas por las fuerzas de inercia [5].
[6] Rozamiento producido por el movimiento de un sólido en el interior de un fluido (en el caso de superficies lubricadas). Este rozamiento depende de la geometría del sólido, de la viscosidad y densidad del fluido y de la velocidad relativa del objeto en el fluido.

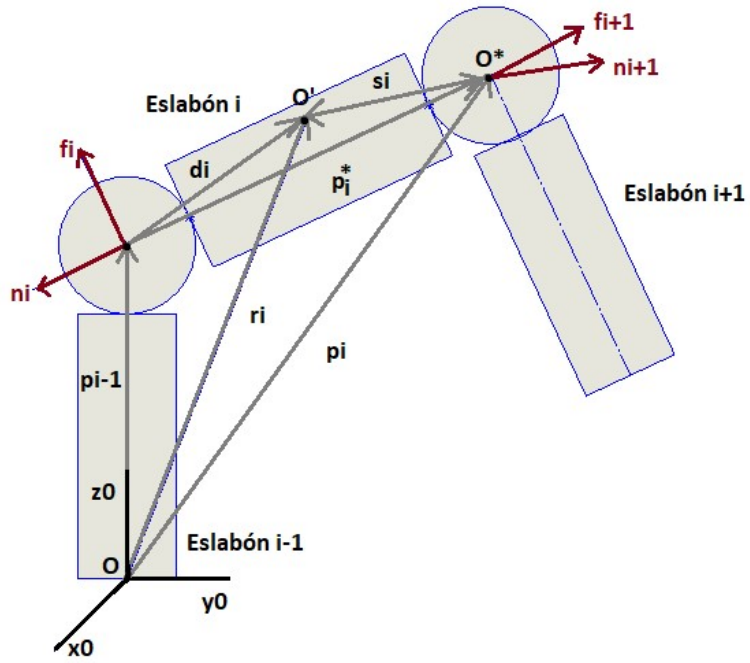

Ilustración 12. Ejemplo de simplificación de robot en el que se muestra las fuerzas y los momentos que actúan sobre el eslabón i.

En la *Ilustración 12* aparece la simplificación de un robot, para la que:

ri: posición del centro de masas del eslabón i con respecto al sistema de coordenadas de la base.

fi: fuerza ejercida en el eslabón i por el eslabón i-1 expresado en el sistema de coordenadas i-1.

ni: momento ejercido en el eslabón i por el eslabón i-1 expresado en el sistema de coordenadas i-1.

Si se hace el equilibrio de fuerzas y pares de la ilustración anterior se obtiene:

$$F_i = {}^{i}f_i - {}^{i+1}f_{i+1} \qquad (59)$$

$$M_i = {}^{i}n_i - {}^{i+1}n_{i+1} + d_i x\, {}^{i}f_i - s_i x\, {}^{i+1}f_{i+1} = \\ = {}^{i}n_i - {}^{i+1}n_{i+1} + (p_{i-1} - r_i) x\, {}^{i}f_i - (p_i - r_i) \cdot {}^{i+1}f_{i+1} \qquad (60)$$

Si se substituye la ecuación 59 a la ecuación 60, se consigue:

$$M_i = {}^{i}n_i - {}^{i+1}n_{i+1} + (p_{i-1} - r_i)x\left(F_i + {}^{i+1}f_{i+1}\right) - (p_i - r_i) \cdot {}^{i+1}f_{i+1} = \\ = {}^{i}n_i - {}^{i+1}n_{i+1} + (p_{i-1} - r_i)xF_i - p_i^* x\, {}^{i+1}f_{i+1} \quad (61)$$

Finalmente, se obtienen las ecuaciones del par y de la fuerza en el eslabón i:

$${}^{i}f_i = F_i + {}^{i+1}f_{i+1} = m_i \cdot {}^{i}a_i + {}^{i+1}f_{i+1} \quad (62)$$

$${}^{i}n_i = {}^{i+1}n_{i+1} + (p_{i-1} - r_i)xF_i - p_i^* x\, {}^{i+1}f_{i+1} + M_i = \\ = {}^{i+1}n_{i+1} + \left(p_i^* + {}^{i}s_i\right)xF_i + p_i^* x\, {}^{i+1}f_{i+1} + {}^{i}I_i \cdot \\ \cdot {}^{i}\dot{\omega}_i + {}^{i}\omega_i x({}^{i}I_i \cdot {}^{i}\omega_i) \quad (63)$$

Para finalizar, se detalla la formulación del modelo dinámico, según si la configuración del eslabón, de la siguiente forma [1]:

N-E8. A partir del desarrollo anterior, se obtiene la fuerza ejercida sobre el eslabón i:

$${}^{i}f_i = m_i \cdot {}^{i}a_i + {}^{i}R_{i+1} \cdot {}^{i+1}f_{i+1} \quad (64)$$

N-E9. Se consigue el par ejercido sobre el eslabón i:

$${}^{i}n_i = {}^{i}R_{i+1} \cdot [{}^{i+1}n_{i+1} + ({}^{i+1}R_i \cdot p_i^*)x\, {}^{i+1}f_{i+1}] + \left(p_i^* + {}^{i}s_i\right)xm_i \cdot {}^{i}a_i \\ + {}^{i}I_i \cdot {}^{i}\dot{\omega}_i + {}^{i}\omega_i x({}^{i}I_i \cdot {}^{i}\omega_i) \quad (65)$$

N-E10. Se obtiene el par aplicado para la articulación i. No se considera el coeficiente de rozamiento viscoso de la articulación (b).

$$\tau_i = \\ = \begin{cases} {}^{i}n_i^T \cdot {}^{i}R_{i-1} \cdot z_0 + b_i \cdot \dot{q}_i & Si\ el\ eslabón\ i\ es\ de\ rotación \\ {}^{i}f_i^T \cdot {}^{i}R_{i-1} \cdot z_0 + b_i \cdot \dot{q}_i & Si\ el\ eslabón\ i\ es\ de\ translación \end{cases} \quad (66)$$

7.2. Pares de gravedad

Otro modo de calcular el par de cada articulación es mediante el calculo de los pares de gravedad:

P-G1. Se determina el centro de gravedad de cada eslabón con respecto a la base:

$$\vec{p}^{0}_{cgi} = A^{0}_{i} * \vec{p}_{cgi} \tag{67}$$

P-G2. Se calcula el momento de cada eslabón. El momento del eslabón i estará afectado por los eslabones siguientes (i+1, i+2, …):

$$\vec{M}_{gi} = \sum_{p=i}^{n} m_p \vec{g} X (\vec{p}^{0}_{cgp} - \vec{p}^{0}_{op-1}) \tag{68}$$

P-G3. Se calcula el par a partir del momento calculado anteriormente:

$$\tau_i = \vec{M}_{gi} . \vec{z}_{i-1} \tag{69}$$

Finalmente, todo este proceso desemboca en el desarrollo del siguiente algoritmo:

```
void Robot::GravityTorque(float q[]){
    float grav[3] = {0, 9.81F, 0};
    Vector<float> g;
    g = grav;
    ForwardKinematics(q);
    Matrix<float> A0(4,4);
    A0 = _Linker[0].A0;

    Vector<float> pcg(4);

    Vector<float> position[_RobotDOF];
    position[0] = zeros(3);
    position[1] = GetPosition(A0);

    Vector<float> z0 [_RobotDOF];
    z0[0](0) = 0;
    z0[0](1) = 0;
    z0[0](2) = 1;

    Vector<float> m(4);
    for (int i = 0; i <_RobotDOF; i++) {
        if (i > 0) {
            z0[i](0) = A0(0,2);
            z0[i](1) = A0(1,2);
            z0[i](2) = A0(2,2);

            A0 = A0 * _Linker[i].A0;
            if (i<_RobotDOF-1) {
                position[i + 1] = GetPosition(A0);
            }
        }
        m(0) = _Linker[i].Cent_Gravity(0) / _Linker[i].Mass;
        m(1) = _Linker[i].Cent_Gravity(1) / _Linker[i].Mass;
        m(2) = _Linker[i].Cent_Gravity(2) / _Linker[i].Mass;
        m(3) = 1;
```

```
        pcg = A0 * m;
        _Motor[i].pcg(0) = pcg(0);
        _Motor[i].pcg(1) = pcg(1);
        _Motor[i].pcg(2) = pcg(2);
    }
    Vector<float> M;
    Vector<float> mg;
    float Tau;
    for (int i =0; i < _RobotDOF; i++) {
        mg = g * _Linker[i].Mass;
        M = mg.crossProduct((_Motor[i].pcg - position[i]));
        for (int j = 0; j < _RobotDOF; j++) {
            if(j>i) {
                mg = g * _Linker[j].Mass;
                M = M + (mg.crossProduct(_Motor[j].pcg)-position[i]);
            }
        }
        Tau = M * z0[i];
        _Linker[i].Tau = Tau;
        M = zeros(3);
    }

}
```

7.3. Comparativa del algoritmo Newton-Euler y Pares de gravedad

Se ha hecho una comparativa entre el algoritmo Newton-Euler y los pares de gravedad para determinar qué algoritmo es más rápido y eficaz para calcular los pares que afectan al motor. Con dicha intención, se ha calculado el tiempo en el microcontrolador (durante 6000 iteraciones) y se ha enviado la información a través de una comunicación serial a una aplicación desarrollada en Java, cuyo propósito es proporcionar un array para posteriormente representar el histograma.

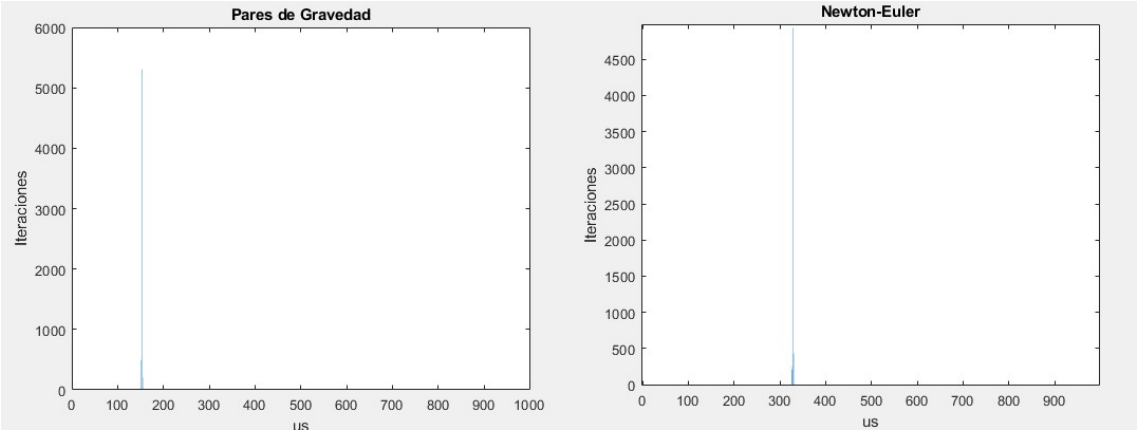

Ilustración 13. Comparativa de tiempos entre el algoritmo de Pares de Gravedad y el Newton-Euler.

En la *Ilustración 13* se puede observar que el algoritmo de pares de gravedad es mucho más rápido que el de Newton-Euler. En este caso, será el usuario quien podrá escoger el algoritmo que más se le ajuste a la aplicación.

8. Control Cinemático

Este apartado resulta de relevante importancia, principalmente porque abordará los métodos necesarios para calcular la trayectoria que describe el movimiento del robot.

Una trayectoria se puede definir como un historial de tiempo a partir del que se detalla la posición, la velocidad y la aceleración.

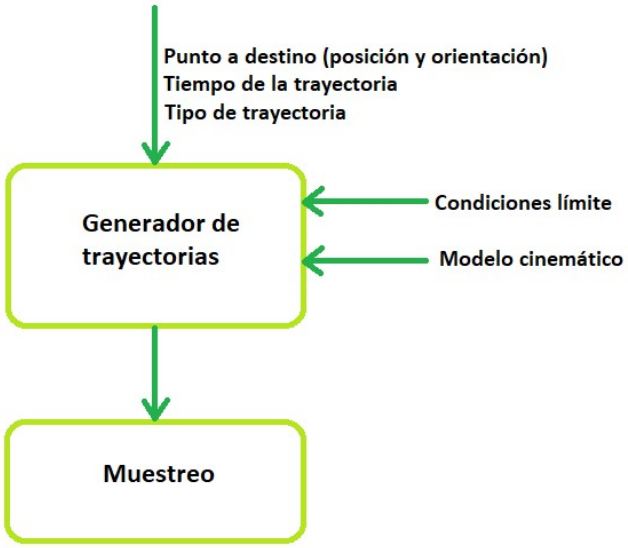

Ilustración 14. Se muestran las fases en que conta el control cinemático.

En la *Ilustración 14* se esquematiza el control cinemático que un generador de trayectorias recibe, como entradas, el punto destino donde tiene que ir el *End Effector* y el tiempo de las trayectorias a alcanzar dicho objetivo. Además, para que el algoritmo sea capaz de generar las trayectorias, se tiene que cumplir unos requisitos físicos en cuanto a la velocidad, aceleración y jerk[7]. Estas condiciones límite vendrán impuestas por el hardware que tenga el usuario. Igualmente, el generador de trayectorias utilizará el modelo cinemático del robot para calcular la posición de cada articulación.

Cada trayectoria debe de ser muestreada en un período de muestreo T, es decir, generando en cada instante kT un vector de coordenadas articulares para realizar el posterior cálculo del control dinámico.

[7] También llamada sobre-aceleración. Se define como el cambio de la aceleración en un determinado período de tiempo. De esta manera, se define el jerk como la derivada de la aceleración con respecto al tiempo.

A continuación, se exponen, de forma general, las funciones que realizará un control cinemático:

1. Dado un programa, interpretar el punto de destino, el tipo de trayectoria y el tiempo a invertir.
2. Muestreo de las trayectorias cartesianas, donde se obtiene un numero finito de puntos de dicha trayectoria. Cada uno de estos puntos vendrá definido por un vector de seis dimensiones (x, y, z, φ, θ, ψ).
3. Para cada punto del vector de seis dimensiones descrito anteriormente, se obtendrán sus correspondientes coordenadas articulares (q_0, q_1… q_n). En este paso puede aparecer una solución no válida para los puntos singulares.
4. Interpolación de los puntos articulares obtenidos, en la que se genera, para cada articulación, una expresión $q_i(t)$ que pase o se aproxime a ellos, siendo una trayectoria capaz de poderse realizar en los motores.
5. Muestreo de la trayectoria articular para generar las referencias para el control dinámico.

A partir del planteamiento descrito, se ha creado una clase llamada *Trajectory*, en la cual cada coordenada articular presentará un objeto de tipo *Trayectory*. En cada coordenada articular se guardarán las posiciones, las velocidades y las aceleraciones, con respecto al tiempo.

```
class Trajectory {
    private:
        Vector<float> pos;
        Vector<float> v;
        Vector<float> a;
        int Contsample;
        int TimSampling;
    public:
        Trajectory();
        Trajectory(int dof, int sampling);
        void Insert(float p, float v, float a);
        float GetXref(int i);
        float GetVref(int i);
        float GetAref(int i);
};
```

8.1. Interpolación y generación de trayectorias

Para que el robot alcance un determinado objetivo, debe moverse desde un punto inicial a un punto final u objetivo. Dicho punto final se podrá alcanzar siguiendo infinitas trayectorias espaciales. En este caso, la evolución temporal de la trayectoria en cada articulación va del punto inicial al final sin tener en consideración la evolución del resto de articulaciones. De esta manera, cada motor tratará de llevar su articulación al punto destino en el menor tiempo posible.

Una vez el usuario determine la posición y orientación del terminal del robot, y aplicando el modelo de cinemática inversa, se obtiene la secuencia de configuraciones articulares por las que el robot tendrá que pasar. La interpolación tendrá el objetivo de unir la sucesión de puntos articulares, siempre garantizando que se cumplan las restricciones de velocidad, de aceleración y de jerk correspondientes para cada motor, con el propósito de garantizar una generación de trayectoria y con suficiente suavidad. De esta forma, se ajustará a una función, cuyos parámetros se obtendrán a partir de las condiciones límite; posiciones, velocidades, aceleraciones y jerk, iniciales y finales.

En los apartados siguientes se analizarán los interpoladores diseñados en este proyecto.

8.1.1. Interpoladores Splines

La interpolación de las Splines es una clase de interpolación en la que se usan polinomios de n orden. Esta se caracteriza por el hecho que el error de interpolación se puede reducir incluso cuando se usan polinomios de bajo orden.

8.1.1.1. Interpolador Splin de quinto orden

Un modo de hacer la interpolación de una trayectoria es a partir del cálculo de un polinomio de quinto orden. De esta manera, tal y como se desarrolla en el *Anexo 15.1*, se pueden encontrar los distintos coeficientes (a, b, c, d, e y f) que conforman dicho polinomio:

$$q(t) = a + b \cdot t + c \cdot t^2 + d \cdot t^3 + e \cdot t^4 + f \cdot t^5 \quad (70)$$

Se ha desarrollado en Matlab un script para posteriormente implementar dicha función en C++ y observar el correcto funcionamiento del algoritmo. En la gráfica siguiente se observa un ejemplo de trayectoria utilizando la interpolación de quinto orden.

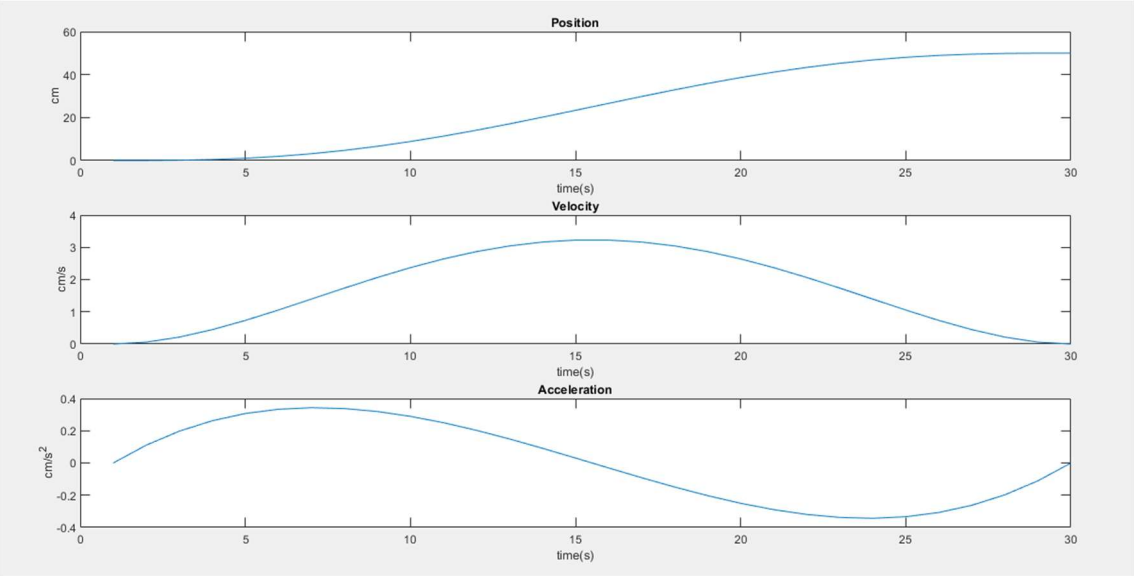

Ilustración 15. Ejemplo de Spline de quinto orden. Spline_5th(0,50,0,0,0,0.01,30,20,20).

Siguiendo el desarrollo anterior, se puede programar el siguiente método, para el cual:

1. q0 = Posición inicial.
2. q1 = Posición final.
3. qd0 = Velocidad inicial.
4. qd1 = Velocidad final.
5. qdd = Aceleración inicial.
6. stime = Período de muestreo (*sampling time*).
7. tfin = Tiempo final.
8. Constraints _v = Limitaciones en cuanto a velocidad.
9. Constraints _a = Limitaciones en cuanto a aceleración.

Las limitaciones de los actuadores en cuanto a la velocidad y a la aceleración vendrán dadas por las características del fabricante. Además, en el algoritmo implementado se ha considerado que la aceleración final sea cero.

```
void MotionCommand::Spline_5th (float q0,float q1, float qd0, float qd1, flo
atqdd,float stime,float tfin,float Constraints_v,float Constraints_a){
      float a, b, c, d, e, f,_q,_qd,_qdd;
```

```
        if(qd0>Constraints_v){
          qd0 = Constraints_v;
        }
        if(qdd>Constraints_a){
          qdd = Constraints_a;
        }

        //Calc the parameters
        f = (6.0F*q1)/powf(tfin,5.0F) - (6.0F*q0)/powf(tfin,5.0F) - (3.0F*qd0)
        /powf(tfin,4.0F) - (3.0F*qd1)/powf(tfin,4.0F) - qdd/(2*powf(tfin,3.0F)
        );
        e = (15.0F*q0)/powf(tfin,4.0F) - (15.0F*q1)/powf(tfin,4.0F)+ (8.0F*qd0
        )/powf(tfin,3.0F) + (7.0F*qd1)/powf(tfin,3.0F) + (3.0F*qdd)/(2.0F*powf
        (tfin,2.0F));
        d = (10.F*q1)/powf(tfin,3.0F) - (10.0F*q0)/powf(tfin,3.0F) - (6.0F*qd0
        )/powf(tfin,2.0F) - (4.0F*qd1)/powf(tfin,2.0F) - (3.0F*qdd)/(2.0F*tfin
        );
        c = qdd/2.0F;
        b = qd0;
        a = q0;

        TotalCycle = (int) tfin/stime;

       //Sampling in units of st (sampling time)
       for(int t=0;t<(int)((tfin/stime)+1);t++){
          _q  = a  + b *((float)t)*stime + c *powf(((float)t)*stime,2.0F) +d *
          powf(((float)t)*stime,3.0F) + e *powf(((float)t)*stime,4.0F) + f *po
          wf(((float)t)*stime,5.0F);

          _qd  = b  + 2.0F*c *((float)t)*stime + 3.0F*d *powf(((float)t)*stime
          ,2.0F) +4.0F*e *powf(((float)t)*stime,3.0F) + 5.0F*f *powf(((float)t
          )*stime,4.0F);
          _qdd  = 2.0F*c  + 6.0F*d *((float)t)*stime + 12.0F*e *powf(((float)t
          )*stime,2.0F) + 20*f *powf(((float)t)*stime,3.0F);
          _Trajectory[index].Insert(_q,_qd,_qdd);
       }
       index++;
}
```

8.1.1.2. Interpolador Splin de séptimo orden

Otra forma de lograr la interpolación de una trayectoria es a partir del cálculo de un polinomio de séptimo orden. De este modo, tal y como se desarrolla en el *Anexo 15.2*, se pueden encontrar los distintos coeficientes (a, b, c, d, e, f, g y h) que conforman dicho polinomio:

$$q(t) = a + b \cdot t + c \cdot t^2 + d \cdot t^3 + e \cdot t^4 + f \cdot t^5 + g \cdot t^6 + h \cdot t^7 \quad (71)$$

Se ha desarrollado en Matlab un script para posteriormente implementar dicha función en C++ y observar el correcto funcionamiento del algoritmo. En la gráfica siguiente se observa un ejemplo de trayectoria utilizando la interpolación de séptimo orden.

```
void MotionCommand::Spline_7th(float q0,float q1, float qd0, float qd1, floa
t qdd,float stime,float tfin,float Constraints_v ,float Constraints_a){

        float a, b, c, d, e, f, g, h,_q,_qd,_qdd;
        if(qd0>Constraints_v){
        qd0 = Constraints_v;
        }
        if(qdd>Constraints_a){
        qdd = Constraints_a;
        }

        //Calc the parameters
        h = (35.0F*q1)/powf(tfin,4.0F)- (35.0F*q0)/powf(tfin,4.0F) - (20.0F*qd
        0)/powf(tfin,3.0F) - (15.0F*qd1)/powf(tfin,3.0F) - (5.0F*qdd)/powf(tfi
        n,2.0F);
        g = (84.0F*q0)/powf(tfin,5.0F) - (84.0F*q1)/powf(tfin,5.0F) + (45.0F*q
        d0)/powf(tfin,4.0F) + (39.0F*qd1)/powf(tfin,4.0F) + (10.0F*qdd)/powf(t
        fin,3.0F);
        f = (70.0F*q1)/powf(tfin,6.0F) - (70.0F*q0)/powf(tfin,6.0F) - (36.0F*q
        d0)/powf(tfin,5.0F) - (34.0F*qd1)/powf(tfin,5.0F) - (15.0F*qdd)/(2.0F*
        powf(tfin,4.0F));
        e = (35.0F*q1)/powf(tfin,4.0F) - (35.0F*q0)/powf(tfin,4.0F) - (20.0F*q
        d0)/powf(tfin,3.0F) - (15.0F*qd1)/powf(tfin,3.0F) - (5.0F*qdd)/powf(tf
        in,2.0F);
        d = 0.0F;
        c = Constraints_a/2.0F;
        b = Constraints_v;
        a = q0;
        TotalCycle = (int) tfin/stime;
        //Sampling in units of st (sampling time)
        for(int t=0;t<(int) tfin/stime;t++){
          _q = a + b*((float)t)*stime + c*powf(((float)t)*stime,2.0F) + d*powf
          (((float)t)*stime,3.0F)+e*powf(((float)t)*stime,4.0F) + f*powf(((flo
          at)t)*stime,5.0F) + g*powf(((float)t)*stime,6.0F)+h*powf(((float)t)*
          stime,7.0F);
          _qd = b + 2.0F*c*((float)t)*stime + 3.0F*d*powf(((float)t)*stime,2.0
          F) +4.0F*e*powf(((float)t)*stime,3.0F) + 5.0F*f*powf(((float)t)*stim
          e,4.0F)+6.0F*g*powf(((float)t)*stime,5.0F) + 7.0F*h*powf(((float)t)*
          stime,6.0F);
          _qdd = 2.0F*c + 6.0F*d*((float)t)*stime + 12.0F*e*powf(((float)t)*st
          ime,2.0F)+20.0F*f*powf(((float)t)*stime,3.0F) + 30.0F*g*powf(((float
          )t)*stime,4.0F)+42.0F*h*powf(((float)t)*stime,5.0F);
          _Trajectory[index].Insert(_q,_qd,_qdd);
        }
        index++;
}
```

De igual forma que el apartado anterior, las limitaciones de los actuadores en cuanto a la velocidad y a la aceleración vendrán dadas por las características del fabricante. Además, en el algoritmo implementado se ha considerado que la aceleración y el jerk final sean cero.

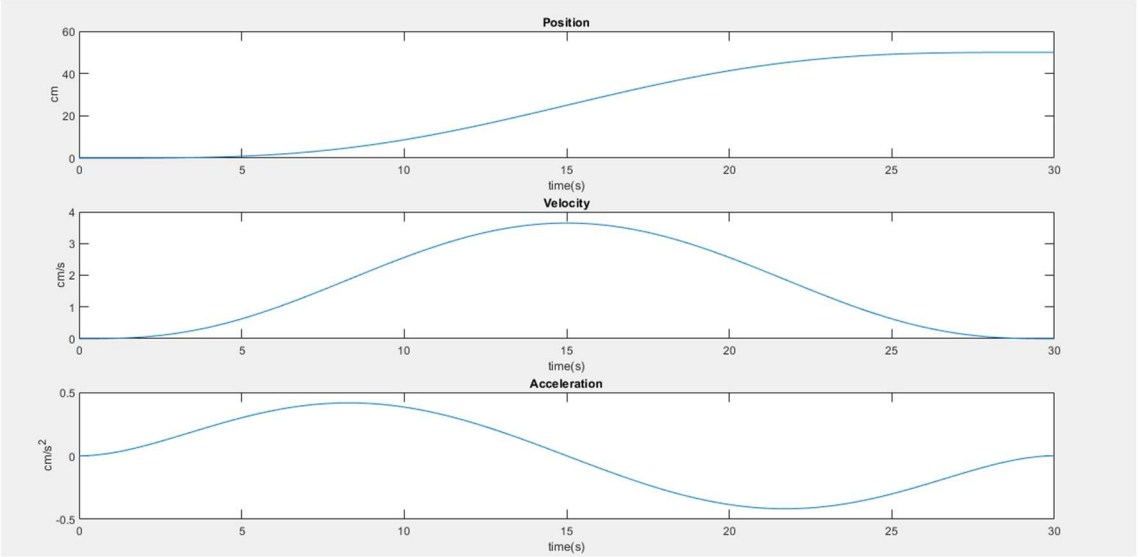

Ilustración 16. Ejemplo de Spline de quinto orden. Spline_7th(0,50,0,0,0,0.01,30,20,20).

8.1.2. Interpolador trapezoidal

El perfil trapezoidal es un método muy común para obtener las trayectorias con un perfil de velocidad continua. Consiste en la utilización de movimientos lineales con mezclas parabólicas. Estas trayectorias se dividen en tres partes (siempre que la posición final sea mayor que la inicial):

a) En la primera parte, la aceleración es positiva y constante y, por lo tanto, la velocidad es una función lineal con respecto al tiempo; la posición es una curva parabólica.
b) En la segunda parte, la aceleración es nula, de esta forma, la velocidad será constante y la posición será una función lineal del tiempo.
c) En la tercera parte se presenta una aceleración negativa constante, la velocidad disminuye de manera lineal; la posición es una función polinomial de grado dos.

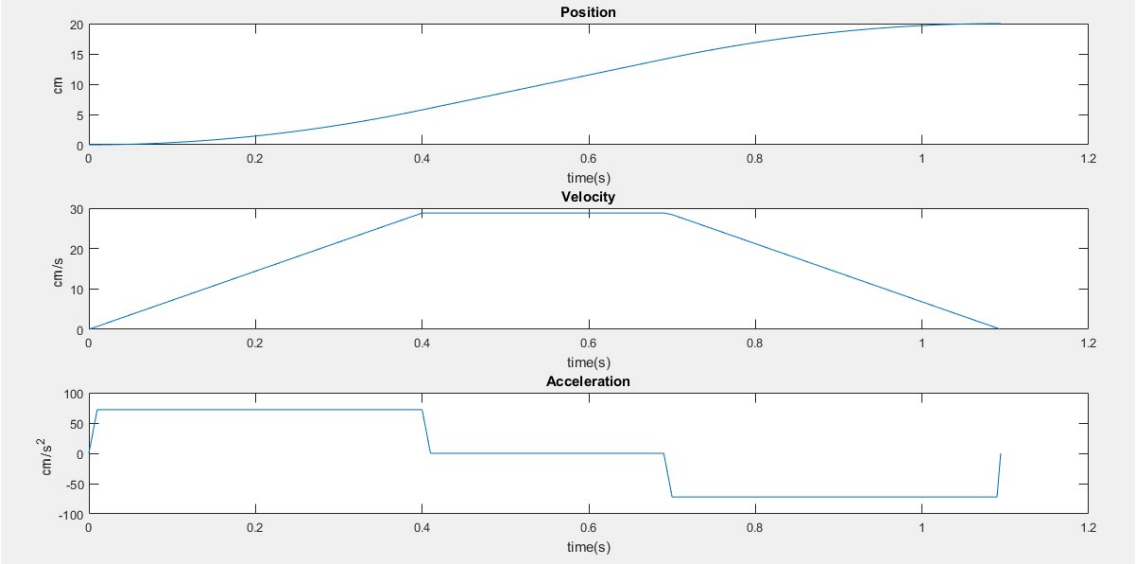

Ilustración 17. Ejemplo de Trapezoide.

Para el cálculo de este algoritmo, se ha utilizado la cinemática clásica del movimiento rectilíneo uniformemente acelerado, describiendo una función a trozos a partir de las tres fases que lo definen.

```
void MotionCommand::Trapezoid(float _q0,float q1, float qd, float qdd,float
stime,float Constraints_v,float Constraints_a){
       float s, signo, t1, t2, tfin, q,qprim,acel;
       s = q1 - _q0;
       signo = sign(s);
       CurrentCycle = 0;

       if (fabsf(s) > (powf(Constraints_v,2.0F) / Constraints_a)){
         t1 = Constraints_v / Constraints_a;
         t2 = fabsf(s) / Constraints_v;
         tfin = fabsf(s) / Constraints_v + Constraints_v / Constraints_a;
       }
       else{
         t1 = sqrtf(fabsf(s) / Constraints_a);
         tfin = 2.0F*t1;
         t2 = 0.0F;
       }
       q = _q0;
       qprim = 0;
       float t=0;
       TotalCycle = (int)tfin/stime+1;
       for(int tim=0; tim<((int)(tfin/stime)+1);tim++){
         t = tim*stime;
         //Calculation of the velocity and position for phase 1 (acceleration
         )
         if (t < t1){
              qdd = signo*Constraints_a;
              qprim=qprim+acel*stime;
              q=q+qprim*stime+0.5F*acel*stime*stime;
         }
```

```
        //Calculation of the velocity and position for phase 2 (constant vel
        ocity)
        else if ((t >= t1) && (t < t2)){
             qdd = 0.0F;
             qprim=signo*qd;
             q=q+qprim*stime;
        }
        //Calculation of the velocity and position for phase 3 (deceleration
        )
        else if ((t >= t2) && (t < tfin)){
             acel = -signo*qdd;
             qprim=qprim + acel*stime;
             q=q + qprim*stime+0.5F*acel*powf(stime,2.0F);
        }else{
             qdd = 0.0F;
             qd = 0.0F;
             q = q1;
        }
        _Trajectory[index].Insert(q,qd,qdd);
      }
      index++;
}
```

8.1.3. Interpolador SCurve

Un perfil de velocidad trapezoidal presenta una aceleración discontinua, de tal manera que esta trayectoria puede generar esfuerzos y tensiones en el sistema mecánico y perjudicar o contribuir en la formación de vibraciones. Así, se tiene que definir un perfil de movimiento más suave, lo que resulta en una utilización de un perfil de movimiento también más suave. De esta forma, para tener una aceleración de manera continua, la velocidad resultante tiene que estar compuesta por segmentos lineales conectados por parábolas.

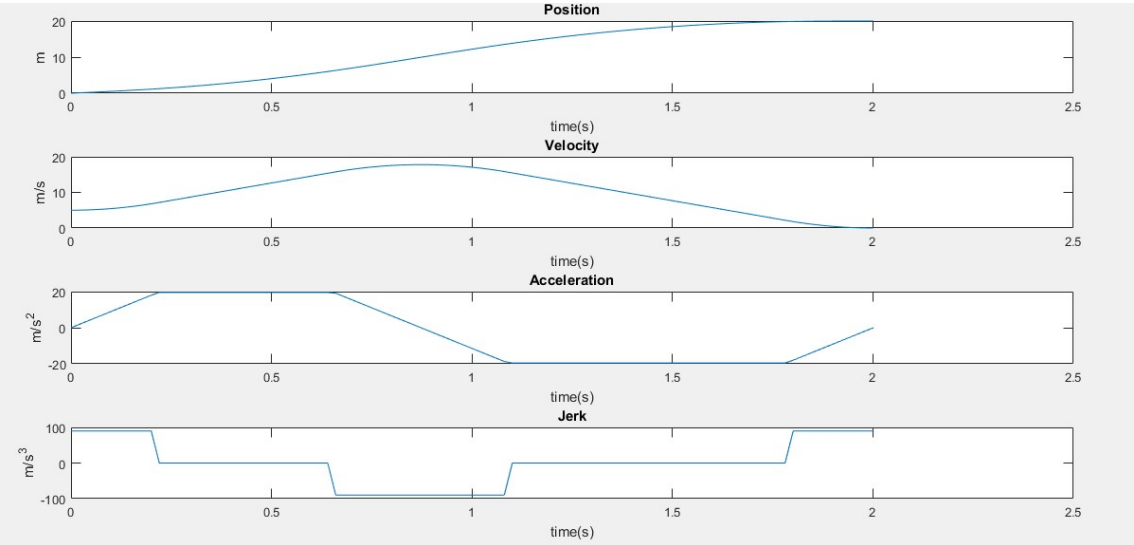

Ilustración 18. Representación de una trayectoria SCurve.

En la *Ilustración 17* del apartado anterior se muestra un perfil de velocidades trapezoidal; la *Ilustración 18* se trata de una *S-Curve*. Si se observan ambas aceleraciones, en el perfil trapezoidal se aprecia una aceleración discontinua en el tiempo, mientras que en el *S-Curve* se observa un triángulo o, en muchos otros casos, un trapecio. A causa de las discontinuidades de la aceleración en el perfil trapezoidal, la velocidad tiene forma de trapecio, mientras que en el otro se observa mucho más suave.

Para calcular la trayectoria de un punto a otro se necesita pasar el punto inicial, el punto final, su velocidad inicial y su velocidad final. Este hecho es de gran importancia para que se adapte bien, así como para garantizar que no haya cambios bruscos, pérdida de control, etc.

A partir del *Anexo 15.3* se ha desarrollado en Matlab un script para implementar posteriormente la función en C++ y observar el funcionamiento del algoritmo.

```
int MotionCommand::SCurve(float q0,float q1, float qd0,float qdf, float qdd,
float stime,float Constraints_v,float Constraints_a, float Constraints_j){
       float _q,_qd,_qdd,_q0,_q1,v0,v1,dq,dv,time_set_velocity,time_reach_acc
       ,Tj,sigma,sigm1,sigm2, vmax,vmin,amax,amin,jmax,jmin,Tj1,Tj2,Ta,Td,Tv,
       T,amax_squared,
         sqrt_delta,alima,alimd,vlim,q,qprim,acel;
         CurrentCycle = 0;
         bool isFeasable;
          if (Constraints_v<abs(qd0)){
                 v0 = sign(qd0)*Constraints_v;
          }
             if (Constraints_v<abs(qdf)){
```

```
                v1 = sign(qdf)*Constraints_v;
        }

        //Check if its feasable
        dq=abs(q1 - q0);
        dv=abs(v1-v0);

        time_set_velocity=sqrtf((dv)/Constraints_j);
        time_reach_acc=Constraints_a/Constraints_j;
           Tj = min(time_set_velocity,time_reach_acc);
           if (Tj < time_reach_acc){
               if (dq<Tj*(v0+v1)){
                   isFeasable = false;
               }
               else{
                    isFeasable = true;
               }
           }
           else if (Tj == time_reach_acc){
                if (dq < (0.5F *(v0+v1)*(Tj+dv/Constraints_a))){
                   isFeasable = false;
                }
                else{
                    isFeasable=true;
                }
           }
           else{
            isFeasable=false;
           }

    if (isFeasable == false){
        printf("Is not feasable this configuration \n");
        return 0;
    }
    //Sign change
        sigma = sign(q1-q0);
        _q0 = sigma*q0;
        _q1 = sigma*q1;
        v0 = sigma*v0;
        v1 = sigma*v1;
        sigm1 = (sigma+1.0F)/2.0F;
        sigm2 = (sigma-1.0F)/2.0F;
        vmax = Constraints_v*sigm1 - Constraints_v*sigm2;
        amax = Constraints_a*sigm1 - Constraints_a*sigm2;
        jmax = Constraints_j*sigm1 - Constraints_j*sigm2;

        vmin = -vmax;
        amin = -amax;
        jmin = -jmax;

      if ((vmax- v0)* jmax < amax* amax){
              Tj1 = sqrtf((vmax - v0)/ jmax);
              Ta = 2.0F * Tj1;
      }
      else{
              Tj1 = amax/jmax;
              Ta =  Tj1 + (vmax - v0)/amax;
      }
```

```
      if ((vmax - v1)* jmax < amax * amax){
            Tj2 = sqrtf((vmax - v1)/ jmax);
            Td = 2.0F* Tj2;
      }
      else{
             Tj2 =  amax / jmax;
             Td = Tj2 + (vmax - v1)/ amax;
      }
         Tv = (_q1 - _q0)/vmax -  Ta*(1.0F + v0 / vmax)/2.0F -  Td *(1.0F +
v1 / vmax)/2.0F;

     if (Tv<0){
         float it = 0;
         float l = 0.99F;
         float max_iter = 200;
         while ((it < max_iter) && (amax >= 0.01F)){
                //Vmax not reach
                Tj1 = amax / jmax;
                Tj2 = Tj1;
                Tj = Tj2;
                amax_squared = amax * amax;
                sqrt_delta = sqrtf( ((amax_squared*amax_squared)/(jmax * jma
x)) + 2.0F*(v0 * v0+  v1 * v1)
                    + amax * (4.0F*(_q1 - _q0) - 2.0F* amax *(v0 + v1)/jmax)
);
                Ta = (amax_squared/jmax - 2.0F * v0 + sqrt_delta)/(2.0F * am
ax);
                Td = (amax_squared/jmax - 2.0F * v1 + sqrt_delta)/(2.0F * am
ax);
                Tv = 0;
                if (Ta < 0.0F){
                    Td = 2.0F*((_q1 - _q0) / (v1 + v0));
                    Tj2 = (jmax *(_q1 - _q0)-sqrtf(jmax *(jmax*(powf((_q1-
_q0),2.0F))+(powf((v1+v0),2.0F))*
                    (v1-v0))))/(jmax*(v1+v0));
                    Ta = 0;
                    Tj1 = 0;
                }
                if(Td < 0.0F){
                    Ta = 2.0F*((_q1-_q0)/(v1+v0));
                    Tj1 = (jmax*(_q1-_q0 )-sqrtf(jmax *(jmax *(powf((_q1 -
_q0 ),2.0F))-(powf((v1 +v0 ),2.0F))*
                    (v1 -v0 ))))/(jmax *(v1 +v0 ));
                    Td  = 0;
                    Tj2  = 0;
                }

                if ((Ta  < 2.0F*Tj1 ) || (Td  < 2.0F*Tj2 )){
                    amax  =amax * l;
                }
                else{
                    break;
                }
        }
     }
     alima  = jmax *Tj1 ;
     alimd  = -jmax  * Tj2 ;
     vmax  = v0  + (Ta  - Tj1 )*alima ;
```

```
      vlim  = vmax ;
      T  = Ta +Tv +Td ;

      qprim = 0;
      float t=0;
      TotalCycle = (int)T/stime+1;
      for(int tim=0; tim<((int)(T/stime)+1);tim++){
         t = tim*stime;
         if (t<Tj1 ){
             _q  = sigma *(_q0 +v0 *t+jmax *powf(t,0.5F));
             _qd = sigma*(v0+jmax*powf(t,2.0F)/2);
             _qdd = sigma*(jmax*t);
         }
         else if (t<(Ta -Tj1 )){
             _q  = sigma *(_q0 +v0 *t+(3.0F*powf(t,2.0F)-
3.0F*Tj1 *t+powf(Tj1 ,2.0F))*(alima /6.0F));
             _qd = sigma*(v0+alima*(t-(Tj1/2.0F)));
             _qdd = sigma*alima;
         }
         else if (t<Ta ){
             _q  = sigma *(_q0 +(vlim +v0 )*(Ta /2.0F)-vlim *(Ta -t)-
jmin *powf((Ta -t),0.5F));
             _qd = sigma*(vlim+jmin*(powf((Ta-t),2.0F)/2.0F));
             _qdd = sigma*(-jmin*(Ta-t));
         }
         else if (t<(Ta +Tv )){
             _q  = sigma *(_q0 +(vlim +v0 )*(Ta /2.0F)+vlim *(t-Ta ));
             _qd = sigma*vlim;
             _qdd = 0;
         }
         else if (t<(T -Td +Tj2 )){
             _q  = sigma *(_q1 -(vlim +v1 )*(Td /2.0F)+vlim *(t-T +Td )-
jmax *(powf((t-T +Td ),0.5F)));
             _qd = sigma*(vlim-jmax*(powf((t-T+Td),2.0F)/2.0F));
             _qdd = sigma*(-jmax*(t-T+Td));
         }
         else if (t<(T -Tj2 )){
             _q  = sigma *(_q1 -(vlim +v1 )*(Td /2.0F)+vlim *(t-
T +Td )+(alimd /6.0F)*(3.0F*powf((t-T +Td ),2.0F)-3.0F*Tj2 *(t-
T +Td )+powf(Tj2 ,2.0F)));
             _qd = sigma*(vlim+alimd*(t-T+Td-(Tj2/2.0F)));
             _qdd = sigma*alimd;
         }
         else if (t<=T ){
             _q  = sigma *(_q1 -v1 *(T -t)-jmax *(powf((T -t),3.0F))/6.0F);
             _qd = sigma*(v1+jmax*(powf((T-t),2.0F))/2.0F);
             _qdd = sigma*(-jmax*(T-t));
         }
         else{
             _q  = sigma *_q0 ;
             _qd = sigma*v1;
             _qdd = 0;
         }
         _Trajectory[index].Insert(_q,_qd,_qdd);
     }
     index++;
     return 1;
}
```

9. Control dinámico

El control cinemático que el robot deberá seguir, ya obtenido gracias al cálculo de las trayectorias, presenta irregularidades en la realidad (debido a rozamientos, holguras, inercias, etc.), por las que trayectoria deseada no coincidirá con la real. El control dinámico tiene como misión que las trayectorias realizadas por el robot sean muy parecidas a las propuestas por el usuario.

A demás, se ha de tener en cuenta que la presencia de mecanismos de reducción, como engranajes, puede introducir fenómenos físicos, no deseados (tales como vibraciones debidas al *backlash*[8] de los engranajes, fricción de los engranajes, etc.), que obstaculicen el desempeño del robot en su tarea requerida.

9.1. Control por corriente

En un robot, el vector de pares τ se obtiene gracias los actuadores electromecánicos, hidráulicos y/o neumáticos. Cada actuador tiene su propia dinámica, es decir, el par o la fuerza entregada es el producto de la transformación dinámica de la entrada del actuador. En el caso de los actuadores electromecánicos, esta entrada puede tratarse de un voltaje o de una corriente.

Este trabajo se interesa por que el usuario adopte una configuración de control por corriente. En esta configuración, la etapa de potencia[9] genera una corriente proporcional a la acción de control. Actualmente en robótica es más frecuente el control por corriente, ya que la corriente es linealmente proporcional al par y, de este modo, permite aplicar técnicas de control por par calculado.

[8] Se puede definir como una holgura o movimiento perdido en un engranaje causada por los huecos que existen en entre los dientes de ambos engranajes. Por lo tanto, consiste en la distancia o ángulo máximo a partir del cual cualquier sistema mecánico puede moverse en una dirección sin transmitir movimiento o aplicar fuerza en la siguiente secuencia mecánica.

[9] Es un dispositivo electrónico que se encarga de la interfase entre el controlador y los actuadores. Transforma la acción de control, que es una señal de baja potencia, en tensión de alta potencia.

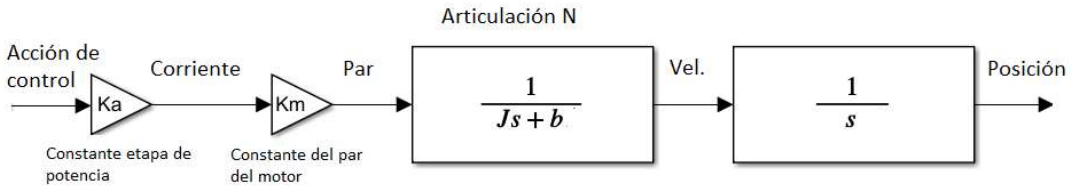

Ilustración 19. Diagrama de bloques de un controlador por corriente

La configuración de la etapa de potencia genera una corriente proporcional a la acción de control, es decir, se genera una tensión de PWM. La anchura de pulso es proporcional a la diferencia entre la corriente de referencia y la real medida por los sensores.

En la *Ilustración 19* se muestra el diagrama de bloques de un controlador de corriente. El controlador envía la corriente de referencia; esta corriente se multiplica por la ganancia de la etapa de potencia, dando la corriente que circulará por el motor. Una vez obtenida la corriente, esta se multiplica por la constante del par del motor, y se obtendrá el par, el cual será directamente proporcional a la corriente que circula por el motor. Se aplica la función de transferencia par/velocidad para obtener la velocidad. Esta función de transferencia vendrá definida en sus polos por el momento de inercia total de la articulación (J) y el rozamiento viscoso (b). Finalmente, se integra la velocidad y se obtiene la posición.

9.1.1. Control Proporcional

El control proporcional es la versión más básica de un controlador, donde se tiene en cuenta el error de posición (ep =qref-q).

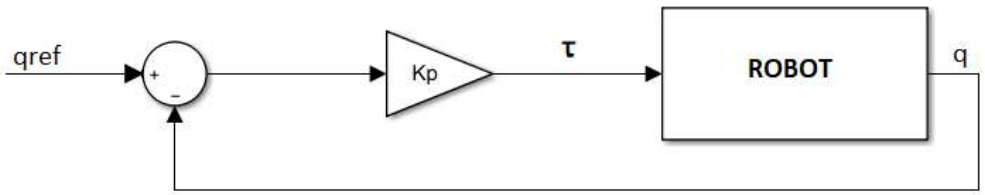

Ilustración 20.Diagrama de bloques de un control P.

Si se analiza para una articulación, se puede deducir la siguiente función de transferencia:

$$\frac{q}{q_{ref}} = \frac{K_p \cdot K_a \cdot \frac{K_m}{J}}{s^2 + \frac{b}{J} \cdot s + \frac{K_p \cdot K_a \cdot K_m}{J}} \quad (72)$$

A partir de lo planteado, se ha desarrollado el método para calcular la acción de control proporcional.

```
float Control::ComputeP(float qref, float qi){
    float u=(((qref - qi) * KP)*KA*KM)/MaxTorque;
    if (u>1){
        u=1;
    }
    if (u<0){
        u=0;
    }
    u = u* Scale + Offset;
    return u;
}
```

9.1.2. Control Proporcional-Derivativo: primera versión

El control proporcional-derivativo es una extensión del control proporcional en el que se tiene en cuenta la referencia de la posición y de la velocidad.

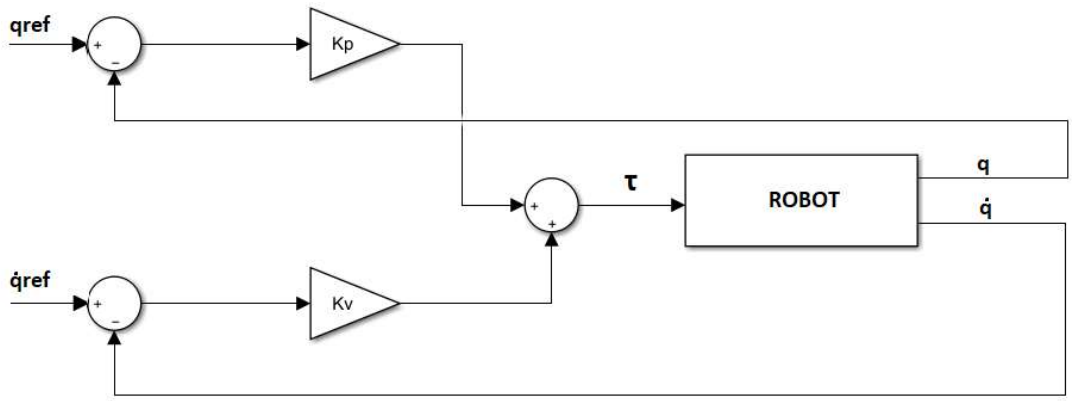

Ilustración 21. Diagrama de bloques de un control PD primera versión.

En la ilustración anterior se muestra el diagrama de bloques de un control proporcional-derivativo, donde:

1. qref es la posición deseada.
2. $\dot{q}$ref es la velocidad deseada.
3. Kp es la ganancia proporcional.
4. Kv es la ganancia derivativa.

De esta manera, la acción de control generada por el regulador es la suma ponderada del error de posición y de su derivada (error de posición).

$$\tau = K_p \cdot (q_{ref} - q) + K_v \cdot (\dot{q}_{ref} - \dot{q}) = K_p \cdot e_p + K_v \cdot e_v \quad (73)$$

Si se analiza para una articulación, se puede deducir la siguiente función de transferencia:

$$\frac{q}{q_{ref}} = \frac{(K_p + K_v \cdot s) \cdot \frac{K_a \cdot K_m}{J}}{s^2 + \left(\frac{b + K_a \cdot K_m \cdot K_v}{J}\right) \cdot s + \frac{K_p \cdot K_a \cdot K_m}{J}} \quad (74)$$

Por una parte, se observa la presencia de un cero en el numerador, lo que puede originar problemas a cambios muy bruscos. Por otra parte, la constante derivativa aparece multiplicando en el denominador; esto permite ajustar el amortiguamiento modificando la constante derivativa.

Se ha desarrollado así el método para calcular la acción de control en la primera versión de PD.

```
float Control::ComputePD_FirstVersion(float qref, float qi, float qdotref, f
loat qdoti){
    float u=(((qref - qi) * KP + (qdotref - qdoti) * KV)*KA*KM)/MaxTorque;
    if (u>1){
        u=1;
    }
    if (u<0){
        u=0;
    }
    u = u* Scale + Offset;
    return u;
}
```

9.1.3. Control Proporcional-Derivativo: segunda versión (PV)

La primera versión del control Proporcional-Derivativo introduce un cero. Esto no es conveniente para entradas de tipo escalón. La segunda versión del control Proporcional-Derivativo (PV) consiste en una extensión de la primera versión del control proporcional derivativo, en la que se tiene en cuenta el *feedback* de la velocidad (y sin una velocidad de referencia).

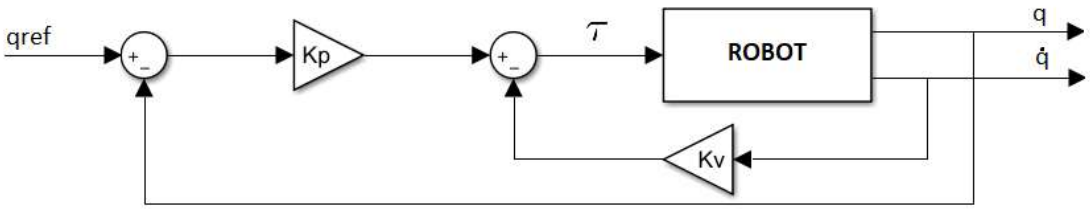

Ilustración 22. Diagrama de bloques de un control PD segunda versión.

En la *Ilustración 22* se muestra el diagrama de bloques de un control PV, donde:

1. qref es la posición deseada.
2. Kp es la ganancia proporcional.
3. Kv es la ganancia derivativa.

De este modo, la acción de control generada por el regulador resulta en la suma ponderada del error de posición y de su derivada (error de posición).

$$\tau = K_p \cdot \left(q_{ref} - q\right) - K_v \cdot \dot{q} \qquad (75)$$

En las aplicaciones reales, el control proporcional derivativo se utiliza en cada articulación, ya que solo depende de la posición y velocidad de la articulación en cuestión y no de las otras articulaciones.

Si se analiza la función de transferencia, se observa que para un regulador PV no existe la presencia de un cero:

$$\frac{q}{q_i} = \frac{K_p \cdot K_a \cdot \frac{K_m}{J}}{s^2 + \left(\frac{b + K_a \cdot K_m \cdot K_v}{J}\right) \cdot s + \frac{K_p \cdot K_a \cdot K_m}{J}} \qquad (76)$$

```
float Control::ComputePD_SecondVersion(float qref, float qi, float qdoti){
    float u=(((qref - qi) * KP  - qdoti * KV)*KA*KM)/MaxTorque;
    if (u>1){
        u=1;
    }
    if (u<0){
        u=0;
    }
    u = u* Scale + Offset;
    return u;
}
```

9.1.4. Control Proporcional-Integral-Derivativo

Los reguladores proporcionales, o proporcional-derivativos, necesitan un error de posición y/o velocidad para generar una acción de control. Esto implica la aparición de un error o retraso para anular las fuerzas de inercia y rozamiento viscoso. Este error es más grande a mayor aceleración requerida. Con el control PD podemos lograr el control de posición para los robots "ideales" que no contienen el termino gravitacional (g(t) = 0). La acción de control tiene como objetivo satisfacer el control de posición y, con dicho fin, se introduce la acción integral, propiamente descrita de un Proporcional-Integral-Derivativo (PID). La ley de control de un PID viene dada por la siguiente ecuación:

$$\tau = K_p \cdot (q_{ref} - q) + K_v \cdot (\dot{q}_{ref} - \dot{q}) + K_i \int (q_{ref} - q) \cdot dt \quad (77)$$

En la actualidad, la mayoría de los manipuladores de robots industriales están controlados por controladores PID, ya que dan solución a los problemas de engranajes.

9.1.5. Control Proporcional-Derivativo con compensación de gravedad

En este apartado se hace mención a un control PD con compensación de gravedad que, a diferencia del PD clásico, se tiene en cuenta el vector de pares gravitacionales (g(q)). Por lo tanto, la ley de control PD con compensación de gravedad viene dado por:

$$\tau = K_p \cdot (q_{ref} - q) + K_v \cdot (\dot{q}_{ref} - \dot{q}) + g(q) \quad (78)$$

A diferencia del control PD, el controlador no hace uso del conocimiento del modelo del manipulador, concretamente de g(q).

Se ha programado así la función correspondiente a un control PD con compensación de la gravedad:

```
float Control::ComputePDG(float qref, float qi, float qdotref, float qdoti,
float Tau){
       float u = (((qref-
       qi) * KP + (qdotref - qdoti) * KV)*KA*KM + Tau)/MaxTorque;
       if (u>1){
         u=1;
       }
       if (u<0){
         u=0;
       }
         u = u* Scale + Offset;
       return u;
}
```

10. Movimiento del robot siguiendo una trayectoria lineal

El lenguaje de programación RAPID[10], que implementan los robots ABB, contiene una función llamada *MoveL,* o movimiento del robot, que sigue una trayectoria lineal. El *MoveL* se utiliza para trasladar el punto central de la herramienta (TCP), en sentido lineal, hacia un punto de destino determinado. Cuando se desea que el TCP permanezca estacionario, esta instrucción puede usarse también para reorientar la herramienta [2].

Se han creado dos métodos correspondientes al *MoveL*: El *MoveL1*, que se calcula a partir de a cinemática directa y se calculan las fuerzas cartesianas, mientras que el *MoveL2* se hace a partir de la cinemática inversa y el cálculo de la Jacobiana inversa.

10.1. MoveL1

Primeramente, se tiene que calcular la posición actual a partir de la lectura de los *encoders* de cada articulación. Seguidamente, se calcula el interpolador trapezoidal. Una vez finalizado el cálculo de los perfiles trapezoidales, para cada período de muestreo se realizarán los siguientes procedimientos:

1. Se leen las coordenadas articulares del robot (q).

[10] RAPID es el lenguaje de programación textual de alto nivel desarrollado por la empresa ABB.

2. Se calcula la cinemática directa de q, y se obtiene la posición y orientación en el extremo del robot.
3. Se calcula la velocidad en cada instante nT.
4. Se calcula la fuerza cartesiana a partir de: Kp*(xref-x)+Kv*(vref-v).
5. Se calcula el torque de gravedad (G).
6. Se calcula la Jacobiana (J) y finalmente se calcula la acción de control como: J*(Fuerza cartesiana)+G.
7. Se mandan las acciones de control a los motores.

10.2. MoveL2

Primeramente, se tiene que calcular la posición actual a partir de la lectura de los *encoders* de cada articulación. Seguidamente, se calcula el interpolador trapezoidal. Una vez finalizado el cálculo de los perfiles trapezoidales, para cada período de muestreo se realizarán los siguientes procedimientos:

- Se leen las coordenadas articulares del robot (q).

- Se obtienen las coordenadas articulares de referencia (qrex) mediante el uso de la cinemática inversa.

- A partir del uso de la Jacobiana inversa, se calculan las velocidades de referencia de las articulaciones (q_prim_ref).

- Se calcula la velocidad en cada instante nT de las articulaciones (q_prim).

- Se realiza el cálculo de las aceleraciones de los motores:

qsec= Kp*(qref-q)+Kv*(qref-q).

- Se calculan los pares del motor a partir de la cinemática inversa y se ejecuta sobre los motores.

11. Movimiento del robot mediante un movimiento de ejes

El *MoveJ* también es una función característica de los robots ABB. El *MoveJ* se utiliza para mover el robot rápidamente de un punto a otro cuando no es imprescindible que el movimiento siga una línea recta. Los ejes del robot y los ejes externos se desplazan hasta la posición de destino a lo largo de una trayectoria no lineal. Todos los ejes alcanzan la posición de destino al mismo tiempo [2].

De forma análoga al *MoveJ* procedente de ABB, se ha desarrollado un método perteneciente a la clase MotionCommand llamada *MoveJ*, teniendo en cuenta que:

1. qf: posición donde se quiere alcanzar (posición final).
2. v0: velocidad inicial.
3. v1: velocidad final.
4. a: aceleración máxima.
5. *samplingtime*: tiempo de muestreo.
6. *time*: tiempo total para alcanzar la posición final.
7. *_trajectory*: tipo de trayectoria.

Una vez escritos los argumentos que conforman dicho método, primeramente, se necesitará determinar la posición en la que se encuentra el robot, es decir, la posición inicial (q0), a partir de la lectura de los *encoders* de cada articulación. A continuación, mediante el uso de la cinemática directa, se determinarán la posición y la orientación del extremo del robot y se calcularán las distintas trayectorias para el método de interpolación elegido en la llamada a la función. Finalmente, se habilitará una interrupción de tiempo con el período de muestreo asignado para que cada período de muestreo vaya recorriendo los puntos de cada trayectoria previamente calculada; una vez finalizada la trayectoria, se deshabilitará automáticamente la interrupción del tiempo y se dará por finalizada la llamada al método.

```
void MotionCommand::MoveJ(End_Effector qf,float v0,float v1,float a,float sa
mplingtime,float time,int zone,Trajectory_Types _trajectory){
    SamplingTime = samplingtime;
    IsFinalized = false;
    PrevPosition = 0;
    float q0[3];
    Vector<float> PositionF(3),PositionInit(3);
    PositionF(0) = qf.x;
```

```
    PositionF(1) = qf.y;
    PositionF(2) = qf.z;

    _Trajectory = new Trajectory[3];


    for (int i=0; i<_Robot->GetDOF();i++){
        if(i<=3){
            _Trajectory[i] = Trajectory(_Robot->GetDOF(),samplingtime);
        }
        q0[i] = _Robot->ReadSensor(i);
    }
    _Robot->ForwardKinematics(q0);
    PositionInit = _Robot->GetEndPosition();

    for (int i=0; i<3;i++){
        switch(_trajectory){
       case TRAPEZOIDAL:
           Trapezoid(PositionInit(i),PositionF(i), v0, a,samplingtime,_Robot
->ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i));
           break;
       case SCURVE:
           SCurve(PositionInit(i),PositionF(i), v0, v1, a,samplingtime,_Robo
t->ConstraintsVelocityMotor(i),_Robot-
>ConstraintsAccelerationMotor(i), _Robot->ConstraintsJerkMotor(i));
           break;
       case SPLINE5:
           Spline_5th(PositionInit(i),PositionF(i), v0, v1, a,samplingtime,t
ime,_Robot->ConstraintsVelocityMotor(i),_Robot-
>ConstraintsAccelerationMotor(i));
           break;
       case SPLINE7:
           Spline_7th(PositionInit(i),PositionF(i), v0, v1, a,samplingtime,t
ime,_Robot->ConstraintsVelocityMotor(i),_Robot-
>ConstraintsAccelerationMotor(i));
           break;
        }
    }
    this->StartMoveL1();
    while(!IsFinalized) {
        asm("nop");
    }
    this->StopMoveL1();


}
```

12. Hardware

La librería desarrollada con los distintos algoritmos que conforman el control de un robot puede ser utilizada con los microcontroladores STM32F429ZIT6U o, en concreto, con la placa de desarrollo STM32F429I-DISC1 (Placa *Discovery*). En este caso, se ha utilizado el entorno de desarrollo Mbed, que se encarga de la compilación de dispositivos basados en microcontroladores de 32 bits del tipo ARM Cortex-M.

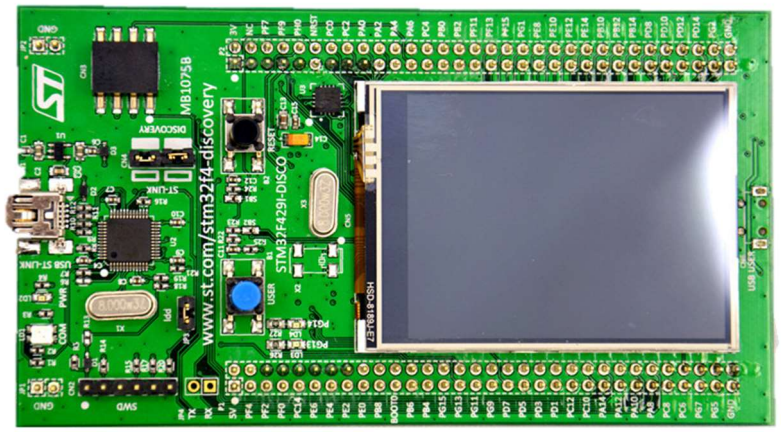

Ilustración 23. Placa de desarrollo STM32F429I-DISCO.

El microcontrolador es un Cortex-M4F con una unidad de punto flotante (FPU[11]), funcionando a 180MHz. La unidad de punto flotante es un aspecto muy valorable en cuanto al diseño de aplicaciones de la robótica, ya que se permite obtener, de manera más eficiente y rápida, las operaciones que conforman el algoritmo.

[11] La unidad de punto flotante, también conocida como coprocesador matemático, está especializada en el cálculo de operaciones en coma flotante. Un número de coma flotante es una notación científica utilizada para representar números reales muy grandes o pequeños de manera eficiente y rápida.

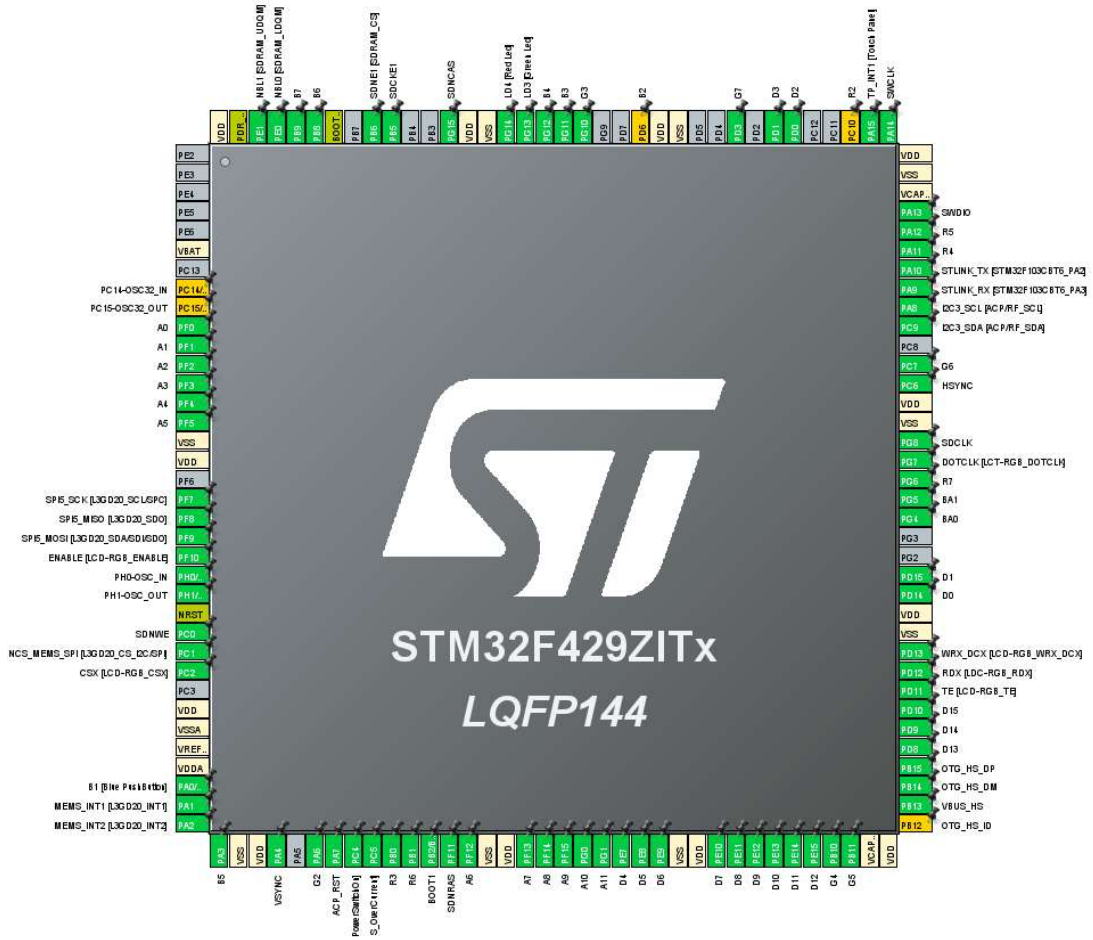

Ilustración 24. En la ilustración se muestra el pinout del STM32F429ZIT con sus correspondientes puertos utilizados en la placa Discovery.

Este microcontrolador incorpora más de 168 entradas y salidas; y más de 21 interfaces de comunicación, entre las cuales destacan las comunicaciones SPI, CAN, I2C, etc.

Al principio se intentó desarrollar una comunicación SPI[12], en la que la placa *Discovery* actuaba como el máster y otra placa de desarrollo para cada articulación que actuaba como esclavo, llamada Nucleo-F303K8.

[12] El protocolo SPI es una interfaz de comunicaciones en serie de manera síncrona que se utiliza para comunicaciones de corta distancia, principalmente en sistemas integrados.

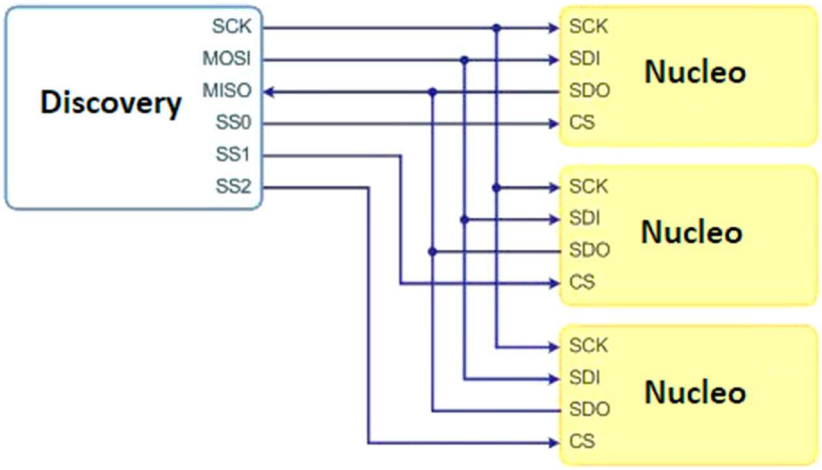

Ilustración 25. Primer planteamiento. Uso de la Discovery como master y placas Nucleo como esclavos para cada articulación de un robot.

Finalmente, se han hecho pruebas mediante comunicación SPI entre ambas placas, pero no ha resultado ser efectivo debido a la falta de sincronización de ambos microcontroladores y a la perdida de datos. Por lo consiguiente, se desarrolló el algoritmo en la propia placa *Discovery*.

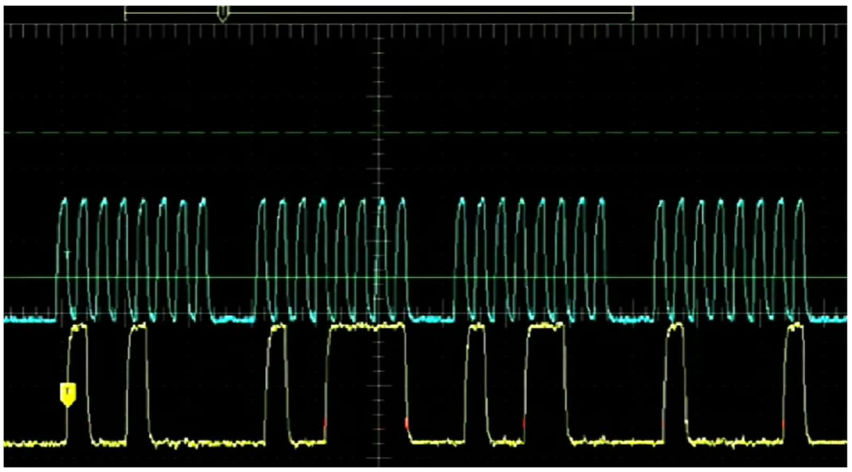

Ilustración 26. Captura del osciloscopio probando el funcionamiento del protocolo SPI, donde en azul aparece el SCLK y en amarillo el MOSI.

La opción de elegir esta placa frente a otras ha sido el hecho que esta incorpora 6 *timers*, cada uno con sus respectivos canales, capaces de manejar señales incrementales (de cuadratura) idóneas para el uso de 1 hasta 4 *encoders* de efecto hall.

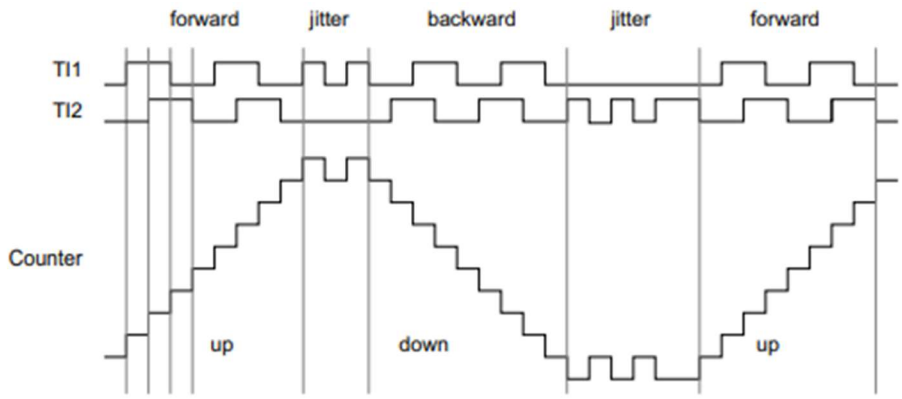

Ilustración 27. Ejemplo de operación de contador en modo de interfaz de encoder. Fuente…

La *Ilustración 27* muestra el funcionamiento del *timer*: en cada vuelta da un pulso, incrementándose en uno el contador hasta que llega a un límite superior. Las señales procedentes del *encoder* (A y B) actúan como incrementales o decrecientes en el contador del *timer*.

De acuerdo con esto, se ha perfeccionado una clase, que toma el nombre de *Sensor.h*, para interactuar con los distintos módulos de hardware que puede implementar el usuario.

```
class Sensor {
        private:
            float rank;
            float scale_factor;
            float offset;
            float value;
            QEI Encoder;
        public:
           Sensor(PinName channelA,PinName channelB,float _rank, float _sFac
tor,float _offset, int _pulsesPerRev);
            float readSensor();
            float computevalue(int value);
            float estimatevelocity (float prevValue,float sampling);
};
```

Esta clase se encarga de la lectura de los sensores, donde el usuario puede elegir los distintos *timers* que presenta la *Discovery*:

Timer	**PINOUT**	
TIM1	PE9	PE11
TIM2/TIM5	PA0	PA1
TIM3	PA6	PA7

TIM4	PD12	PD13
TIM8	PC6	PC7

El usuario deberá introducir las distintas características del robot, en cuanto a matriz de inercias, constantes proporcionales, derivativas (entre otras), y también la configuración de las entradas y salidas. En ese caso, deberá ir al fichero *Features.h* para la configuración pertinente de los periféricos.

Una de las desventajas observadas durante la ejecución del código es que la *Discovery* presenta *single-presicion floating-point* (que corresponde al formato IEEE 754[13]). Eso puede ocasionar que la mantisa contenga menos números, lo que hace que se pierda precisión al hacer un alto computo. Ese fenómeno se ha observado comparando la dinámica Newton-Euler con un robot de dos articulaciones, y uno de cinco con la librería de Peter Corke en Matlab.

```
14 -    q = [1,2];
15 -    qd=[0,3];
16 -    qdd=[1,2];
17 -    L(1) = Link('revolute','alpha', 1.5708,'d',0,'a',0,'m',0,'r',[0,0,0],'I',I1);
18 -    L(2) = Link('revolute','alpha',0,'d',0,'a',0.4318,'m',17.4,'r',[0,0,0],'I',I2);
19 -    R= SerialLink(L);
20 -    T = R.rne(q, qd, qdd)
```

```
Command Window

  T =

      0.5618  -24.1839
```

```
int main() {
    float q[2] = {1, 2};
    float qprim[2] = {0, 3};
    float qsec[2] = {1,2};
    const Link::Links_Type Linker[2] ={Link::REVOLUTION,Link::REVOLUTION};
    Robot p560(ToolMatrix,BaseMatrix,DH_Table,LinkLenght,dyn_mat,MotorParam,ControlGain,RobotDOF,Linker,Constraints);
    p560.EulerNewton(q,qprim,qsec);

    return 0;
}
```

```
Tau: 0.561834 -24.183880
```

Ilustración 28. Comparativa algoritmo Newton-Euler diseñado con las librerías de Peter Corke en Matlab par a un brazo de dos articulaciones.

En la *Ilustración 28* se observa la comparativa de un robot de dos articulaciones con las librerías de Peter Corke, en este caso, a diferencia de la *Ilustración 29*, no se

[13] El estándar IEEE 754 es un estándar técnico para la aritmética de punto flotante por el Instituto de Ingenieros Eléctricos y Electrónicos.

aprecia error. El alto coste computacional, aumentando en n el número de articulaciones, experimenta un incremento del error en la FPU.

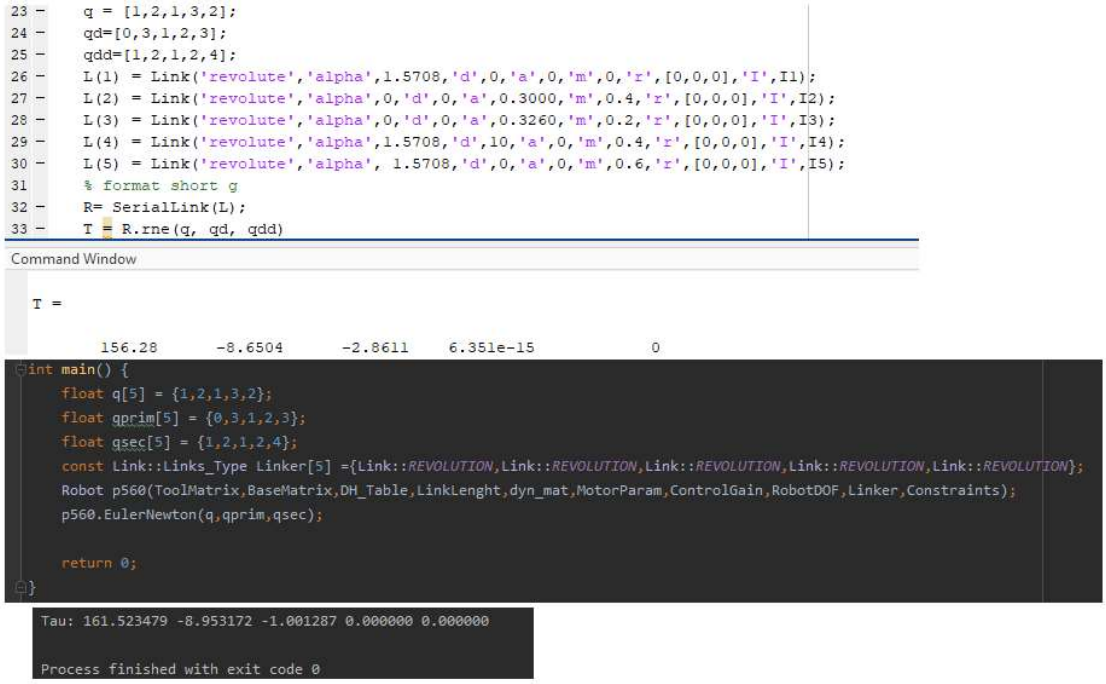

Ilustración 29. Comparativa algoritmo Newton-Euler diseñado con las librerías de Peter Corke en Matlab par a un brazo de cinco articulaciones. Donde se puede apreciar el error producido por el formato IEEE 754

Otra de las clases desarrolladas es la llamada *Driver.h*, que se encarga de enviar las distintas posiciones a los distintos drivers de los motores. El usuario podrá elegir el tipo de comunicación en función del módulo de hardware que tenga. En este caso, se puede escoger entre comunicación CAN[14] (hasta 1 Mbit/s), SPI o enviar a través de un conversor digital analógico (DAC). En el protocolo SPI existen SPI1, SPI4, SPI5 y SPI6 y se comunican hasta 45 Mbits/s; y el SPI2 y SPI3 se comunican hasta 22.5 Mbit/s.

```
class Driver {
    private:
        TypeCommunication communication;
        CAN *can;
        SPI *spi;
        AnalogOut *dac;
    public:
        Driver(PinName adrress);
        Driver(PinName RX,PinName TX);
```

[14] También conocido como Controller Area Network, es un protocolo de comunicación muy efectivo gracias a sus capacidades de alta velocidad, confiabilidad de largo alcance e inmunidad al ruido. Por estas razones, la comunicación CAN se ha convertido en el estándar en tecnologías automotrices y entornos de alto ruido.

```
        Driver(PinName MOSI,PinName MISO,PinName sclk, int frecuency);
        int sendmessage(char data[8]);
        int sendmessage(float data);
        ~Driver();
};
```

13. Conclusiones

En este proyecto se ha podido estudiar y programar el desarrollo de una librería de bajo coste mediante la placa *Discovery* y detallar las limitaciones que esta presenta; los próximos puntos han resultado decisivos en su progreso:

1. Planteamiento del proyecto: Al principio se ideó conectar mediante SPI la placa *Discovery* y la *Núcleo 32*. Se observó que no era eficaz en cuanto a sincronización de ambas. Esta desventaja condujo a la integración de todo el algoritmo en la placa *Discovery*, limitando así su funcionalidad en un determinado robot de n grados de libertad.
2. Desarrollo del proyecto: El desarrollo ha sido costoso por dificultades globales en cuanto al coronavirus, dado que no se ha podido aplicar en un robot real. Por lo tanto, cada una de las partes se ha comprobado con Matlab y la Toolbox de Peter Corke, observando así las limitaciones de la placa, como por ejemplo del FPU. Otro punto decisivo ha sido la creación de las clases matrices y vectores para posteriormente desarrollar, de forma clara, el algoritmo.

Posteriormente, ya más a nivel personal, la elaboración de este Trabajo Fin de Máster ha resultado en un verdadero e importante reto para mí, ya que he tenido que reunir y aplicar todas las competencias que me habían sido presentadas a lo largo de toda mi carrera académica. Llegar a las conclusiones es un punto de reflexión que me lleva a valorar todo el esfuerzo y la labor diarios, una tarea que me ha hecho crecer tanto profesional como mentalmente, solidificando y perfeccionando así el aparente primerizo concepto que antes tenía de lo que podía llegar a ser un robot.

13.1. Perspectivas futuras

Por un lado, el uso de redes inalámbricas con protocolos como TCP o UDP podría presentar mejoras, sobre todo en cuanto a sus aplicaciones. Últimamente, al ser un desarrollo de una librería abierta, hay muchas perspectivas futuras con respecto a las distintas configuraciones de robots posibles, y también a los distintos controladores, *encoders*, y a las comunicaciones que hay en el mercado. Este programa podría servir de inspiración, ya fuera para adaptar las librerías, en un principio solo desarrolladas para el hardware STM32, a otras placas de desarrollo, tales como Raspberry o Beaglebone, donde el hecho de presentar múltiples núcleos configuraría una ventaja en la eficiencia del algoritmo.

14. Biblografía

[1] A. Barrientos, Fundamentos de robótica, McGrawHill, 2007.

[2] ABB, «Manual de referencia técnica-Instrucciones, funciones y tipos de datos RAPID».

[3] L. M. C. Biagiotti, Trajectory Planning for Automatic Machines and Robots, Springer, 2008.

[4] G. a. L. Fu, Robotics: Control, Sensing, Vision and Intelligence, McGrawHill, 1987.

[5] C. Lanczos, The Variational Principles of Mechanics, University of Toronto Press, 1952.

[6] STMicroelectronics, «RM0090,» Discovery Reference Manual.

[7] P. Corke, «https://petercorke.com/toolboxes/robotics-toolbox/,» [En línea].