15. Anexos

15.1. Desarrollo y obtención de los parámetros del polinomio de quinto orden

A partir del polinomio de quinto orden se pueden obtener los coeficientes que definen dicha expresión.

$$q(t) = a + b \cdot t + c \cdot t^2 + d \cdot t^3 + e \cdot t^4 + f \cdot t^5 \quad (79)$$

Para encontrar una solución particular para el polinomio descrito, se tienen que conocer las condiciones de contorno, de tal forma que:

$$q_0 = q(t = 0) = a \quad (80)$$

$$q_f = q(t = tf) = a + b \cdot tf + c \cdot tf^2 + d \cdot tf^3 + e \cdot tf^4 + f \cdot tf^5 \quad (81)$$

$$v_0 = q'(t = 0) = b \quad (82)$$

$$v_f = q'(t = tf) = 5 \cdot f \cdot tf^4 + 4 \cdot e \cdot tf^3 + 3 \cdot d \cdot tf^2 + 2 \cdot c \cdot tf + b \quad (83)$$

$$a_0 = q''(t = 0) = 2 \cdot c \quad (84)$$

$$a_f = q''(t = tf) = 20 * f * tf^3 + 12 * e * tf^2 + 6 * d * tf + 2 * c \quad (85)$$

Si el desarrollo anterior se expresa de manera matricial, se pueden obtener los coeficientes de este polinomio:

$$\begin{bmatrix} q_0 \\ q_f \\ v_0 \\ v_f \\ a_0 \\ a_{fin} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & tf & tf^2 & tf^3 & tf^4 & tf^5 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 \cdot tf & 3 \cdot tf^2 & 4 \cdot tf^3 & 5 \cdot tf^4 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 6 \cdot tf & 12 \cdot tf^2 & 20 \cdot tf^3 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} \quad (86)$$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & tf & tf^2 & tf^3 & tf^4 & tf^5 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 \cdot tf & 3 \cdot tf^2 & 4 \cdot tf^3 & 5 \cdot tf^4 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 6 \cdot tf & 12 \cdot tf^2 & 20 \cdot tf^3 \end{bmatrix}^{-1} \cdot \begin{bmatrix} q_0 \\ q_f \\ v_0 \\ v_f \\ a_0 \\ a_{fin} \end{bmatrix} \quad (87)$$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ -\frac{10}{tf^3} & \frac{10}{tf^3} & -\frac{6}{tf^2} & -\frac{6}{tf^2} & -\frac{3}{2 \cdot tf} & \frac{1}{2 \cdot tf} \\ \frac{15}{tf^4} & -\frac{15}{tf^4} & \frac{8}{tf^3} & \frac{7}{tf^3} & \frac{3}{2 \cdot tf^2} & -\frac{1}{tf^2} \\ -\frac{6}{tf^5} & \frac{6}{tf^5} & -\frac{3}{tf^4} & -\frac{3}{tf^4} & -\frac{1}{2 \cdot tf^3} & \frac{1}{2 \cdot tf^3} \end{bmatrix} \cdot \begin{bmatrix} q_0 \\ q_f \\ v_0 \\ v_f \\ a_0 \\ a_{fin} \end{bmatrix} \quad (88)$$

15.2. Desarrollo y obtención de los parámetros del polinomio de séptimo orden

A partir del polinomio de séptimo orden, se obtienen los coeficientes que definen dicha expresión.

$$q(t) = a + b \cdot t + c \cdot t^2 + d \cdot t^3 + e \cdot t^4 + f \cdot t^5 + g \cdot t^6 + h \cdot t^7 \quad (89)$$

Para encontrar una solución particular para el polinomio descrito, se tienen que conocer las condiciones de contorno, de tal forma que:

$$q_0 = q(t = 0) = a \quad (90)$$

$$q_f = q(t = tf) = a + b \cdot tf + c \cdot tf^2 + d \cdot tf^3 + e \cdot tf^4 + f \cdot tf^5 \quad (91)$$

$$v_0 = q'(t = 0) = b \quad (92)$$

$$v_f = q'(t = tf) = = 7 \cdot h \cdot tf^6 + 6 \cdot g \cdot tf^5 + 5 \cdot f \cdot tf^4 + 4 \cdot e \cdot tf^3 + 3 \cdot d \cdot tf^2 + 2 \cdot c \cdot tf + b \quad (93)$$

$$a_0 = q''(t = 0) = 2 \cdot c \quad (94)$$

$$a_f = q''(t = tf) = = 42 \cdot h \cdot tf^5 + 30 \cdot g \cdot tf^4 + 20 \cdot f \cdot tf^3 + 12 \cdot e \cdot tf^2 + 6 \cdot d \cdot tf + 2 \cdot c \quad (95)$$

$$j_0 = q'''(t = 0) = 6 \cdot d \quad (96)$$

$$j_f = q'''(t = tf) = = 210 \cdot h \cdot tf^4 + 120 \cdot g \cdot tf^3 + 60 \cdot f \cdot tf^2 + 24 \cdot e \cdot tf + 6 \cdot d \quad (97)$$

Si el desarrollo anterior se expresa de manera matricial, se logran obtener los coeficientes de este polinomio:

$$\begin{bmatrix} q_0 \\ q_f \\ v_0 \\ v_f \\ a_0 \\ a_f \\ j_0 \\ j_f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & tf & tf^2 & tf^3 & tf^4 & tf^5 & tf^6 & tf^7 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 \cdot tf & 3 \cdot tf^2 & 4 \cdot tf^3 & 5 \cdot tf^4 & 6 \cdot tf^5 & 7 \cdot tf^6 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 6 \cdot tf & 12 \cdot tf^2 & 20 \cdot tf^3 & 30 \cdot tf^4 & 42 \cdot tf^5 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 24 \cdot tf & 60 \cdot tf^2 & 120 \cdot tf^3 & 210 \cdot tf^4 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} \quad (98)$$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} & 0 \\ -\frac{35}{tf^4} & \frac{35}{tf^4} & -\frac{20}{tf^3} & -\frac{15}{tf^3} & -\frac{5}{tf^2} & \frac{5}{2 \cdot tf^2} & -\frac{2}{3 \cdot tf} & -\frac{1}{6 \cdot tf} \\ \frac{84}{tf^5} & -\frac{84}{tf^5} & \frac{45}{tf^4} & \frac{39}{tf^4} & \frac{10}{tf^3} & -\frac{7}{tf^3} & \frac{1}{tf^2} & \frac{1}{2 \cdot tf^2} \\ -\frac{70}{tf^6} & \frac{70}{tf^6} & -\frac{36}{tf^5} & -\frac{34}{tf^5} & -\frac{15}{2 \cdot tf^4} & \frac{13}{2 \cdot tf^4} & -\frac{2}{3 \cdot tf^3} & -\frac{1}{2 \cdot tf^3} \\ \frac{20}{tf^7} & -\frac{20}{tf^7} & \frac{10}{tf^6} & \frac{10}{tf^6} & \frac{2}{tf^5} & -\frac{2}{tf^5} & \frac{1}{6 \cdot tf^4} & \frac{1}{6 \cdot tf^4} \end{bmatrix} \cdot \begin{bmatrix} q_0 \\ q_f \\ v_0 \\ v_f \\ a_0 \\ a_f \\ j_0 \\ j_f \end{bmatrix}$$

(99)

15.3. Desarrollo y obtención de la SCurve

Para el desarrollo de la *SCurve* se considerará un jerk discontinuo en el eje temporal. De este modo, se pueden distinguir tres fases [3]:

1. Fase de aceleración: durante el intervalo de tiempo 0 a Ta. En esta fase la aceleración tiene carácter lineal.
2. Fase de velocidad máxima: durante el intervalo Ta y Ta+Tv. Se caracteriza por la velocidad constante.
3. Fase de deceleración: durante el intervalo Ta+Tv y T, de tal manera que el tiempo total de la trayectoria vendrá dado por:

$$T = T_a + T_v + T_d \qquad \textit{(100)}$$

Donde:

Ta: tiempo de aceleración.

Tv: tiempo de velocidad constante.

Td: tiempo de deceleración.

En esta fase se observa perfiles opuestos a la fase de aceleración.

Cuando el usuario tenga que utilizar este método, tendrá que asignar unas limitaciones referentes a las características técnicas de los motores en cuanto a velocidad, aceleración y jerk máximos. Primeramente, se deberá verificar si la trayectoria es capaz de realizarse. Si, por ejemplo, tenemos un punto inicial y final, cuya diferencia es muy pequeña con respecto a la diferencia de velocidades inicial y final, no se podrá realizar. Con dicho fin, se debe comprobar siempre si es posible realizar la trayectoria con jerk (un pulso positivo y otro negativo). Por lo tanto:

$$T_j = min\left(\sqrt{\frac{v_1 - v_0}{j_{max}}}, \frac{a_{max}}{j_{max}}\right) \quad (101)$$

De esta manera, se comprueba si la trayectoria es factible, si la aceleración alcanza su máximo valor y si puede existir el jerk por paso por cero; la trayectoria será factible si:

$$q_1 - q_0 > \begin{cases} Tj \cdot (v_0 + v_1) & si\ T_j < \frac{a_{max}}{j_{max}} \\ \frac{1}{2} \cdot (v_0 + v_1) \cdot \left[Tj + \frac{v1 - v0}{a_{max}}\right] & si\ \ T_j = \frac{a_{max}}{j_{max}} \end{cases} \quad (102)$$

La expresión anterior demuestra si es posible hacer el cálculo de los parámetros de la trayectoria. En este caso, para definir el máximo valor de velocidad durante el movimiento, existirán dos casos:

Caso 1: En este caso se verifica si la máxima aceleración se alcanza por las siguientes condiciones:

Tiempo de aceleración:

$$Si\ (v_{max} - v_0) \cdot j_{max} < a_{max}^2 \rightarrow a_{max}\ no\ es\ alcanzada \quad (103)$$

Si la aceleración es alcanzada en el intervalo de aceleración, se utiliza la siguiente expresión:

$$T_{j1} = \sqrt{\frac{v_{max} - v_0}{j_{max}}} \quad (104)$$

donde T_{j1} es el tiempo en el cual el jerk es constante durante la fase de aceleración:

$$T_a = 2 \cdot T_{j1} \quad (105)$$

Contrariamente, si no se alcanza, se utilizarán las siguientes expresiones:

$$T_{j1} = \frac{a_{max}}{j_{max}} \quad (106)$$

$$T_a = T_{j1} + \frac{v_{max} - v_0}{a_{max}} \quad (107)$$

Tiempo de desaceleración:

$$Si\ (v_{max} - v_1) \cdot j_{max} < a_{max}^2 \rightarrow a_{min}\ no\ es\ alcanzada \quad (108)$$

Si la aceleración es alcanzada en el intervalo de desaceleración se utiliza la siguiente expresión:

$$T_{j2} = \sqrt{\frac{v_{max} - v_0}{j_{max}}} \quad (109)$$

donde T_{j2} es el tiempo en el cual el jerk es constante durante la fase de desaceleración:

$$T_d = 2 \cdot T_{j2} \quad (110)$$

De otra forma, si no se alcanza, se utilizará la siguiente expresión:

$$T_{j2} = \frac{a_{max}}{j_{max}} \quad (111)$$

$$T_d = T_{j2} + \frac{v_{max} - v_1}{a_{max}} \quad (112)$$

Finalmente, es posible determinar el tiempo T_v en el intervalo de velocidad constante:

$$T_v = \frac{q_1 - q_0}{v_{max}} - \frac{T_a}{2} \cdot \left(1 + \frac{v_0}{v_{max}}\right) - \frac{T_d}{2} \cdot \left(1 + \frac{v_1}{v_{max}}\right) \quad (113)$$

Caso 2: En este caso, si la velocidad constante no existe, la duración de la aceleración y desaceleración se calculará según:

$$T_j = T_{j1} = T_{j2} = \frac{a_{max}}{j_{max}} \quad (114)$$

$$T_a = \frac{\frac{a_{max}^2}{j_{max}} - 2 \cdot v_0 + \sqrt{delta}}{2 \cdot a_{max}} \quad (115)$$

$$T_a = \frac{\frac{a_{max}^2}{j_{max}} - 2 \cdot v_1 + \sqrt{delta}}{2 \cdot a_{max}} \quad (116)$$

donde:

$$delta = \frac{a_{max}^4}{j_{max}^2} + 2 \cdot (v_0^2 + v_1^2) + a_{max} \cdot \cdot \left(4 \cdot (q_1 - q_0) - 2 \cdot \frac{a_{max}}{j_{max}} \cdot (v_0 + v_1)\right) \quad (117)$$

Tanto en el primer como en el segundo caso es posible que la máxima aceleración no se alcance (tanto a_{min} como a_{max}). Esto sucede en aplicaciones en las que existe un desplazamiento pequeño si la aceleración máxima es elevada, o la velocidad inicial está cerca de la máxima velocidad permitida. En estos casos, el segmento de aceleración constante no está presente. Dado esto, se tiene que garantizar que la trayectoria tenga tiempo suficiente para acelerar (desde v_0 a v_{lim}) y decelerar (de v_{lim} a v_1).

Puede suceder que T_a o T_d llegue a ser negativo. En este caso, dependiendo de los valores de velocidad inicial y final, solo será necesario un intervalo de aceleración o desaceleración; $T_a < 0$ quiere decir que el intervalo de aceleración no estará presente.

Finalmente, si se pone a cero el tiempo de aceleración se podrá recalcular los otros intervalos:

$$T_d = 2 \cdot \frac{q_1 - q_0}{v_1 + v_0} \qquad (118)$$

$$T_{j2} = \\ = \frac{j_{max} \cdot (q_1 - q_0) - \sqrt{j_{max} \cdot (j_{max} \cdot (q_1 - q_0)^2 + (v_1 + v_0)^2 \cdot (v_1 - v_0)}}{j_{max} \cdot (v_1 + v_0)} \qquad (119)$$

main.h

```
1  #ifndef MAIN_H
2  #define MAIN_H
3
4      #include "mbed.h"
5
6      #include "Ro/Vector.h"
7      #include "Ro/Matrix.h"
8
9      #include "Features.h"
10     #include "Ro/Robot.h"
11
12     #include "Ro/MotionCommand.h"
13     #include "Ro/Control.h"
14
15     #include "Ro/Driver.h"
16     #include "Ro/TransformationMatrix.h"
17 #endif //MAIN_H
```

main.cpp

```
1  #include "main.h"
2
3  int main() {
4      float q[2]={1,2};
5      const Link::Links_Type Linker[2] ={Link::REVOLUTION,Link::REVOLUTION};
6      Robot p560(ToolMatrix,BaseMatrix,DH_Table,LinkLenght,dyn_mat,MotorParam,
   ControlGain,RobotDOF,Linker,Constraints,PinSensorA, PinSensorB,SensorProperties,
   PinDriver);
7
8      return 0;
9  }
```

```
1  //
2  // Archivo cabezera para la configuración del eslabón.
3  //
4
5
6  #ifndef UNTITLED_LINK_H
7  #define UNTITLED_LINK_H
8  #include "Vector.h"
9  #include "Matrix.h"
10
11 class Link {
12 public:
13     enum Links_Type {REVOLUTION, LINEAR};
14     Link();
15     float ReturnLenght();
16
17 //private:
18     friend class Robot;
19     Matrix<float> Inertia_tensor;
20     Vector<float> Cent_Gravity;
21     enum Links_Type Type;
22     float theta;
23     float alpha;
24     float d;
25     float a;
26     Matrix<float> A0;
27
28     float Lenght;
29     float Mass;
30     float Tau;
31
32     Vector<float> w;
33     Vector<float> wd;
34     Vector<float> vd;
35     Vector<float> acc;
36     Vector<float> f;
37     Vector<float> M;
38
39
40 };
41
42
43 #endif //UNTITLED_LINK_H
44
```

```
1  //
2  // Archivo cabezera donde se almacenan las especificaciones de cada motor.
3  //
4
5  #ifndef UNTITLED_MOTOR_H
6  #define UNTITLED_MOTOR_H
7  #include "Vector.h"
8  #include "Sensor.h"
9  #include "Control.h"
10 #include "Driver.h"
11
12 class Motor {
13     float Degrees_Kinematics;
14 private:
15     friend class Robot;
16     Vector<float> pcg;
17     float Constrains_v;
18     float Constrains_a;
19     float Constrains_j;
20     Sensor* _Sensor;
21     Control* _Control;
22     Driver* _Driver;
23 public:
24     Motor();
25 };
26
27
28 #endif //UNTITLED_MOTOR_H
29
```

```
1  //
2  // Archivo cabezera donde se muestran los distintos métodos a utilizar para la clase Robot.
3  //
4
5
6  #ifndef UNTITLED_ROBOT_H
7  #define UNTITLED_ROBOT_H
8  #include "mbed.h"
9
10 #include "Vector.h"
11 #include "Matrix.h"
12
13 #include "Link.h"
14 #include "Motor.h"
15
16 #include "Trajectory.h"
17 #include "End_Effector.h"
18 #include <cmath>
19
20 class Robot {
21
22 public:
23     Robot(float _ToolMatrix[][4],float _BaseMatrix[][4],const float DH_Table[], const float
    lenght[],float dyn_mat [][10],const float MotorParam[],const float ControlGain[],int
    RobotDOF,const Link::Links_Type link [], const float MotorConstraints[],PinName
    PinSensorA[], PinName PinSensorB[],const float SensorProperties[], PinName PinDriver[]);
24     Robot(Matrix<float>& _ToolMatrix,Matrix<float>& _BaseMatrix,const float DH_Table
    [], const float lenght[],float dyn_mat [][10],const float MotorParam[],const float
    ControlGain[],int RobotDOF,const Link::Links_Type link [], const float MotorConstraints[],
    PinName PinSensorA[], PinName PinSensorB[],const float SensorProperties[], PinName
    PinDriver[]);
25     ~Robot();
26
27     Vector<float> GetTorque();
28     int GetDOF();
29
30     float ReadSensor(int num);
31     float ReadVelocity(float prevValue,float sampling,int num);
32
33     Vector<float> ReturnSaturationControl(Vector<float> value);
34     Vector<float> ReturnComputeP(Vector<float> qref, Vector<float> qi);
35     Vector<float> ReturnComputePD_FirstVersion(Vector<float> qref, Vector<float> qi,
    Vector<float> qdotref, Vector<float> qdoti);
36     Vector<float> ReturnComputePD_SecondVersion(Vector<float> qref, Vector<float> qi,
    Vector<float> qdoti);
37     Vector<float> ReturnComputePID(Vector<float> qref, Vector<float> qi, Vector<float>
    qdotref, Vector<float> qdoti);
38     Vector<float> ReturnComputePDG(Vector<float> qref, Vector<float> qi, Vector<float>
    qdotref, Vector<float> qdoti, Vector<float> Tau);
39
40     int SendData(Vector<float> data);
41
42     Matrix <float> GetRotation(int num);
43     Matrix <float> GetRotation(Matrix<float> mat);
44     Vector <float> GetPosition(Matrix<float> vect);
45
```

```
46      Link GetLink(int num);
47
48      float ConstraintsVelocityMotor(int num);
49      float ConstraintsAccelerationMotor(int num);
50      float ConstraintsJerkMotor(int num);
51
52      Vector<float> GetKP();
53      Vector<float> GetKV();
54      Vector<float> GetKI();
55      Vector<float> GetEndPosition();
56      Matrix<float> GetJacobian();
57
58
59      void ForwardKinematics(const float thetaOffset[]);
60      Vector<float> InverseKinematics(End_Effector* EFF_q0, int conf);
61      void Jacobian(float q[]);
62      void EulerNewton(const float q[],const float qd[],const float qdd[]);
63      void GravityTorque(float q[]);
64
65
66  private:
67      Matrix <float> ToolMatrix;
68      Matrix <float> BaseMatrix;
69      Matrix <float> _Jacobian;
70
71      Link *_Linker;
72      Motor *_Motor;
73
74      int _RobotDOF;
75  };
76
77
78  #endif //UNTITLED_ROBOT_H
79
```

```
1  //
2  // Archivo cabezera para la configuración de los drivers que actúan sobre los motores.
3  //
4
5  #ifndef DIRVER_H
6  #define DIRVER_H
7     #include "mbed.h"
8     enum TypeCommunication {D_CAN, D_DAC,D_SPI};
9  class Driver {
10    private:
11       TypeCommunication communication;
12       CAN *can;
13       SPI *spi;
14       AnalogOut *dac;
15    public:
16       Driver(PinName adrress);
17       Driver(PinName RX,PinName TX);
18       Driver(PinName MOSI,PinName MISO,PinName sclk, int frecuency);
19       int sendmessage(char data[8]);
20       int sendmessage(float data);
21       ~Driver();
22 };
23 #endif //DIRVER_H
```

```
1  //
2  // Algoritmo perteneciente a la clase eslabón, donde se detallan las caracteristicas del propio
   eslabón.
3  //
4
5  #include "Link.h"
6  Link::Link() : A0(4,4){
7
8      theta = 0;
9      alpha = 0;
10     d = 0;
11     a = 0;
12
13
14     Lenght = 0;
15     Mass = 0;
16     Tau = 0;
17 }
18 float Link::ReturnLenght(){
19     return Lenght;
20 }
```

```
1  //
2  // Archivo cabezera de la clase matriz.
3  //
4
5  #ifndef UNTITLED_MATRIX_H
6  #define UNTITLED_MATRIX_H
7
8
9  #ifndef AA_MATRIX_H
10 #define AA_MATRIX_H
11 #include <iostream>
12 #include "Vector.h"
13 using namespace std;
14
15
16 template<class T>
17 class Matrix{
18 protected:
19     int rows;
20     int columns;
21     static int objectCount;
22     T **n;
23
24 public:
25     Matrix();
26     Matrix(int rows,int columns);
27     ~Matrix();
28
29     void PrintResult();
30     void Reconfigurate(int rows,int columns);
31     T Determinant();
32     Matrix<T> Cofactor();
33     Matrix<T> Inverse();
34     Matrix<T> PseudoInverse();
35     Matrix<T> Transpose();
36
37     Matrix& operator=(const Matrix &other );
38     Matrix& operator=(const T param[][2]);
39     Matrix& operator=(const T param[][3]);
40     Matrix& operator=(const T param[][4]);
41     Matrix operator+(const Matrix &other);
42     Matrix operator*(const Matrix& other);
43     Vector<T> operator*(Vector<T>& other);
44     T& operator()(int row, int col);
45
46
47 };
48
49 #endif //AA_MATRIX_H
50
51 #endif //UNTITLED_MATRIX_H
52
```

```
1  //
2  // Archivo cabezera para la configuración de los sensores.
3  //
4
5  #ifndef SENSOR_H
6  #define SENSOR_H
7     #include "mbed.h"
8     #include "QEI/QEI.h"
9     using namespace std;
10    class Sensor {
11       private:
12          float rank;
13          float scale_factor;
14          float offset;
15          float value;
16          QEI Encoder;
17       public:
18          Sensor(PinName channelA,PinName channelB,float _rank, float _sFactor,float _offset
    , int _pulsesPerRev);
19          float readSensor();
20          float computevalue(int value);
21          float estimatevelocity (float prevValue,float sampling);
22    };
23 #endif //SENSOR_H
```

Vector.h

```
1  //
2  // Archivo cabezera de la clase vector.
3  //
4
5  #ifndef UNTITLED_VECTOR_H
6  #define UNTITLED_VECTOR_H
7  #include <iostream>
8  #include <cstdio>
9  using namespace std;
10
11 template<class T>
12 class Vector {
13    private:
14       int dimension;
15       static int objectCount;
16       T *n;
17    public:
18       Vector();
19       Vector(int dimension);
20       ~Vector();
21    int getDimension();
22    void PrintResult();
23    Vector<T> crossProduct(Vector<T> const& VectorB);
24
25
26    Vector& operator=(const Vector &other );
27
28    Vector& operator=(T param[]);
29
30    Vector operator+(Vector<T> other);
31    Vector operator-(Vector<T>& other);
32    T operator*(Vector<T>& other);
33    Vector operator*(T factor);
34    T& operator()(int dim);
35 };
36
37 Vector<float> zeros(int num);
38
39
40 Vector<float> crossProduct(Vector<float> VectorA,Vector<float> VectorB);
41 #endif //UNTITLED_VECTOR_H
42
```

```
1  //
2  // Archivo cabezera para un control de un brazo robot.
3  //
4  #ifndef CONTROL_H
5  #define CONTROL_H
6     #include "mbed.h"
7     using namespace std;
8     class Control{
9        private:
10          float KP;
11          float KV;
12          float KI;
13          float KA;
14          float KM;
15          float Scale;
16          float Offset;
17          float MaxTorque;
18          float SamplingTime;
19          float PrevErr;
20       public:
21          Control(float _KP, float _KV, float _KI,float _KA,float _KM ,float _SamplingTime,
   float _Scale, float _Offset,float _MaxTorque);
22          float GetKP();
23          float GetKV();
24          float GetKI();
25          float SaturationControl(float value);
26          float ComputeP(float qref, float qi);
27          float ComputePD_FirstVersion(float qref, float qi, float qdotref, float qdoti);
28          float ComputePD_SecondVersion(float qref, float qi, float qdoti);
29          float ComputePID(float qref, float qi, float qdotref, float qdoti);
30          float ComputePDG(float qref, float qi, float qdotref, float qdoti, float Tau);
31    };
32 #endif //CONTROL_H
```

```cpp
//
// Algoritmo donde se muestran las distintas condiciones límite de cada clase motor.
//

#include "Motor.h"
Motor::Motor(){
    Constrains_v = 0;
    Constrains_a = 0;
    Constrains_j = 0;
}
```

```
1  //
2  // Algoritmo principal donde se calcula la parte cinemática y dinámica de un robot.
3  //
4
5  #include "Ro/Robot.h"
6  Robot::Robot(float _ToolMatrix[][4],float _BaseMatrix[][4],const float DH_Table[], const
   float lenght[],float dyn_mat [][10],const float MotorParam[],const float ControlGain[],int
   RobotDOF,const Link::Links_Type link [], const float MotorConstraints[],PinName
   PinSensorA[], PinName PinSensorB[], const float SensorProperties[], PinName PinDriver[]):
   ToolMatrix(4,4),BaseMatrix(4,4),_Jacobian(6,RobotDOF){
7     _RobotDOF = RobotDOF;
8     ToolMatrix = _ToolMatrix;
9     BaseMatrix = _BaseMatrix;
10
11    _Linker = new Link[_RobotDOF];
12    _Motor = new Motor[_RobotDOF];
13
14    for(int i=0;i<_RobotDOF; i++){
15
16        _Motor[i]._Control= new Control(ControlGain[i*3], ControlGain[i*3+1], ControlGain[i
   *3+2],
17                                        MotorParam[i*5],MotorParam[i*5+1] ,20, MotorParam[i*5+2],
   MotorParam[i*5+3],MotorParam[i*5+4]);
18        _Motor[i]._Sensor= new Sensor(PinSensorA[i], PinSensorB[i],SensorProperties[(i*4)],
   SensorProperties[(i*4)+1],SensorProperties[(i*4)+2], SensorProperties[(i*4)+3]);
19        _Motor[i]._Driver= new Driver(PinDriver[i]);
20
21        _Motor[i].Constrains_v = MotorConstraints[(i*3)];
22        _Motor[i].Constrains_a = MotorConstraints[(i*3)+1];
23        _Motor[i].Constrains_j = MotorConstraints[(i*3)+2];
24
25        _Linker[i].Type = link[i];
26        _Linker[i].Lenght = lenght[i];
27        _Linker[i].theta = DH_Table[((i*4)+0)];
28        _Linker[i].alpha = DH_Table[((i*4)+1)];
29        _Linker[i].d = DH_Table[((i*4)+2)];
30        _Linker[i].a = DH_Table[((i*4)+3)];
31
32        _Linker[i].Inertia_tensor(0,0) = dyn_mat[i][1]+dyn_mat[i][2];
33        _Linker[i].Inertia_tensor(0,1) = dyn_mat[i][3];
34        _Linker[i].Inertia_tensor(0,2) = dyn_mat[i][4];
35
36        _Linker[i].Inertia_tensor(1,0) = dyn_mat[i][3];
37        _Linker[i].Inertia_tensor(1,1) = dyn_mat[i][0]+dyn_mat[i][2];
38        _Linker[i].Inertia_tensor(1,2) = dyn_mat[i][5];
39
40        _Linker[i].Inertia_tensor(2,0) = dyn_mat[i][4];
41        _Linker[i].Inertia_tensor(2,1) = dyn_mat[i][5];
42        _Linker[i].Inertia_tensor(2,2) = dyn_mat[i][1]+dyn_mat[i][0];
43
44        _Linker[i].Cent_Gravity(0)= dyn_mat[i][6];
45        _Linker[i].Cent_Gravity(1) = dyn_mat[i][7];
46        _Linker[i].Cent_Gravity(2) = dyn_mat[i][8];
47
48        _Linker[i].Mass = dyn_mat[i][9];
49        for(int x=0; x<4; x++) {
```

```
50            for (int y = 0; y < 4; y++) {
51                _Linker[i].A0(x,y)=0;
52            }
53        }
54    }
55 }
56 Robot::Robot(Matrix<float>& _ToolMatrix,Matrix<float>& _BaseMatrix,const float
   DH_Table[], const float lenght[],float dyn_mat [][10],const float MotorParam[],const float
   ControlGain[],int RobotDOF,const Link::Links_Type link [], const float MotorConstraints
   [],PinName PinSensorA[], PinName PinSensorB[],const float SensorProperties[], PinName
   PinDriver[]):ToolMatrix(4,4),BaseMatrix(4,4),_Jacobian(6,RobotDOF){
57    _RobotDOF = RobotDOF;
58    ToolMatrix = _ToolMatrix;
59    BaseMatrix = _BaseMatrix;
60
61    _Linker = new Link[_RobotDOF];
62    _Motor = new Motor[_RobotDOF];
63
64    for(int i=0;i<_RobotDOF; i++){
65        _Motor[i]._Control= new Control(ControlGain[i*3], ControlGain[i*3+1], ControlGain[
   i*3+2],
66                                        MotorParam[i*5],MotorParam[i*5+1] ,20, MotorParam[i*5+2
   ], MotorParam[i*5+3],MotorParam[i*5+4]);
67        _Motor[i]._Sensor= new Sensor(PinSensorA[i], PinSensorB[i],SensorProperties[(i*4)],
   SensorProperties[(i*4)+1],SensorProperties[(i*4)+2], SensorProperties[(i*4)+3]);
68        _Motor[i]._Driver= new Driver(PinDriver[i]);
69
70        _Motor[i].Constrains_v = MotorConstraints[(i*3)];
71        _Motor[i].Constrains_a = MotorConstraints[(i*3)+1];
72        _Motor[i].Constrains_j = MotorConstraints[(i*3)+2];
73
74        _Linker[i].Type = link[i];
75        _Linker[i].Lenght = lenght[i];
76        _Linker[i].theta = DH_Table[((i*4)+0)];
77        _Linker[i].alpha = DH_Table[((i*4)+1)];
78        _Linker[i].d = DH_Table[((i*4)+2)];
79        _Linker[i].a = DH_Table[((i*4)+3)];
80
81        _Linker[i].Inertia_tensor(0,0) = dyn_mat[i][1]+dyn_mat[i][2];
82        _Linker[i].Inertia_tensor(0,1) = dyn_mat[i][3];
83        _Linker[i].Inertia_tensor(0,2) = dyn_mat[i][4];
84
85        _Linker[i].Inertia_tensor(1,0) = dyn_mat[i][3];
86        _Linker[i].Inertia_tensor(1,1) = dyn_mat[i][0]+dyn_mat[i][2];
87        _Linker[i].Inertia_tensor(1,2) = dyn_mat[i][5];
88
89        _Linker[i].Inertia_tensor(2,0) = dyn_mat[i][4];
90        _Linker[i].Inertia_tensor(2,1) = dyn_mat[i][5];
91        _Linker[i].Inertia_tensor(2,2) = dyn_mat[i][1]+dyn_mat[i][0];
92
93        _Linker[i].Cent_Gravity(0)= dyn_mat[i][6];
94        _Linker[i].Cent_Gravity(1) = dyn_mat[i][7];
95        _Linker[i].Cent_Gravity(2) = dyn_mat[i][8];
96
97        _Linker[i].Mass = dyn_mat[i][9];
98
```

```
99          for(int x=0; x<4; x++) {
100             for (int y = 0; y < 4; y++) {
101                 _Linker[i].A0(x,y)=0;
102             }
103         }
104     }
105 }
106 Robot::~Robot(){
107     for(int i=0;i<_RobotDOF; i++){
108         delete[] _Motor[i]._Control;
109         delete[] _Motor[i]._Sensor;
110         delete[] _Motor[i]._Driver;
111     }
112     delete[] _Linker;
113     delete[] _Motor;
114 }
115 Vector<float> Robot::GetTorque(){
116     Vector<float> result(_RobotDOF);
117     for(int i=0; i<_RobotDOF; i++){
118         result(i) = _Linker[i].Tau;
119     }
120     return result;
121 }
122 Matrix <float> Robot::GetRotation(int num){
123     Matrix<float> result(3,3);
124     result = _Linker[num].A0;
125     return result;
126 }
127 Matrix <float> Robot::GetRotation(Matrix<float> mat){
128     Matrix<float> result(3,3);
129     result = mat;
130     return result;
131 }
132 Vector <float>  Robot::GetPosition(Matrix<float> mat){
133     Vector<float> result(3);
134     for(int i=0; i<3;i++) {
135         result(i) = mat(i,3);
136     }
137     return result;
138 }
139
140 Link Robot::GetLink(int num){
141     return _Linker[num];
142 }
143 Matrix<float> Robot::GetJacobian(){
144     return _Jacobian;
145 }
146
147 float Robot::ConstraintsVelocityMotor(int num){
148     return _Motor[num].Constrains_v;
149 }
150 float Robot::ConstraintsAccelerationMotor(int num){
151     return _Motor[num].Constrains_a;
152 }
153 float Robot::ConstraintsJerkMotor(int num){
154     return _Motor[num].Constrains_j;
```

```
155 }
156
157 Vector<float> Robot::GetKP(){
158     Vector<float> result(_RobotDOF);
159     for(int i=0; i<_RobotDOF; i++){
160         result(i) = _Motor[i]._Control->GetKP();
161     }
162     return result;
163 }
164 Vector<float> Robot::GetKV(){
165     Vector<float> result(_RobotDOF);
166     for(int i=0; i<_RobotDOF; i++){
167         result(i) = _Motor[i]._Control->GetKV();
168     }
169     return result;
170 }
171 Vector<float> Robot::GetKI(){
172     Vector<float> result(_RobotDOF);
173     for(int i=0; i<_RobotDOF; i++){
174         result(i) = _Motor[i]._Control->GetKI();
175     }
176     return result;
177 }
178
179
180 Vector<float> Robot::GetEndPosition(){
181     Vector<float> result(3);
182     Matrix<float> A(4,4);
183     A = _Linker[0].A0;
184     for(int i=1; i<_RobotDOF; i++){
185         A = A*_Linker[i] .A0;
186     }
187     result(0) = A(0,3);
188     result(1) = A(1,3);
189     result(2) = A(2,3);
190     return result;
191 }
192 int Robot::GetDOF(){
193     return _RobotDOF;
194 }
195 float Robot::ReadSensor(int num){
196     return _Motor[num]._Sensor->readSensor();
197 }
198 float Robot::ReadVelocity(float prevValue,float sampling,int num){
199     return _Motor[num]._Sensor->estimatevelocity (prevValue,sampling);
200 }
201 Vector<float> Robot::ReturnSaturationControl(Vector<float> value){
202     Vector<float> result(_RobotDOF);
203     for(int i=0; i<_RobotDOF; i++){
204         result(i) = _Motor[i]._Control->SaturationControl(value(i));
205     }
206     return result;
207 }
208 Vector<float> Robot::ReturnComputeP(Vector<float> qref, Vector<float> qi){
209     Vector<float> result(_RobotDOF);
210     for(int i=0; i<_RobotDOF; i++){
```

```
211         result(i) = _Motor[i]._Control->ComputeP(qref(i), qi(i));
212     }
213     return result;
214 }
215 Vector<float> Robot::ReturnComputePD_FirstVersion(Vector<float> qref, Vector<float>
    qi, Vector<float> qdotref, Vector<float> qdoti){
216     Vector<float> result(_RobotDOF);
217     for(int i=0; i<_RobotDOF; i++){
218         result(i) = _Motor[i]._Control->ComputePD_FirstVersion(qref(i), qi(i), qdotref(i), qdoti
    (i));
219     }
220     return result;
221 }
222 Vector<float> Robot::ReturnComputePD_SecondVersion(Vector<float> qref, Vector<float
    > qi, Vector<float> qdoti){
223     Vector<float> result(_RobotDOF);
224     for(int i=0; i<_RobotDOF; i++){
225         result(i) = _Motor[i]._Control->ComputePD_SecondVersion(qref(i), qi(i), qdoti(i));
226     }
227     return result;
228 }
229 Vector<float> Robot::ReturnComputePID(Vector<float> qref, Vector<float> qi, Vector<
    float> qdotref, Vector<float> qdoti){
230     Vector<float> result(_RobotDOF);
231     for(int i=0; i<_RobotDOF; i++){
232         result(i) = _Motor[i]._Control->ComputePID(qref(i), qi(i), qdotref(i), qdoti(i));
233     }
234     return result;
235 }
236 Vector<float> Robot::ReturnComputePDG(Vector<float> qref, Vector<float> qi, Vector<
    float> qdotref, Vector<float> qdoti, Vector<float> Tau){
237     Vector<float> result(_RobotDOF);
238     for(int i=0; i<_RobotDOF; i++){
239         result(i) = _Motor[i]._Control->ComputePDG(qref(i), qi(i), qdotref(i), qdoti(i), Tau(i));
240     }
241     return result;
242 }
243 int Robot::SendData(Vector<float> data){
244     int isCorrect;
245     for(int i=0; i<_RobotDOF; i++){
246         isCorrect = _Motor[i]._Driver->sendmessage(data(i));
247         if(isCorrect == 0){
248             return 0;
249         }
250     }
251     return 1;
252 }
253 void Robot::ForwardKinematics(const float thetaOffset[]){
254     float STheta,CTheta,SAlpha,CAlpha,theta,d;
255     for (int i=0; i<_RobotDOF;i++) {
256         if ( _Linker[i].Type == Link::REVOLUTION) {
257             theta = thetaOffset[i];
258             d =  _Linker[i].d;
259         }else{
260             d =  thetaOffset[i];
261             theta =  _Linker[i].theta;
```

```
262         }
263         STheta = sinf(theta);
264         CTheta = cosf(theta);
265         SAlpha = sinf(  _Linker[i].alpha);
266         CAlpha = cosf(  _Linker[i].alpha);
267         if( (CAlpha>-0.000001F)&&(CAlpha<0.000001F)){
268            CAlpha = 0;
269         }
270         if( (SAlpha>-0.000001F)&&(SAlpha<0.000001F)){
271            SAlpha = 0;
272         }
273         if( (STheta>-0.000001F)&&(STheta<0.000001F)){
274            STheta = 0;
275         }
276         if( (CTheta>-0.000001F)&&(CTheta<0.000001F)){
277            CTheta = 0;
278         }
279         for (int x = 0; x < 4; x++) {
280            for (int y = 0; y < 4; y++) {
281               if (x == 0 && y == 0) {
282                  _Linker[i] .A0 (x,y)  = CTheta;
283               } else if (x == 0 && y == 1) {
284                  _Linker[i] .A0  (x,y)  = -1 * CAlpha * STheta;
285               } else if (x == 0 && y == 2) {
286                  _Linker[i] .A0  (x,y)  = SAlpha * STheta;
287               } else if (x == 0 && y == 3) {
288                  _Linker[i] .A0  (x,y)  =  _Linker[i] .a * CTheta;
289               } else if (x == 1 && y == 0) {
290                  _Linker[i] .A0  (x,y)  = STheta;
291               } else if (x == 1 && y == 1) {
292                  _Linker[i] .A0  (x,y)  = CAlpha * CTheta;
293               } else if (x == 1 && y == 2) {
294                  _Linker[i] .A0  (x,y)  = -1 * SAlpha * CTheta;
295               } else if (x == 1 && y == 3) {
296                  _Linker[i] .A0  (x,y)  =  _Linker[i] .a * STheta;
297               } else if (x == 2 && y == 1) {
298                  _Linker[i] .A0  (x,y)  = SAlpha;
299               } else if (x == 2 && y == 2) {
300                  _Linker[i] .A0  (x,y)  = CAlpha;
301               } else if (x == 2 && y == 3) {
302                  _Linker[i] .A0  (x,y)  = d;
303               } else if (x == 3 && y == 3) {
304                  _Linker[i] .A0  (x,y)  = 1;
305               }
306               else{
307                  _Linker[i] .A0  (x,y) =0;
308               }
309               if( ( _Linker[i] .A0  (x,y) >-0.000001F)&&( _Linker[i] .A0  (x,y) )<0.000001F){
310                  _Linker[i] .A0  (x,y) =0;
311               }
312
313            }
314         }
315      }
316   }
317
```

```
318  Vector<float> Robot::InverseKinematics(End_Effector* EFF_q0, int conf){
319    /*USER SPECIFICATIONS*/
320    Vector<float> result (_RobotDOF);
321    float Xf = EFF_q0->x;
322    float Yf = EFF_q0->y;
323    float fi = EFF_q0->pitch;
324    float theta = EFF_q0->roll;
325
326    float L1 = _Linker[0].ReturnLenght();
327    float L2 = _Linker[1].ReturnLenght();
328    float L3 = _Linker[2].ReturnLenght();
329    float L4 = _Linker[3].ReturnLenght();
330
331    //Wrist points
332    float Cfi = cosf(fi);
333    float Sfi = sinf(fi);
334    if ((Cfi > -0.000001F) && (Cfi < 0.000001F)) {
335       Cfi = 0;
336    }
337    if ((Sfi > -0.000001F) && (Sfi < 0.000001F)) {
338       Sfi = 0;
339    }
340    float Xm = Xf - L4 * Cfi;
341    float Ym = Yf - L4 * Sfi;
342
343    //Calc articular coordinates
344    //q4
345    result(3) = theta;
346
347    //q2
348    float C2 = ((powf(Xm, 2.0F)) + (powf(Ym, 2.0F)) - (powf(L1, 2.0F)) - (powf(L2, 2.0F
     ))) / (2 * L1 * L2);
349    float S2 = sqrtf(1 - powf(C2, 2.0F));
350    float q21 = atan2f(S2, C2);
351    float q22 = atan2f(-S2, C2);
352    result (1) = q21;
353
354    //q1
355    float alpha = atan2f((L2 * sinf(result(1))), (L1 + L2 * cosf(result(1))));
356    float beta_1 = atan2f(Ym, Xm);
357    float beta_2 = atan2f(Ym, -Xm);
358
359    float q1;
360    if (conf == 0) {
361       //Elbow up
362       q1 = beta_1 - alpha;
363    } else {
364       //Elbow down
365       q1 = beta_2 - alpha;
366    }
367    result (0)= q1;
368    //q3
369    result (2) = fi-result(0)-result(1);
370    return result;
371 }
372 void Robot::Jacobian(float q[]){
```

```
373     ForwardKinematics(q);
374     Matrix<float> A0(4,4);
375     A0 = BaseMatrix;
376     for(int i=0; i<_RobotDOF; i++){
377        A0 = A0 *_Linker[i].A0;
378     }
379     struct Vector<float> p0_dof;
380     p0_dof(0) = A0(0,3);
381     p0_dof(1) = A0(1,3);
382     p0_dof(2) = A0(2,3);
383
384     Vector<float> p[_RobotDOF];
385     Vector<float>  pA0;
386     pA0(0) = BaseMatrix(0,3);
387     pA0(1) = BaseMatrix(1,3);
388     pA0(2) = BaseMatrix(2,3);
389
390     p[0] = p0_dof - pA0;
391
392     Vector<float> z[_RobotDOF];
393     Vector<float> z0;
394     z0(0) = BaseMatrix(0,2);
395     z0(1) = BaseMatrix(1,2);
396     z0(2) = BaseMatrix(2,2);
397
398     z[0]=z0;
399
400
401     Matrix<float> A01;
402     A01 = BaseMatrix * _Linker[0].A0;
403     Vector<float> Jv(3);
404     Vector<float> Jw(3);
405
406     Matrix<float> J(6,_RobotDOF);
407     if(_Linker[0].Type == Link::REVOLUTION){
408        Jv = z[0].crossProduct(p[0]);
409        Jw = z[0];
410     }else{
411        Jv = z[0];
412        Jw = zeros(3);
413     }
414        J(1,0) = Jv(0);
415        J(2,0) = Jv(1);
416        J(3,0) = Jv(2);
417        J(4,0) = Jw(0);
418        J(5,0) = Jw(1);
419        J(6,0) = Jw(2);
420
421     for(int i=1;i<_RobotDOF;i++){
422        pA0(0) = A01(0,3);
423        pA0(1) = A01(1,3);
424        pA0(2) = A01(2,3);
425
426        z0(0) = A01(0,2);
427        z0(1) = A01(1,2);
428        z0(2) = A01(2,2);
```

```
429
430         p[i] = p0_dof - pA0;
431         z[i] = z0;
432
433         if(_Linker[i].Type == Link::REVOLUTION){
434             Jv = z[i].crossProduct(p[i]);
435             Jw = z[i];
436         }else{
437             Jv = z[i];
438             Jw = zeros(3);
439         }
440
441         A01 = A01 * _Linker[i].A0;
442         J(0,i) = Jv(0);
443         J(1,i) = Jv(1);
444         J(2,i) = Jv(2);
445         J(3,i) = Jw(0);
446         J(4,i) = Jw(1);
447         J(5,i) = Jw(2);
448     }
449     _Jacobian = J;
450 }
451
452 void Robot::EulerNewton(const float q[],const float qd[],const float qdd[]){
453     Matrix<float> Rtot(3,3);
454
455     //Get inverse rotation
456     ForwardKinematics(q);
457     Matrix<float> R3_post_inv = GetRotation(0).Inverse();
458     Rtot = GetRotation(0);
459
460     //Vector join Si-1 and Si
461     Vector<float> R_pi(3);
462
463     R_pi(0) = _Linker[0].a;
464     if (( R_pi(0) > -0.000001F) && (R_pi(0)) < 0.000001F) {
465         R_pi(0) = 0;
466     }
467
468     R_pi(1) = _Linker[0].d*sinf(_Linker[0].alpha);
469     if (( R_pi(1) > -0.000001F) && (R_pi(1)) < 0.000001F) {
470         R_pi(1) = 0;
471     }
472
473     R_pi(2) = _Linker[0].d*cosf(_Linker[0].alpha);
474     if (( R_pi(2) > -0.000001F) && (R_pi(2)) < 0.000001F) {
475         R_pi(2) = 0;
476     }
477
478     //Center of gravity
479     Vector<float> s;
480     if(_Linker[0].Mass !=0 ) {
481         s(0) = _Linker[0].Cent_Gravity(0) / _Linker[0].Mass;
482         s(1) = _Linker[0].Cent_Gravity(1) / _Linker[0].Mass;
483         s(2) = _Linker[0].Cent_Gravity(2) / _Linker[0].Mass;
484     }else{
```

```
485         s = zeros(3);
486     }
487
488     //
489     float Z0[3] = {0,0,1};
490     Vector<float> z0;
491     z0 = Z0;
492
493     float g0[3] = {0,9.81F,0};
494     Vector<float> g;
495     g = g0;
496
497     Vector<float> w,wd,vd,a,z03qd,z03qdd;
498
499     if (_Linker[0].Type == Link::REVOLUTION){
500         z03qd = z0 * qd[0];
501         z03qdd = z0 * qdd[0];
502        w  =  R3_post_inv * z03qd;
503
504        wd  =  R3_post_inv * z03qdd;
505
506         vd = (wd.crossProduct(R_pi)) + w.crossProduct(w.crossProduct(R_pi)) + (R3_post_inv
    * g);
507     }else{
508         z03qdd = (z0 * qdd[0]) + g;
509         w = zeros(3);
510         wd = zeros(3);
511         vd = R3_post_inv * z03qdd;
512     }
513     a = (wd.crossProduct(s)) + (w.crossProduct(w.crossProduct(s))) + (vd);
514
515     _Linker[0].w =  w;
516     _Linker[0].wd = wd;
517     _Linker[0].vd = vd;
518     _Linker[0].acc = a;
519
520     Vector<float> aux;
521
522     for (int i=1; i<_RobotDOF; i++){
523         //Get inverse rotation
524         Rtot = Rtot * GetRotation(i);
525         R3_post_inv = GetRotation(i).Inverse();
526
527         //Vector join Si-1 and Si
528         R_pi(0) = _Linker[i].a;
529         R_pi(1) = _Linker[i].d*sinf(_Linker[i].alpha);
530         R_pi(2) = _Linker[i].d*cosf(_Linker[i].alpha);
531
532         //Center of gravity
533         s(0) = _Linker[i].Cent_Gravity(0)/ _Linker[i].Mass;
534         s(1) = _Linker[i].Cent_Gravity(1)/ _Linker[i].Mass;
535         s(2) = _Linker[i].Cent_Gravity(2)/ _Linker[i].Mass;
536
537         z03qd = z0 * qd[i];
538         z03qdd = z0 * qdd[i];
539
```

```
540         if (_Linker[i].Type == Link::REVOLUTION){
541 
542             aux = _Linker[i-1].w + z03qd;
543             w  = R3_post_inv * aux;
544             aux = _Linker[i-1].wd + z03qdd;
545             wd = R3_post_inv * aux + (_Linker[i-1].w.crossProduct(z03qd));
546             vd = (wd.crossProduct(R_pi)) + (w.crossProduct(w.crossProduct(R_pi))) + (
    R3_post_inv * _Linker[i-1].vd);
547 
548         }else{
549 
550             w = R3_post_inv * _Linker[i-1].w;
551             wd = R3_post_inv * _Linker[i-1].wd;
552             aux = z03qdd + _Linker[i-1].vd;
553             vd = R3_post_inv * aux + (w.crossProduct(R_pi)) + ((w * 2.0F).crossProduct(
    R3_post_inv*z0*qd[i]))
554                 + (w.crossProduct(w.crossProduct(R_pi)));
555         }
556 
557         a = (wd.crossProduct(s)) + (w.crossProduct(w.crossProduct(s))) + vd;
558         _Linker[i].w = w;
559         _Linker[i].wd = wd;
560         _Linker[i].vd = vd;
561         _Linker[i].acc = a;
562 
563     }
564 
565     //Vector join Si-1 and Si
566     R_pi(0) = _Linker[_RobotDOF-1].a;
567     R_pi(1) = _Linker[_RobotDOF-1].d*sinf(_Linker[_RobotDOF-1].alpha);
568     R_pi(2) = _Linker[_RobotDOF-1].d*cosf(_Linker[_RobotDOF-1].alpha);
569 
570     //Center of gravity
571     s(0) = _Linker[_RobotDOF-1].Cent_Gravity(0)/ _Linker[_RobotDOF-1].Mass;
572     s(1) = _Linker[_RobotDOF-1].Cent_Gravity(1)/ _Linker[_RobotDOF-1].Mass;
573     s(2) = _Linker[_RobotDOF-1].Cent_Gravity(2)/ _Linker[_RobotDOF-1].Mass;
574 
575 
576 
577     // Fuerza ejercida sobre i en el cdm
578     Vector<float> f_cdm;
579     Vector<float> M_cdm;
580     Vector<float> f;
581     Vector<float> M;
582 
583     f_cdm = _Linker[_RobotDOF-1].acc * _Linker[_RobotDOF-1].Mass;
584     M_cdm = (_Linker[_RobotDOF-1].Inertia_tensor * _Linker[_RobotDOF-1].wd)  +
585           (_Linker[_RobotDOF-1].w.crossProduct(_Linker[_RobotDOF-1].Inertia_tensor *
    _Linker[_RobotDOF-1].w));
586 
587 
588     f = f_cdm * -1;
589 
590     M = (R_pi + s).crossProduct(f_cdm) + M_cdm;
591 
592     float Tau;
```

```
593     if (_Linker[_RobotDOF-1].Type == Link::REVOLUTION){
594        Tau = (R3_post_inv.Transpose() * M) * z0;
595     }else{
596        Tau = (R3_post_inv.Transpose() * f) * z0;
597     }
598     _Linker[_RobotDOF-1].Tau = Tau;
599     _Linker[_RobotDOF-1].f = f;
600     _Linker[_RobotDOF-1].M = M;
601
602     Matrix<float> R3_post;
603     Matrix<float> R3_inv;
604     Matrix<float> Rtot_inv;
605     Matrix<float> R3_inv_ant;
606     Vector<float> R_pi_post;
607     Vector<float> A;
608
609     for (int i=_RobotDOF-2; i>=0; i--){
610
611        R3_post = GetRotation(i+1);
612        printf("Hello");
613        R3_inv = GetRotation(i+1).Inverse();
614        Rtot = Rtot * R3_inv;
615        Rtot_inv = Rtot.Inverse();
616
617        R3_inv_ant = GetRotation(i).Inverse();
618
619        //Vector join Si-1 and Si
620        R_pi(0) = _Linker[i].a;
621        R_pi(1) = _Linker[i].d*sinf(_Linker[i].alpha);
622        R_pi(2) = _Linker[i].d*cosf(_Linker[i].alpha);
623        R_pi_post = R3_inv * R_pi;
624
625        //Center of gravity
626        s(0) = _Linker[i].Cent_Gravity(0)/ _Linker[i].Mass;
627        s(1) = _Linker[i].Cent_Gravity(1)/ _Linker[i].Mass;
628        s(2) = _Linker[i].Cent_Gravity(2)/ _Linker[i].Mass;
629
630
631        f_cdm = _Linker[i].acc * _Linker[i].Mass;
632        M_cdm = (_Linker[i].Inertia_tensor * _Linker[i].wd)  +
633              (_Linker[i].w.crossProduct(_Linker[i].Inertia_tensor * _Linker[i].w));
634
635        f = (R3_post * _Linker[i+1].f) + f_cdm;
636        A   =   (_Linker[i+1].M + R_pi_post.crossProduct(_Linker[i+1].f));
637        M = (R3_post * A) + ((R_pi + s).crossProduct(f_cdm) + M_cdm);
638
639        if (_Linker[i].Type == Link::REVOLUTION){
640           Tau = (R3_inv_ant.Transpose() * M) * z0;
641        }else {
642           Tau = (R3_inv_ant.Transpose() * f) * z0;
643        }
644
645        _Linker[i].Tau=Tau;
646        _Linker[i].f = f;
647        _Linker[i].M = M;
648
```

```
649     }
650
651 }
652
653 void Robot::GravityTorque(float q[]){
654     float grav[3] = {0, 9.81F, 0};
655     Vector<float> g;
656     g = grav;
657     ForwardKinematics(q);
658     Matrix<float> A0(4,4);
659     A0 = _Linker[0].A0;
660
661     Vector<float> pcg(4);
662
663     Vector<float> position[_RobotDOF];
664     position[0] = zeros(3);
665     position[1] = GetPosition(A0);
666
667     Vector<float> z0 [_RobotDOF];
668     z0[0](0) = 0;
669     z0[0](1) = 0;
670     z0[0](2) = 1;
671
672     Vector<float> m(4);
673     for (int i = 0; i <_RobotDOF; i++) {
674         if (i > 0) {
675             z0[i](0) = A0(0,2);
676             z0[i](1) = A0(1,2);
677             z0[i](2) = A0(2,2);
678
679             A0 = A0 * _Linker[i].A0;
680             if (i<_RobotDOF-1) {
681                 position[i + 1] = GetPosition(A0);
682             }
683         }
684         m(0) = _Linker[i].Cent_Gravity(0) / _Linker[i].Mass;
685         m(1) = _Linker[i].Cent_Gravity(1) / _Linker[i].Mass;
686         m(2) = _Linker[i].Cent_Gravity(2) / _Linker[i].Mass;
687         m(3) = 1;
688
689
690         pcg = A0 * m;
691         _Motor[i].pcg(0) = pcg(0);
692         _Motor[i].pcg(1) = pcg(1);
693         _Motor[i].pcg(2) = pcg(2);
694     }
695     Vector<float> M;
696     Vector<float> mg;
697     float Tau;
698     for (int i =0; i < _RobotDOF; i++) {
699         mg = g * _Linker[i].Mass;
700         M = mg.crossProduct((_Motor[i].pcg - position[i]));
701         for (int j = 0; j < _RobotDOF; j++) {
702             if(j>i) {
703                 mg = g * _Linker[j].Mass;
704                 M = M + (mg.crossProduct(_Motor[j].pcg)-position[i]);
```

```
705             }
706         }
707         Tau = M * z0[i];
708         _Linker[i].Tau = Tau;
709         M = zeros(3);
710     }
711
712 }
```

```cpp
//
// Algoritmo para la configuración de los drivers que actúan sobre los motores.
//

#include "Driver.h"

Driver::Driver(PinName adrress){
    communication = D_DAC;
    dac = new AnalogOut(adrress);
}

Driver::Driver(PinName RX,PinName TX){
    communication = D_CAN;
    can = new CAN(RX,TX);

}

Driver::Driver(PinName MOSI,PinName MISO,PinName sclk, int frecuency){
    communication = D_SPI;
    spi = new SPI(MOSI, MISO, sclk);
    spi->format(8,3);
    spi->frequency(frecuency);
}

int Driver::sendmessage(char data[8]){
   int isCorrect;
   char received[8];
   switch(communication){
     case D_CAN:
        isCorrect=can->write(CANMessage(1337, data, 8));
        break;
     case D_SPI:
        isCorrect = spi->write(data,8,received,8);
        break;
     default:
        return 0;
   }
   if(isCorrect){
      return 1;
   }else{
      return 0;
   }
}

int Driver::sendmessage(float data){
   if (communication==D_DAC){
      dac->write(data);
      return 1;
   }
   return 0;
}
Driver::~Driver(){
   delete can, dac, spi;
   }
```

```
1  //
2  // Archivo cabezera donde el usuario introduce las magnitudes físicas según el robot.
3  //
4
5  #ifndef UNTITLED_FEATURES_H
6  #define UNTITLED_FEATURES_H
7  #include "mbed.h"
8     #define DOF 2
9     int RobotDOF = 2;
10
11 PinName PinSensorA [DOF]={PA_0,PA_1};
12 PinName PinSensorB [DOF]={PA_2,PA_3};
13 PinName PinDriver [DOF]={PA_4,PA_5};
14 float Constraints[DOF*3] = {
15       // Velocity    Acceleration     Jerk
16          1,          2,          3,
17          1,          2,          3
18 };
19
20
21 const float SensorProperties[DOF*4]={
22       /*  Rank       Factor      Offset      PulsesPerRev*/
23 /*1*/     0,         0,          0,          0,
24 /*2*/     0,         0,          0,          0
25 };
26
27 float dyn_mat [DOF][10]={
28    //     Ix         Iy          Iz          Ixy       Ixz       Iyz   mx          my        mz     mass
29       {0.0056F,   0.0150F,   0.0150F,       1,        3,        2,  -0.8200F,       0,        0,        2
   },
30       {0.0028F,   0.0056F,   0.0056F,       0,        0,        0,  -0.1950F,       0,        0,        1
   }
31 };
32 float  ToolMatrix[4][4]={
33       {1,    2,     3,     10},
34       {4,    5,     6,     11}
35 };
36 float  BaseMatrix[4][4]={
37       {1,    2,     3,     10},
38       {4,    5,     6,     11}
39 };
40
41 const float DH_Table[DOF*4]={
42       /*     theta       alpha       d           a */
43       0,         1.5708F,     0,          0,
44       0,            0,        0,        0.3000F
45 };
46
47 const float MotorParam[DOF*5]={
48       /*  KA      KM      Scale       Offset      MaxTorque*/
49       0,     0,     0,          0,          0,
50       0,     0,     0,          0,          0
51 };
52
53 const float ControlGain[DOF*3]={
54       /*  KP          KD          KI*/
```

```
55  /*1*/     0,      0,      0,
56  /*2*/     0,      0,      0
57  };
58
59
60  const float LinkLenght[DOF]={2,2};
61
62
63  #endif //UNTITLED_FEATURES_H
```

```
1  //
2  // Algoritmo utilizado para la creación de matrices y sus operaciones.
3  //
4
5
6  #include "Matrix.h"
7
8  template<class T>
9  Matrix<T>::Matrix(){
10         this->rows = 3;
11         this->columns = 3;
12         n = new T *[this->rows];
13         for (int i = 0; i <this->rows ; i++){
14            n[i] = new T[this->columns];
15            for (int j = 0; j < this->columns; j++){
16               n[i][j] = 0;
17            }
18         }
19 }
20
21 template<class T>
22 Matrix<T>::Matrix(int rows,int columns){
23         objectCount++;
24         this->rows = rows;
25         this->columns = columns;
26         n = new T *[this->rows];
27         for (int i = 0; i <this->rows ; i++){
28            n[i] = new T[this->columns];
29            for (int j = 0; j < this->columns; j++){
30               n[i][j] = 0;
31            }
32         }
33 }
34
35 template<class T>
36 Matrix<T>::~Matrix() {
37    for(int i = 0; i < this->rows; ++i) {
38       delete[] n[i];
39    }
40    //Free the array of pointers
41    delete[] n;
42 }
43
44 template<class T>
45 T Matrix<T>::Determinant(){
46    T det = 0;
47    if(this->rows == this->columns) {
48       switch (this->rows) {
49          case 2: {
50             det = n[0][0] * n[1][1] - n[0][1] * n[1][0];
51             break;
52          }
53          case 3: {
54             T a = n[0][0];
55             T b = n[0][1];
56             T c = n[0][2];
```

```
57              T d = n[1][0];
58              T e = n[1][1];
59              T f = n[1][2];
60              T g = n[2][0];
61              T h = n[2][1];
62              T i = n[2][2];
63              det = (a * e * i + b * f * g + c * d * h);
64              det = det - a * f * h;
65              det = det - b * d * i;
66              det = det - c * e * g;
67              break;
68           }
69           case 4: {
70              Matrix<T> *temp[4];
71              for (int i = 0; i < 4; i++)
72                 temp[i] = new Matrix<T>(3, 3);
73              for (int k = 0; k < 4; k++) {
74                 for (int i = 1; i < 4; i++) {
75                    int j1 = 0;
76                    for (int j = 0; j < 4; j++) {
77                       if (k == j)
78                          continue;
79                       temp[k]->n[i - 1][j1++] = this->n[i][j];
80                    }
81                 }
82              }
83              det = this->n[0][0] * temp[0]->Determinant() - this->n[0][1] * temp[1]->
    Determinant() +
84                  this->n[0][2] * temp[2]->Determinant()
85                  - this->n[0][3] * temp[3]->Determinant();
86              break;
87           }
88           case 5: {
89              Matrix<T> *temp[5];
90              for (int i = 0; i < 5; i++)
91                 temp[i] = new Matrix<T>(4, 4);
92              for (int k = 0; k < 5; k++) {
93                 for (int i = 1; i < 5; i++) {
94                    int j1 = 0;
95                    for (int j = 0; j < 5; j++) {
96                       if (k == j)
97                          continue;
98                       temp[k]->n[i - 1][j1++] = this->n[i][j];
99                    }
100                }
101             }
102             det = this->n[0][0] * temp[0]->Determinant() - this->n[0][1] * temp[1]->
    Determinant()
103                 + this->n[0][2] * temp[2]->Determinant() - this->n[0][3] * temp[3]->
    Determinant()
104                 + this->n[0][4] * temp[4]->Determinant();
105             break;
106          }
107          case 6:
108          case 7:
109          case 8:
```

```
110           case 9:
111           case 10:
112           case 11:
113           case 12:
114           default: {
115              Matrix **temp = new Matrix *[rows];
116              for (int i = 0; i < rows; i++)
117                 temp[i] = new Matrix(rows - 1, rows - 1);
118              for (int k = 0; k < rows; k++) {
119                 for (int i = 1; i < rows; i++) {
120                    int j1 = 0;
121                    for (int j = 0; j < rows; j++) {
122                       if (k == j)
123                          continue;
124                       temp[k]->n[i - 1][j1++] = this->n[i][j];
125                    }
126                 }
127              }
128              for (int k = 0; k < rows; k++) {
129                 if ((k % 2) == 0) {
130                    det = det + (this->n[0][k] * temp[k]->Determinant());
131                 } else {
132                    det = det - (this->n[0][k] * temp[k]->Determinant());
133                 }
134              }
135              for (int i = 0; i < rows; i++)
136                 delete temp[i];
137              delete[] temp;
138             break;
139           }
140        }
141     }
142     return det;
143  }
144
145  template<class T>
146  Matrix<T> Matrix<T>::Cofactor(){
147     Matrix<T> cofactor(this->rows, this->columns);
148
149     if (this->rows != this->columns)
150        return cofactor;
151     if (this->rows < 2)
152        return cofactor;
153     else if (this->rows == 2){
154        for (int i = 0; i < this->rows; i++) {
155           for (int j = 0; j < this->columns; j++){
156              cofactor(i, j) =this->n[((this->rows-1)-i)][((this->columns-1)-j)];
157              if((i+j)%2 != 0) {
158                 cofactor(i, j) = -1*cofactor(i, j);
159              }
160           }
161        }
162        return cofactor;
163     }
164     else if (this->rows >= 3){
165        int DIM = this->rows;
```

```
166         Matrix<T> ***temp = new Matrix<T>**[DIM];
167         for (int i = 0; i < DIM; i++)
168            temp[i] = new Matrix<T>*[DIM];
169         for (int i = 0; i < DIM; i++)
170            for (int j = 0; j < DIM; j++)
171               temp[i][j] = new Matrix<T>(DIM - 1, DIM - 1);
172         for (int k1 = 0; k1 < DIM; k1++){
173            for (int k2 = 0; k2 < DIM; k2++){
174               int i1 = 0;
175               for (int i = 0; i < DIM; i++){
176                  int j1 = 0;
177                  for (int j = 0; j < DIM; j++){
178                     if (k1 == i || k2 == j)
179                        continue;
180                     temp[k1][k2]->n[i1][j1++]
181                           = this->n[i][j];
182                  }
183                  if (k1 != i)
184                     i1++;
185               }
186            }
187         }
188         bool flagPositive = true;
189         for (int k1 = 0; k1 < DIM; k1++){
190            flagPositive = ((k1 % 2) == 0);
191            for (int k2 = 0; k2 < DIM; k2++){
192               if (flagPositive == true){
193                  cofactor.n[k1][k2]
194                        = temp[k1][k2]->Determinant();
195                  flagPositive = false;
196               }
197               else{
198                  cofactor.n[k1][k2]
199                        = -temp[k1][k2]->Determinant();
200                  flagPositive = true;
201               }
202            }
203         }
204         for (int i = 0; i < DIM; i++)
205            for (int j = 0; j < DIM; j++)
206               delete temp[i][j];
207         for (int i = 0; i < DIM; i++)
208            delete[] temp[i];
209         delete[] temp;
210      }
211      return cofactor;
212   }
213   template<class T>
214   Matrix<T> Matrix<T>::Inverse(){
215      Matrix<T> cofactor(this->rows, this->columns);
216      Matrix<T> inv(this->rows, this->columns);
217      if (this->rows != this->columns)
218         return inv;
219
220      T det = this->Determinant();
221      cofactor = this->Cofactor();
```

```
222     if(det != 0) {
223         for (int i = 0; i < this->rows; i++) {
224             for (int j = 0; j < this->columns; j++) {
225                 inv.n[j][i] = cofactor.n[i][j] / det;
226             }
227         }
228     }
229     return inv;
230 }
231
232 template<class T>
233 Matrix<T>  Matrix<T>::PseudoInverse(){
234     Matrix<T> trans(this->columns, this->rows);
235     Matrix<T> A(this->rows, this->columns);
236     A = *this;
237     trans = Transpose();
238     Matrix<T> result = Matrix<T>(trans.rows, trans.columns);
239     if((trans*A).Determinant()!=0){
240         result = ((trans * A).Inverse())*trans;
241     }else if((A*trans).Determinant()!=0){
242         result = trans *((A*trans).Inverse());
243     }else{
244         printf("Error Matrix PseudoInverse\n");
245     }
246     return result;
247
248 }
249
250 template<class T>
251 Matrix<T> Matrix<T>::Transpose(){
252     Matrix<T> trans(this->columns, this->rows);
253         for (int i = 0; i < this->columns; i++) {
254             for (int j = 0; j < this->rows; j++) {
255                 trans.n[i][j] = this->n[j][i];
256             }
257         }
258
259     return trans;
260 }
261 template<class T>
262 void Matrix<T>::PrintResult(){
263     for (int x = 0; x<this->rows; x++) {
264         for (int y = 0; y < this->columns; y++) {
265                 printf(" %f ", n[x][y]);
266         }
267         printf("\n");
268     }
269     printf("\n");
270 }
271 template<class T>
272 void  Matrix<T>::Reconfigurate(int rows,int columns){
273     Matrix<T>::~Matrix();
274     this->rows = rows;
275     this->columns = columns;
276     n = new T *[this->rows];
277     for (int i = 0; i <this->rows ; i++){
```

```
278         n[i] = new T[this->columns];
279         for (int j = 0; j < this->columns; j++){
280             n[i][j] = 0;
281         }
282     }
283 }
284 template<class T>
285 Matrix<T>&  Matrix<T>::operator=(const Matrix &other ){
286         if(this->columns<=other.columns && this->rows<=other.rows){
287             for(int i=0;i<this->rows;i++){
288                 for(int j=0;j<this->columns;j++){
289                     this->n[i][j]=other.n[i][j];
290                     }
291                 }
292         }else{
293             printf("ERROR: Matrix A = Matrix B. B has more dimensions than A \n");
294         }
295          return (*this);
296 }
297
298 template<class T>
299 Matrix<T>& Matrix<T>::operator=(const T param[][2]){
300     for(int i=0;i<this->rows;i++){
301         for(int j=0;j<this->columns;j++){
302             this->n[i][j]=param[i][j];
303         }
304     }
305     return (*this);
306 }
307
308 template<class T>
309 Matrix<T>& Matrix<T>::operator=(const T param[][3]){
310     for(int i=0;i<this->rows;i++){
311         for(int j=0;j<this->columns;j++){
312             this->n[i][j]=param[i][j];
313         }
314     }
315     return (*this);
316 }
317
318 template<class T>
319 Matrix<T>& Matrix<T>::operator=(const T param[][4]){
320     for(int i=0;i<this->rows;i++){
321         for(int j=0;j<this->columns;j++){
322             this->n[i][j]=param[i][j];
323         }
324     }
325     return (*this);
326 }
327
328
329 template<class T>
330 Matrix<T>  Matrix<T>::operator+(const Matrix &other){
331     Matrix<T> result = Matrix<T>(rows, other.columns);
332     if (this->rows == other.columns && this->columns == other.rows) {
333         for (int i = 0; i < this->rows; i++) {
```

```
334             for (int j = 0; j < this->columns; j++) {
335                 result.n[i][j] = other.n[i][j] + this->n[i][j];
336             }
337         }
338     }
339         return result;
340 }
341
342 template<class T>
343 Matrix<T>  Matrix<T>::operator*(const Matrix& other) {
344         Matrix<T> result = Matrix<T>(this->rows, other.columns);
345         if (this->columns == other.rows) {
346             for (int i = 0; i < this->rows; i++) {
347                 for (int j = 0; j < this->columns; j++) {
348                     for (int k = 0; k < this->columns; k++) {
349                         result.n[i][j] += this->n[i][k]*other.n[k][j];
350                     }
351                 }
352             }
353         }
354         return result;
355 }
356 template<class T>
357 Vector<T> Matrix<T>::operator*(Vector<T>& other){
358     Vector<T> result = Vector<T>(other.getDimension());
359     for (int x=0;x<other.getDimension();x++){
360         result(x)= (this->n[x][0]*other(0)+this->n[x][1]*other(1)+this->n[x][2]*other(2));
361         if( (result (x)>-0.01)&&(result (x))<0.01){
362             result (x) = 0;
363         }
364     }
365     return result;
366
367 }
368 template<class T>
369 T& Matrix<T>::operator()(int row, int col) {
370         return n[row][col];
371 }
372
373 template<class T>
374 int Matrix<T>::objectCount = 0;
375
376 template class Matrix<float>;
377 template class Matrix<int>;
378 template class Vector<float>;
379 template class Vector<int>;
```

```cpp
//
// Archivo encargado de obtener información de los distintos sensores ubicados en cada articulación.
//

#include "Sensor.h"
using namespace std;

    Sensor::Sensor(PinName channelA,PinName channelB,float _rank, float _sFactor,float _offset, int _pulsesPerRev):Encoder(channelA,channelB, NC, _pulsesPerRev){
        rank = _rank;
        scale_factor = _sFactor;
        offset = _offset;
    }

    float Sensor::readSensor(){
        return (Encoder.getRevolutions());
    }
    float Sensor::computevalue(int value){
        return (value*scale_factor-offset);
    }

    float Sensor::estimatevelocity (float prevValue,float sampling){
        return ((Encoder.getRevolutions()-prevValue)/sampling);
    }

```

```
1  //
2  // Algoritmo utilizado para la creación de vectores y sus operaciones.
3  //
4
5  #include "Ro/Vector.h"
6  template<class T>
7  Vector<T>::Vector() {
8     objectCount++;
9     this->dimension = 3;
10    n = new T [this->dimension];
11    for (int i = 0; i <this->dimension ; i++){
12       n[i] = 0;
13    }
14 }
15
16 template<class T>
17 Vector<T>::Vector(int dimension) {
18    objectCount++;
19    this->dimension = dimension;
20    n = new T [this->dimension];
21    for (int i = 0; i <this->dimension ; i++){
22       n[i] = 0;
23    }
24 }
25
26 template<class T>
27 Vector<T>::~Vector(){
28    delete [] n;
29 }
30
31 template<class T>
32 int Vector<T>::getDimension(){
33    return dimension;
34 }
35
36 template<class T>
37 void Vector<T>::PrintResult(){
38    for (int x = 0; x<this->dimension; x++) {
39       printf(" %f ", n[x]);
40    }
41    printf("\n");
42 }
43
44
45 template<class T>
46 Vector<T> Vector<T>::crossProduct(Vector<T> const& VectorB){
47    Vector<T> result(3);
48    if(this->dimension == 3) {
49       result(0) = (n[1] * VectorB.n[2] - n[2] * VectorB.n[1]);
50       result(1) = -(n[0] * VectorB.n[2] - n[2] * VectorB.n[0]);
51       result(2) = (n[0] * VectorB.n[1] - n[1] * VectorB.n[0]);
52       for (int x = 0; x < 3; x++) {
53          if (( result(x) > -0.000001) && (result(x)) < 0.000001) {
54             result(x) = 0;
55          }
56       }
```

```
57      }
58      return result;
59  }
60
61
62
63
64  template<class T>
65  Vector<T>&  Vector<T>::operator=(const Vector &other ){
66      if(this->dimension==other.dimension){
67          for(int i=0;i<this->dimension;i++){
68              this->n[i] = other.n[i];
69          }
70      }else{
71          printf("ERROR: Vector A = Vector B. B has more dimensions than A");
72      }
73      return (*this);
74  }
75
76
77  template<class T>
78  Vector<T>& Vector<T>::operator=(T param[]){
79      for(int i=0;i<this->dimension;i++){
80          this->n[i]=param[i];
81      }
82      return (*this);
83
84  }
85  template<class T>
86  Vector<T>  Vector<T>::operator+(Vector<T> other){
87      Vector<T> result(this->dimension);
88      if(this->dimension == other.dimension) {
89          for (int i = 0; i < this->dimension; i++) {
90              result.n[i] = other.n[i] + this->n[i];
91          }
92      }else{
93          printf("ERROR: Vector A = Vector B. B has more dimensions than A");
94      }
95      return result;
96  }
97
98  template<class T>
99  Vector<T>  Vector<T>::operator-(Vector<T>& other){
100     Vector<T> result(this->dimension);
101     if(this->dimension== other.dimension) {
102         for (int i = 0; i < this->dimension; i++) {
103             result.n[i] = other.n[i] - this->n[i];
104         }
105     }else{
106         printf("ERROR: Vector A = Vector B. B has more dimensions than A");
107     }
108     return result;
109 }
110
111 template<class T>
112 T Vector<T>::operator*(Vector<T>& other) {
```

```
113     T ans=0;
114     if(this->dimension == other.dimension) {
115         for (int i = 0; i < this->dimension; i++) {
116             ans = ans + (other.n[i] * this->n[i]);
117         }
118     }else{
119         printf("ERROR: Vector A = Vector B. B has more dimensions than A");
120     }
121     return ans;
122 }
123
124 template<class T>
125 Vector<T>  Vector<T>::operator*(T factor){
126     Vector<T> result(this->dimension);
127     for (int i = 0; i < this->dimension; i++) {
128         result.n[i] = factor * this->n[i];
129     }
130     return result;
131 }
132
133 template<class T>
134 T& Vector<T>::operator()(int dim){
135     return n[dim];
136 }
137
138 template<class T>
139 int Vector<T>::objectCount = 0;
140
141
142 Vector<float> zeros(int num){
143     Vector<float> result(num);
144     return result;
145 }
146 //template<class T>
147 Vector<float> crossProduct(Vector<float> VectorA,Vector<float> VectorB){
148     Vector<float> result(3);
149
150     //if(VectorA->dimension == 3) {
151     result(0) = (VectorA(1) * VectorB(2) - VectorA(2) * VectorB(1));
152     result(1) = -(VectorA(0) * VectorB(2) - VectorA(2) * VectorB(0));
153     result(2) = (VectorA(0) * VectorB(1) - VectorA(1) * VectorB(0));
154     for (int x = 0; x < 3; x++) {
155         if (( result(x) > -0.000001) && (result(x)) < 0.000001) {
156             result(x) = 0;
157         }
158     }
159     return result;
160 }
161
162 template class Vector<float>;
163 template class Vector<int>;
```

```
1  //
2  // Algoritmo para un control de un brazo robot.
3  //
4  #include "Control.h"
5
6  Control::Control(float _KP, float _KV, float _KI,float _KA,float _KM ,float
   _SamplingTime, float _Scale, float _Offset,float _MaxTorque){
7     KP = _KP;
8     KV = _KV;
9     KI = _KI;
10    KA = _KA;
11    KM = _KM;
12    Scale = _Scale;
13    Offset = _Offset;
14    MaxTorque = _MaxTorque;
15    SamplingTime = _SamplingTime;
16    PrevErr = 0;
17 }
18 float Control::GetKP(){
19    return KP;
20 }
21 float Control::GetKV(){
22    return KV;
23 }
24 float Control::GetKI(){
25    return KI;
26 }
27 float Control::SaturationControl(float value){
28    float u = (value*KA*KM)/MaxTorque;
29    if (u>1){
30       u=1;
31    }
32    if (u<0){
33       u=0;
34    }
35    u = u* Scale + Offset;
36    return u;
37 }
38 float Control::ComputeP(float qref, float qi){
39    float u=(((qref - qi) * KP)*KA*KM)/MaxTorque;
40    if (u>1){
41       u=1;
42    }
43    if (u<0){
44       u=0;
45    }
46    u = u* Scale + Offset;
47    return u;
48 }
49 float Control::ComputePD_FirstVersion(float qref, float qi, float qdotref, float qdoti){
50    float u=(((qref - qi) * KP + (qdotref - qdoti) * KV)*KA*KM)/MaxTorque;
51    if (u>1){
52       u=1;
53    }
54    if (u<0){
55       u=0;
```

```
56      }
57      u = u* Scale + Offset;
58      return u;
59  }
60  float Control::ComputePD_SecondVersion(float qref, float qi, float qdoti){
61      float u=(((qref - qi) * KP  - qdoti * KV)*KA*KM)/MaxTorque;
62      if (u>1){
63          u=1;
64      }
65      if (u<0){
66          u=0;
67      }
68      u = u* Scale + Offset;
69      return u;
70  }
71  float Control::ComputePID(float qref, float qi, float qdotref, float qdoti){
72
73      float IntegralErr = PrevErr + (qref - qi);
74      PrevErr = IntegralErr;
75
76      float u=(((qref-qi) * KP + (qdotref - qdoti) * KV + IntegralErr * KI)*KA*KM)/
    MaxTorque;
77
78      if (u>1){
79          u=1;
80      }
81      if (u<0){
82          u=0;
83      }
84      u = u* Scale + Offset;
85      return u;
86  }
87  float Control::ComputePDG(float qref, float qi, float qdotref, float qdoti, float Tau){
88      float u = (((qref-qi) * KP + (qdotref - qdoti) * KV)*KA*KM + Tau)/MaxTorque;
89      if (u>1){
90          u=1;
91      }
92      if (u<0){
93          u=0;
94      }
95      u = u* Scale + Offset;
96      return u;
97  }
```

```
1  //
2  // Archivo cabezera que contiene la información de una trayectoria.
3  //
4
5  #ifndef UNTITLED_TRAJECTORY_H
6  #define UNTITLED_TRAJECTORY_H
7
8  #include "Vector.h"
9  #include "Matrix.h"
10
11 class Trajectory {
12    private:
13       Vector<float> pos;
14       Vector<float> v;
15       Vector<float> a;
16       int Contsample;
17       int TimSampling;
18    public:
19       Trajectory();
20       Trajectory(int dof, int sampling);
21       void Insert(float p, float v, float a);
22       float GetXref(int i);
23       float GetVref(int i);
24       float GetAref(int i);
25 };
26
27
28 #endif //UNTITLED_TRAJECTORY_H
```

```
1  #ifndef UNTITLED_END_EFFECTOR_H
2  #define UNTITLED_END_EFFECTOR_H
3
4  //
5  // Algoritmo donde se detallan las especificaciones del End Effector.
6  //
7
8  class End_Effector {
9  public:
10     //Position
11     float x;
12     float y;
13     float z;
14
15     //Orientation
16     float roll;
17     float pitch;
18     float yaw;
19
20     End_Effector();
21 };
22
23
24 #endif //UNTITLED_END_EFFECTOR_H
25
```

```
1  //
2  // Archivo donde se almacena la información respecto a las trayectorias generadas.
3  //
4
5  #include "Trajectory.h"
6  Trajectory::Trajectory(){
7  }
8  Trajectory::Trajectory(int dof, int sampling):pos(sampling),v(sampling),a(sampling){
9        TimSampling = sampling;
10       Contsample=0;
11 }
12 void Trajectory::Insert(float p, float v, float a){
13    if(Contsample<TimSampling){
14       this->pos(Contsample) = p;
15       this->v(Contsample) = v;
16       this->a(Contsample) = a;
17       Contsample++;
18    }
19 }
20 float Trajectory::GetXref(int i){
21    return pos(i);
22 }
23 float Trajectory::GetVref(int i){
24    return pos(i);
25 }
26 float Trajectory::GetAref(int i){
27    return pos(i);
28 }
```

```
1  //
2  // Archivo cabezera para el comando de movimiento.
3  //
4
5  #ifndef MOTIONCOMMAND_H
6  #define MOTIONCOMMAND_H
7     #include "mbed.h"
8     #include "Ro/Robot.h"
9     #include "Ro/Trajectory.h"
10
11 class MotionCommand{
12    private:
13    Robot* _Robot;
14    float* Initial_point;
15    float* Final_point;
16    float SamplingTime;
17    Vector<float> PrevPosition;
18    Trajectory* _Trajectory;
19    int CurrentCycle;
20    int TotalCycle;
21    bool IsFinalized;
22
23    int Zone;
24    int index;
25    Ticker TimInterrupt;
26
27    void Trapezoid(float q0,float q1, float qd, float qdd,float stime,float Constraints_v,float
   Contrains_a);
28    int SCurve(float q0,float q1, float qd0,float qdf, float qdd,float stime,float Constraints_v,
   float Constraints_a, float Constraints_j);
29    void Spline_5th (float q0,float q1, float qd0, float qd1, float qdd,float stime,float tfin,float
    Constraints_v ,float Constraints_a);
30    void Spline_7th(float q0,float q1, float qd0, float qd1, float qdd,float stime,float tfin,float
   Constraints_v ,float Constraints_a);
31    void TimFunctionMoveL1();
32    void StartMoveL1();
33    void StopMoveL1();
34
35    void TimFunctionMoveL2();
36    void StartMoveL2();
37    void StopMoveL2();
38
39    public:
40
41    enum Trajectory_Types {TRAPEZOIDAL, SCURVE, SPLINE5, SPLINE7};
42
43    MotionCommand(Robot* Robot1);
44    ~MotionCommand();
45
46    void MoveJ(End_Effector qf,float v0,float v1,float a,float samplingtime,float time,int
   zone,Trajectory_Types _trajectory);
47    void MoveL1(End_Effector qf, float v, float a,float samplingtime);
48    void MoveL2(End_Effector qf, float v, float a,float samplingtime);
49
50
51 };
```

```
52
53  template <typename T>
54  inline T  sign (T  a);
55
56
57  #endif //MOTIONCOMMAND_H
```

```
1  //
2  // Archivo cabezera para la la determinación del End Effector.
3  //
4
5  #include "End_Effector.h"
6  End_Effector::End_Effector(){
7
8  }
```

```
1  //
2  // Algoritmo donde se detallan los distintos interpoladores para alcancar el punto deseado,
   trayectorias (MoveL y MoveJ).
3  //
4
5  #include "MotionCommand.h"
6  template <typename T>
7  inline T  sign (T  a) {
8    return (a > 0 ? 1:-1);
9  }
10
11 MotionCommand::MotionCommand(Robot* Robot1):PrevPosition(Robot1->GetDOF()){
12     _Robot = Robot1;
13     IsFinalized = false;
14     index = 0;
15 }
16 MotionCommand::~MotionCommand(){
17     delete[] _Trajectory;
18 }
19 void MotionCommand::MoveJ(End_Effector qf,float v0,float v1,float a,float samplingtime,
   float time,int zone,Trajectory_Types _trajectory){
20     SamplingTime = samplingtime;
21     IsFinalized = false;
22     PrevPosition = 0;
23     float q0[3];
24     Vector<float> PositionF(3),PositionInit(3);
25     PositionF(0) = qf.x;
26     PositionF(1) = qf.y;
27     PositionF(2) = qf.z;
28
29     _Trajectory = new Trajectory[3];
30
31
32     for (int i=0; i<_Robot->GetDOF();i++){
33        if(i<=3){
34           _Trajectory[i] = Trajectory(_Robot->GetDOF(),samplingtime);
35        }
36        q0[i] = _Robot->ReadSensor(i);
37     }
38     _Robot->ForwardKinematics(q0);
39     PositionInit = _Robot->GetEndPosition();
40
41     for (int i=0; i<3;i++){
42        switch(_trajectory){
43        case TRAPEZOIDAL:
44           Trapezoid(PositionInit(i),PositionF(i), v0, a,samplingtime,_Robot->
   ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i));
45           break;
46        case SCURVE:
47           SCurve(PositionInit(i),PositionF(i), v0, v1, a,samplingtime,_Robot->
   ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i), _Robot->
   ConstraintsJerkMotor(i));
48           break;
49        case SPLINE5:
50           Spline_5th(PositionInit(i),PositionF(i), v0, v1, a,samplingtime,time,_Robot->
   ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i));
```

```
51            break;
52          case SPLINE7:
53            Spline_7th(PositionInit(i),PositionF(i), v0, v1, a,samplingtime,time,_Robot->
    ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i));
54            break;
55          }
56      }
57      this->StartMoveL1();
58      while(!IsFinalized) {
59        asm("nop");
60      }
61      this->StopMoveL1();
62
63
64  }
65  void MotionCommand::MoveL1(End_Effector qf, float v, float a,float samplingtime){
66      SamplingTime = samplingtime;
67      IsFinalized = false;
68      PrevPosition = 0;
69      float q0[_Robot->GetDOF()];
70      Vector<float> PositionF(3),PositionInit(3);
71      PositionF(0) = qf.x;
72      PositionF(1) = qf.y;
73      PositionF(2) = qf.z;
74      _Trajectory = new Trajectory[3];
75
76
77      for (int i=0; i<_Robot->GetDOF();i++){
78        if(i<=3){
79          _Trajectory[i] = Trajectory(_Robot->GetDOF(),samplingtime);
80        }
81        q0[i] = _Robot->ReadSensor(i);
82      }
83      _Robot->ForwardKinematics(q0);
84      PositionInit = _Robot->GetEndPosition();
85
86      for (int i=0; i<3;i++){
87        Trapezoid(PositionInit(i),PositionF(i), v, a,samplingtime,_Robot->
    ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i));
88      }
89      this->StartMoveL1();
90      while(!IsFinalized) {
91        asm("nop");
92      }
93      this->StopMoveL1();
94  }
95
96  void MotionCommand::MoveL2(End_Effector qf, float v, float a,float samplingtime){
97      SamplingTime = samplingtime;
98      IsFinalized = false;
99      PrevPosition = 0;
100     float q0[3];
101     Vector<float> PositionF(3),PositionInit(3);
102     PositionF(0) = qf.x;
103     PositionF(1) = qf.y;
104     PositionF(2) = qf.z;
```

```
105     _Trajectory = new Trajectory[3];
106
107
108     for (int i=0; i<_Robot->GetDOF();i++){
109         if(i<=3){
110             _Trajectory[i] = Trajectory(_Robot->GetDOF(),samplingtime);
111         }
112         q0[i] = _Robot->ReadSensor(i);
113     }
114     _Robot->ForwardKinematics(q0);
115     PositionInit = _Robot->GetEndPosition();
116
117     for (int i=0; i<3;i++){
118         Trapezoid(PositionInit(i),PositionF(i), v, a,samplingtime,_Robot->
    ConstraintsVelocityMotor(i),_Robot->ConstraintsAccelerationMotor(i));
119     }
120     this->StartMoveL2();
121     while(!IsFinalized) {
122         asm("nop");
123     }
124     this->StopMoveL2();
125 }
126
127
128 void MotionCommand::Trapezoid(float _q0,float q1, float qd, float qdd,float stime,float
    Constraints_v,float Constraints_a){
129     float s, signo, t1, t2, tfin, q,qprim,acel;
130     s = q1 - _q0;
131     signo = sign(s);
132     CurrentCycle = 0;
133
134         if (fabsf(s) > (powf(Constraints_v,2.0F) / Constraints_a)){
135             t1 = Constraints_v / Constraints_a;
136             t2 = fabsf(s) / Constraints_v;
137             tfin = fabsf(s) / Constraints_v + Constraints_v / Constraints_a;
138         }
139         else{
140             t1 = sqrtf(fabsf(s) / Constraints_a);
141             tfin = 2.0F*t1;
142             t2 = 0.0F;
143         }
144         q = _q0;
145         qprim = 0;
146         float t=0;
147         TotalCycle = (int)tfin/stime+1;
148         for(int tim=0; tim<((int)(tfin/stime)+1);tim++){
149                 t = tim*stime;
150                 //Calculation of the velocity and position for phase 1 (acceleration)
151                 if (t < t1){
152                     qdd = signo*Constraints_a;
153                     qprim=qprim+acel*stime;
154                     q=q+qprim*stime+0.5F*acel*stime*stime;
155                 }
156                 //Calculation of the velocity and position for phase 2 (constant velocity)
157                 else if ((t >= t1) && (t < t2)){
158                     qdd = 0.0F;
```

```
159                 qprim=signo*qd;
160                 q=q+qprim*stime;
161             }
162             //Calculation of the velocity and position for phase 3 (deceleration)
163             else if ((t >= t2) && (t < tfin)){
164                 acel = -signo*qdd;
165                 qprim=qprim + acel*stime;
166                 q=q + qprim*stime+0.5F*acel*powf(stime,2.0F);
167             }else{
168                 qdd = 0.0F;
169                 qd = 0.0F;
170                 q = q1;
171             }
172             _Trajectory[index].Insert(q,qd,qdd);
173         }
174         index++;
175 }
176
177
178 void MotionCommand::StartMoveL1(){
179     TimInterrupt.attach(this,&MotionCommand::TimFunctionMoveL1,SamplingTime);
180 }
181  void MotionCommand::StopMoveL1(){
182     TimInterrupt.detach();
183 }
184 void MotionCommand::TimFunctionMoveL1(){
185     if(CurrentCycle <=  TotalCycle ){
186         Vector<float> x(_Robot->GetDOF()),v(_Robot->GetDOF());
187         Vector<float> Kp(_Robot->GetDOF()),Kv(_Robot->GetDOF());
188         int isCorrect;
189         float q[_Robot->GetDOF()],xref[_Robot->GetDOF()], vref[_Robot->GetDOF()];
190
191
192         for (int i=0; i<_Robot->GetDOF();i++){
193             q[i] = _Robot->ReadSensor(i);
194             xref[i] =  _Trajectory[i].GetXref(CurrentCycle);
195             vref[i] =  _Trajectory[i].GetVref(CurrentCycle);
196         }
197
198         _Robot->ForwardKinematics(q);
199         x = _Robot->GetEndPosition();
200
201         if(CurrentCycle == 0){
202             v(0) = 0;
203             v(1) = 0;
204             v(2) = 0;
205         }else{
206             v(0) = (x(0) - PrevPosition(0)) / SamplingTime;
207             v(1) = (x(1) - PrevPosition(1)) / SamplingTime;
208             v(2) = (x(2) - PrevPosition(2)) / SamplingTime;
209         }
210
211         Kp = _Robot->GetKP();
212         Kv = _Robot->GetKV();
213
214         Vector<float> CartesianForce (6);
```

```
215         Vector<float> G (_Robot->GetDOF());
216         Matrix<float> Jacobian(6,_Robot->GetDOF());
217         Matrix<float> JacobianTranspose(_Robot->GetDOF(),6);
218 
219         Vector<float> Control(_Robot->GetDOF());
220 
221         for (int i=0; i<6;i++){
222           if(i>=3){
223             CartesianForce(i) = 0;
224             continue;
225           }
226           CartesianForce(i) = Kp(i) * (xref[i] -x(i)) + Kv(i) * (vref[i] -v(i));
227         }
228 
229         _Robot->GravityTorque(q);
230         G = _Robot->GetTorque();
231 
232         _Robot->Jacobian(q);
233         Jacobian = _Robot->GetJacobian();
234         JacobianTranspose = Jacobian.Transpose();
235 
236         Control = (JacobianTranspose * CartesianForce) + G;
237 
238         Vector<float> SaturationControl(_Robot->GetDOF());
239 
240         SaturationControl = _Robot->ReturnSaturationControl(Control);
241 
242         isCorrect = _Robot->SendData(SaturationControl);
243 
244         PrevPosition = x;
245         CurrentCycle++;
246       }else{
247         IsFinalized = true;
248       }
249 
250 }
251 
252 
253 
254 void MotionCommand::StartMoveL2(){
255   TimInterrupt.attach(this,&MotionCommand::TimFunctionMoveL2,SamplingTime);
256 }
257  void MotionCommand::StopMoveL2(){
258   TimInterrupt.detach();
259 }
260 void MotionCommand::TimFunctionMoveL2(){
261   if(CurrentCycle <=  TotalCycle ){
262         float _q[_Robot->GetDOF()],qsec[_Robot->GetDOF()], q_prim[_Robot->GetDOF()];
263         Vector<float> Kp(_Robot->GetDOF()),Kv(_Robot->GetDOF()),q(_Robot->GetDOF
    ()),qref(_Robot->GetDOF()), q_prim_ref(_Robot->GetDOF()), q_sec_ref(_Robot->GetDOF
    ()),
264         xref(6), x_prim_ref(6), x_sec_ref(6),Tau(_Robot->GetDOF());
265         int isCorrect;
266         End_Effector Eref;
267 
268         for(int i=0; i<6;i++){
```

```
269             xref(i) =  _Trajectory[i].GetXref(CurrentCycle);
270             x_prim_ref(i) =  _Trajectory[i].GetVref(CurrentCycle);
271             x_sec_ref(i) =  _Trajectory[i].GetAref(CurrentCycle);
272         }
273         for (int i=0; i<_Robot->GetDOF();i++){
274             q(i) = _q[i] = _Robot->ReadSensor(i);
275 
276             if(CurrentCycle == 0){
277                 q_prim[i] = 0;
278             }else{
279                 q_prim[i] = (q(i) - PrevPosition(0)) / SamplingTime;
280             }
281 
282         }
283 
284         Kp = _Robot->GetKP();
285         Kv = _Robot->GetKV();
286 
287         Eref.x = xref(0);
288         Eref.y = xref(1);
289         Eref.z = xref(2);
290         Eref.roll = xref(3);
291         Eref.pitch = xref(4);
292         Eref.yaw = xref(5);
293 
294         qref = _Robot->InverseKinematics(&Eref,0);
295         Matrix<float> PseudoInverse(_Robot->GetDOF(),6);
296         Matrix<float> Jacobian(6,_Robot->GetDOF());
297         _Robot->Jacobian(_q);
298         Jacobian = _Robot->GetJacobian();
299 
300         if(_Robot->GetDOF() == 6){
301             q_prim_ref = Jacobian.Inverse() * x_prim_ref;
302         }else{
303             q_prim_ref = Jacobian.PseudoInverse() * x_prim_ref;
304         }
305 
306 
307         for (int i=0; i<6;i++){
308             qsec[i] = Kp(i) * (qref(i) -q(i)) + Kv(i) * (q_prim_ref(i) -q_prim[i]);
309         }
310 
311 
312         _Robot->EulerNewton(_q,q_prim,qsec);
313         Tau = _Robot->GetTorque();
314         Vector<float> SaturationControl(_Robot->GetDOF());
315 
316         SaturationControl = _Robot->ReturnSaturationControl(Tau);
317 
318         isCorrect = _Robot->SendData(SaturationControl);
319 
320         PrevPosition = _q;
321         CurrentCycle++;
322     }else{
323         IsFinalized = true;
324     }
```

```
325
326 }
327
328 int MotionCommand::SCurve(float q0,float q1, float qd0,float qdf, float qdd,float stime,
    float Constraints_v,float Constraints_a, float Constraints_j){
329         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,
330             sqrt_delta,alima,alimd,vlim,q,qprim,acel;
331         CurrentCycle = 0;
332         bool isFeasable;
333          if (Constraints_v<abs(qd0)){
334             v0 = sign(qd0)*Constraints_v;
335          }
336            if (Constraints_v<abs(qdf)){
337             v1 = sign(qdf)*Constraints_v;
338          }
339
340          //Check if its feasable
341          dq=abs(q1 - q0);
342          dv=abs(v1-v0);
343
344          time_set_velocity=sqrtf((dv)/Constraints_j);
345          time_reach_acc=Constraints_a/Constraints_j;
346            Tj = min(time_set_velocity,time_reach_acc);
347            if (Tj < time_reach_acc){
348               if (dq<Tj*(v0+v1)){
349                  isFeasable = false;
350               }
351               else{
352                   isFeasable = true;
353               }
354            }
355            else if (Tj == time_reach_acc){
356                if (dq < (0.5F *(v0+v1)*(Tj+dv/Constraints_a))){
357                  isFeasable = false;
358                }
359                else{
360                   isFeasable=true;
361                }
362            }
363            else{
364             isFeasable=false;
365            }
366
367     if (isFeasable == false){
368        printf("Is not feasable this configuration \n");
369        return 0;
370     }
371     //Sign change
372        sigma = sign(q1-q0);
373        _q0 = sigma*q0;
374        _q1 = sigma*q1;
375        v0 = sigma*v0;
376        v1 = sigma*v1;
377        sigm1 = (sigma+1.0F)/2.0F;
378        sigm2 = (sigma-1.0F)/2.0F;
```

```
379         vmax = Constraints_v*sigm1 - Constraints_v*sigm2;
380         amax = Constraints_a*sigm1 - Constraints_a*sigm2;
381         jmax = Constraints_j*sigm1 - Constraints_j*sigm2;
382
383         vmin = -vmax;
384         amin = -amax;
385         jmin = -jmax;
386
387       if ((vmax- v0)* jmax < amax* amax){
388             Tj1 = sqrtf((vmax - v0)/ jmax);
389             Ta = 2.0F * Tj1;
390       }
391       else{
392             Tj1 = amax/jmax;
393             Ta =  Tj1 + (vmax - v0)/amax;
394       }
395       if ((vmax - v1)* jmax < amax * amax){
396            Tj2 = sqrtf((vmax - v1)/ jmax);
397            Td = 2.0F* Tj2;
398       }
399       else{
400             Tj2 =  amax / jmax;
401             Td = Tj2 + (vmax - v1)/ amax;
402       }
403         Tv = (_q1 - _q0)/vmax -  Ta*(1.0F + v0 / vmax)/2.0F -  Td *(1.0F + v1 / vmax)/2.0F;
404
405      if (Tv<0){
406         float it = 0;
407         float l = 0.99F;
408         float max_iter = 200;
409         while ((it < max_iter) && (amax >= 0.01F)){
410               //Vmax not reach
411               Tj1 = amax / jmax;
412               Tj2 = Tj1;
413               Tj = Tj2;
414               amax_squared = amax * amax;
415               sqrt_delta = sqrtf( ((amax_squared*amax_squared)/(jmax * jmax)) + 2.0F*(v0 * v0
    +  v1 * v1)
416                     + amax * (4.0F*(_q1 - _q0) - 2.0F* amax *(v0 + v1)/jmax));
417               Ta = (amax_squared/jmax - 2.0F * v0 + sqrt_delta)/(2.0F * amax);
418               Td = (amax_squared/jmax - 2.0F * v1 + sqrt_delta)/(2.0F * amax);
419               Tv = 0;
420               if (Ta < 0.0F){
421                  Td = 2.0F*((_q1 - _q0) / (v1 + v0));
422                  Tj2 = (jmax *(_q1 - _q0)-sqrtf(jmax *(jmax*(powf((_q1-_q0),2.0F))+(powf((v1
    +v0),2.0F))*
423                  (v1-v0))))/(jmax*(v1+v0));
424                  Ta = 0;
425                  Tj1 = 0;
426               }
427               if(Td < 0.0F){
428                  Ta = 2.0F*((_q1-_q0)/(v1+v0));
429                  Tj1 = (jmax*(_q1-_q0 )-sqrtf(jmax *(jmax *(powf((_q1 -_q0 ),2.0F))-(powf((v1
    +v0 ),2.0F))*
430                  (v1 -v0 ))))/(jmax *(v1 +v0 ));
431                  Td  = 0;
```

```
432                 Tj2  = 0;
433               }
434
435               if ((Ta  < 2.0F*Tj1 ) || (Td  < 2.0F*Tj2 )){
436                 amax  =amax * l;
437               }
438               else{
439                 break;
440               }
441         }
442       }
443       alima  = jmax *Tj1 ;
444       alimd  = -jmax  * Tj2 ;
445       vmax  = v0  + (Ta  - Tj1 )*alima ;
446       vlim  = vmax ;
447       T  = Ta +Tv +Td ;
448
449       qprim = 0;
450       float t=0;
451       TotalCycle = (int)T/stime+1;
452       for(int tim=0; tim<((int)(T/stime)+1);tim++){
453         t = tim*stime;
454         if (t<Tj1 ){
455           _q  = sigma *(_q0 +v0 *t+jmax *powf(t,0.5F));
456           _qd = sigma*(v0+jmax*powf(t,2.0F)/2);
457           _qdd = sigma*(jmax*t);
458         }
459         else if (t<(Ta -Tj1 )){
460           _q  = sigma *(_q0 +v0 *t+(3.0F*powf(t,2.0F)-3.0F*Tj1 *t+powf(Tj1 ,2.0F))*(alima /
    6.0F));
461           _qd = sigma*(v0+alima*(t-(Tj1/2.0F)));
462           _qdd = sigma*alima;
463         }
464         else if (t<Ta ){
465           _q  = sigma *(_q0 +(vlim +v0 )*(Ta /2.0F)-vlim *(Ta -t)-jmin *powf((Ta -t),0.5F));
466           _qd = sigma*(vlim+jmin*(powf((Ta-t),2.0F)/2.0F));
467           _qdd = sigma*(-jmin*(Ta-t));
468         }
469         else if (t<(Ta +Tv )){
470           _q  = sigma *(_q0 +(vlim +v0 )*(Ta /2.0F)+vlim *(t-Ta ));
471           _qd = sigma*vlim;
472           _qdd = 0;
473         }
474         else if (t<(T -Td +Tj2 )){
475           _q  = sigma *(_q1 -(vlim +v1 )*(Td /2.0F)+vlim *(t-T +Td )-jmax *(powf((t-T +Td ),
    0.5F)));
476           _qd = sigma*(vlim-jmax*(powf((t-T+Td),2.0F)/2.0F));
477           _qdd = sigma*(-jmax*(t-T+Td));
478         }
479         else if (t<(T -Tj2 )){
480           _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)));
481           _qd = sigma*(vlim+alimd*(t-T+Td-(Tj2/2.0F)));
482           _qdd = sigma*alimd;
483         }
484         else if (t<=T ){
```

```
485             _q  = sigma *(_q1 -v1 *(T -t)-jmax *(powf((T -t),3.0F))/6.0F);
486             _qd = sigma*(v1+jmax*(powf((T-t),2.0F))/2.0F);
487             _qdd = sigma*(-jmax*(T-t));
488         }
489         else{
490             _q  = sigma *_q0 ;
491             _qd = sigma*v1;
492             _qdd = 0;
493         }
494         _Trajectory[index].Insert(_q,_qd,_qdd);
495     }
496     index++;
497     return 1;
498 }
499
500 void MotionCommand::Spline_5th (float q0,float q1, float qd0, float qd1, float qdd,float
    stime,float tfin,float Constraints_v ,float Constraints_a){
501     float a, b, c, d, e, f,_q,_qd,_qdd;
502
503     if(qd0>Constraints_v){
504         qd0 = Constraints_v;
505     }
506     if(qdd>Constraints_a){
507         qdd = Constraints_a;
508     }
509     //Calc the parameters
510     f  = (6.0F*q1)/powf(tfin,5.0F) - (6.0F*q0)/powf(tfin,5.0F) - (3.0F*qd0)/powf(tfin,4.0F)
511     - (3.0F*qd1)/powf(tfin,4.0F) - qdd/(2*powf(tfin,3.0F));
512     e  = (15.0F*q0)/powf(tfin,4.0F) - (15.0F*q1)/powf(tfin,4.0F)+ (8.0F*qd0)/powf(tfin,3.0F)
513     + (7.0F*qd1)/powf(tfin,3.0F) + (3.0F*qdd)/(2.0F*powf(tfin,2.0F));
514     d  = (10.F*q1)/powf(tfin,3.0F) - (10.0F*q0)/powf(tfin,3.0F) - (6.0F*qd0)/powf(tfin,2.0F)
515     - (4.0F*qd1)/powf(tfin,2.0F) - (3.0F*qdd)/(2.0F*tfin);
516     c  = qdd/2.0F;
517     b  = qd0;
518     a  = q0 ;
519
520     TotalCycle = (int) tfin/stime;
521
522     //Sampling in units of st (sampling time)
523     for(int t=0;t<(int)((tfin/stime)+1);t++){
524         _q  = a  + b *((float)t)*stime + c *powf(((float)t)*stime,2.0F) +
525                   d *powf(((float)t)*stime,3.0F) + e *powf(((float)t)*stime,4.0F) + f *powf(((
    float)t)*stime,5.0F);
526         _qd  = b  + 2.0F*c *((float)t)*stime + 3.0F*d *powf(((float)t)*stime,2.0F) +
527                   4.0F*e *powf(((float)t)*stime,3.0F) + 5.0F*f *powf(((float)t)*stime,4.0F);
528         _qdd  = 2.0F*c  + 6.0F*d *((float)t)*stime + 12.0F*e *powf(((float)t)*stime,2.0F) +
529                   20*f *powf(((float)t)*stime,3.0F);
530         _Trajectory[index].Insert(_q,_qd,_qdd);
531     }
532     index++;
533 }
534
535
536 void MotionCommand::Spline_7th(float q0,float q1, float qd0, float qd1, float qdd,float
    stime,float tfin,float Constraints_v ,float Constraints_a){
537   float a, b, c, d, e, f, g, h,_q,_qd,_qdd;
```

```
538     if(qd0>Constraints_v){
539         qd0 = Constraints_v;
540     }
541     if(qdd>Constraints_a){
542         qdd = Constraints_a;
543     }
544
545     //Calc the parameters
546     h = (35.0F*q1)/powf(tfin,4.0F)- (35.0F*q0)/powf(tfin,4.0F) - (20.0F*qd0)/powf(tfin,3.0F
    ) - (15.0F*qd1)/powf(tfin,3.0F) - (5.0F*qdd)/powf(tfin,2.0F);
547     g = (84.0F*q0)/powf(tfin,5.0F) - (84.0F*q1)/powf(tfin,5.0F) + (45.0F*qd0)/powf(tfin,4.0F
    ) + (39.0F*qd1)/powf(tfin,4.0F) + (10.0F*qdd)/powf(tfin,3.0F);
548     f = (70.0F*q1)/powf(tfin,6.0F) - (70.0F*q0)/powf(tfin,6.0F) - (36.0F*qd0)/powf(tfin,5.0F
    ) - (34.0F*qd1)/powf(tfin,5.0F) - (15.0F*qdd)/(2.0F*powf(tfin,4.0F));
549     e = (35.0F*q1)/powf(tfin,4.0F) - (35.0F*q0)/powf(tfin,4.0F) - (20.0F*qd0)/powf(tfin,3.0F
    ) - (15.0F*qd1)/powf(tfin,3.0F) - (5.0F*qdd)/powf(tfin,2.0F);
550     d = 0.0F;
551     c = Constraints_a/2.0F;
552     b = Constraints_v;
553     a = q0;
554     TotalCycle = (int) tfin/stime;
555     //Sampling in units of st (sampling time)
556     for(int t=0;t<(int) tfin/stime;t++){
557         _q = a + b*((float)t)*stime + c*powf(((float)t)*stime,2.0F) + d*powf(((float)t)*
    stime,3.0F) +
558                                 e*powf(((float)t)*stime,4.0F) + f*powf(((float)t)*stime,5.0F
    ) + g*powf(((float)t)*stime,6.0F)
559                                 + h*powf(((float)t)*stime,7.0F);
560         _qd = b + 2.0F*c*((float)t)*stime + 3.0F*d*powf(((float)t)*stime,2.0F) +
561                                   4.0F*e*powf(((float)t)*stime,3.0F) + 5.0F*f*powf(((float)t
    )*stime,4.0F) +
562                                   6.0F*g*powf(((float)t)*stime,5.0F) + 7.0F*h*powf(((float)t
    )*stime,6.0F);
563         _qdd = 2.0F*c + 6.0F*d*((float)t)*stime + 12.0F*e*powf(((float)t)*stime,2.0F) +
564                                   20.0F*f*powf(((float)t)*stime,3.0F) + 30.0F*g*powf(((float
    )t)*stime,4.0F) +
565                                   42.0F*h*powf(((float)t)*stime,5.0F);
566         _Trajectory[index].Insert(_q,_qd,_qdd);
567     }
568     index++;
569
570 }
571
572
```

```
1  //
2  // Archivo cabezera de las matrices de transformación.
3  //
4
5
6  #ifndef UNTITLED_TRANSFORMATIONMATRIX_H
7  #define UNTITLED_TRANSFORMATIONMATRIX_H
8  #include "Matrix.h"
9  #include "cmath"
10
11 class TransformationMatrix : public Matrix<float> {
12 private:
13     void Clone(Matrix<float> Mat);
14     Matrix<float> RotXInner(float thetaX);
15     Matrix<float> RotYInner(float thetaY);
16     Matrix<float> RotZInner(float thetaZ);
17
18 public:
19     TransformationMatrix();
20     Matrix<float>& RotX(float thetaX);
21     Matrix<float>& RotY(float thetaY);
22     Matrix<float>& RotZ(float thetaZ);
23
24     //Euler angles
25     Matrix<float>& EulerXZX(float thetaX1,float thetaZ2,float thetaX3);
26     Matrix<float>& EulerXYX(float thetaX1,float thetaY2,float thetaX3);
27     Matrix<float>& EulerYXY(float thetaY1,float thetaX2,float thetaY3);
28     Matrix<float>& EulerYZY(float thetaY1,float thetaZ2,float thetaY3);
29     Matrix<float>& EulerZYZ(float thetaZ1,float thetaY2,float thetaZ3);
30     Matrix<float>& EulerZXZ(float thetaZ1,float thetaX2,float thetaZ3);
31
32
33     //Tait–Bryan angles
34     Matrix<float>& EulerXZY(float thetaX1,float thetaZ2,float thetaY3);
35     Matrix<float>& EulerXYZ(float thetaX1,float thetaY2,float thetaZ3);
36     Matrix<float>& EulerYXZ(float thetaY1,float thetaX2,float thetaZ3);
37     Matrix<float>& EulerYZX(float thetaY1,float thetaZ2,float thetaX3);
38     Matrix<float>& EulerZYX(float thetaZ1,float thetaY2,float thetaX3);
39     Matrix<float>& EulerZXY(float thetaZ1,float thetaX2,float thetaY3);
40
41     Matrix<float>& Translation(float dx, float dy, float dz);
42
43 };
44
45
46 #endif //UNTITLED_TRANSFORMATIONMATRIX_H
47
```

```
1  //
2  // Archivo capaz de utilizar matrices de transformación para por ejemplo determinar el End
   Effector.
3  //
4
5  #include "Ro/TransformationMatrix.h"
6  void TransformationMatrix::Clone(Matrix<float> Mat){
7     for (int i = 0; i <this->rows ; i++){
8        for (int j = 0; j < this->columns; j++){
9           this->n[i][j] = Mat(i,j);
10       }
11    }
12 }
13 TransformationMatrix::TransformationMatrix():Matrix<float>(4,4){}
14
15 Matrix<float>& TransformationMatrix::RotX(float thetaX){
16    float STheta = sinf(thetaX);
17    float CTheta = cosf( thetaX);
18    for(int x=0;x<4;x++){
19       for(int y=0; y<4; y++){
20          if ((x == 0 && y == 0) || (x == 3  && y == 3) ) {
21             this->n[x][y]=1;
22          } else if (x == 1 && y == 1) {
23             this->n[x][y]=CTheta;
24          } else if (x == 2 && y == 1) {
25             this->n[x][y]=STheta;
26          } else if (x == 1 && y == 2) {
27             this->n[x][y]=-STheta;
28          } else if (x == 2 && y == 2) {
29             this->n[x][y]=CTheta;
30          }else{
31             this->n[x][y]=0;
32          }
33          if( (this->n[x][y]>-0.000001F)&&(this->n[x][y]<0.001F) ){
34             this->n[x][y]=0;
35          }
36       }
37    }
38    return *this;
39 }
40 Matrix<float>& TransformationMatrix::RotY(float thetaY){
41    float STheta = sinf(thetaY);
42    float CTheta = cosf( thetaY);
43    for(int x=0;x<4;x++){
44       for(int y=0; y<4; y++){
45          if ((x == 1 && y == 1) || (x == 3  && y == 3) ) {
46             this->n[x][y]=1;
47          } else if (x == 0 && y == 0) {
48             this->n[x][y]=CTheta;
49          } else if (x == 0 && y == 2) {
50             this->n[x][y]=STheta;
51          } else if (x == 2 && y == 0) {
52             this->n[x][y]=-STheta;
53          } else if (x == 2 && y == 2) {
54             this->n[x][y]=CTheta;
55          }else{
```

```
56              this->n[x][y]=0;
57           }
58           if( (this->n[x][y]>-0.000001F)&&(this->n[x][y]<0.001F) ){
59              this->n [x][y]=0;
60           }
61        }
62     }
63     return *this;
64  }
65  Matrix<float>& TransformationMatrix::RotZ(float thetaZ){
66     float STheta = sinf(thetaZ);
67     float CTheta = cosf( thetaZ);
68     for(int x=0;x<4;x++){
69        for(int y=0; y<4; y++){
70           if ((x == 2 && y == 2) || (x == 3  && y == 3) ) {
71              this->n[x][y]=1;
72           } else if (x == 0 && y == 0) {
73              this->n[x][y]=CTheta;
74           } else if (x == 0 && y == 1) {
75              this->n[x][y]=-STheta;
76           } else if (x == 1 && y == 0) {
77              this->n[x][y]=STheta;
78           } else if (x == 1 && y == 1) {
79              this->n[x][y]=CTheta;
80           }else{
81              this->n[x][y]=0;
82           }
83           if( (this->n[x][y]>-0.000001F)&&(this->n[x][y]<0.001F) ){
84              this->n [x][y]=0;
85           }
86        }
87     }
88     return *this;
89  }
90
91
92  Matrix<float> TransformationMatrix::RotXInner(float thetaX){
93     Matrix<float> Mat(4,4);
94     float STheta = sinf(thetaX);
95     float CTheta = cosf( thetaX);
96     for(int x=0;x<4;x++){
97        for(int y=0; y<4; y++){
98           if ((x == 0 && y == 0) || (x == 3  && y == 3) ) {
99              Mat(x,y)=1;
100          } else if (x == 1 && y == 1) {
101             Mat(x,y)=CTheta;
102          } else if (x == 2 && y == 1) {
103             Mat(x,y)=STheta;
104          } else if (x == 1 && y == 2) {
105             Mat(x,y)=-STheta;
106          } else if (x == 2 && y == 2) {
107             Mat(x,y)=CTheta;
108          }else{
109             Mat(x,y)=0;
110          }
111          if( (Mat(x,y)>-0.000001F)&&(Mat(x,y)<0.001F) ){
```

```
112                 Mat(x,y)=0;
113             }
114         }
115     }
116     return Mat;
117 }
118 Matrix<float> TransformationMatrix::RotYInner(float thetaY){
119     Matrix<float> Mat(4,4);
120     float STheta = sinf(thetaY);
121     float CTheta = cosf( thetaY);
122     for(int x=0;x<4;x++){
123         for(int y=0; y<4; y++){
124             if ((x == 1 && y == 1) || (x == 3  && y == 3) ) {
125                 Mat(x,y)=1;
126             } else if (x == 0 && y == 0) {
127                 Mat(x,y)=CTheta;
128             } else if (x == 0 && y == 2) {
129                 Mat(x,y)=STheta;
130             } else if (x == 2 && y == 0) {
131                 Mat(x,y)=-STheta;
132             } else if (x == 2 && y == 2) {
133                 Mat(x,y)=CTheta;
134             }else{
135                 Mat(x,y)=0;
136             }
137             if( (Mat(x,y)>-0.000001F)&&(Mat(x,y)<0.001F) ){
138                 this->n [x][y]=0;
139             }
140         }
141     }
142     return Mat;
143 }
144 Matrix<float> TransformationMatrix::RotZInner(float thetaZ){
145     Matrix<float> Mat(4,4);
146     float STheta = sinf(thetaZ);
147     float CTheta = cosf( thetaZ);
148     for(int x=0;x<4;x++){
149         for(int y=0; y<4; y++){
150             if ((x == 2 && y == 2) || (x == 3  && y == 3) ) {
151                 Mat(x,y)=1;
152             } else if (x == 0 && y == 0) {
153                 Mat(x,y)=CTheta;
154             } else if (x == 0 && y == 1) {
155                 Mat(x,y)=-STheta;
156             } else if (x == 1 && y == 0) {
157                 Mat(x,y)=STheta;
158             } else if (x == 1 && y == 1) {
159                 Mat(x,y)=CTheta;
160             }else{
161                 Mat(x,y)=0;
162             }
163             if( (Mat(x,y)>-0.000001F)&&(Mat(x,y)<0.001F) ){
164                 Mat(x,y)=0;
165             }
166         }
167     }
```

```
168     return Mat;
169 }
170
171 Matrix<float>& TransformationMatrix::EulerXZX(float thetaX1,float thetaZ2,float thetaX3
    ){
172     Clone(RotXInner(thetaX1)*RotZInner(thetaZ2)*RotXInner(thetaX3));
173     return *this;
174 }
175 Matrix<float>& TransformationMatrix::EulerXYX(float thetaX1,float thetaY2,float
    thetaX3){
176     Clone(RotZInner(thetaX1)*RotXInner(thetaY2)*RotZInner(thetaX3));
177     return *this;
178 }
179 Matrix<float>& TransformationMatrix::EulerYXY(float thetaY1,float thetaX2,float
    thetaY3){
180     Clone (RotZInner(thetaY1)*RotXInner(thetaX2)*RotZInner(thetaY3));
181     return *this;
182 }
183 Matrix<float>& TransformationMatrix::EulerYZY(float thetaY1,float thetaZ2,float thetaY3
    ){
184     Clone (RotZInner(thetaY1)*RotXInner(thetaZ2)*RotZInner(thetaY3));
185     return *this;
186 }
187 Matrix<float>& TransformationMatrix::EulerZYZ(float thetaZ1,float thetaY2,float thetaZ3
    ){
188     Clone (RotZInner(thetaZ1)*RotXInner(thetaY2)*RotZInner(thetaZ3));
189     return *this;
190 }
191 Matrix<float>& TransformationMatrix::EulerZXZ(float thetaZ1,float thetaX2,float thetaZ3
    ){
192     Clone (RotZInner(thetaZ1)*RotXInner(thetaX2)*RotZInner(thetaZ3));
193     return *this;
194 }
195
196 Matrix<float>& TransformationMatrix::EulerXZY(float thetaX1,float thetaZ2,float thetaY3
    ){
197     Clone (RotZInner(thetaX1)*RotXInner(thetaZ2)*RotZInner(thetaY3));
198     return *this;
199 }
200 Matrix<float>& TransformationMatrix::EulerXYZ(float thetaX1,float thetaY2,float thetaZ3
    ){
201     Clone (RotZInner(thetaX1)*RotXInner(thetaY2)*RotZInner(thetaZ3));
202     return *this;
203 }
204 Matrix<float>& TransformationMatrix::EulerYXZ(float thetaY1,float thetaX2,float thetaZ3
    ){
205     Clone (RotZInner(thetaY1)*RotXInner(thetaX2)*RotZInner(thetaZ3));
206     return *this;
207 }
208 Matrix<float>& TransformationMatrix::EulerYZX(float thetaY1,float thetaZ2,float thetaX3
    ){
209     Clone (RotZInner(thetaY1)*RotXInner(thetaZ2)*RotZInner(thetaX3));
210 }
211 Matrix<float>& TransformationMatrix::EulerZYX(float thetaZ1,float thetaY2,float thetaX3
    ){
212     Clone (RotZInner(thetaZ1)*RotXInner(thetaY2)*RotZInner(thetaX3));
```

```
213     return *this;
214 }
215 Matrix<float>& TransformationMatrix::EulerZXY(float thetaZ1,float thetaX2,float thetaY3
    ){
216     Clone (RotZInner(thetaZ1)*RotXInner(thetaX2)*RotZInner(thetaY3));
217     return *this;
218 }
219
220
221 Matrix<float>& TransformationMatrix::Translation(float dx, float dy, float dz){
222     for(int x=0;x<4;x++){
223         for(int y=0; y<4; y++){
224             if ( (x == 0 && y == 0) || (x == 1 && y == 1) || (x == 2 && y == 2)  || (x == 3  && y
     == 3) ) {
225                 this->n[x][y]=1;
226             } else if (x == 0 && y == 3) {
227                 this->n[x][y]=dx;
228             } else if (x == 1 && y == 3) {
229                 this->n[x][y]=dy;
230             } else if (x == 2 && y == 3) {
231                 this->n[x][y]=dz;
232             }else{
233                 this->n[x][y]=0;
234             }
235         }
236     }
237     return *this;
238 }
239
240
241
```