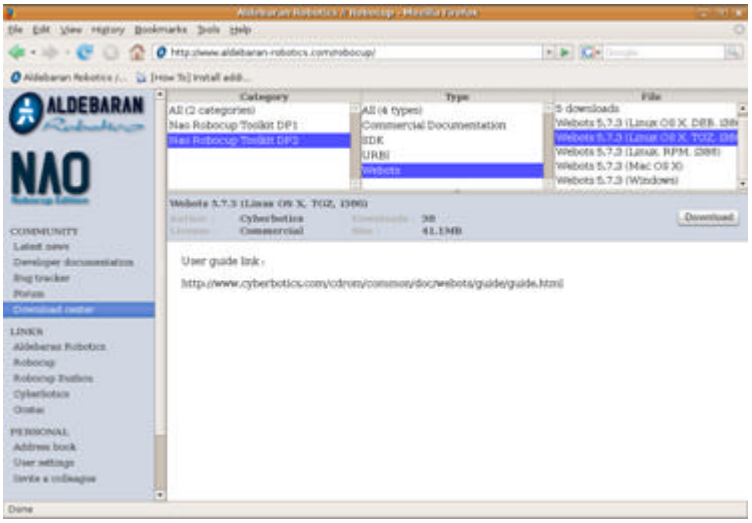
Simulación del robot Aldebaran Nao en Webots 5.7.3

El contenido de este documento describe la instalación y ejecución de la aplicación Webots 5.7.3 bajo la plataforma Ubuntu 7.10.

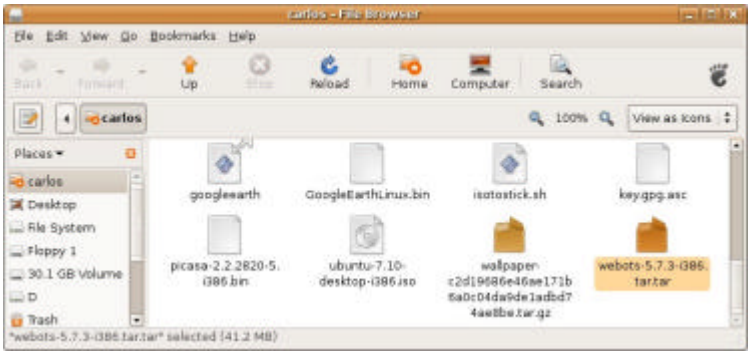
Instalación del software Webots y paquete de desarrollo

De la sección 'Download center' dentro del sitio http://www.aldebaran-robotics.com/robocup/ es posible descargar la 5.7.3 del software Webots. Dicha aplicación es una versión a prueba de Webots pero que permite el ser utilizada de forma ilimitada para la simulación del robot Aldebara Nao dentro del entorno de la Standart Platform de RoboCup.

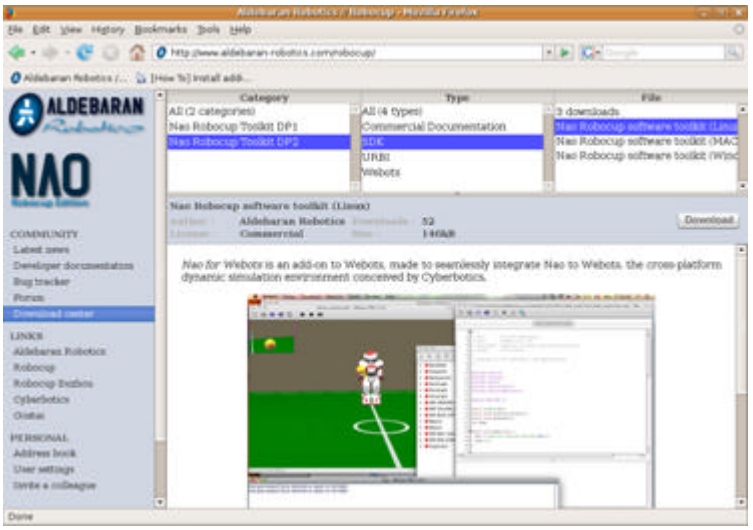


Una vez que se ha accedido a 'Download center' seleccionar en el menú 'Category' la opción 'Nao Robocup Tookit DP2'; en la sección 'Type', 'Webots' y finalmente en la sección File seleccionar 'Webots 5.7.3 (Linux OS X, TGZ, i386)'. Presionar el botón 'Download' para comenzar la descarga, el archivo se llama 'webots-5.7.3-i386.tar.tar'

Una vez que se ha descargado webots-5.7.3-i386.tar.tar, se debe colocar dicho archivo en la carpeta donde se desea instalar Webots. En este ejemplo se utilizo el Home Folder de una sesión llamada carlos, esto es, /home/carlos/ pero en adelante se referirá a dicho directorio como /home/[user]/.



Seleccionar el archivo y pulsar el botón derecho del mouse, seleccionar la opción 'Extract Here'. Se creará el directorio webots en esta carpeta.



También del 'Download center' descargar el 'Nao Robocup Software Toolkit (Linux)', este paquete se encuentra también dentro la categoría 'Nao Robocup Tookit DP2' pero dentro del 'Type' llamado 'SDK'. Una vez descargado descomprimir y copiar la carpeta nao_robocup a la carpeta /home/[user]/webots/projects/contest/. Se pedirá permiso para sobrescribir los archivos ya existentes pues en dicho directorio existe ya una carpeta con el nombre nao_robocup. se debe aceptar la sobre escritura.

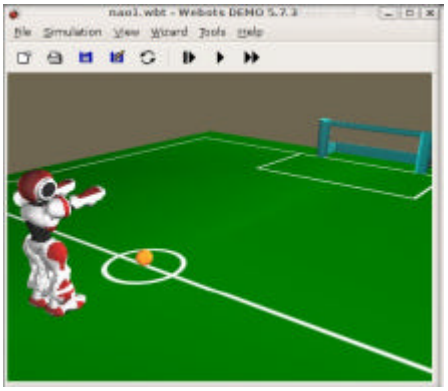
Ahora se debe acceder a la carpeta /home/[user]/webots/projects/contest/nao_robocup/controllers/nao_soccer_player_red/. Una vez ahí se debe cambiar el nombre del archivo nao_soccer_player_red_.c por cualquier otro. Esto es necesario pues Webots tomará este archivo como el controlador del robot, nosotros deseamos que el controlador sea el archivo nao_soccer_player_red_.cpp.

Para ejecutar Webots 5.7.3 seleccionar el archivo webots en /home/[user]/webots.

Simulacion de la Standart Platform League en Webots 5.7.3

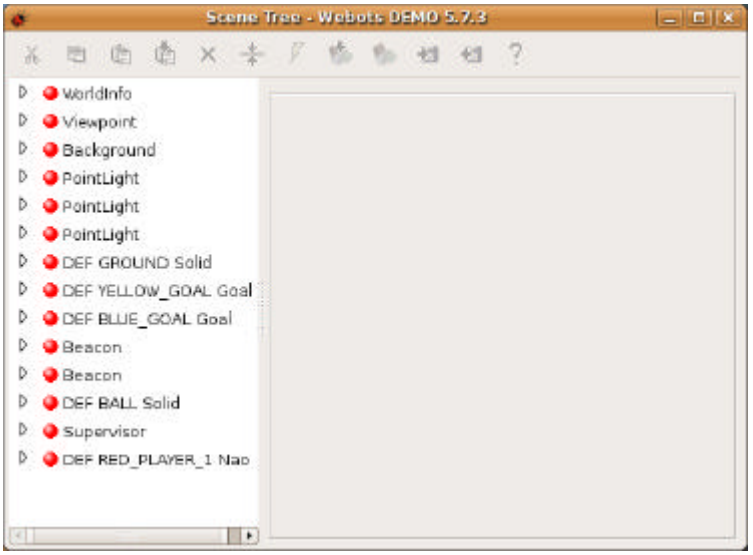
El simulador Webots consta de 3 ventanas, la principal muestra el entorno en que se desenvuelve el robot. La de 'Text Editor' permite modificar y compilar el código que controla el o los robots. La de 'Log' despliega información que se genera durante la simulación.

Dentro del directorio /home/[user]/webots/projects/contest/nao_robocup/worlds/ se encuentran varios archivos con extensión .wbt. Cada uno de ellos describe las características del espacio en las que el o los robots interactuaran durante la simulación. Por ejemplo, el archivo nao1.wbt consta de un solo robot Nao en una cancha de la Standart Platform League mientras que el archivo nao2.wbt contiene dos robots, uno de cada equipo, en la misma cancha.

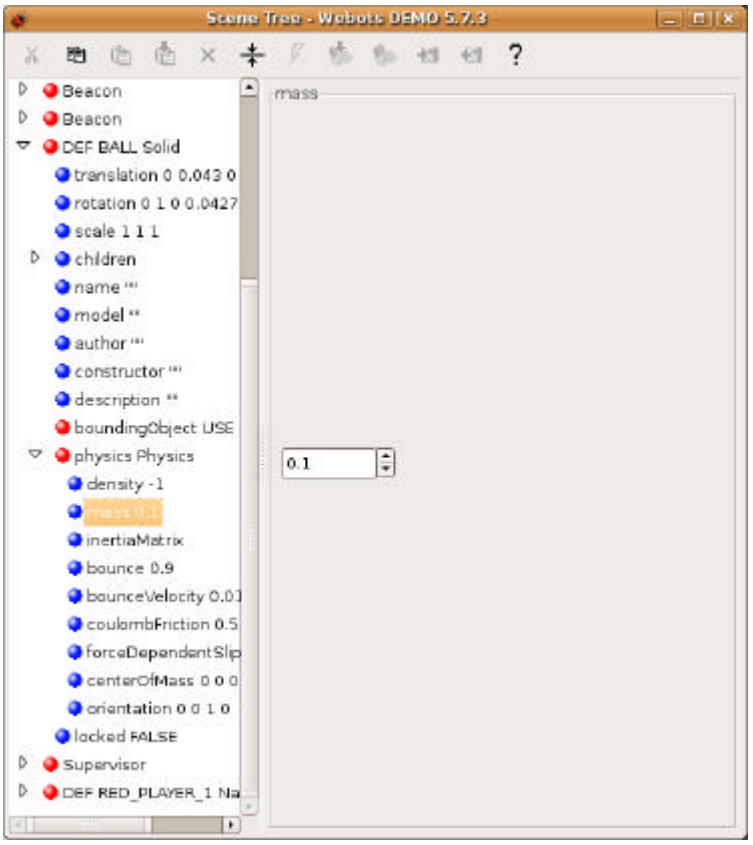


Cuando algún archivo .wbt es cargado, se despliega en la ventana 'Text Editor' el código del controlador del robot. Para el archivo nao1.wbt, el comportamiento del robot está definido en nao_soccer_player_red.cpp, este archivo se encuentran en la carpeta /home/user]/webots/projects/contest/nao_robocup/controllers/nao_soccer_player_blue/. En el caso del nao2.wbt, el control del robot azul esta definido en nao_soccer_player_blue.cpp dentro de la carpeta /home/[user]/webots/projects/contest/nao_robocup/controllers/nao_soccer_player_blue/.

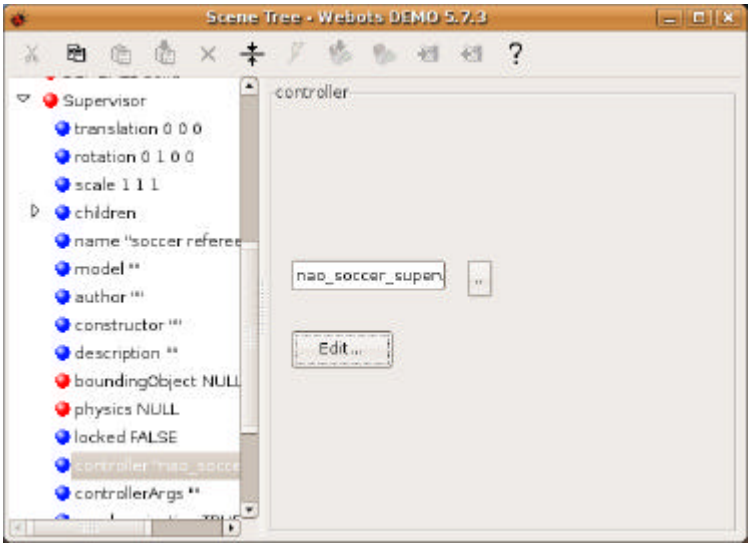
Es posible en cualquier momento cambiar el controlador de alguno o todos los los robots, para ello, en la ventana principal hay que acceder al menú Tools y en este seleccionar la opción 'Scene Tree'.



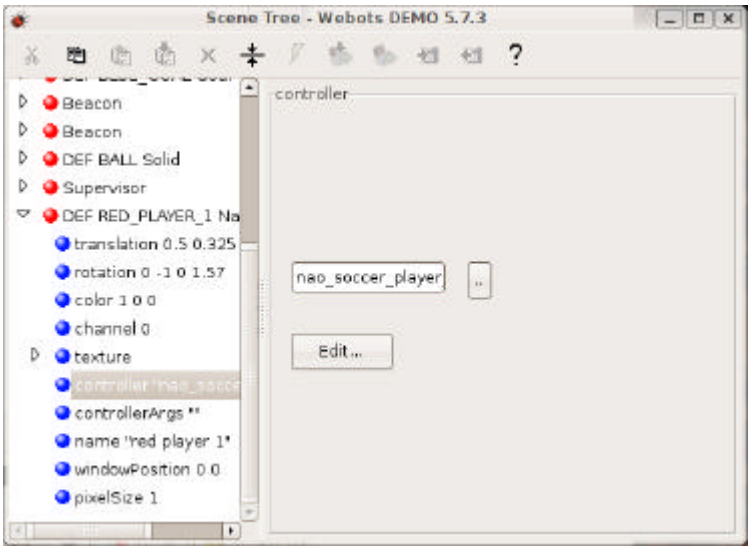
La ventana 'Scene Tree' permite acceder a la configuración de los objetos que están presentes en el entorno de simulación, estos incluyen el o los robots, la pelota, las porterías, etc. Por ejemplo, si se selecciona el objeto 'DEF BALL_Solid'-'physics PHYSICS' se pueden modificar las propiedades físicas de la pelota; lo anterior, por supuesto, modificará el comportamiento de la pelota durante la simulación. Lo mismo puede hacerse para todos los objetos incluidos en el entorno a simular.



A partir de este punto se trabajará con el archivo nao01.wbt. Dentro del 'Scene Tree' bajo la opción 'Supervisor'-'Controller'. podemos modificar el código que controla elementos del entorno, por ejemplo, la modificación del marcador ante un gol. Por defecto, el controlador es el archivo nao_soccer_supervisor.c que se encuentra en /home/user]/webots/projects/contest/nao_robocup/controllers/nao_soccer_supervisor/. Es posible modificar este controlador pero no cambiarlo por otro pues la versión 5.7.3 de Webots liberada para la simulación de la Standart Platform League solo permite construir los objetos nao_soccer_supervisor, nao_soccer_player_red y nao_soccer_player_blue.



El controlador del robot esta definido bajo la opción 'DEF RED_PLAYER_1 Nao'-'controller'. Si el código del archivo controlador no se desplegó en la ventana 'Text Editor' al cargar el archivo nao1.wbt, es posible abrirlo pulsando sobre el botón Edit.



Demos del Nao Robocup Tookit DP2

El 'Nao Robocup Tookit DP2' contiene varios demos donde se puede observar el funcionamiento del robot Nao bajo distintas circunstancias. Para poder ejecutar dichos demos es necesario hacer algunas modificaciones al código del archivo nao_soccer_player_red.cpp.

Una de las primeras cosas que puede notar al revisar el código es la definición de algunas constantes, dichas constantes son usadas mas adelante en el código para controlar la ejecución de los demos. Si desea que el demo se ejecute, la constante debe ser definida en 1, si no se desea ejecutar algún demo en particular, la constante que le corresponde debe ser definida en 0.

```
#define DEMO_CAMERA 1
#define DEMO_MOTION 1
#define DEMO_WALK 1
#define DEMO_FSR 1
#define DEMO_INERTIAL 1
#define DEMO_DANCE 1
```

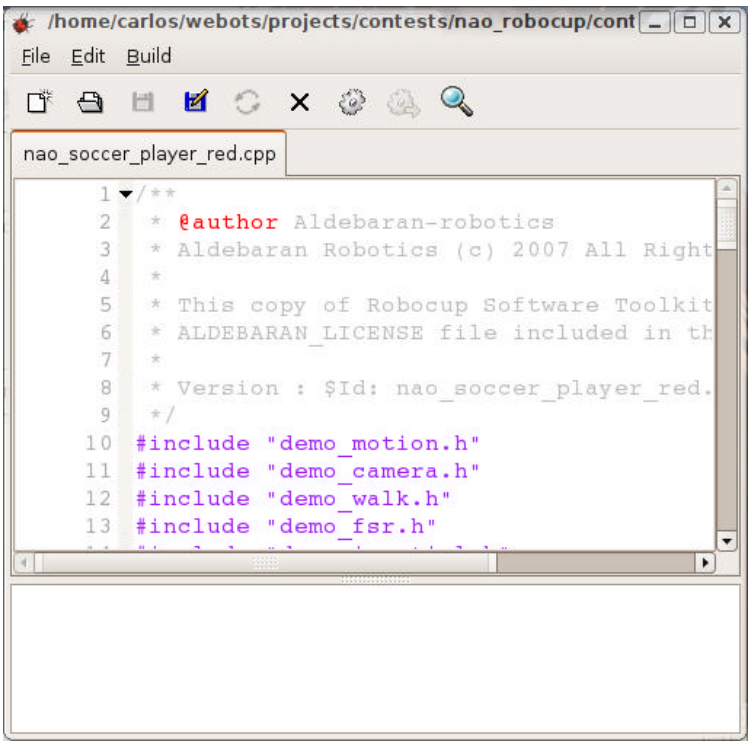
En el método executeBehavior se encuentran las instrucciones ifdef utilizadas para controlar el acceso a cada demo.

```
void *executeBehaviour (void *pNothing)
{
  ALNaoProxy nao;
  nao.oSleep(1);
/**************
 * DEMO CAMERA
***************/
#ifdef DEMO_CAMERA
  demo_camera();
#endif /*!DEMO_CAMERA*/
...
}
```

Estas instrucciones deben ser comentadas o eliminadas de modo que el método executeBehavior quede como sigue:

```
void *executeBehaviour (void *pNothing)
{
  ALNaoProxy nao;
  nao.oSleep(1);

if (DEMO_CAMERA)
  demo_camera();

if (DEMO_MOTION)
  demo_motion();

if (DEMO_WALK)
  demo_walk();

if (DEMO_FSR)
  demo_fsr();

if (DEMO_INERTIAL)
  demo_inertial();

if (DEMO_DANCE)
  demo_dance();

  pthread_exit(NULL);
  return NULL;
}
```

Ahora es necesario construir de nuevo el controlar con los cambios realizados. Para ello se debe seleccionar el icono con la figura de un engrane o acceder a la opción Make dentro del menú Build.



Con lo anterior es posible ya observar las simulaciones incluidas en el Nao Robocup Tookit DP2. A continuación se describirá el funcionamiento de los ejemplos demo_camera y demo_motion.

demo_camera

El demo que muestra el funcionamiento de la cámara esta definido en el archivo demo_camera.cpp. En dicho archivo el método que controla la ejecución del demo es demo_camera(), este método utiliza a su vez los métodos oFindBall y oMoveBall para identificar la bola e indicar al supervisor que mueva la pelota, respectivamente.

El método oFindBall sirve para determinar si un píxel es de color naranja, para ello se pasan a dicho método enteros con el valor de los componentes Rojo, Verde y Azul de un píxel. Cabe señalar que esto se hace así en este demo pues el simulador regresa la imagen del archivo como una colección de bytes que describen la imagen en el espacio RGB. Así, en este método se considera como naranja todo lo que este en el rango 101-255 en R, en 111-255 en G y 30-70 en B

```
...
inline bool oFindBall(int pRed, int pGreen, int pBlue )
{
  // Mervellous world, where everything is well colored ;)
  if(pRed <= 255 && pRed > 100 &&
    pGreen <= 255 && pGreen > 110 &&
    pBlue < 70 && pBlue > 30
    )
  {
    //we got a ball !!!!
    return true;
  }
  return false;
}
...
```

En demo_camera, el ciclo de identificación se realiza 150 veces; 120 veces de ellas la bola se estará moviendo en círculo, el movimiento es enviado al supervisor utilizando el método oMoveBall.

```
...
void demo_camera()
{
   ALCameraProxy cam;
   ALMotionProxy motion;
    ...
  std::vector<float> pos;
    ...
   for (unsigned short i=0;i<150;i++)
   {
    if (i<120)
      oMoveBall(-0.2f, 0.5f+0.20*cosf( 2*3.14*(i)/60 ), 0.20*sinf( 2*3.14*(i)/60 ) );
   // Get the image backImage is in RGB mode, according to webots. But, cameraProxy/ will provide conversion methods.
   const uint8* img = cam.oGet();

   int yBall = -1;    // mean of Y ball pos in the image, init with a non sens value
   int xBall = -1;    // mean of X ball pos in the image, init with a non sens value
   int pixelBall = 0; // Number of pixel with the good color.

   // Looking for orange colors in RGB mode.
   int r, g, b;
   for( unsigned short i = 0; i < height; i++)
   {
     for( unsigned short j = 0; j < width; j++)
     {
       r = *img++; g = *img++; b = *img++;
       if( oFindBall( r, g, b ) )
       {
         xBall += j;
         yBall += i;
         pixelBall++;
```

```
        }
      }
    }
    ...
   if( pixelBall > minPixelFound )
    {
      // compute mean position
      xBall /= pixelBall;
      yBall /= pixelBall;


      // camera is 60 deg width
      float degX = (2.0f * (float)xBall / (float)width - 1.0f) * 30.0f;
      // and 45 deg height
      float degY = (2.0f * (float)yBall / (float)height - 1.0f) * 22.5f;
       ....

      // prepare to move the head in the direction of the ball.
      // note that Yaw rotation of the head (lateral rotation) is positive on the left (trigonometric convention)
      // and our degY is compute with anti-trigonometric convention
      pos.clear();
      pos.push_back( degreesToRadians(-degX) + motion.oGetAngle("HeadYaw") );
      pos.push_back( degreesToRadians(degY) + motion.oGetAngle("HeadPitch"));

      // execute movement
      if(motion.oPostGotoChainAngles("Head", pos, 0.2f, "linear"))
        robot_console_printf("execeeded speed\n");

    }
}
```

Cada una de las 150 veces, mediante el método cam.oGet se obtiene la imagen de la cámara. cam es un objeto del tipo ALCameraProxy, dicho tipo de objeto nos permite acceder a información relacionada con la cámara.

Una vez que se tiene la imagen, mediante un par de ciclos se recorre el contenido de la imagen extrayendo el valor en R, G y B para cada píxel. Como se mencionó anteriormente, Webots regresa la imagen de la cámara del Nao en el espacio de color RGB; la imagen que regresa tiene un tamaño de 160x120 píxeles. Es importante tener en cuenta esto para el desarrollo pues en realidad el Nao regresa la imagen en el espacio de color YUV y con un tamaño de 640x480 y está parte del código debe ser modificado para ser probada en el Nao.

Finalmente, para cada píxel, utilizando el método oFindBall se determina si es de color naranja. Tras recorrer la imagen, si se encontraron mas de 20 píxel naranjas se considera que se ha encontrado una pelota, en este caso se mueve la cabeza del Nao buscando centrar la pelota en la imagen de la cámara.

Para efectuar el movimiento se utiliza el objeto motion de la clase ALMotionProxy. El metodo oPostGotoChainAngle de dicha clase recibe 4 parámetros. El primero es una cadena de caracteres con el nombre de la parte a mover, en este caso “Head” indica que se moverán ambos motores de la cabeza. El segundo parámetro es un stl de tipo vector que contiene el valor en radianes de cada motor a mover, en este caso contiene solo dos valores, uno para el motor HeadYaw y otro para HeadPitch. El tercer parámetro es el tiempo en segundos en el que deseamos se realice el movimiento. El último parámetro es una cadena de caracteres que indica que tipo de interpolación de realizara entre los valores actuales de los motores y los que se desean, en este caso se trata de una interpolación lineal.

Para calcular la siguiente posición, se utilizan la función oGetAngle de la clase ALMotionProxy para obtener el valor en radianes de un motor, el nombre del motor es el parámetro de dicha función. Tambien se emplea la función degreeToRadians para convertir de grados a radianes.

demo_motion

En el demo_motion podemos observar el movimiento de las extremidades del robot nao. Para ello se hace uso también del objeto motion de tipo ALMotionProxy.

Al inicio del demo, se coloca el valor de todos los motores en una posición que llamaremos inicial, se utiliza el stl::vector positions para almacenar el valor deseado para los 22 motores y se establecen mediante el método oGotoBodyAngles; este método recibe como parámetros el stl, el tiempo de interpolación y el tipo de interpolación. La función bloquea la ejecución hasta que los motores se han establecidos en la posición deseada.

```
void demo_motion ()
{
  ALMotionProxy motion;
  ...
  // prepare the initial position of the robot :
  std::vector<float> positions;
  positions.push_back( degreesToRadians(0.0f)   );   //  HeadYaw
  positions.push_back( degreesToRadians(0.0f)   );   //  HeadPitch
  positions.push_back( degreesToRadians(120.0f) );   //  LShoulderPitch
  positions.push_back( degreesToRadians(20.0f)  );   //  LShoulderRoll
  positions.push_back( degreesToRadians(-80.0f) );   //  LElbowYaw
  positions.push_bac k( degreesToRadians(-80.0f) );   //  LElbowRoll
  positions.push_back( degreesToRadians(0.0f)   );   //  LHipYawPitch
  //see note 1.
  positions.push_back( degreesToRadians(-20.0f) );   //  LHipPitch
  positions.push_back( degreesToRadians(0.0f)   );   //  LHipRoll
  positions.push_back( degreesToRadians(40.0f)  );   //  LKneePitch
  positions.push_back( degreesToRadians(-20.0f) );   //  LAnklePitch
  positions.push_back( degreesToRadians(0.0f)   );   //  LAnkleRoll
  positions.push_back( degreesToRadians(0.0f)   );   //  RHipYawPitch
  //see note 1.
  positions.push_back( degreesToRadians(-20.0f) );   //  RHipPitch
  positions.push_back( degreesToRadians(0.0f)   );   //  RHipRoll
  positions.push_back( degreesToRadians(40.0f)  );   //  RKneePitch
  positions.push_back( degreesToRadians(-20.0f) );   //  RAnklePitch
  positions.push_back( degreesToRadians(0.0f)   );   //  RAnkleRoll);
  positions.push_back( degreesToRadians(120.0f) );   //  RShoulderPitch
  positions.push_back( degreesToRadians(-20.0f) );   //  RShoulderRoll
```

```
  positions.push_back( degreesToRadians(80.0f)  );   //  RElbowYaw
  positions.push_back( degreesToRadians(80.0f)  );   //  RElbowRoll


  // Note 1 :
  // LHipYawPitch and RHipYawPitch share one motor. Priority is always
  // given to LHipYawPitch in the case that both are sent commands.

  // go to the init position in 2 sec
  motion.oGotoBodyAngles( positions, 2.0f, "smooth" );

  std::vector<float> leftArmPositions;
  std::vector<float> rightArmPositions;

  // prepare the first motion :
  leftArmPositions.push_back(0.0f);
  leftArmPositions.push_back(0.0f);
  leftArmPositions.push_back(0.0f);
  leftArmPositions.push_back(0.0f);

  rightArmPositions.push_back(0.0f);
  rightArmPositions.push_back(0.0f);
  rightArmPositions.push_back(0.0f);
  rightArmPositions.push_back(0.0f);

  // go to first position
  motion.oPostGotoChainAngles("LeftArm",leftArmPositions, 1.0f, "smooth");
  motion.oGotoChainAngles("RightArm",rightArmPositions, 1.0f, "smooth");
  ...
}
```

El siguiente movimiento se realiza de manera similar a como se realizó en demo_camera. En este caso se utilizan dos stl::vector, uno para el brazo izquierdo y otro para el derecho. Los identificadores para la función de movimiento son “LeftArm” y “RightArm”, respectivamente, Los vectores de posicion deben contener, en cada caso, los valores para los 4 motores que hay en un brazo. La funcion PostGotoChainAngles fija los valores pero no bloquea la ejecución, la función oGotoChainAngles si se bloquea, así logramos que ambos brazos se muevan simultáneamente, tras el movimiento se fijan en el vector los valores para la siguiente posición y se ejecutan utilizando los funciones en ese orden.

Uso del sonar

Para acceder al sonar es necesario declarar un objeto del tipo ALSonarProxy; esta clase contiene las funciones oGetLeft y oGetRight, las funciones devuelven la distancia, en metros, hasta algún objeto localizado a los costados del Nao. Un ejemplo de su uso puede ser observado en demo_camera.

Sensores de fuerza

El robot Nao cuenta con 8 sensores de fuerza, cada pie cuenta con 4 de ellos. En cada pie, se ubican dos al frente y dos atrás. La clase ALFSRProxy sirve para obtener el valor de la fuerza ejercida en cada uno. Un ejemplo de su uso puede observarse en el demo_frs.
La clase ALFSRProxy cuenta con las funciones oGet y oGets, la primera regresa el valor en Newtons de la fuerza ejercida sobre uno de los sensores, el identificador del sensor deseado se debe proporcionar como parámetro; la segunda función regresa un stl:vector con los valores en Newtons de todos los sensores.

Uso de los LEDs

El control de los LEDs del Nao se realiza mediante la clase ALLEDProxy, dicha clase contiene métodos como oOn y oOff para encender y apagar un determinado LED u oSet para cambiar el color de un LED, pasando como parámetro los valores R, G y B del color que se desee.
Webots no posee soporte para simular el uso de los LEDs del robot Nao.