Appendix D: Improved Steering Controller and Wheel

```
/*
 *  Gazebo - Outdoor Multi-Robot Simulator
 *  Copyright (C) 2003
 *     Nate Koenig & Andrew Howard
 *
 *  This program is free software; you can redistribute it and/or
modify
 *  it under the terms of the GNU General Public License as published
by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
1307  USA
 *
 */
/*
 *  An improved steering controller for a four-wheeled vehicle
 *  Author: J. C. Allen
 *  Date: 26 March 2010
 *  Based on "General steering controller for any number of wheels and
configuration",
 *   Jordi Polo, dated 23 Dec 2007
 */

#include "Global.hh"
#include "XMLConfig.hh"
#include "Model.hh"
#include "Simulator.hh"
#include "gazebo.h"
#include "GazeboError.hh"
#include "ControllerFactory.hh"
#include "Steering_Position2d.hh"
#include "Wheel.hh"
#include "ODEBody.hh"
#include <string.h>


using namespace gazebo;

GZ_REGISTER_STATIC_CONTROLLER("steering_position2d",
Steering_Position2d);

enum {DRIVE, STEER, FULL};

////////////////////////////////////////////////////////////////////////
////////
```

```
// Constructor
Steering_Position2d::Steering_Position2d(Entity *parent )
    : Controller(parent)
{
  this->myParent = dynamic_cast<Model*>(this->parent);

  if (!this->myParent)
    gzthrow("Steering_Position2d controller requires a Model as its
parent");

  Param::Begin(&this->parameters);
  // Load control parameters used by the steering controller
  this->velocityOffset = new ParamT<double>("velocityOffset", 0.0, 0);
  this->useSwaybars = new ParamT<bool>("useSwaybars", true, 0);
  this->swayForce = new ParamT<double>("swayForce", 300.0, 0);
  this->swayForceLimit = new ParamT<double>("swayForceLimit", 15.0, 0);
  this->useConstantVelocityMode = new
ParamT<bool>("useConstantVelocityMode", false, 0);
  this->useConstantSteeringAngleMode = new
ParamT<bool>("useConstantSteeringAngleMode", false, 0);
  this->constantSteeringAngle = new
ParamT<double>("constantSteeringAngle", 0.0, 0);
  this->useSafeVelocity = new ParamT<bool>("useSafeVelocity", true, 0);
  this->useTurnRadius = new ParamT<bool>("useTurnRadius", false, 0);
  this->turnRadius = new ParamT<double>("turnRadius", 46.1, 0);

  // Load default values for the steering controller
  this->defaultTorque = new ParamT<double>("torque", 1000.0, 0);
  this->defaultSteerTorque = new ParamT<double>("steerTorque", 1000.0,
0);

  // Load vehicle characteristics used by the steering controller
  this->tc = new ParamT<double>("turningCircle",11.491, 0);
  this->tw = new ParamT<double>("trackWidth", 1.580, 0);
  this->wb = new ParamT<double>("wheelBase", 2.619, 0);
  this->ssf = new ParamT<double>("ssf", 1.17, 0);
  this->vf = new ParamT<double>("velocityFinal", 27.778, 0);
  this->vt = new ParamT<double>("velocityFinalTime", 5.0, 0);
  this->tr = new ParamT<double>("tireRadius", 0.368, 0);
  this->sw = new ParamT<double>("sectionWidth", 0.235, 0);
  Param::End();

  this->enableMotors = true;

  this->prevUpdateTime = Simulator::Instance()->GetSimTime();

  this->sa=0;
  this->v=0;
}

//////////////////////////////////////////////////////////////////////
/////////
// Destructor
Steering_Position2d::~Steering_Position2d()
```

```
{
  delete this->velocityOffset;
  delete this->useSwaybars;
  delete this->swayForce;
  delete this->swayForceLimit;
  delete this->useConstantVelocityMode;
  delete this->useConstantSteeringAngleMode;
  delete this->constantSteeringAngle;
  delete this->useSafeVelocity;
  delete this->useTurnRadius;
  delete this->turnRadius;

  delete this->defaultTorque;
  delete this->defaultSteerTorque;

  delete this->tc;
  delete this->tw;
  delete this->wb;
  delete this->ssf;
  delete this->vf;
  delete this->vt;
  delete this->tr;
  delete this->sw;
}

////////////////////////////////////////////////////////////////////////
/////////
// Load the controller
void Steering_Position2d::LoadChild(XMLConfigNode *node)
{
  XMLConfigNode *childNode;
  std::string jointName, type;
  double torque, steerTorque;
  double g, r;

  this->myIface = dynamic_cast<PositionIface*>(this-
>GetIface("position"));

  // Load control parameters used by the steering controller
  this->velocityOffset->Load(node);
  this->useSwaybars->Load(node);
  this->swayForce->Load(node);
  this->swayForceLimit->Load(node);
  this->useConstantVelocityMode->Load(node);
  this->useConstantSteeringAngleMode->Load(node);
  this->constantSteeringAngle->Load(node);
  this->useSafeVelocity->Load(node);
  this->useTurnRadius->Load(node);
  this->turnRadius->Load(node);

  // Load default values for the steering controller
  this->defaultTorque->Load(node);
  this->defaultSteerTorque->Load(node);
```

```
  // Load vehicle characteristics used by the steering controller
  this->tc->Load(node);
  this->tw->Load(node);
  this->wb->Load(node);
  this->ssf->Load(node);
  this->vf->Load(node);
  this->vt->Load(node);
  this->tr->Load(node);
  this->sw->Load(node);

  std::cout<<"\n\nLoading the controller...\n";
  std::cout<<"  Load control parameters used by the steering
controller...\n";
  std::cout<<"    useSwaybars: "<<**this->useSwaybars<<"\n";
  if (**this->useSwaybars)
  {
  std::cout<<"      swayForce: "<<**this->swayForce<<"\n";
  std::cout<<"      swayForceLimit: "<<**this->swayForceLimit<<"\n";
  }
  std::cout<<"    useConstantVelocityMode: "<<**this-
>useConstantVelocityMode<<"\n";
  std::cout<<"    useConstantSteeringAngleMode: "<<**this-
>useConstantSteeringAngleMode<<"\n";
  if (**this->useConstantSteeringAngleMode)
    std::cout<<"      constantSteeringAngle: "<<**this-
>constantSteeringAngle<<"\n";
  std::cout<<"    useSafeVelocity: "<<**this->useSafeVelocity<<"\n";
  if (**this->useSafeVelocity)
    std::cout<<"      velocityOffset: "<<**this->velocityOffset<<"
m/s\n";
  std::cout<<"    useTurnRadius: "<<**this->useTurnRadius<<"\n";
  if (**this->useTurnRadius)
    std::cout<<"      turnRadius: "<<**this->turnRadius<<"\n";
  std::cout<<"  Load default values for the steering controller...\n";
  std::cout<<"    defaultTorque: "<<**this->defaultTorque<<"\n";
  std::cout<<"    defaultSteerTorque: "<<**this-
>defaultSteerTorque<<"\n";
  std::cout<<"  Load vehicle characteristics used by the steering
controller...\n";
  std::cout<<"    turningCircle: "<<**this->tc<<" m\n";
  std::cout<<"    trackWidth: "<<**this->tw<<" m\n";
  std::cout<<"    wheelBase: "<<**this->wb<<" m\n";
  std::cout<<"    ssf: "<<**this->ssf<<"\n";
  std::cout<<"    velocityFinal: "<<**this->vf<<" m/s\n";
  std::cout<<"    velocityFinalTime: "<<**this->vt<<" s\n";
  std::cout<<"    tireRadius: "<<**this->tr<<" m\n";
  std::cout<<"    sectionWidth: "<<**this->sw<<" m\n";
  std::cout<<"\nLoading the joints...\n";

  childNode = node->GetChild("wheel");

  while (childNode)
  {
    // Load default values for individual wheels.  These values
```

```
override the default values for the steering
    //  controller, above.
    jointName = childNode->GetString("jointName", "", 1);
    type = childNode->GetString("type", "", 1);
    torque = childNode->GetDouble("torque", **this->defaultTorque, 0);
    if (type != "drive")
      steerTorque = childNode->GetDouble("steerTorque", **this-
>defaultSteerTorque, 0);

    std::cout<<"  Loading: "<<jointName<<"\n";
    std::cout<<"    type: "<<type<<"\n";
    std::cout<<"    torque: "<<torque<<"\n";
    if (type != "drive")
      std::cout<<"    steerTorque: "<<steerTorque<<"\n";

    Wheel *wheel=new Wheel();

    if (type == "drive")
    {
      wheel->Connect(this->myParent->GetJoint(jointName), DRIVE);
      wheel->SetTorque(torque);
    }
    else
    {
      if (type == "steer")
      {
        wheel->Connect(this->myParent->GetJoint(jointName), STEER);
        wheel->SetTorque(0); // If the wheel is not full, FMax2 should
be 0 otherwise joint will lock
      }
      else
      {
        wheel->Connect(this->myParent->GetJoint(jointName), FULL);
        wheel->SetTorque(torque);
      }
      wheel->SetSteerTorque(steerTorque);
    }
    wheels.push_back(wheel);

    childNode= childNode->GetNext("wheel");
  }

  // Calculate vehicle characteristics used by the steering controller
  g = 9.80665; // acceleration due to gravity

  std::cout<<"\nCalculated vehicle characteristics used by the steering
controller...\n";

  // Calculate maximum velocity and maximum angular velocity at vehicle
center of gravity
  if (**this->useTurnRadius)
    r = **this->turnRadius + (**this->tw + **this->sw) / 2;
  else
    r = (**this->tc + **this->tw + **this->sw) / 2;
```

```
  std::cout<<"  Radius used to calculate maximum velocity, maximum
angular velocity, and maximum steering angle at vehicle center of
gravity: "<<r<<" m\n";

  wcgMax = sqrt(**this->ssf * g / r);

  if (**this->useSafeVelocity)
    vcgMax = sqrt(**this->ssf * r * g) + **this->velocityOffset;
  else
    vcgMax = **this->vf;

  std::cout<<"  Maximum velocity at vehicle center of gravity:
"<<vcgMax<<" m/s\n";
  std::cout<<"  Maximum angular velocity at vehicle center of gravity:
"<<wcgMax<<" rad/s\n";

  // Calculate maximum steering angle at vehicle center of gravity
  sacgMax = atan(**this->wb / r);

  std::cout<<"  Maximum steering angle at vehicle center of gravity:
"<<sacgMax<<" rad\n";

  // Calculate constant acceleration
  a = **this->vf / **this->vt;

  std::cout<<"  Acceleration: "<<a<<" m/s^2\n\n";

  vcg0 = 0;
  acg0 = 0;
}

////////////////////////////////////////////////////////////////////////
/////////
// Initialize the controller
void Steering_Position2d::InitChild()
{
  // Reset odometric pose
  this->odomPose[0] = 0.0;
  this->odomPose[1] = 0.0;
  this->odomPose[2] = 0.0;

  this->odomVel[0] = 0.0;
  this->odomVel[1] = 0.0;
  this->odomVel[2] = 0.0;
}

////////////////////////////////////////////////////////////////////////
/////////
// Reset the controller
void Steering_Position2d::ResetChild()
{
  // Reset odometric pose
  this->odomPose[0] = 0.0;
  this->odomPose[1] = 0.0;
```

```
  this->odomPose[2] = 0.0;

  this->odomVel[0] = 0.0;
  this->odomVel[1] = 0.0;
  this->odomVel[2] = 0.0;
}

////////////////////////////////////////////////////////////////////////
/////////
// Update the controller
void Steering_Position2d::UpdateChild()
{
  // local variables for tire radius, track width, and wheelbase
  double tr, tw, wb;
  // turning radius of wheels 1 through 4 and vehicle center of gravity
  double r1, r2, r3, r4, rcg;
  // linear distance traveled by wheels 1 through 4 and vehicle center
of gravity
  double d1 = 0.0, d2 = 0.0, d3, d4, dcg = 0.0;
  // linear distance traveled by wheels 1 and 2 (used to calculate
vehicle pose)
  double o1, o2;
  // linear velocity of wheels 1 through 4 and vehicle center of
gravity
  double v1, v2, v3, v4, vcg;
  // tangent angles of wheels 3 and 4 and vehicle center of gravity to
circles with turning radii of
  //  r3, r4, and rcg
  double a3, a4, acg = 0.0;
  // angular velocity of the STEER OR FULL wheel joints (wheels 3 and
4) or vehicle center of gravity if steering
  //  angle is zero
  double wa = 0.0;
  // angular velocity of the DRIVE wheel joints (wheels 1 and 2) or
FULL wheel joints (wheels 3 and 4) or vehicle
  //  center of gravity if steering angle is zero in the xz-plane (in
the direction of travel)
  double w = 0.0;

  Time dt;
  int count;

  tr = **this->tr;
  tw = **this->tw;
  wb = **this->wb;

  this->GetPositionCmd();

  dt = Simulator::Instance()->GetSimTime() - this->prevUpdateTime;
  this->prevUpdateTime = Simulator::Instance()->GetSimTime();

  std::vector<Wheel*>::iterator iter;

  // Calculate the current velocity at vehicle cg
```

```
  if (v >= 0)
  {
    if (v > 0.2) // we want to be able to "turn" the steering wheel
without acceleration
    {
      vcg = vcg0 + a * ( v - 0.2 ) / ( 0.5 - 0.2 ) * dt.Double();
      if (vcg > vcgMax)
        vcg = vcgMax;
    }
    else // useConstantVelocityMode controls whether the vehicle
"coasts" to a stop or maintains constant velocity
         //  when the "gas pedal" is not depressed
    {
      if (**this->useConstantVelocityMode) // maintain constant
velocity
        vcg = vcg0;
      else // "coast" to a stop
      {
        vcg = vcg0 - a * dt.Double();
        if (vcg < 0)
          vcg = 0;
      }
    }
  }
  else // v < 0
  {
    vcg = vcg0 - 3 * a * v / -0.1 * dt.Double();
    if (vcg < -vcgMax / 5)
      vcg = -vcgMax / 5;
  }

  // Calculate the distance at vehicle cg
  dcg = vcg * dt.Double();

  if (**this->useConstantSteeringAngleMode)
    sa = **this->constantSteeringAngle;

  // Calculate the angle at vehicle cg
  if (sa < -DTOR(10)) // right turn
  {
    if (sa < -sacgMax)
      sa = -sacgMax;
    if (acg > sa)
      acg = acg0 - wcgMax * dt.Double();
    if (acg <= sa)
      acg = sa;
  }
  else if (sa > DTOR(10)) // left turn
  {
    if (sa > +sacgMax)
      sa = +sacgMax;
    if (acg < sa)
      acg = acg0 + wcgMax * dt.Double();
    if (acg >= sa)
```

```
      acg = sa;
  }
  else // sa == 0
    acg = acg0;

  if ( fabs(acg) < 0.0001 ) // if fabs( acg ) < 0.0001, acg is
effectively 0, so we set rcg greater than the diameter of the earth
    rcg = 999999999;
  else if ( fabs(acg) < 0.05 ) // if fabs( acg ) < 0.05, we use small
angle approximation (alpha = tan(alpha))
    rcg = fabs( wb / acg );
  else
    rcg = fabs( wb / tan(acg) );

  count = 0;
  for (iter=this->wheels.begin(); iter!=this->wheels.end(); iter++)
  {
    if (this->enableMotors)
    {
      // Calculate the turning radius for each wheel
      if (sa < 0) // right turn
      {
        r1 = rcg + tw / 2;
        r2 = rcg - tw / 2;
        if (count == 0) // left_front_wheel_hinge
        {
          r3 = sqrt( r1 * r1 + wb * wb);
          a3 = acg * r3 / rcg;
          wa = a3;
          d3 = dcg * r3 / rcg;
          v3 = d3 / dt.Double();
          w = v3 / tr;
        }
        else if (count == 1) // right_front_wheel_hinge
        {
          r4 = sqrt( r2 * r2 + wb * wb);
          a4 = acg * r4 / rcg;
          wa = a4;
          d4 = dcg * r4 / rcg;
          v4 = d4 / dt.Double();
          w = v4 / tr;
        }
        else if (count == 2) // left_rear_wheel_hinge
        {
          wa = 0;
          d1 = dcg * r1 / rcg;
          v1 = d1 / dt.Double();
          w = v1 / tr;
        }
        else if (count == 3) // right_rear_wheel_hinge
        {
          wa = 0;
          d2 = dcg * r2 / rcg;
          v2 = d2 / dt.Double();
```

```
    w = v2 / tr;
  }
}
else if (sa > 0) // left turn
{
  r1 = rcg - tw / 2;
  r2 = rcg + tw / 2;
  if (count == 0) // left_front_wheel_hinge
  {
    r3 = sqrt( r1 * r1 + wb * wb);
    a3 = acg * r3 / rcg;
    wa = a3;
    d3 = dcg * r3 / rcg;
    v3 = d3 / dt.Double();
    w = v3 / tr;
  }
  else if (count == 1) // right_front_wheel_hinge
  {
    r4 = sqrt( r2 * r2 + wb * wb);
    a4 = acg * r4 / rcg;
    wa = a4;
    d4 = dcg * r4 / rcg;
    v4 = d4 / dt.Double();
    w = v4 / tr;
  }
  else if (count == 2) // left_rear_wheel_hinge
  {
    wa = 0;
    d1 = dcg * r1 / rcg;
    v1 = d1 / dt.Double();
    w = v1 / tr;
  }
  else if (count == 3) // right_rear_wheel_hinge
  {
    wa = 0;
    d2 = dcg * r2 / rcg;
    v2 = d2 / dt.Double();
    w = v2 / tr;
  }
}
else // sa == 0
{
  wa = 0;
  w = vcg / tr;
}

(*iter)->Update(-w, -wa, **this->updatePeriodP);

if (**this->useSwaybars)
{
  Swaybars();
}

count += 1;
```

```
    }
    else
    {
      (*iter)->Stop();
    }
  }

  vcg0 = vcg;
  acg0 = acg;

  o1 = dt.Double() * tr * this->myParent-
>GetJoint("left_rear_wheel_hinge")->GetVelocity(0);
  o2 = dt.Double() * tr * this->myParent-
>GetJoint("right_rear_wheel_hinge")->GetVelocity(0);

  // Compute odometric pose
  this->odomPose[0] += (o1 + o2) / 2 * cos( this->odomPose[2] );
  this->odomPose[1] += (o1 + o2) / 2 * sin( this->odomPose[2] );
  this->odomPose[2] += (o1 - o2) / tw;

  // Compute odometric instantaneous velocity
  this->odomVel[0] = (o1 + o2) / 2 / dt.Double();
  this->odomVel[1] = 0.0;
  this->odomVel[2] = (o1 - o2) / tw / dt.Double();

  this->PutPositionData();
}

////////////////////////////////////////////////////////////////////////
/////////
// Finalize the controller
void Steering_Position2d::FiniChild()
{
}

////////////////////////////////////////////////////////////////////////
///////
// Get commands from the external interface
void Steering_Position2d::GetPositionCmd()
{
  if (this->myIface->Lock(1))
  {
    this->v = this->myIface->data->cmdVelocity.pos.x;
    this->sa = this->myIface->data->cmdVelocity.yaw;

    this->enableMotors = this->myIface->data->cmdEnableMotors > 0;

    this-myIface->Unlock();
  }
}

////////////////////////////////////////////////////////////////////////
///////
// Update the data in the interface
```

```
void Steering_Position2d::PutPositionData()
{
  if (this->myIface->Lock(1))
  {
    // TODO: Data timestamp
    this->myIface->data->head.time = Simulator::Instance()-
>GetSimTime().Double();

    this->myIface->data->pose.pos.x = this->odomPose[0];
    this->myIface->data->pose.pos.y = this->odomPose[1];
    this->myIface->data->pose.yaw = NORMALIZE(this->odomPose[2]);

    this->myIface->data->velocity.pos.x = this->odomVel[0];
    this->myIface->data->velocity.yaw = this->odomVel[2];

    // TODO
    this->myIface->data->stall = 0;

    this-myIface->Unlock();
  }
}

////////////////////////////////////////////////////////////////////////
///////
// "Anti-sway bar" implementation
void Steering_Position2d::Swaybars()
{
  Vector3 wheelAnchor;
  Vector3 bodyAnchor;
  Vector3 axis;
  Vector3 force;
  double displacement, amt;

  std::string hinge[4];

  hinge[0] = "left_front_wheel_hinge";
  hinge[1] = "right_front_wheel_hinge";
  hinge[2] = "left_rear_wheel_hinge";
  hinge[3] = "right_rear_wheel_hinge";

  for (int i = 0; i < 4; i++)
  {
    bodyAnchor = this->myParent->GetJoint(hinge[i])->GetAnchor(0);
    wheelAnchor = this->myParent->GetJoint(hinge[i])->GetAnchor(1);
    axis = this->myParent->GetJoint(hinge[i])->GetAxis(1);

    displacement = (bodyAnchor.z - wheelAnchor.z) * axis.z;
    if (displacement > 0)
    {
      amt = displacement * **this->swayForce;

      if (amt > **this->swayForceLimit)
      {
        amt = **this->swayForceLimit;
```

```
      }
      // "downforce"
      force.Set(-axis.x * amt, -axis.y * amt, -axis.z * amt);
      this->myParent->GetJoint(hinge[i])->GetJointBody(1)-
>SetForce(force);
      // "upforce"
      force.Set(axis.x * amt, axis.y * amt, axis.z * amt);
      this->myParent->GetJoint(hinge[i^1])->GetJointBody(1)-
>SetForce(force);
    }
  }
}
```

```
/*
 *  Gazebo - Outdoor Multi-Robot Simulator
 *  Copyright (C) 2003
 *     Nate Koenig & Andrew Howard
 *
 *  This program is free software; you can redistribute it and/or
modify
 *  it under the terms of the GNU General Public License as published
by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
1307  USA
 *
 */
/*
 *  An improved wheel for a four-wheeled vehicle
 *  Author: J. C. Allen
 *  Date: 26 March 2010
 *  Based on "Wheel that can not be steered",
 *   Jordi Polo, dated 18 Dec 2007
 */

#include "Global.hh"
#include "XMLConfig.hh"
#include "Model.hh"
#include "Body.hh"
#include "Joint.hh"
#include "World.hh"
#include "gazebo.h"
#include "GazeboError.hh"
#include "ControllerFactory.hh"
#include "Steering_Position2d.hh"
#include "Wheel.hh"
#include <string>

using namespace gazebo;

enum {DRIVE, STEER, FULL};

//////////////////////////////////////////////////////////////////////
/////////
// Constructor
Wheel::Wheel()
{
}
```

```
//////////////////////////////////////////////////////////////////////
/////////
// Destructor
Wheel::~Wheel()
{
  delete this->joint;
}

//////////////////////////////////////////////////////////////////////
/////////
// Connects the wheel to a given Joint
void Wheel::Connect(Joint *joint, int type)
{
  this->joint = joint;
  this->type = type;

  if (!this->joint)
  {
    std::ostringstream stream;
    stream << "The controller couldn't get the joint " <<this->joint-
>GetName();
    gzthrow(stream.str());
  }

  // avoid an initial impulse to the joints that would make the vehicle
flip
  this->joint->SetAttribute(Joint::FUDGE_FACTOR, 0, 0.1);
}

//////////////////////////////////////////////////////////////////////
/////////
// Stops the wheel
void Wheel::Stop()
{
  switch (this->type)
  {
    case DRIVE:
      this->joint->SetVelocity(0, 0);
      this->joint->SetMaxForce(0, 0);
      break;
    case STEER:
      this->joint->SetVelocity(0, 0);
      this->joint->SetMaxForce(0, 0);
      this->joint->SetVelocity(1, 0);
      this->joint->SetMaxForce(1, 0);
      break;
    default:
      this->joint->SetVelocity(0, 0);
      this->joint->SetMaxForce(0, 0);
      this->joint->SetVelocity(1, 0);
      this->joint->SetMaxForce(1, 0);
  }
}
```

```
////////////////////////////////////////////////////////////////////////
/////////
// Set the torque
void Wheel::SetTorque(double newTorque)
{
  this->torque = newTorque;

  switch (this->type)
  {
    case DRIVE:
      this->joint->SetMaxForce(0, this->torque);
      break;
    case STEER:
      this->joint->SetMaxForce(1, this->torque);
      break;
    default:
      this->joint->SetMaxForce(1, this->torque);
  }
}

////////////////////////////////////////////////////////////////////////
/////////
// Get the torque
double Wheel::GetTorque()
{
  return this->torque;
}

////////////////////////////////////////////////////////////////////////
/////////
// Set the steering torque
void Wheel::SetSteerTorque(double newTorque)
{
  this->steerTorque = newTorque;

  switch (this->type)
  {
    case DRIVE: // drive wheels have no steering axis
      break;
    case STEER:
      this->joint->SetMaxForce(0, this->steerTorque);
      break;
    default:
      this->joint->SetMaxForce(0, this->steerTorque);
  }
}

////////////////////////////////////////////////////////////////////////
/////////
// Get the steering torque
double Wheel::GetSteerTorque()
{
  return this->steerTorque;
```

```
}

//////////////////////////////////////////////////////////////////////////
/////////
// Update the wheel
void Wheel::Update(double speed, double steer, double rate)
{
  switch (this->type)
  {
    case DRIVE:
      this->joint->SetVelocity(0, speed);
      break;
    case STEER:
      this->joint->SetVelocity(0, rate * (steer - this->joint-
>GetAngle(0).GetAsRadian()));
      break;
    default:
      this->joint->SetVelocity(0, rate * (steer - this->joint-
>GetAngle(0).GetAsRadian()));
      this->joint->SetVelocity(1, speed);
  }
}
```