```
*  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
*  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
*  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
*  POSSIBILITY OF SUCH DAMAGE.
*
* Author: Eitan Marder-Eppstein
*********************************************************************/
#include <navfn/navfn_ros.h>
#include <pluginlib/class_list_macros.h>
#include <costmap_2d/cost_values.h>
#include <costmap_2d/costmap_2d.h>
#include <sensor_msgs/point_cloud2_iterator.h>

//register this planner as a BaseGlobalPlanner plugin
PLUGINLIB_EXPORT_CLASS(navfn::NavfnROS, nav_core::BaseGlobalPlanner)

namespace navfn {

  NavfnROS::NavfnROS()
    : costmap_(NULL),  planner_(), initialized_(false), allow_unknown_(true) {}

  NavfnROS::NavfnROS(std::string name, costmap_2d::Costmap2DROS* costmap_ros)
    : costmap_(NULL),  planner_(), initialized_(false), allow_unknown_(true) {
      //initialize the planner
      initialize(name, costmap_ros);
  }

  NavfnROS::NavfnROS(std::string name, costmap_2d::Costmap2D* costmap, std::string global_
    : costmap_(NULL),  planner_(), initialized_(false), allow_unknown_(true) {
      //initialize the planner
      initialize(name, costmap, global_frame);
  }

  void NavfnROS::initialize(std::string name, costmap_2d::Costmap2D* costmap, std::string 
    if(!initialized_){
      costmap_ = costmap;
      global_frame_ = global_frame;
      planner_ = boost::shared_ptr<NavFn>(new NavFn(costmap_->getSizeInCellsX(), costmap_-

      ros::NodeHandle private_nh("~/" + name);

      plan_pub_ = private_nh.advertise<nav_msgs::Path>("plan", 1);

      private_nh.param("visualize_potential", visualize_potential_, false);

      //if we're going to visualize the potential array we need to advertise
      if(visualize_potential_)
        potarr_pub_ = private_nh.advertise<sensor_msgs::PointCloud2>("potential", 1);

      private_nh.param("allow_unknown", allow_unknown_, true);
      private_nh.param("planner_window_x", planner_window_x_, 0.0);
      private_nh.param("planner_window_y", planner_window_y_, 0.0);
      private_nh.param("default_tolerance", default_tolerance_, 0.0);

      make_plan_srv_ =  private_nh.advertiseService("make_plan", &NavfnROS::makePlanServic

      initialized_ = true;
    }
    else
      ROS_WARN("This planner has already been initialized, you can't call it twice, doing
  }

  void NavfnROS::initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros){
    initialize(name, costmap_ros->getCostmap(), costmap_ros->getGlobalFrameID());
  }

  bool NavfnROS::validPointPotential(const geometry_msgs::Point& world_point){
    return validPointPotential(world_point, default_tolerance_);
  }

  bool NavfnROS::validPointPotential(const geometry_msgs::Point& world_point, double toler
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return false;
    }

    double resolution = costmap_->getResolution();
    geometry_msgs::Point p;
    p = world_point;

    p.y = world_point.y - tolerance;

    while(p.y <= world_point.y + tolerance){
      p.x = world_point.x - tolerance;
      while(p.x <= world_point.x + tolerance){
        double potential = getPointPotential(p);
        if(potential < POT_HIGH){
          return true;
        }
        p.x += resolution;
      }
      p.y += resolution;
    }

    return false;
  }

  double NavfnROS::getPointPotential(const geometry_msgs::Point& world_point){
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return -1.0;
    }

    unsigned int mx, my;
    if(!costmap_->worldToMap(world_point.x, world_point.y, mx, my))
      return DBL_MAX;

    unsigned int index = my * planner_->nx + mx;
    return planner_->potarr[index];
  }

  bool NavfnROS::computePotential(const geometry_msgs::Point& world_point){
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return false;
    }

    //make sure to resize the underlying array that Navfn uses
    planner_->setNavArr(costmap_->getSizeInCellsX(), costmap_->getSizeInCellsY());
    planner_->setCostmap(costmap_->getCharMap(), true, allow_unknown_);

    unsigned int mx, my;
    if(!costmap_->worldToMap(world_point.x, world_point.y, mx, my))
      return false;

    int map_start[2];
    map_start[0] = 0;
    map_start[1] = 0;

    int map_goal[2];
    map_goal[0] = mx;
    map_goal[1] = my;

    planner_->setStart(map_start);
    planner_->setGoal(map_goal);

    return planner_->calcNavFnDijkstra();
  }

  void NavfnROS::clearRobotCell(const geometry_msgs::PoseStamped& global_pose, unsigned ir
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return;
    }

    //set the associated costs in the cost map to be free
    costmap_->setCost(mx, my, costmap_2d::FREE_SPACE);
  }

  bool NavfnROS::makePlanService(nav_msgs::GetPlan::Request& req, nav_msgs::GetPlan::Respo
    makePlan(req.start, req.goal, resp.plan.poses);

    resp.plan.header.stamp = ros::Time::now();
    resp.plan.header.frame_id = global_frame_;

    return true;
  }

  void NavfnROS::mapToWorld(double mx, double my, double& wx, double& wy) {
    wx = costmap_->getOriginX() + mx * costmap_->getResolution();
    wy = costmap_->getOriginY() + my * costmap_->getResolution();
  }

  bool NavfnROS::makePlan(const geometry_msgs::PoseStamped& start,
      const geometry_msgs::PoseStamped& goal, std::vector<geometry_msgs::PoseStamped>& pla
    return makePlan(start, goal, default_tolerance_, plan);
  }

  bool NavfnROS::makePlan(const geometry_msgs::PoseStamped& start,
      const geometry_msgs::PoseStamped& goal, double tolerance, std::vector<geometry_msgs:
    boost::mutex::scoped_lock lock(mutex_);
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return false;
    }

    //clear the plan, just in case
    plan.clear();

    ros::NodeHandle n;

    //until tf can handle transforming things that are way in the past... we'll require th
    if(goal.header.frame_id != global_frame_){
      ROS_ERROR("The goal pose passed to this planner must be in the %s frame.  It is inst
                global_frame_.c_str(), goal.header.frame_id.c_str());
      return false;
    }

    if(start.header.frame_id != global_frame_){
      ROS_ERROR("The start pose passed to this planner must be in the %s frame.  It is ins
                global_frame_.c_str(), start.header.frame_id.c_str());
      return false;
    }

    double wx = start.pose.position.x;
    double wy = start.pose.position.y;

    unsigned int mx, my;
    if(!costmap_->worldToMap(wx, wy, mx, my)){
      ROS_WARN("The robot's start position is off the global costmap. Planning will always
      return false;
    }

    //clear the starting cell within the costmap because we know it can't be an obstacle
    clearRobotCell(start, mx, my);

    //make sure to resize the underlying array that Navfn uses
    planner_->setNavArr(costmap_->getSizeInCellsX(), costmap_->getSizeInCellsY());
    planner_->setCostmap(costmap_->getCharMap(), true, allow_unknown_);

    int map_start[2];
    map_start[0] = mx;
    map_start[1] = my;

    wx = goal.pose.position.x;
    wy = goal.pose.position.y;

    if(!costmap_->worldToMap(wx, wy, mx, my)){
      if(tolerance <= 0.0){
        ROS_WARN_THROTTLE(1.0, "The goal sent to the navfn planner is off the global costm
        return false;
      }
      mx = 0;
      my = 0;
    }

    int map_goal[2];
    map_goal[0] = mx;
    map_goal[1] = my;

    planner_->setStart(map_goal);
    planner_->setGoal(map_start);

    //bool success = planner_->calcNavFnAstar();
    planner_->calcNavFnDijkstra(true);

    double resolution = costmap_->getResolution();
    geometry_msgs::PoseStamped p, best_pose;
    p = goal;

    bool found_legal = false;
    double best_sdist = DBL_MAX;

    p.pose.position.y = goal.pose.position.y - tolerance;

    while(p.pose.position.y <= goal.pose.position.y + tolerance){
      p.pose.position.x = goal.pose.position.x - tolerance;
      while(p.pose.position.x <= goal.pose.position.x + tolerance){
        double potential = getPointPotential(p.pose.position);
        double sdist = sq_distance(p, goal);
        if(potential < POT_HIGH && sdist < best_sdist){
          best_sdist = sdist;
          best_pose = p;
          found_legal = true;
        }
        p.pose.position.x += resolution;
      }
      p.pose.position.y += resolution;
    }

    if(found_legal){
      //extract the plan
      if(getPlanFromPotential(best_pose, plan)){
        //make sure the goal we push on has the same timestamp as the rest of the plan
        geometry_msgs::PoseStamped goal_copy = best_pose;
        goal_copy.header.stamp = ros::Time::now();
        plan.push_back(goal_copy);
      }
      else{
        ROS_ERROR("Failed to get a plan from potential when a legal potential was found. T
      }
    }

    if (visualize_potential_)
    {
      // Publish the potentials as a PointCloud2
      sensor_msgs::PointCloud2 cloud;
      cloud.width = 0;
      cloud.height = 0;
      cloud.header.stamp = ros::Time::now();
      cloud.header.frame_id = global_frame_;
      sensor_msgs::PointCloud2Modifier cloud_mod(cloud);
      cloud_mod.setPointCloud2Fields(4, "x", 1, sensor_msgs::PointField::FLOAT32,
                                        "y", 1, sensor_msgs::PointField::FLOAT32,
                                        "z", 1, sensor_msgs::PointField::FLOAT32,
                                        "pot", 1, sensor_msgs::PointField::FLOAT32);
      cloud_mod.resize(planner_->ny * planner_->nx);
      sensor_msgs::PointCloud2Iterator<float> iter_x(cloud, "x");

      PotarrPoint pt;
      float *pp = planner_->potarr;
      double pot_x, pot_y;
      for (unsigned int i = 0; i < (unsigned int)planner_->ny*planner_->nx ; i++)
      {
        if (pp[i] < 10e7)
        {
          mapToWorld(i%planner_->nx, i/planner_->nx, pot_x, pot_y);
          iter_x[0] = pot_x;
          iter_x[1] = pot_y;
          iter_x[2] = pp[i]/pp[planner_->start[1]*planner_->nx + planner_->start[0]]*20;
          iter_x[3] = pp[i];
          ++iter_x;
        }
      }
      potarr_pub_.publish(cloud);
    }

    //publish the plan for visualization purposes
    publishPlan(plan, 0.0, 1.0, 0.0, 0.0);

    return !plan.empty();
  }

  void NavfnROS::publishPlan(const std::vector<geometry_msgs::PoseStamped>& path, double r
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return;
    }

    //create a message for the plan
    nav_msgs::Path gui_path;
    gui_path.poses.resize(path.size());

    if(path.empty()) {
      //still set a valid frame so visualization won't hit transform issues
        gui_path.header.frame_id = global_frame_;
      gui_path.header.stamp = ros::Time::now();
    } else {
      gui_path.header.frame_id = path[0].header.frame_id;
      gui_path.header.stamp = path[0].header.stamp;
    }

    // Extract the plan in world co-ordinates, we assume the path is all in the same frame
    for(unsigned int i=0; i < path.size(); i++){
      gui_path.poses[i] = path[i];
    }

    plan_pub_.publish(gui_path);
  }

  bool NavfnROS::getPlanFromPotential(const geometry_msgs::PoseStamped& goal, std::vector<
    if(!initialized_){
      ROS_ERROR("This planner has not been initialized yet, but it is being used, please c
      return false;
    }

    //clear the plan, just in case
    plan.clear();

    //until tf can handle transforming things that are way in the past... we'll require th
    if(goal.header.frame_id != global_frame_){
      ROS_ERROR("The goal pose passed to this planner must be in the %s frame.  It is inst
                global_frame_.c_str(), goal.header.frame_id.c_str());
      return false;
    }

    double wx = goal.pose.position.x;
    double wy = goal.pose.position.y;

    //the potential has already been computed, so we won't update our copy of the costmap
    unsigned int mx, my;
    if(!costmap_->worldToMap(wx, wy, mx, my)){
      ROS_WARN_THROTTLE(1.0, "The goal sent to the navfn planner is off the global costmap
      return false;
    }

    int map_goal[2];
    map_goal[0] = mx;
    map_goal[1] = my;

    planner_->setStart(map_goal);

    planner_->calcPath(costmap_->getSizeInCellsX() * 4);

    //extract the plan
    float *x = planner_->getPathX();
    float *y = planner_->getPathY();
    int len = planner_->getPathLen();
    ros::Time plan_time = ros::Time::now();

    for(int i = len - 1; i >= 0; --i){
      //convert the plan to world coordinates
      double world_x, world_y;
      mapToWorld(x[i], y[i], world_x, world_y);

      geometry_msgs::PoseStamped pose;
      pose.header.stamp = plan_time;
      pose.header.frame_id = global_frame_;
      pose.pose.position.x = world_x;
      pose.pose.position.y = world_y;
      pose.pose.position.z = 0.0;
      pose.pose.orientation.x = 0.0;
```

```
        pose.pose.orientation.y = 0.0;
        pose.pose.orientation.z = 0.0;
        pose.pose.orientation.w = 1.0;
        plan.push_back(pose);
      }

      //publish the plan for visualization purposes
      publishPlan(plan, 0.0, 1.0, 0.0, 0.0);
      return !plan.empty();
    }
  };
```