### **base\_local\_planner**

# **构造函数与初始化**

#include <tf/transform\_listener.h>

#include <costmap\_2d/costmap\_2d\_ros.h>

#include <base\_local\_planner/trajectory\_planner\_ros.h>

.......

tf::TransformListener tf(ros::Duration(10));

costmap\_2d::Costmap2DROS costmap("my\_costmap", tf);

base\_local\_planner::TrajectoryPlannerROS tp;

首先生成在trajectory\_planner\_ros.cpp

中的base\_local\_planner::TrajectoryPlannerROS

实例tp，base\_local\_planner::TrajectoryPlannerROS

是对base\_local\_planner::TrajectoryPlanner的ROS包装。

在生成实例的过程中声明一些变量，最重要的是参数如下：

WorldModel\* world\_model\_; //控制器将会使用的世界模型

TrajectoryPlanner\* tc\_; //轨迹控制器本体，同时也声明了许多许多个将要用的参数，简直没法展开说，要用的时候再说吧

轨迹控制器本体在trajectory\_planner.cpp中，这里真正的涉及到了轨迹的生成与计算，以后我们会再见到它。

tp.initialize("my\_trajectory\_planner", &tf, &costmap);

我们可以对比move\_base.cpp129-132行：

tc\_ = blp\_loader\_.createInstance(local\_planner);

ROS\_INFO("Created local\_planner %s", local\_planner.c\_str());

tc\_->initialize(blp\_loader\_.getName(local\_planner), &tf\_, controller\_costmap\_ros\_);

//多了这一句，在nav\_core中定义这个虚函数，然而base\_local\_planner并没有重载，所以这里并没有意义。

tc\_->setGlobalCostmap(planner\_costmap\_ros\_);

从initialize开始真正的base\_local\_planner，即trajectory\_planner\_ros.cpp第84-245行。首先将会是很多很多的参数读取以及纠正不科学的参数。在读取这些参数以后，我们终于开始正式生成轨迹和评价。

在第230行正式对base\_local\_planner进行声明：

tc\_ = new TrajectoryPlanner(\*world\_model\_, \*costmap\_, footprint\_spec\_,

acc\_lim\_x\_, acc\_lim\_y\_, acc\_lim\_theta\_, sim\_time, sim\_granularity, vx\_samples, vtheta\_samples, pdist\_scale,

gdist\_scale, occdist\_scale, heading\_lookahead, oscillation\_reset\_dist, escape\_reset\_dist, escape\_reset\_theta, holonomic\_robot,

max\_vel\_x, min\_vel\_x, max\_vel\_th\_, min\_vel\_th\_, min\_in\_place\_vel\_th\_, backup\_vel,

dwa, heading\_scoring, heading\_scoring\_timestep, meter\_scoring, simple\_attractor, y\_vels, stop\_time\_buffer, sim\_period\_, angular\_sim\_granularity);

其构造函数在trajectory\_planner\_ros.cpp第141-172行，将许多flag定义为false，之后是187行：

costmap\_2d::calculateMinAndMaxDistances(footprint\_spec\_, inscribed\_radius\_, circumscribed\_radius\_);

计算了costmap中需要使用的内接圆与外切圆半径。至此，构造函数与初始化结束。

# **速度与路径计算：setPlan**

setPlan在trajectory\_planner\_ros.cpp中的第372-第525行，在这里将会计算机器人实际运行的速度，发布局部路径规划，可以说是base\_local\_planner的核心。

首先将机器人的姿态以及全局路径规划转换到global\_frame\_上。

if (!costmap\_ros\_->getRobotPose(global\_pose)) {

return false;

}

std::vector<geometry\_msgs::PoseStamped> transformed\_plan;

//get the global plan in our frame

if (!transformGlobalPlan(\*tf\_, global\_plan\_, global\_pose, \*costmap\_, global\_frame\_, transformed\_plan)) {

ROS\_WARN("Could not transform the global plan to the frame of the controller");

return false;

}

第395-419行，得到机器人当前速度以及终点的(x,y,θ \thetaθ)。第422-473行，查看机器人是否到达终点，如果到达了，就让机器人停下来。第475-478行：

//将全局路径拷贝进来，并认为全军路径的最后一个点就是终点。

tc\_->updatePlan(transformed\_plan);

//计算应该跟随什么轨迹，给定当前机器人位置和朝向，计算机器人应该跟随的好轨迹。

Trajectory path = tc\_->findBestPath(global\_pose, robot\_vel, drive\_cmds);

findBestPath在trajectory\_planner.cpp中第906至第977行，终于我们开始正式计算轨迹以及速度分量，将函数摘抄如下：

//计算应该跟随什么轨迹，给定当前机器人位置和朝向，计算机器人应该跟随的好轨迹。

Trajectory TrajectoryPlanner::findBestPath(tf::Stamped<tf::Pose> global\_pose, tf::Stamped<tf::Pose> global\_vel,

tf::Stamped<tf::Pose>& drive\_velocities){

//将当前机器人位置和方向转变成float形式的向量

Eigen::Vector3f pos(global\_pose.getOrigin().getX(), global\_pose.getOrigin().getY(), tf::getYaw(global\_pose.getRotation()));

Eigen::Vector3f vel(global\_vel.getOrigin().getX(), global\_vel.getOrigin().getY(), tf::getYaw(global\_vel.getRotation()));

//重置地图

path\_map\_.resetPathDist();

goal\_map\_.resetPathDist();

//暂时移走机器人footprint上的障碍物

std::vector<base\_local\_planner::Position2DInt> footprint\_list =

footprint\_helper\_.getFootprintCells(

pos,

footprint\_spec\_,

costmap\_,

true);

//将初始footprint上的所有cell标记

for (unsigned int i = 0; i < footprint\_list.size(); ++i) {

path\_map\_(footprint\_list[i].x, footprint\_list[i].y).within\_robot = true;

}

//确保我们根据全局计划更新map并且计算代价

path\_map\_.setTargetCells(costmap\_, global\_plan\_);

goal\_map\_.setLocalGoal(costmap\_, global\_plan\_);

ROS\_DEBUG("Path/Goal distance computed");

//找到代价最低的轨迹。输入分别是目前机器人位置，速度以及加速度限制，生成诸多可能轨迹，选取其中打分最高的。这里也就是最关键的一步。

Trajectory best = createTrajectories(pos[0], pos[1], pos[2],

vel[0], vel[1], vel[2],

acc\_lim\_x\_, acc\_lim\_y\_, acc\_lim\_theta\_);

ROS\_DEBUG("Trajectories created");

/\*

//If we want to print a ppm file to draw goal dist

char buf[4096];

sprintf(buf, "base\_local\_planner.ppm");

FILE \*fp = fopen(buf, "w");

if(fp){

fprintf(fp, "P3\n");

fprintf(fp, "%d %d\n", map\_.size\_x\_, map\_.size\_y\_);

fprintf(fp, "255\n");

for(int j = map\_.size\_y\_ - 1; j >= 0; --j){

for(unsigned int i = 0; i < map\_.size\_x\_; ++i){

int g\_dist = 255 - int(map\_(i, j).goal\_dist);

int p\_dist = 255 - int(map\_(i, j).path\_dist);

if(g\_dist < 0)

g\_dist = 0;

if(p\_dist < 0)

p\_dist = 0;

fprintf(fp, "%d 0 %d ", g\_dist, 0);

}

fprintf(fp, "\n");

}

fclose(fp);

}

\*/

// 如果最后打分如果小于0，说明所有的路径都不可用

if(best.cost\_ < 0){

drive\_velocities.setIdentity();

}

else{

tf::Vector3 start(best.xv\_, best.yv\_, 0);

drive\_velocities.setOrigin(start);

tf::Matrix3x3 matrix;

matrix.setRotation(tf::createQuaternionFromYaw(best.thetav\_));

drive\_velocities.setBasis(matrix);

}

return best;

}

## **轨迹返回：createTrajectories**

createTrajectories是整个dwa\_local\_controller以及base\_local\_controller的关键，它在trajectory\_planner\_ros第535-第902行。

首先第539-第560行，读取数据，设定这一次仿真中可以到达的最大线速度和角速度。

第563-第569行，首先根据vx速度以及θ \thetaθ的采样数，分别计算出每一次线速度和角速度的变化值dvx和dvtheta。并把最小的vx、vy、vθ \thetaθ作为初始采样。

第584-第645行，如果机器人没有在逃离，首先假设机器人会向前，那么按照按照顺序对所有采样到的vx，对每一个vx都去尝试vθ \thetaθ使用generateTrajectory生成轨迹并打分，保留分数最高的轨迹。如果是柔性机器人那么还有vy方向的采样和打分。第655-第697行，机器人不向前只旋转的情况下，对这些轨迹进行打分。保留下分数最高的轨迹。

第700-第846行，检查最优轨迹的分数是否大于0，也就是正常。如果正常的话，那么为了抑制震荡影响，当机器人在某方向移动时，对下一个周期的与其相反方向标记为无效，直到机器人从标记震荡的位置处离开一定距离，（声明与定义在trajectory\_planner.h中第271-第275行），返回最佳轨迹。

最后，当最优轨迹不可用时，就让机器人缓慢退后。

## **轨迹生成与打分：generateTrajectory**

该函数定义在trajectory\_planner.h中第247-249行，连带说明如下：

//double vx\_samp, double vy\_samp, double vtheta\_samp

//分别为这次采样中使用的vx, yv, vtheta，生成的轨迹以引用的形式传回。

void generateTrajectory(double x, double y, double theta, double vx, double vy,

double vtheta, double vx\_samp, double vy\_samp, double vtheta\_samp, double acc\_x, double acc\_y,

double acc\_theta, double impossible\_cost, Trajectory& traj);

这一部分可以说是base\_local\_planner的核心，所以我全部拷贝过来，慢慢做解释：

void TrajectoryPlanner::generateTrajectory(

double x, double y, double theta,

double vx, double vy, double vtheta,

double vx\_samp, double vy\_samp, double vtheta\_samp,

double acc\_x, double acc\_y, double acc\_theta,

double impossible\_cost,

Trajectory& traj) {

// 确保运行一半的时候参数不会改变

boost::mutex::scoped\_lock l(configuration\_mutex\_);

//记录初始时刻的x、y、theta、vx、vy、vtheta

double x\_i = x;

double y\_i = y;

double theta\_i = theta;

double vx\_i, vy\_i, vtheta\_i;

vx\_i = vx;

vy\_i = vy;

vtheta\_i = vtheta;

//计算速度的模

double vmag = hypot(vx\_samp, vy\_samp);

//计算仿真的步数

int num\_steps;

if(!heading\_scoring\_) {

num\_steps = int(max((vmag \* sim\_time\_) / sim\_granularity\_, fabs(vtheta\_samp) / angular\_sim\_granularity\_) + 0.5);

} else {

num\_steps = int(sim\_time\_ / sim\_granularity\_ + 0.5);

}

//我们希望至少有一步，即使一步都没有，我们也希望能为当前位置打分

if(num\_steps == 0) {

num\_steps = 1;

}

double dt = sim\_time\_ / num\_steps;

double time = 0.0;

//创造一个潜在的轨迹

traj.resetPoints();

traj.xv\_ = vx\_samp;

traj.yv\_ = vy\_samp;

traj.thetav\_ = vtheta\_samp;

traj.cost\_ = -1.0;

//初始化这个轨迹的cost

double path\_dist = 0.0;

double goal\_dist = 0.0;

double occ\_cost = 0.0;

double heading\_diff = 0.0;

for(int i = 0; i < num\_steps; ++i){

unsigned int cell\_x, cell\_y;

//不希望路径跑出已知地图

if(!costmap\_.worldToMap(x\_i, y\_i, cell\_x, cell\_y)){

traj.cost\_ = -1.0;

return;

}

//检查当前点路径的合法性

double footprint\_cost = footprintCost(x\_i, y\_i, theta\_i);

//如果遇到了障碍物，那么会返回一个负数

if(footprint\_cost < 0){

traj.cost\_ = -1.0;

return;

//更新occ\_cost

occ\_cost = std::max(std::max(occ\_cost, footprint\_cost), double(costmap\_.getCost(cell\_x, cell\_y)));

//如果只是想简简单单的跟随终点的话，这里调整为true即可

if (simple\_attractor\_) {

goal\_dist = (x\_i - global\_plan\_[global\_plan\_.size() -1].pose.position.x) \*

(x\_i - global\_plan\_[global\_plan\_.size() -1].pose.position.x) +

(y\_i - global\_plan\_[global\_plan\_.size() -1].pose.position.y) \*

(y\_i - global\_plan\_[global\_plan\_.size() -1].pose.position.y);

} else {

bool update\_path\_and\_goal\_distances = true;

// 是否为朝向打分？

if (heading\_scoring\_) {

if (time >= heading\_scoring\_timestep\_ && time < heading\_scoring\_timestep\_ + dt) {

heading\_diff = headingDiff(cell\_x, cell\_y, x\_i, y\_i, theta\_i);

} else {

update\_path\_and\_goal\_distances = false;

}

}

if (update\_path\_and\_goal\_distances) {

//更新路径与目标的距离

path\_dist = path\_map\_(cell\_x, cell\_y).target\_dist;

goal\_dist = goal\_map\_(cell\_x, cell\_y).target\_dist;

//如果一个路径上的点没法明确到达在终点，他就是无效的

if(impossible\_cost <= goal\_dist || impossible\_cost <= path\_dist){

// ROS\_DEBUG("No path to goal with goal distance = %f, path\_distance = %f and max cost = %f",

// goal\_dist, path\_dist, impossible\_cost);

traj.cost\_ = -2.0;

return;

}

}

}

//这个点有效，加入轨迹

traj.addPoint(x\_i, y\_i, theta\_i);

//计算速度

vx\_i = computeNewVelocity(vx\_samp, vx\_i, acc\_x, dt);

vy\_i = computeNewVelocity(vy\_samp, vy\_i, acc\_y, dt);

vtheta\_i = computeNewVelocity(vtheta\_samp, vtheta\_i, acc\_theta, dt);

//计算位置

x\_i = computeNewXPosition(x\_i, vx\_i, vy\_i, theta\_i, dt);

y\_i = computeNewYPosition(y\_i, vx\_i, vy\_i, theta\_i, dt);

theta\_i = computeNewThetaPosition(theta\_i, vtheta\_i, dt);

//增加时间

time += dt;

}

double cost = -1.0;

//更新cost

if (!heading\_scoring\_) {

cost = pdist\_scale\_ \* path\_dist + goal\_dist \* gdist\_scale\_ + occdist\_scale\_ \* occ\_cost;

} else {

cost = occdist\_scale\_ \* occ\_cost + pdist\_scale\_ \* path\_dist + 0.3 \* heading\_diff + goal\_dist \* gdist\_scale\_;

}

traj.cost\_ = cost;

}

流程图：

