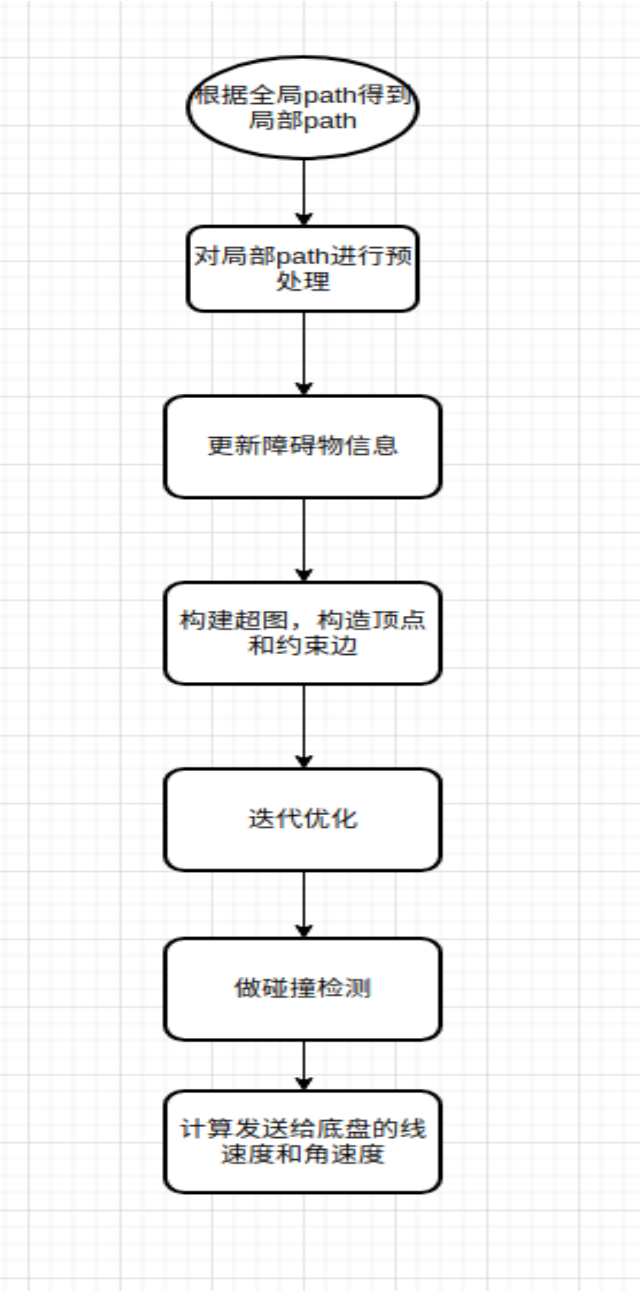
Teb 算法研究

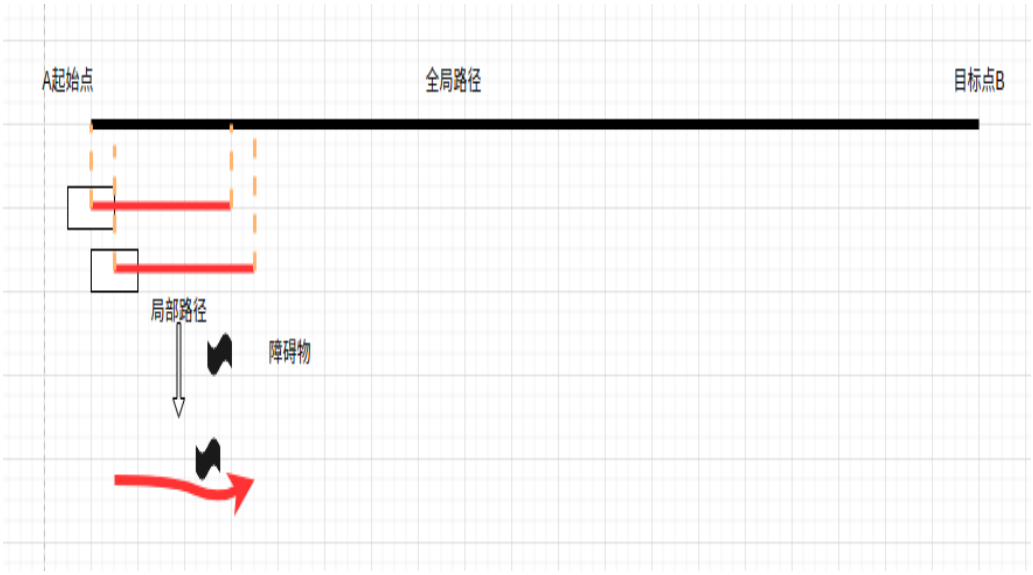
1.理论部分概述

teb 算法流程图：



teb 算法

teb 算法上游接口图示：



teb 基础理论：

time eletic band(时间橡皮筋)：起始点、目标点,teb 的初始轨迹是全局规划的 path 上截取的一段距离，将这段距离内的点作 teb 规划的初始的一些点位信息，然后这些点之间的时间间隔是相同的，将这段时间总的消耗作为一个代价值考虑到规划的路径中去，针对每个不同的优化项，每次改动的点数不会太多，所以较为灵活，同时这也使得这种算法能够正在扩展目标函数和约束。

teb 主要是考虑了时间因素和动态约束，将有限的机器人的速度和加速度描述成一个多目标优化框架问题。利用 g2o 的图优化方法来求解。

teb 中考虑的目标函数主要分为两大类：

约束：速度(只和临近的两三个位姿相关)、加速度限制（同上）

目标：最快、避障（只会考虑距离车附近几个点的信息）、以及距离最短

-----最快和距离最短是全局考虑的因素所以需要动态调节所有的参数

----遵守机器人的非完整约束涉及到两个相邻的位姿，俩位姿位于一个常曲率的普通圆弧上

非完整约束运动学

teb 同时考虑原始路径的 via-point 的到达和避免静态或者动态的障碍。这俩目标函数 相似，不同之处在于 via-point 吸引橡皮筋，而障碍物排斥它

teb 算法的目标函数可以表示为：

$$f(B) = \sum_{k} \gamma_k f_k(B) \qquad (4)$$

$$B^* = \underset{B}{\operatorname{argmin}} f(B) \qquad (5)$$

平常的图优化的表示形式是：

$$\begin{aligned}\mathbf{F}(\mathbf{x}) &= \sum_{k=\langle i,j\rangle\in\mathcal{C}} \underbrace{\mathbf{e}_k(\mathbf{x}_i,\mathbf{x}_j,\mathbf{z}_{ij})^T\boldsymbol{\Omega}_k\mathbf{e}_k(\mathbf{x}_i,\mathbf{x}_j,\mathbf{z}_{ij})}_{\mathbf{F}_k} \\ \mathbf{x}^* &= \min_x \mathbf{F}(\mathbf{x})\end{aligned}$$

图优化中，顶点是待估计的参数，观测数据构成了边。Teb 的待估参数就是 B，也就是位姿和时间差的组合，边表示目标函数 fk,并连接多个顶点。

这个图的顶点 vertexs 是橡皮筋的状态(机器人的姿态和时间)，图的边是我们自己定义的优化目标函数。

g2o 框架在批处理模式下优化 teb,每次迭代都会生成一个新的图。在一个控制周期内对轨迹修改进行多次迭代.g2o 框架提供了两种求解器：CHOLMOD 和 CSparse。默认使用的是 CSparse

对优化的 teb 进行验证，如果不符合条件，则停止或者重新调用运动规划器

其运行时间依赖于 teb 的维数，teb 的维数随位姿数线性增长。

二、teb 算法具体步骤：

1.给出机器人的姿态信息，其中一个点的信息 Q(i)={xi,yi,の i}

Q(i):表示该点的姿态信息

x(i),y(i):具体的位置信息

の(i):方位角信息

$$f(B) = \sum_k \gamma_k f_k(B) \quad (4)$$

$$B^* = \underset{B}{\operatorname{argmin}} f(B) \quad (5)$$

2.在点和点之间增加时间间隔， 产生 n-1 个时间差值 Ti,这个 Ti 的差值都是不同的：

τ= Ti (i = 0,1...,n-1)

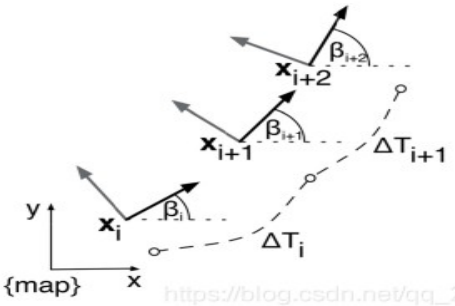
3.所以就可以用一个变量 B 来表示这个点的姿态信息和时间信息和：

B = (Q,τ)

4.以 B 作为因变量，构造一个加权多目标函数 f(B)来最取最优的 B*:

f(B) = r1* B1 + r2 *B2 + ...

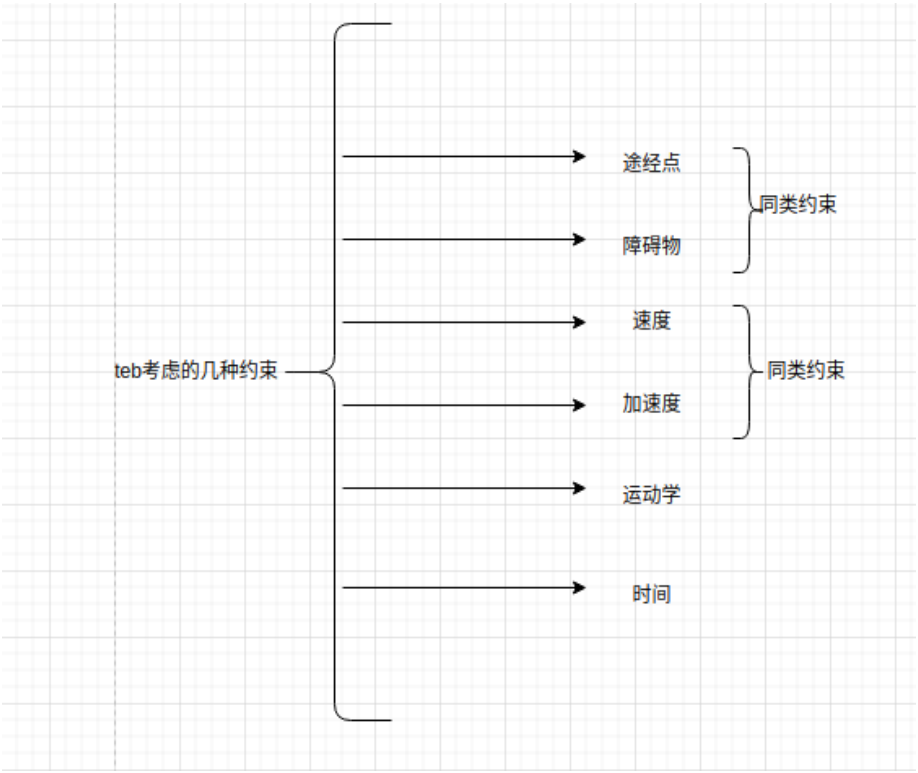
5. 图示：



6.目标函数的大部分分量相对于 B 都是局部的，只依赖几个连续的点的信息，导致稀疏矩阵。

teb 算法目标函数的分类：

目标函数图示：



1.waypoints 和障碍物的吸引与排斥，设置了一个惩罚函数来实现：

$$e_{\Gamma}(x, x_r, \epsilon, S, n) \simeq \begin{cases} \left(\frac{x-(x_r-\epsilon)}{S}\right)^n & \text{if } x > x_r - \epsilon \\ 0 & \text{otherwise} \end{cases}$$

x:因素

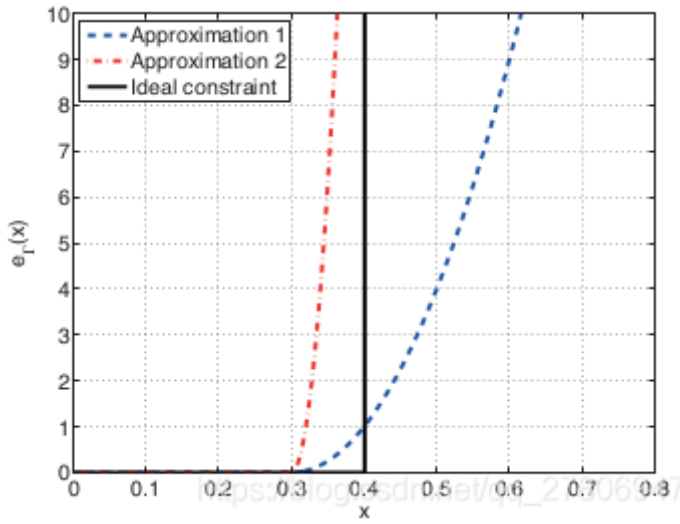
xr:阈值

S:缩放因子

n:多项式阶数

ϵ: 表示近似值附近的一个小位移

其中，xr 为极限值，S、n 和 ϵ 影响近似的准确度。举例来说的话就是：



在“Approximation 1”中，*S*=0.1、*n*=2 和$\varepsilon = 0.1$；在“Approximation 2”中（更强的近似），*S*=0.05、*n*=2 和$\varepsilon = 0.1$。上述示例显示了当约束$x_r = 0.4$时的近似值。

从图中可以看到，随着 x 的不断变化，其 e 值会产生不同的惩罚值。

$$f_{path} = e_\Gamma(d_{min,j}, r_{p_{max}}, \epsilon, S, n)$$

对于 waypoints 来说就是不要让这个值太大。

$$f_{ob} = e_\Gamma(-d_{min,j}, -r_{o_{min}}, \epsilon, S, n)$$

之所以这里是负值，是因为它要保证距离障碍物一定的距离。

速度和加速度：

线速度：

$$v_i \simeq \frac{1}{\Delta T_i} \| \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{pmatrix} \|$$

角速度：

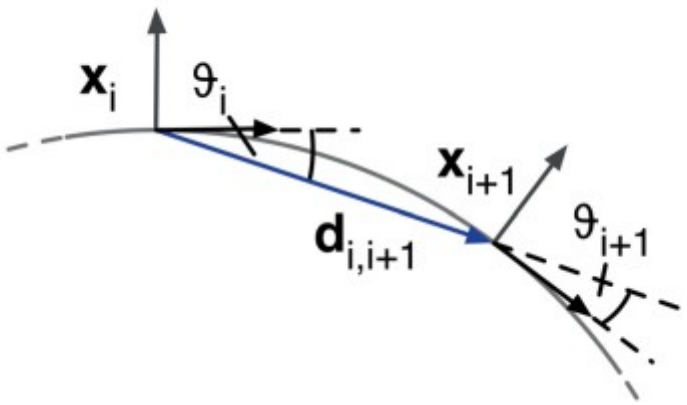
$$\omega_i \simeq \frac{\beta_{i+1} - \beta_i}{\Delta T_i}$$

加速度：

$$a_i \quad = \quad \frac{2(v_{i+1} - v_i)}{\Delta T_i + \Delta T_{i+1}}$$

通过这几个公式能够计算出在每一个时刻的动力学特性，在给定的速度和加速度的范围内，设定给定的 xr 的值，然后通过这几个公式可以计算出对应的速度和加速度的值，之后带入到惩罚函数中去可以计算出损失，所以这里的动力学约束都不是硬约束，都是软约束。

运动学约束：



差分驱动的机器人只能在机器人当前方向上运动。这个运动学约束导致了机器人的平滑路径是由一系列分段圆弧组成的。所以两个相邻的超图中的点的信息处于具有固定曲率的同一段圆弧上，所以要满足以下公式的条件：

$$\vartheta_i \quad = \quad \vartheta_{i+1}$$

$$\Leftrightarrow \begin{pmatrix} \cos\beta_i \\ \sin\beta_i \\ 0 \end{pmatrix} \times \mathbf{d}_{i,i+1} \quad = \quad \mathbf{d}_{i,i+1} \times \begin{pmatrix} \cos\beta_{i+1} \\ \sin\beta_{i+1} \\ 0 \end{pmatrix}$$

方向向量是：

$$\mathbf{d}_{i,i+1} := \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \\ 0 \end{pmatrix}$$

目标函数：

$$f_k(\mathbf{x}_i, \mathbf{x}_{i+1}) = \left\| \left[\begin{pmatrix} \cos\beta_i \\ \sin\beta_i \\ 0 \end{pmatrix} + \begin{pmatrix} \cos\beta_{i+1} \\ \sin\beta_{i+1} \\ 0 \end{pmatrix} \right] \times \mathbf{d}_{i,i+1} \right\|^2$$

将目标函数转化成下面的公式：

$$|(cos\theta_1 + cos\theta_2)(y_2 - y_1) - (sin\theta_1 + sin\theta_2)(x_2 - x_1)|$$

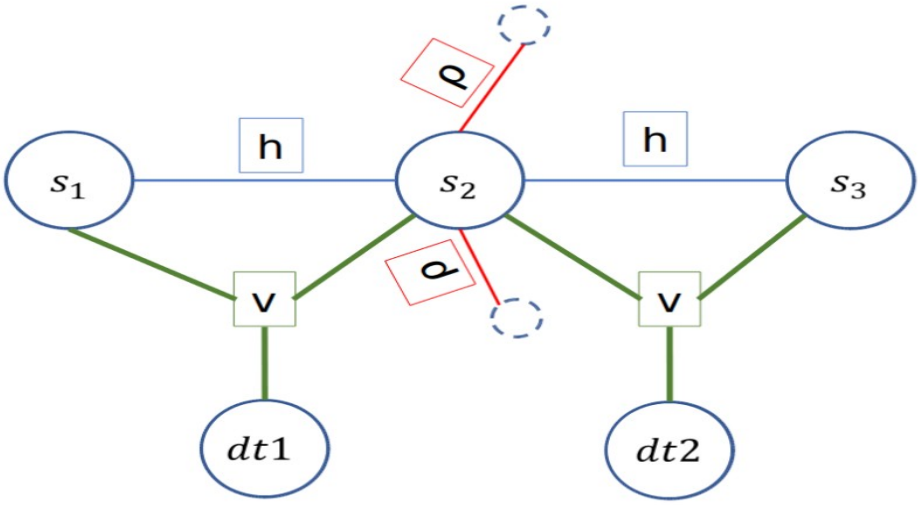
惩罚值越小表示按照运动学约束走的越好。

时间因素：

$$f_k = (\sum_{i=1}^{n} \Delta T_i)^2$$

利用 g2o 通用图优化法来进行求解：

在得到这些目标函数的约束表达之后，调用 g2o 来进行迭代求解:



目标函数包含的几种类型详细说明：

(1).waypoints and obstacles

实现跟随已知的全局规划路径和避障

(2).velocity and accleration

根据点的姿态和时间信息相应的能够计算出对应的速度和加速度的信息（有一定的范围）

(3).non-holonomic kinematics（非完整运动学)

希望小车的运动轨迹是相对来说是比较平滑的，控制量是车速和转角，

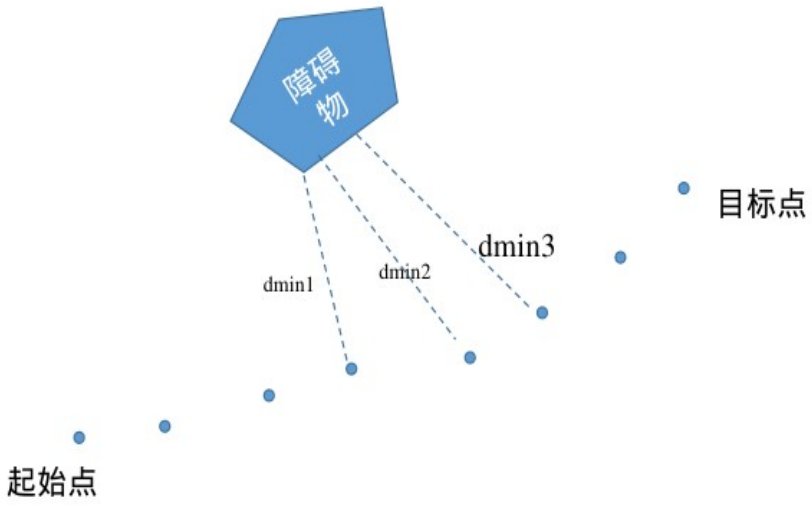
阿克曼结构有最小转弯半径，差速/全向轮车 0

(4).fastest path

在起始点和目标点之间是等间隔取的点，希望整体的时间效率最高。

6. 障碍物约束说明

假设此障碍物距离给出的路径点的几个相对较近的距离为 dmin1,dmin2,dmin3,最近的为 dmin1.



规则：

1.点与点之间的时间间隔是相等的

2.每个点带有对应姿态信息

3.每个约束只会影响其周围的几个连续状态，而不是整条 path

三、代码分析

1.决定局部规划目标点的姿态

```
robot_goal_.x() = transformed_plan.back().pose.position.x;
robot_goal_.y() = transformed_plan.back().pose.position.y;
// Overwrite goal orientation if needed
if (cfg_.trajectory.global_plan_overwrite_orientation)
{
  robot_goal_.theta() = estimateLocalGoalOrientation(global_plan_, transformed_plan.back(), goal_idx, tf_plan_to_global);
  // overwrite/update goal orientation of the transformed plan with the actual goal (enable using the plan as initialization)
  tf2::Quaternion q;
  q.setRPY(0, 0, robot_goal_.theta());
  tf2::convert(q, transformed_plan.back().pose.orientation);
```

决定了局部规划时目标点的姿态是否用局部规划最后一个点的姿态，或者使用局部 path 上姿态 yaw 的均值来作为目标点的姿态

局部规划的最后一个点就是所谓的 local_goal,当遇到需要终点转弯时，local_goal 的朝向不是最终的 goal 的朝向，仍然是规划的

2. movebase 加载局部路径规划器

```
// prune global plan to cut off parts of the past (spatially before the robot)
ROS_INFO_STREAM("the global plan size is :" << global_plan_.size());
pruneGlobalPlan(*tf_, robot_pose, global_plan_, cfg_.trajectory.global_plan_prune_distance);
ROS_INFO_STREAM("prune  global plan size is :" << global_plan_.size());
// Transform global plan to the frame of interest (w.r.t. the local costmap)
```

pruneGlobalPlan:截取全局路径的一部分组委局部规划的初始轨迹，主要是将机器人已经过去的位姿删除。通过设置参数 dist_behind_robot，单位是 m，默认是 1，不能小于地图的 cellsize。在调用后，一般在规划几次后就会发生变化。并不是一直不变的。

3. 静态障碍和自定义障碍

```
// Update obstacle container with costmap information or polygons provided by a costmap_converter plugin
if (costmap_converter_)
  updateObstacleContainerWithCostmapConverter();
else
  updateObstacleContainerWithCostmap();

// also consider custom obstacles (must be called after other updates, since the container is not cleared)
updateObstacleContainerWithCustomObstacles();
```

updateObstacleContainerWithCunstomObstacles()功能为用户发布的障碍物消息

分别由参数 include_costmap_obstacles 和 costmap_converter_plugin 决定，动态障碍 include_dynamic_obstalces.之后对应 buildGraph 中的 AddEdgesObstacles 与 AddEdgesDynamicObstacles

```cpp
void TebLocalPlannerROS::updateObstacleContainerWithCostmap()
{
  // Add costmap obstacles if desired
  if (cfg_.obstacles.include_costmap_obstacles)
  {
    Eigen::Vector2d robot_orient = robot_pose_.orientationUnitVec();

    for (unsigned int i=0; i<costmap_->getSizeInCellsX()-1; ++i)
    {
      for (unsigned int j=0; j<costmap_->getSizeInCellsY()-1; ++j)
      {
        if (costmap_->getCost(i,j) == costmap_2d::LETHAL_OBSTACLE)
        {
          Eigen::Vector2d obs;
          costmap_->mapToWorld(i,j,obs.coeffRef(0), obs.coeffRef(1));

          // check if obstacle is interesting (e.g. not far behind the robot)
          Eigen::Vector2d obs_dir = obs-robot_pose_.position();
          // if(obs_dir.norm() < 1.5)    //改变最大速度约束
        //   cfg_.robot.max_vel_x = 0.3;
        // 如果norm值过大 或者  obs_dir和robot_orient的点乘积小于0,  不算障碍
          if ( obs_dir.dot(robot_orient) < 0 && obs_dir.norm() > cfg_.obstacles.costmap_obstacles_behind_robot_dist  )
            continue;

          obstacles_.push_back(ObstaclePtr(new PointObstacle(obs)));
        }
      }
    }
  }
```

筛除一些距离车较远的障碍物

```cpp
// Add custom obstacles obtained via message
boost::mutex::scoped_lock l(custom_obst_mutex_);

if (!custom_obstacle_msg_.obstacles.empty())
{
  // We only use the global header to specify the obstacle coordinate system instead of individual ones
  Eigen::Affine3d obstacle_to_map_eig;
  try
  {
    geometry_msgs::TransformStamped obstacle_to_map =  tf_->lookupTransform(global_frame_, ros::Time(0),
                                                       custom_obstacle_msg_.header.frame_id, ros::Time(0),
                                                       custom_obstacle_msg_.header.frame_id, ros::Duration(cfg_.robot.transfo
    obstacle_to_map_eig = tf2::transformToEigen(obstacle_to_map);
  }
  catch (tf::TransformException ex)
  {
    ROS_ERROR("%s",ex.what());
    obstacle_to_map_eig.setIdentity();
  }

  for (size_t i=0; i<custom_obstacle_msg_.obstacles.size(); ++i)
  {
    if (custom_obstacle_msg_.obstacles.at(i).polygon.points.size() == 1 && custom_obstacle_msg_.obstacles.at(i).radius > 0 ) // circle
    {
      Eigen::Vector3d pos( custom_obstacle_msg_.obstacles.at(i).polygon.points.front().x,
                           custom_obstacle_msg_.obstacles.at(i).polygon.points.front().y,
                           custom_obstacle_msg_.obstacles.at(i).polygon.points.front().z );
      obstacles_.push_back(ObstaclePtr(new CircularObstacle( (obstacle_to_map_eig * pos).head(2), custom_obstacle_msg_.obstacles.at(i).radius)));
    }
    else if (custom_obstacle_msg_.obstacles.at(i).polygon.points.size() == 1 ) // point
    {
      Eigen::Vector3d pos( custom_obstacle_msg_.obstacles.at(i).polygon.points.front().x,
                           custom_obstacle_msg_.obstacles.at(i).polygon.points.front().y,
                           custom_obstacle_msg_.obstacles.at(i).polygon.points.front().z );
      obstacles_.push_back(ObstaclePtr(new PointObstacle( (obstacle_to_map_eig * pos).head(2) )));
    }
    else if (custom_obstacle_msg_.obstacles.at(i).polygon.points.size() == 2 ) // line
    {
```

在自定义障碍物时可以针对接收到的障碍物是点、线、或者是带有轮廓的进行判断从而得到不同的障碍物，如果是带有速度的，设置速度并存储。

4. teb optimal 的路径规划

```cpp
  bool success = planner_->plan(transformed_plan, &robot_vel_, cfg_.goal_tolerance.free_goal_vel);
  if (!success)
  {
    planner_->clearPlanner(); // force reinitialization for next time
    ROS_WARN("teb_local_planner was not able to obtain a local plan for the current setting.");

    ++no_infeasible_plans_; // increase number of infeasible solutions in a row
    time_last_infeasible_plan_ = ros::Time::now();
    last_cmd_ = cmd_vel.twist;
    message = "teb_local_planner was not able to obtain a local plan";
    return mbf_msgs::ExePathResult::NO_VALID_CMD;
  }
```

这里的 clearPlanner()不会停止规划，而是强制下次重新初始化 planner,要停止继续规划，只能 return.

```cpp
bool TebOptimalPlanner::plan(const std::vector<geometry_msgs::PoseStamped>& initial_plan, const geometry_msgs::Twist* start_vel, bool free_goal_vel)
{
  ROS_ASSERT_MSG(initialized_, "Call initialize() first.");
  if (!teb_.isInit())
  {
    teb_.initTrajectoryToGoal(initial_plan, cfg_->robot.max_vel_x, cfg_->robot.max_vel_theta, cfg_->trajectory.global_plan_overwrite_orientation,
      cfg_->trajectory.min_samples, cfg_->trajectory.allow_init_with_backwards_motion);
  }
  else // warm start
  {
    PoseSE2 start_(initial_plan.front().pose);
    PoseSE2 goal_(initial_plan.back().pose);
    if (teb_.sizePoses()>0
        && (goal_.position() - teb_.BackPose().position()).norm() < cfg_->trajectory.force_reinit_new_goal_dist
        && fabs(g2o::normalize_theta(goal_.theta() - teb_.BackPose().theta())) < cfg_->trajectory.force_reinit_new_goal_angular) // actual warm start!
      teb_.updateAndPruneTEB(start_, goal_, cfg_->trajectory.min_samples); // update TEB
    else // goal too far away -> reinit
    {//当导航较远的goal时，则首先clear轨迹序列，然后重新起点终点插值
      ROS_DEBUG("New goal: distance to existing goal is higher than the specified threshold. Reinitalizing trajectories.");
      teb_.clearTimedElasticBand();
      teb_.initTrajectoryToGoal(initial_plan, cfg_->robot.max_vel_x, cfg_->robot.max_vel_theta, cfg_->trajectory.global_plan_overwrite_orientation,
        cfg_->trajectory.min_samples, cfg_->trajectory.allow_init_with_backwards_motion);
    }
  }
  //从里程计获取的速度
  if (start_vel)
    setVelocityStart(*start_vel);
  if (free_goal_vel)
    setVelocityGoalFree();
  else
    vel_goal_.first = true; // we just reactivate and use the previously set velocity (should be zero if nothing was modified)

  // now optimize
  return optimizeTEB(cfg_->optim.no_inner_iterations, cfg_->optim.no_outer_iterations);
}
```

根据输入的初始路径，设置轨迹的起始点和终点，以及中间点的 yaw 的信息，之后调用 optimizeTEB()函数。杆关键函数是 buildGraph 和优化 OptimizeGraph 两个步骤

将轨迹优化问题建成了一个 g2o 图优化问题，通过公 g2o 中关于大规模系数矩阵的优化算法来解决，涉及到构建超图。将机器人的姿态和时间作为节点，将目标函数和约束作为边。建图时将路径的一系列约束加入进去，比如与障碍物保持一定距离，速度、加速度限制、时间最小约束、最小转弯半径、旋转方向约束等。这些转换成代价函数放在 g2o 中，g2o 优化时会使得这些代价函数达到最小。

5. 局部规划时路径的初始化

```cpp
bool TimedElasticBand::initTrajectoryToGoal(const std::vector<geometry_msgs::PoseStamped>& plan, double max_vel_x, double max_vel_theta, bool estimate_orient, int min_samples, bool guess_backwards_motion)
{
  if (!isInit())
  {
    PoseSE2 start(plan.front().pose);
    PoseSE2 goal(plan.back().pose);

    addPose(start); // add starting point with given orientation
    setPoseVertexFixed(0,true); // StartConf is a fixed constraint during optimization

    bool backwards = false;
    if (guess_backwards_motion && (goal.position()-start.position()).dot(start.orientationUnitVec()) < 0) // check if the goal is behind the start pose (w.r.t. start orientation)
      backwards = true;
    // TODO: dt - max_vel_x_backwards for backwards motions

    for (int i=1; i<(int)plan.size()-1; ++i)
    {
      double yaw;
      if (estimate_orient)
      {
        // get yaw from the orientation of the distance vector between pose_{i+1} and pose_{i}
        double dx = plan[i+1].pose.position.x - plan[i].pose.position.x;
        double dy = plan[i+1].pose.position.y - plan[i].pose.position.y;
        yaw = std::atan2(dy,dx);
        if (backwards)
          yaw = g2o::normalize_theta(yaw+M_PI);
      }
      else
      {
        yaw = tf::getYaw(plan[i].pose.orientation);
      }
      PoseSE2 intermediate_pose(plan[i].pose.position.x, plan[i].pose.position.y, yaw);
      double dt = estimateDeltaT(BackPose(), intermediate_pose, max_vel_x, max_vel_theta);
      addPoseAndTimeDiff(intermediate_pose, dt);
    }

    // if number of samples is not larger than min_samples, insert manually
    if ( sizePoses() < min_samples-1 )
    {
      ROS_DEBUG("initTEBtoGoal(): number of generated samples is less than specified by min_samples. Forcing the insertion of more samples...");
      while (sizePoses() < min_samples-1) // subtract goal point that will be added later
      {
        // simple strategy: interpolate between the current pose and the goal
        PoseSE2 intermediate_pose = PoseSE2::average(BackPose(), goal);
        double dt = estimateDeltaT(BackPose(), intermediate_pose, max_vel_x, max_vel_theta);
        addPoseAndTimeDiff( intermediate_pose, dt ); // let the optimier correct the timestep (TODO: better initialization
      }
    }

    // Now add final state with given orientation
    double dt = estimateDeltaT(BackPose(), goal, max_vel_x, max_vel_theta);
    addPoseAndTimeDiff(goal, dt);
    setPoseVertexFixed(sizePoses()-1,true); // GoalConf is a fixed constraint during optimization
  }
  else // size!=0
  {
```

在这里会对初始间隔的路径点根据给定的速度去求出两两点之间的时间值，然后将 pose 和时间值信息存储起来，得到一系列节点。dt 的单位是秒，不是时间戳，是路径点之间的时间差。所有点都插入 pose_vec_,但是除了起点外的 timestep 都插入 timediff_vec_.

```
void TimedElasticBand::updateAndPruneTEB(boost::optional<const PoseSE2&> new_start, boost::optional<const PoseSE2&> new_goal, int min_samples)
{
  // first and simple approach: change only start confs (and virtual start conf for inital velocity)
  // TEST if optimizer can handle this "hard" placement

  if (new_start && sizePoses()>0)
  {
    // find nearest state (using l2-norm) in order to prune the trajectory
    // (remove already passed states)
    double dist_cache = (new_start->position()- Pose(0).position()).norm();
    double dist;
    int lookahead = std::min<int>( sizePoses()-min_samples, 10); // satisfy min_samples, otherwise max 10 samples

    int nearest_idx = 0;
    for (int i = 1; i<=lookahead; ++i)
    {
      dist = (new_start->position()- Pose(i).position()).norm();
      if (dist<dist_cache)
      {
        dist_cache = dist;
        nearest_idx = i;
      }
      else break;
    }

    // prune trajectory at the beginning (and extrapolate sequences at the end if the horizon is fixed)
    if (nearest_idx>0)
    {
      // nearest_idx is equal to the number of samples to be removed (since it counts from 0 ;-) )
      // WARNING delete starting at pose 1, and overwrite the original pose(0) with new_start, since Pose(0) is fixed during optimization!
      deletePoses(1, nearest_idx);  // delete first states such that the closest state is the new first one
      deleteTimeDiffs(1, nearest_idx); // delete corresponding time differences
    }

    // update start
    Pose(0) = *new_start;
  }

  if (new_goal && sizePoses()>0)
  {
    BackPose() = *new_goal;
  }
};
```

在获取到初始的的点之后，同样需要对图的点数进行更新处理，将 new_start 后面的点进行删除，得到一张新的节点图。

6. optimizeTEB 的流程循环操作(多次图优化)：

```
bool TebOptimalPlanner::optimizeTEB(int iterations_innerloop, int iterations_outerloop, bool compute_cost_afterwards,
                                    double obst_cost_scale, double viapoint_cost_scale, bool alternative_time_cost)
{
  if (cfg_->optim.optimization_activate==false)
    return false;

  bool success = false;
  optimized_ = false;

  double weight_multiplier = 1.0;

  // TODO(roesmann): we introduced the non-fast mode with the support of dynamic obstacles
  //                 (which leads to better results in terms of x-y-t homotopy planning).
  //                 however, we have not tested this mode intensively yet, so we keep
  //                 the legacy fast mode as default until we finish our tests.
  bool fast_mode = !cfg_->obstacles.include_dynamic_obstacles;

  for(int i=0; i<iterations_outerloop; ++i)
  {
    if (cfg_->trajectory.teb_autosize)
    {
      //teb_.autoResize(cfg_->trajectory.dt_ref, cfg_->trajectory.dt_hysteresis, cfg_->trajectory.min_samples, cfg_->trajectory.max_samples);
      teb_.autoResize(cfg_->trajectory.dt_ref, cfg_->trajectory.dt_hysteresis, cfg_->trajectory.min_samples, cfg_->trajectory.max_samples, fast_mode);

    }

    success = buildGraph(weight_multiplier);
    if (!success)
    {
        clearGraph();
        return false;
    }
    success = optimizeGraph(iterations_innerloop, false);
    if (!success)
    {
        clearGraph();
        return false;
    }
    optimized_ = true;

    if (compute_cost_afterwards && i==iterations_outerloop-1) // compute cost vec only in the last iteration
      computeCurrentCost(obst_cost_scale, viapoint_cost_scale, alternative_time_cost);

    clearGraph();

    weight_multiplier *= cfg_->optim.weight_adapt_factor;
  }
```

一、teb.autoResize()在外循环里

二、buildGraph 函数

三、optimizeGraph 内循环就是 g2o 的 optimize 函数的参数

四、computeCurrentCost 函数(目前的图优化的花费，只在最后一次外循环时进行一次)

五、clearGraph 函数

外循环通过调用 autoResize()来根据时间分辨率调整轨迹，应根据 cpu 的控制速率要求来定

内循环调用 optimizeGraph()进行优化，内部循环的时间一般经过 2-6 次

左后调用 computeCurrentCost()计算轨迹花费，将图中所有边的 error 的平方和构成 cost

autoResize

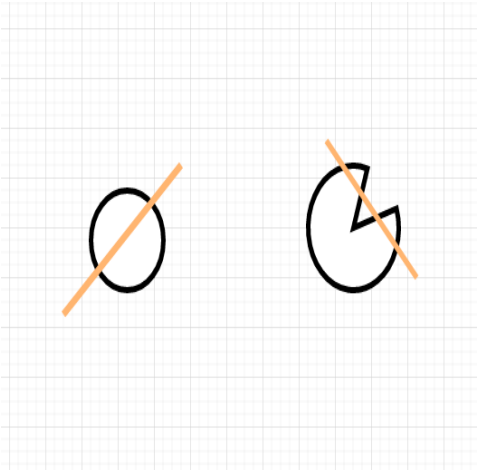
一般不用快速模式，进行 100 次外循环

```
void TimedElasticBand::autoResize(double dt_ref, double dt_hysteresis, int min_samples, int max_samples, bool fast_mode)
{
  ROS_ASSERT(sizeTimeDiffs() == 0 || sizeTimeDiffs() + 1 == sizePoses());
  /// iterate through all TEB states and add/remove states!
  bool modified = true;

  for (int rep = 0; rep < 100 && modified; ++rep) // actually it should be while(), but we want to make sure to not get stuck i
  {
    modified = false;

    for(int i=0; i < sizeTimeDiffs(); ++i) // TimeDiff connects Point(i) with Point(i+1)
    {
      if(TimeDiff(i) > dt_ref + dt_hysteresis && sizeTimeDiffs()<max_samples)
      {
        // Force the planner to have equal timediffs between poses (dt_ref +/- dt_hyteresis).
        // (new behaviour)
        if (TimeDiff(i) > 2*dt_ref)
        {
          double newtime = 0.5*TimeDiff(i);

          TimeDiff(i) = newtime;
          insertPose(i+1, PoseSE2::average(Pose(i),Pose(i+1)) );
          insertTimeDiff(i+1,newtime);

          i--; // check the updated pose diff again
          modified = true;
        }
        else
        {
          if (i < sizeTimeDiffs() - 1)
          {
            timediffs().at(i+1)->dt()+= timediffs().at(i)->dt() - dt_ref;
          }
          timediffs().at(i)->dt() = dt_ref;
        }
      }
      else if(TimeDiff(i) < dt_ref - dt_hysteresis && sizeTimeDiffs()>min_samples) // only remove samples if size is larger tha
      {
        //ROS_DEBUG("teb_local_planner: autoResize() deleting bandpoint i=%u, #TimeDiffs=%lu",i,sizeTimeDiffs());

        if(i < ((int)sizeTimeDiffs()-1))
        {
          TimeDiff(i+1) = TimeDiff(i+1) + TimeDiff(i);
          deleteTimeDiff(i);
          deletePose(i+1);
          i--; // check the updated pose diff again
        }
        else
        { // last motion should be adjusted, shift time to the interval before
          TimeDiff(i-1) += TimeDiff(i);
          deleteTimeDiff(i);
          deletePose(i);
        }

        modified = true;
      }
    }
```

这个其实是对 initTrajectoryToGoal 中建立的弹性带进行调整，不能让位姿点和 deltT 太密集，也不能太稀疏。min_samples 和 max_samples 是另外一组限制条件，dt_ref 和 dt_hysteresis 是另一组

teb 算法是否能找到最优解

teb 算法带的都是一些软约束，teb 求解的话是一个求一个非凸函数，所以得到的是局部最优解。



针对 teb 算法天然带有倒车问题的处理：

1.加上 waypoints 的选项，设定途径点之间不同的间隔，然后加大 weight_kinematics_forward_dirve 的权重，基本可以避免倒车现象，或者加大 weight_kinematics_forward_drive,权重太大会降低收敛速度，将 max_vel_x_backwards 减小到 0，至少减少到参数 penalty_epsilon 相同，同样能够达到相同的效果。

2.由于是差速车，所以考虑在得到全局 path 的初始位置的几个点计算出一个初始的 yaw，以及获取到当前车自身实际的 yaw，如果偏差大于设定的阈值，则先转到规划的 path 的方向上，之后再调用 teb 算法处理。

这种处理有两种方法，第一种是计算出来偏差之后，虚拟的再构造出来一个目标点，这个目标点的位置信息是由起始点的位置得到的，yaw 的信息是由全局 path 的前几个点位信息得到的，有了起始点和目标点，再调用一次 teb 算法，得出了对应的速度信息，下发给底盘，后面的流程按照正常的 teb 算法进行处理即可。

另外一种方式是只要检测到阈值之后，在下发给底盘的速度上给线速度为 0，yaw 还按照实际的 teb 输出进行，这同样能够避免倒车的问题。