**Overview of RAMP ROS Nodes**

This document is meant to be a brief overview of our implementation of the RAMP framework. For a more detailed explanation, please refer to the IROS 2016 paper. Each process involved in our implementation is described, and some pseudocode is given for the various procedures in each process.

First, pseudo-code for the RAMP algorithm is given below.

**RAMP Algorithm:**

i = 0;

Initialize a population *P* of trajectories;

While not is not reached do:

Simultaneously do **Move**, **Plan**, and **Sense**:

**Move**:

If no expected collision on *T\_best* before next control cycle then:  
 Move along *T\_best;*  
 else  
 Pause motion;

**Plan**:

If end of ith control cycle then:

i = i+1;  
 *T\_best* = best trajectory in population;  
 Update *P* with new starting state;

Call Modification();

**Sense**:

If end of ith sensing cycle then:

Update planner with new obstacle information  
 Evaluate *P;*

**Modification:**

Randomly select modification operator *o*;

Randomly select a trajectory from population *P*

Apply *o* to selected trajectory  
 Evaluate new trajectory

If new trajectory is better than worst trajectory in *P* then:

If trajectory is feasible then:

Replace a random non-best feasible member of *P* with new trajectory;

Else:

Replace a random non-best infeasible member of *P* with new trajectory;

Below is the list of processes involved in our RAMP implementation. The source code for our RAMP implementation can be found at <https://github.com/sterlingm/ramp>. The most up-to-date branch is “integrate\_sensing\_circles”.

Three of our processes are service hosts that only listen for srv requests. Those processes are trajectory\_generator, trajectory\_evaluation, and path\_modification.

For each process, there is a brief explanation of the process’ purpose, and some pseudo-code of the process’ main procedure(s). All pseudo-code lines in red text are parts of the code that are involved in some kind of inter-process communication (publishing or subscribing). All processes execute in parallel.

**Ramp Planner**

This process has three main procedures – control cycles, planning cycles, and sensing cycles. They are implemented as callback functions for a ros::Timer. The control cycles send the latest best trajectory for the robot to move on. The control cycles also update the starting state of the population. The planning cycles perform modifications to the population, and perform a motion error compensation procedure. Sensing cycles update the latest obstacle information, and re-evaluate the population based on the updated information.

**Control Cycle**

Find best trajectory, T\_best, from the population

Send T\_best to robot for execution (Publish RampTrajectory msg)

// Adapt the holonomic population:

For each trajectory:

Update knot points on the path

Update curve information

Plan a trajectory with updated information (Call trajectory\_generator service)

// Find non-holonomic population:

For each trajectory:

Find transition from current trajectory onto T (Call trajectory\_generator service)

Evaluate the population (Call trajectory\_evaluation service)

Set time for next control cycle t\_cc = earliest starting time of a trajectory in the population

**Planning Cycle**

Adjust starting state and offset trajectories based on motion error

Do modification (Call path\_modification service)

Compute trajectory for modified path (Call trajectory\_generator service)

Find transition trajectory for the new trajectory (Call trajectory\_generator service)

Evaluate new trajectory (Call trajectory\_evaluation service)

Attempt to insert new trajectory into population

If modification changed population:

Set next control cycle time = earliest starting time of a trajectory in the population

**Sensing Cycle**

Receive latest obstacle information from ramp\_sensing

Generate obstacle trajectories (call trajectory\_generator service)

Evaluate current trajectory (T\_best) with new obstacle trajectories (call trajectory\_evaluation service)

Evaluate population with new obstacle trajectories (call trajectory\_evaluation service)

**Imminent Collision Timer Callback**

If current trajectory (T\_best) is not feasible && time until first collision < D // (D=some time threshold)

result = true

Else

result = false

Set ros parameter = result

**Update Msg Callback (MotionState)**

Receive latest robot state from ramp\_control in form of MotionState

Transform odometry coordinates into world coordinates

Set latest state of the robot

**Trajectory Generator (ROS Service)**

The trajectory\_generator process receives an array of TrajectoryRequest msgs, and generates a trajectory for each request. The TrajectoryRequest msg contains a Path and Bezier Curve information. It utilizes the Reflexxes library for trajectory generation.

Input: TrajectorySrv::Request

Output: TrajectorySrv::Response

For each request:

Given curve information, compute points on the curve using Reflexxes library

Convert holonomic input Path p to a non-holonomic path that contains Bezier Curve knot points

For each knot point *i* in path *p*

If *p[i-1]:p*[*i*] is a curve:

Push on curve points

Else:

Call Reflexxes library until knot point *i* is reached

**Trajectory Evaluation (ROS Service)**

The process trajectory\_evaluation hosts a service for computing the fitness and feasibility of a trajectory. This input to this process, EvalutionSrv::Request, is an array of EvalutionRequests, which contain a trajectory to evaluation, a list of obstacle trajectories to use for collision detection, robot and obstacle radii information, and other misc. variables used in evaluating.

Input: EvaluationSrv::Request

Output: EvaluationSrv::Response

For each request:

Compute feasibility by running collision detection against all obstacle trajectories

Compute fitness based on various trajectory elements, such as time and distance to obstacles

Output EvaluationSrv::Response

**Path Modification (ROS Service)**

Input: ModificationRequest::Request

Output: ModificationRequest::Response

The path\_modification process is responsible for applying genetic operators to a Path. The available operators are insert, delete, change, swap, and crossover. There is little else involved in this process. It does not call any other processes.

Receive input of ModificationRequest::Request

If operator == Insert:

Find random index i

Generate random position q

Insert q at index i+1

Else if operator == Delete:

Choose index i at random

Delete knot point at index i

Else if operator == Change:

Find knot point i

Generate random position q

Replace knot point at index i with q

Else if operator == Swap:

Find two random indices i\_0 and i\_1

Swap the states at those indices

Else if operator == Crossover:

Find two random indices i\_0 and i\_1

Crossover paths at those indices

Output ModificationRequest::Response

**Ramp Control**

ramp\_control subscribes to a topic “ramp/bestTrajec” to listen for a trajectory to execute. The main loop of the process goes through each point on the trajectory and publishes velocity commands to move the robot from each point to the next. This process also subscribes to topics that contain the latest motion state based on the robot’s wheel encoders, builds a ramp\_msgs/MotionState object from this information, and then publishes the object on topic “ramp/update” to update the planner.

**RampTrajectory Msg callback:**

Receive a trajectory to move on

For trajectory points p=1:end

Compute longitudinal and angular velocities to move to from point p-1 to p

Compute time to stop moving at those velocities

**Main Loop:**

For each trajectory point i:

Block movement until no imminent collision (Check ROS parameter server)

While time < end\_times[i]:

Set velocity values

Publish Twist msg

**Odometry Msg Callback**

Receive latest odometry information

Convert to a MotionState msg and store in class object

**Update State Timer Callback**

Publish the latest state of the robot (MotionState msg)

**Ramp Sensing**

The ramp\_sensing process receives sensing information by subscribing to a topic “costmap\_node/costmap/costmap” to obtain costmaps, which are type nav\_msgs/OccupancyGrid. The process determines obstacle position and velocities based on a costmap, and then publish that information on topic “ramp/obstacles”.

Input: nav\_msgs/OccupancyGrid

Output: ramp\_msgs/ObstacleList

**OccupancyGrid msg Callback**

Receive costmap

// Detect circles

Detect obstacle blobs in the costmap

For each blob:

Determine a circle fit for each obstacle

Update Kalman filter to estimate actual best position for circle

Predict the obstacle’s velocity

Build a ramp\_msgs/Obstacle object based on circle

Add Obstacle to ramp\_msgs/ObstacleList

**Obstacle Timer Callback**

Publish ramp\_msgs/ObstacleList msg on “ramp/obstacles” topic