Desired State Generation

Wyatt Newman
February, 2015

A feedback control system requires sensory feedback and an input command (a reference input). For steering algorithms, we will use odometry messages for feedback, and we will need a corresponding input command topic. Odometry messages are typically fast (e.g. 50Hz updates) and smooth, although subject to slow drift. Our command messages should be comparably fast and smooth—and expressed in the same reference frame as odometry messages. We will refer to the input commands as "desired state."

The desired state messages should be a relatively high-frequency stream (e.g. 50 Hz), should be smooth —including respecting inertia—and should contain desired values of x, y, heading, speed and angular velocity (for planar motion; more dimensions are needed to describe 6DOF trajectories). Although only these 5 values are needed for our mobile robots, we will utilize generic, higher-dimensional ROS messages (and simply ignore the irrelevant fields).

The illustrative approach described here is the program "example_des_state_generator" (in a package of the same name). This code compiles and runs, but requires modifications (notably, speed profiling and halt detection) before it is suitable for use on real robots.

The approach assumes starting from a polyline path plan, which is re-interpreted as a sequence of "path" segments. These path segments are traversed incrementally with small, rapid updates. The resulting states—including both pose and twist—are published on the topic "desState."

The following description starts from the top level: how to receive a path plan. At the next level, a path plan is translated into a queue of path segments (which are more dynamically feasible). At the lowest level, path segments are used to generate a corresponding stream of desired states.

Communicating Polyline Plans: Generation of a stream of desired states originates from a higher-level plan. In ROS, the conventional datatype for a path plan is a "Path", defined within the "nav_msgs" package. A "Path" consists of a variable-length vector of objects of type geometry_msgs/PoseStamped[]. A "Pose" is defined in the "geometry_msgs" package, and it consists of two fields: x,y,z values of "position" and x,y,z,w fields of a quaternion specifying "orientation." Equivalently, we can think of a "path" as consisting of a sequence of vertices describing a polyline. For planar motion, this polyline lies in the x-y plane. For most wheeled vehicles, specification of orientation is redundant. To move from vertex v1 to vertex v2, the robot must have a heading along the vector from v1 to v2. Any other specification of orientation would be kinematically inconsistent.

A path defined as a polyline implies a strategy for continuous motion (which is a prerequisite for feasible motion control). Construction of a path can be manual (e.g., by hard-coding a sequence of poses within a static a-priori map), or it can be automatic (e.g. using any of a wide variety of path-planning algorithms). The topic of path planning should be separated from the topic of desired-state generation, since these are problems that differ significantly in level of abstraction and demands on timing. For desired-state generation, we thus assume that a path is to be provided—by whatever means is appropriate.

For the interface between the path planner and the desired-state generator, we must define how paths get communicated. The approach proposed here is to have the path planner communicate with the

desired-state generator via a ROS service, with the expectation that the path planner will deliver subgoals incrementally, as needed. By deferring communication of longer-term subgoals, it will be easier for the planner to invoke re-planning (e.g., as unexpected obstacles are encountered). The planner should have the option of providing sub-goals incrementally—as well as the option of telling the desired-state generator to halt immediately and flush all sub-goals currently in its queue.

Dividing the problem along these lines, the objective of the desired-state generator is as follows. Given a (typically, limited-horizon) plan consisting of a queue of sub-goal poses, output a dynamically-feasible stream of intermediate states at an acceptably-high update rate.

It should be noted a "path" specifies only space; it does not specify timing. A "trajectory" is an augmentation of a "path" that incorporates timing. To be a dynamically-feasible trajectory, the sequence of desired states should conform to constraints on maximum velocities and accelerations (both translational and rotational). A polyline path has a discontinuity of the first derivative at each vertex. Correspondingly, an attempt to follow a polyline at constant speed would require instantaneous changes in (vector) velocities at each vertex. Such motion would require infinite acceleration and infinite angular acceleration. The coarse polyline plan must be re-interpreted to describe it as a feasible trajectory. The example code performs this by translating a path polyline into a queue of path "segments" that are dynamically realistic.

Converting a Polyline Path into Path Segments: To follow a polyline feasibly, it is necessary to come to a complete halt at each vertex, then re-orient the robot (spin in place) with a viable trajectory (e.g., using triangular angular-velocity profile) to point towards the next vertex. Motions of this type are thus: move towards vertex; come to halt at vertex subgoal; spin in place to face the next subgoal; repeat.

Trajectory plans of this type are, at least, dynamically feasible. However, such plans are also often unnecessarily slow. More graceful and more efficient motion would include curved path segments, e.g. by blending sequential line segments with smooth, circular arcs. The example desired-state generation code provided only performs polyline motions (translate or spin in place); extensions to accommodate curved path segments are desirable. In anticipation, an "ARC" segment type is also defined.

In our CWRU repository, we have defined a "pathSegment" message as follows:

The following custom message definition, **PathSegment.msg**, may be used to publish and receive descriptions of path segments.

```
#Reference frame that the following coordinates should be interpreted relative to
and timestamp to use when transforming
Header header

#Segment type enums
int8 LINE = 1
int8 ARC = 2
int8 SPIN_IN_PLACE = 3

#Segment type
int8 seg_type

#Segment number
uint32 seg_number

#Segment length
float64 seg_length

#Reference point
geometry_msgs/Point ref_point

#Initial tangent angle
geometry_msgs/Quaternion init_tan_angle

#Curvature
float64 curvature

#Speed limits for this segment
geometry_msgs/Twist max_speeds
geometry_msgs/Twist min_speeds

#Acceleration limit for this segment (m/s^2 for lines/arcs, rads/s^2 for spin-in-
place)
float64 accel_limit

#Deceleration limit for this segment
float64 decel_limit
```

The above message type describes a single path segment, currently capable of describing line segments, circular arcs and spin-in-place commands.

The fields are defined in the following table (see M.S. Thesis by Eric Perko, Jan, 2013, EECS). One important assumption about these path segments is that, for a given sequence of segments, they will already be blended together to remove any discontinuities in the geometric parameterization between the end of a segment and the beginning of the following segment.

Name	Description
Header	This is a standard ROS header type that contains information such as the reference frame the rest of the fields are in and the timestamp for when the path segment was generated
Segment type	An integer enum representing the type of this segment, such as a straight line segment, constant curvature arc segment or spin-in-place segment.
Segment number	The ID number of this segment.
Segment length	The length of the segment. Whether it is in meters or radians depends on the segment type
Reference point	The reference point for this path segment. Interpretation depends on the segment type
Initial tangent angle	The initial tangent angle for this segment. Interpretation depends on the segment type
curvature	The curvature of this segment. Exact interpretation depends on the segment type. For straight lines, curvature should be 0.0
Max speed	A pair of the maximum translational speed and maximum rotational speed to be used for this segment
Min speed	A pair of the minimum translational speed and minimum rotational speed to be used for this segment
Acceleration limit	The acceleration limit for this segment. Whether it is in m/s^2 or rads/s^2 depends on the segment type
Deceleration limit	The deceleration limit for this segment. Whether it is in m/s^2 or rads/s^2 depends on the segment type

Of the fields listed in the above table, the ones requiring the most explanation are the reference point and the initial tangent angle.

The reference point can have the following meanings, depending on segment type. For a straight line segment, the reference point is the start point of the line segment. For a constant curvature arc, the reference point is the center of the circle that the arc belongs to (and the radius of this circle is 1/curvature). For a spin-in-place segment, the reference point is the point about which to spin.

The initial tangent angle can have the following meanings, depending on segment type. For a straight line segment, the initial tangent angle is the direction of the line segment. For a constant curvature arc, the initial tangent angle defines the actual point along the circle where the the arc segment begins. For a spin-in-place segment, the initial tangent angle is the starting angle for the spin.

In the example code provided, path subgoals are processed one at a time, as needed. These subgoals are transformed into odom coordinates at the last moment, in order to minimize effects of odometry drift (and assuming a localization node periodically updates the odom-to-map transform).

More generally, multiple path poses would be considered in order to construct blended path segments. In the example code, however, only LINE and SPIN-IN-PLACE segments are constructed. The function:

```
vec_of_path_segs = build_spin_then_line_path_segments(start_pose_wrt_odom, goal_pose_wrt_odom);
```

returns a vector of objects of type pathSegment—in this case, always consisting of two path segments: a spin-in-place followed by a line-segment. More generally, this function could return an arbitrary number of path segments that produce a more desirable trajectory. At present, though, each path subgoal leads to two path segments. These segments are pushed onto the path-segment queue for processing of incremental desired states.

A coarse path subgoal is popped from the path queue whenever the segment queue is depleted. If the path queue is also depleted, the robot remains at rest at the last subgoal. The service "appendPathService", provided by example_des_state_generator, remains available to receive more path subgoals whenever a higher-level planner is ready to provide them.

Generating Desired States from Path Segments: At each iteration of the main loop, there is always a current path segment—although this may simply be "HALT." The member function update_des_state() is responsible for incrementing the desired state along the current path segment and publishing the result to the topic "desState". The message published is the same message type as used by odom: nav_msgs::Odometry. This message contains the current desired x, y, heading (a quaternion) and twist (speed and angular velocity). The intent is that a steering node could subscribe to both the desState topic and the odom topic and use this information to compare desired state to measured state, resulting in a control action that makes course corrections relative to the nominal, planned speed and spin rate.

"Crawling" along a path segment is updated at a relatively high frequency, defined in the header file as UPDATE_RATE. The member methods "compute_speed_profile()" and "compute_omega_profile()" are intended to perform speed profiling, e.g. trapezoidal or triangular speed modulation. At present, these functions do ***not*** perform graceful accel/decel. These need to be implemented. Additionally, these functions should be made aware of robot halt conditions, such as E-stop, detected obstacle, or software halt command.

Given the current computed values of speed and angular velocity, progress along a line segment can be computed as:

```
double delta_s = current_speed_des_*dt_; //incremental forward move distance; a scalar
current_seg_length_to_go_ -= delta_s; // plan to move forward by this much
```

For spin-in-place, the updated distance-to-go is:

```
double delta_phi = current_omega_des_*dt_; //incremental rotation--could be + or -
current_seg_length_to_go_ -= fabs(delta_phi); // decrement the (absolute) distance (rotation) to go
```

When the distance-to-go (whether meters or radians) has counted down to zero (or negative, if overshoots), then the current segment following is complete.

Whether a line or spin in place, values are computed for the current desired twist (speed and angular velocity), desired heading (converted to a quaternion), and desired x,y position. These components are used to populate a "desired state", which is an object of type nav_msgs::Odometry. The corresponding message is published on the topic "desState."

When a segment is completed, the function "unpack_next_path_segment()" is called. This function pops the next path segment from the path queue (if available) and initializes conditions for the new segment. If no path segments remain in the queue, this routine attempts to pop a new path subgoal from the path queue to construct and push new path segments onto the segment queue.

Running the Example Code : This code can be run with cwruBot simulation. Once speed profiling is implemented, it should be suitable for use on real robots. Start up this node with:

rosrun example_des_state_generator des_state_generator

The desired-state generator will suspend itself in the class constructor until it receives a valid "odom"message. Once odom is alive, it will attempt to send streams of desired state. However, it will need to receive a path plan before it knows where to go.

An illustrative node that sends a path plan to the desired-state generator can be run as:

rosrun example_des_state_generator example_path_sender

This node contains a ROS service client that sends a "request" that contains a ROS message of type nav_msgs/Path (a variable-length vector of time-stamped poses). The points thus transmitted will be appended to the buffer owned by the state generator node.

This example code has 3 hard-coded poses. You can change the pose coordinates, if desired, and and more subgoal poses. More generally, this node would be replaced by a path planner or an interactive user interface. Nonetheless, this example illustrates the means of communicating a path to the state generator.

The state generator also provides a ROS service "flushPathService", which causes the desired-state generator to clear its current queue of subgoals, to support dynamic replanning.

Sending out desired states does not cause the robot to move. However, the desired-state information is necessary for a steering algorithm, which is a separate package to be discussed.