EXPLORING AND MAPPING CONFINED AND SENSOR-CHALLENGED ENVIRONMENTS WITH PROPRIOCEPTION OF AN ARTICULATED MOBILE ROBOT

by

Jacob Everist

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

April 2015

Dedication

to family

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

In many real-world environments such as flooded pipes or caves, exteroceptive sensors, such as vision, range or touch, often fail to give any useful information for robotic tasks. This may result from complete sensor failure or incompatibility with the ambient environment. We investigate how proprioceptive sensors can help robots to successfully explore, map and navigate in these types of challenging environments.

Our approach is to use a snake robot with proprioceptive joint sensors capable of knowing its complete internal posture. From this posture over time, we are able to sweep out the free space of confined pipe-like environments. With the free space information, we incrementally build a map. The success of mapping is determined by the ability to re-use the map for navigation to user-directed destinations.

We address the following challenges: 1) How does the robot move and locomote in a confined environment without exteroception? 2) How is the distance traveled by a snake robot measured with no odometry and no exteroception? 3) How does the robot sense the environment with only proprioception? 4) How is the map built with the information available? 5) How is the map corrected for visiting the same location twice, i.e. loop-closing? 6) How does the robot navigate and explore with the constructed map?

In order to move through the environment, the robot needs to have a solution for motion planning and collision-reaction. In an exteroceptive approach, we would detect and avoid obstacles at a distance. If collisions were made, touch sensors could detect them and we could react appropriately. With only proprioceptive sensors, indirect methods are needed for reacting to obstacles. A series of motion methods are used to solve these

problems including compliant locomotion, compliant path-following, safe-anchoring, stability assurance, slip detection, and dead-end detection. We show results demonstrating their effectiveness.

While moving through the environment, the snake robot needs some means of measuring the distance traveled. Wheeled robots usually have some form of shaft encoder to measure rotations of the wheels to estimate distance traveled. In addition, range or vision sensors are capable of tracking changes in the environment to estimate position of the robot. GPS is not feasible because of its unreliable operation in underground environments. Our proprioceptive approach achieves motion estimation by anchoring multiple points of the body with the environment and kinematically tracking the change in distance as the snake contracts and extends. This gives us a rudimentary motion estimation method whose effectiveness we measure in a series of experiments.

Some method is needed to sense the environment. In an exteroceptive approach, vision and range sensors would give us a wealth of information about the obstacles in the environment at great distances. Touch sensors would give us a binary status of contact with an obstacle or not. With only proprioceptive joint sensors, our approach is to kinematically compute the occupancy of the robots body in the environment and record the presence of free space over time. Using this information, we can indirectly infer obstacles on the boundary of free space. We show our approach and the sensing results for a variety of environmental configurations.

Combining the multiple snapshots of the local sensed free space, the robot needs a means of building a map. In the exteroceptive approach, one would use the ability to sense landmark features at a distance and identify and merge their position across several views. However, in a proprioceptive approach, all sensed information is local. The opportunities for spotting landmark features between consecutive views are limited. Our approach is to use the pose graph representation for map-building and add geometric constraints between poses that partially overlap. The constraints are made from a

combination of positional estimates and the alignment of overlapping poses. We show the quality of maps in a variety of environments.

In the event of visiting the same place twice, we wish to detect the sameness of the location and correct the map to make it consistent. This is often called the loop-closing or data association problem. In the case of exteroceptive mapping, this is often achieved by finding a correspondence between sets of landmark features that are similar and then merging the two places in the map. In the proprioceptive case, the availability of landmark features is scarce. We instead develop an approach that exploits the properties of confined environments and detects loop-closing events by comparing local environmental topologies. We show the results of loop-closing events in a variety of environmental junctions.

Once we have constructed the map, the next step is to use the map for exploration and navigation purposes. In the exteroceptive case, navigating with the map is a localization problem, comparing the sensed environment with the mapped one. This is also the approach in the proprioceptive case. However, the localization accuracy along the length of a followed path is more uncertain, so our path-following algorithms are necessarily more robust to this eventuality. In the exteroceptive case, the act of exploration can be achieved by navigating to the boundaries of the map with no obstacles. In the proprioceptive case, exploration is achieved by following every tunnel or path until a dead-end is detected. We show some environments that we successfully map and navigate, as well as some environments that require future work.

Chapter 1

Introduction

In this dissertation, we develop an algorithmic approach for robotically mapping confined environments using internal sensors only. Some challenging environments such as underground caves and pipes are impossible to explore and map with external sensors. Such environments often cause external sensors to fail because they are too confined, too hostile for the sensors, or incompatible to the fluid medium. These environments are also impossible for humans to explore for similar reasons. Robotic solutions are possible, but such environments often cause available external sensors to fail. A robotic solution that can explore and map without external sensors would allow access to previously inaccessible environments.

For example, an approach that is invariant to ambient fluid (air, water, muddy water, etc.) would significantly impact cave exploration and underground science by the access to still deeper and smaller cave arteries regardless of whether they are submerged in water. Search and rescue operations would have additional tools for locating survivors in complicated debris piles or collapsed mines. Utility service workers would be able to map and inspect pipe networks without having to excavate or evacuate the pipes.

1.1 Challenges

1.1.1 Limitations of Exteroceptive Sensors

Exteroceptive sensors are sensors that are affixed externally and designed to sense information about the robot's environment. They're biological equivalents include such things as vision, hearing, touch, taste and smell. The robot equivalents include touch, cameras, sonar, and laser range-finders.

Currently we cannot effectively explore and map these environments with exteroceptive sensors. Current state-of-the-art mapping methods depend on exteroceptive sensors such as vision, range, or touch sensors that are sensitive to the ambient properties of the environment. Sonar and laser-range finders are sensitive to the ambient fluid properties for which they are calibrated. They are also often ineffective due to occlusions, distortions, reflections, and time-of-flight issues. Changing visibility, lighting conditions and ambient fluid opacity impact the usefulness of vision sensors. Hazardous environments may also damage fragile external touch sensors.

1.1.2 Tracking Position

In addition to the challenges of sensing the environment, we also have the challenge of tracking the robot's position in the confined environment. Due to the enclosed nature of the environment, global positioning information is often unavailable. Therefore, any mapping method will need an accurate motion estimation approach for the robot. Wheeled dead-reckoning may be very difficult or impossible if the environment is highly unstructured or otherwise untenable to a wheeled locomotion approach. Even a well designed motion estimation approach will be susceptible to accumulation of errors and will require localization techniques that will reduce and bound the positional error of the robot.

1.1.3 Proprioceptive Sensors

Given these restrictions, developing an approach that will work on all the most challenging conditions forces us to rely on a strictly proprioceptive approach for collecting environmental information and building the map. Such proprioceptive sensors include accelerometers, gyros, INS (inertial navigation systems), and joint angle sensors. Examples of biological proprioception includes sense of hunger, fatigue, shortness of breath, pain, sense of hot, sense of cold, perceived configuration of the body, and numerous others.

Exteroceptive mapping approaches directly sense large sweeps of the environment at a distance and allow us to make multiple observations of environmental features from multiple poses, integrating this into a map. With a proprioceptive approach, the information is by definition internal to the robot's body and local to the robot's position. Any inferred information about the environment is limited to the immediate vicinity of the robot's pose. The challenge is to explore the environment and construct a map with fundamentally less information.

1.1.4 Void Space and Features

Limited information about the immediate vicinity of an articulated robot can be obtained by sweeping the robot's body to cover all of the void space that is reachable. By taking posture snapshots while the robot is sweeping and plotting them into an image, we can create a rudimentary void space image of the local environment. The challenge then becomes how to utilize this particular type of data in a robotic mapping approach.

In the exteroceptive approach (external sensors), one would use the ability to sense landmark features at a distance and identify and merge their position across several views. However, in a proprioceptive approach, all sensed information is local. Not only will the opportunities for spotting landmark features between consecutive views be limited, but we must also define what exactly we will use as landmark features. In exteroceptive mapping, the boundaries of the environment are sensed and landmark features are extracted that represent corners, curves, structures, and visual characteristics. With proprioceptive sensors, we present the different kinds of features that can be extracted and used from void space information.

1.1.5 Map Representation

How is a map built and how is it represented? How can it be used and how accurate can it be? The map is built to void space data taken from proprioception. The map must

be sufficiently accurate to navigate the environment to a given destination. What kinds of environmental characteristics can it capture and what are its limitations?

1.1.6 Data Association

In the event of visiting the same place twice, we wish to detect the sameness of the location and correct the map to make it consistent. This is often called the loop-closing or data association problem. In the case of exteroceptive mapping, this is often achieved by finding a correspondence between sets of landmark features that are similar and then merging the two places in the map. In the proprioceptive case, the availability of landmark features is scarce. The current best-practice methods for solving the data association problem depend on an abundance of landmark features with which multiple combinations of associations are attempted until an optimal correspondence is found. In our approach, with only a few landmarks to use, performing data association in this way becomes ineffective at best, destructive at worst. We determine a different approach to data association using void space information.

1.2 Related Work

The mapping of confined spaces is a recurring theme in the research literature. Work has been done to navigate and map underground abandoned mines [Thrun04] with large wheeled robots traveling through the constructed corridors. Although the environment is large enough and clear enough to make use of vision and range sensors, its location underground makes the use of GPS technologies problematic. This work can be applied to confined environments where exteroceptive sensors are still applicable.

Robotic pipe inspection has been studied for a long time [Fukuda89], but usually requires that the pipe be evacuated. The robotic solutions are often tailored for a specific pipe size and pipe structure with accompanying sensors that are selected for the

task. Often the solutions are manually remote-controlled with a light and camera for navigation.

Other work has focused on the exploration of flooded subterranean environments such as sinkholes [Gary08] and beneath sea-ice or glaciers. This involves the use of a submersible robot and underwater sonar for sensing.

Other work has approach the problem of identifying features in the absence of external sensors. This work comes from the literature on robotic grasping and is described as contact detection. That is, the identification and reconstruction of object contacts from the joint sensors of the grasping end effector. Various methods to this approach are described in [Kaneko], [Grupen], [Haidacher], [Mimura].

Our earlier paper [Everist09] established the basic concept of mapping using void space information with a rudimentary motion estimation technique. This early work attempted to merge the information from contact detection with the new void space concept. We did not yet have an approach for localization, feature extraction, and data association.

Another researcher [Mazzini11] tackled the problem of mapping a confined and opaque environment of a pipe junction in an oil well bore hole deep underground where high temperature, high pressure, and mud make external sensors infeasible. Mazzini's approach focuses on mapping one local area of the underground oil well. They use physical probing with a robotic finger to reconstruct the walls of the pipe. The robot remains anchored and immobile so there are no need for motion estimation. Our dissertation differs from this work in that we use void space information instead of contact or obstacle information, our robot is mobile instead of anchored to one location, and that we are exploring and constructing a complete map instead of one local environment of interest.

1.3 Approach

To address these challenges, our approach is to develop algorithmic methods for utilizing proprioceptive joint angle information on an articulated mobile robot to enable exploration and mapping of a confined environment. From a series of complete posture information sets, we build up void space information into a posture image and integrate it into a whole map. We hypothesize that with the posture images taken at multiple robot poses in the environment, we can integrate them into a map; we believe this approach will serve as effective alternative for mapping confined environments when exteroceptive sensors have failed or are unavailable.

Our claim here is that the type of robot used and its method of locomotion are not important so long as the robot is articulated and with many joints. The more joint sensors the robot has, the better capable the robot is of mapping.

In this design we use a simulator that enables us to rapidly develop and test different algorithmic techniques without becoming overly worried about the hardware and locomotive aspects of the robot. We use a simplified flat pipe-like environment with a snake robot that is capable of anchoring to the sides of the walls and pushing and pulling itself through the pipe on a near frictionless floor.

We do not simulate any sensor-inhibiting fluid in the environment or sensor-failure scenarios, nor do use any type of external sensor, be it touch sensors, vision sensors, or range sensors such laser range-finders and sonar. Additionally, no global positioning system is used since it is not practical to expect GPS to function underground. The only sensors we permit ourselves to use are joint sensors.

For capturing the posture images, we examine a number of difficulties and pitfalls to capturing this information. Notably, the prevalence of slip of the robot can corrupt the posture image because the robot loses its stable frame of reference to the environment. We develop a number of techniques for safely capturing the void space information,

reducing the likelihood of slip, detecting slip, and mitigating the damage if slip should occur.

We examine several possibilities for extracting landmark features from the posture images. We try several traditional forms of features such as corners and edges, but in the end we develop our own macro-feature which is the overall shape of the local environment captured by the posture image and reduced to the form of a medial axis.

When the robot is moving through the environment, we must also have techniques for estimating the motion between the poses at which we capture the posture images. We show a highly accurate but very slow method [Sec Motion 1] for our particular snake robot morphology, and compare it to a coarse but fast motion estimation method [Sec Motion 2]. We discover that the coarse method is sufficient so long as we rely on the macro-features of the posture images to constrain the possibilities.

Given these basics, we then delve into the heart of the matter: the actual integration of poses, motion estimates, and posture images into a coherent map. We develop and experiment with a few techniques using the constraints from the macro-features in a variety of ways [Sec Constrain 1]. We finally settle on an aggregate pose constraint method where the whole set of past poses is used to generate a feature that constrains the next pose [Sec Constrain 2].

We show how our mapping approach works in a variety of different environments for example, a single path pipe where there are no branches in the environment. The environment only contains a single path that includes a different variety of turns and angles. We then complicate matters by adding in different types of junctions such as T-junctions, Y-junctions, and X-junctions. We do not yet consider environments that have loops in them.

The adding of junctions adds significant complications and requires us to have a better understanding of the structure of our map. We establish a new type of mapping structure called a shoot map. A shoot is a section of the environment that is a single path. The shoot has an origin and a termination point which indicates either the boundary

of the environment or more space yet to be explored. New shoots are created when a junction is detected, where the new shoot's origin is at the location of the parent shoot where the junction was detected. That representation allows us to deal with the restrictions on the type of information we can acquire and the often incompleteness of the posture image capture with respect to the actual local environment.

Finally, given the fundamental mapping structures and methods we have developed, we cast them into a probabilistic framework. This allows us to keep track of different map possibilities, discarding improbable ones, and selecting the most probable. Our mapping approach has certain ambiguities in the type of environmental information we can acquire. The probabilistic framework keeps track of these ambiguities until they can be resolved when more information is acquired.

1.4 Experimental System

Robot setup, joints and geometry, environmental parameters, simulation.

We explicitly chose to develop our approach in a simulation environment in order to rapidly develop a theoretical framework to the problem of mapping without external sensing. In future work, we will apply these techniques to physical robots.

We choose to use the Bullet physics engine (version 2.77) because it is free, open source, mature, and actively developed. Though it is primarily designed for games and animation, it is acceptable to our robotics application. Our primary requisites are simulation stability and accurate modeling of friction and contact forces. True friction and contact modeling are not available to us without specially designed simulation software at a monetary and performance speed cost. Our choice of Bullet gives us believable results so long as we do not demand too much and try to simulate challenging situations (e.g. high speed, high mass, 1000s of contact points).

We focus our attention on environments such as the one depicted in Figure 1.1. We build a flat plane and place a number of vertical walls to create a pipe-like maze

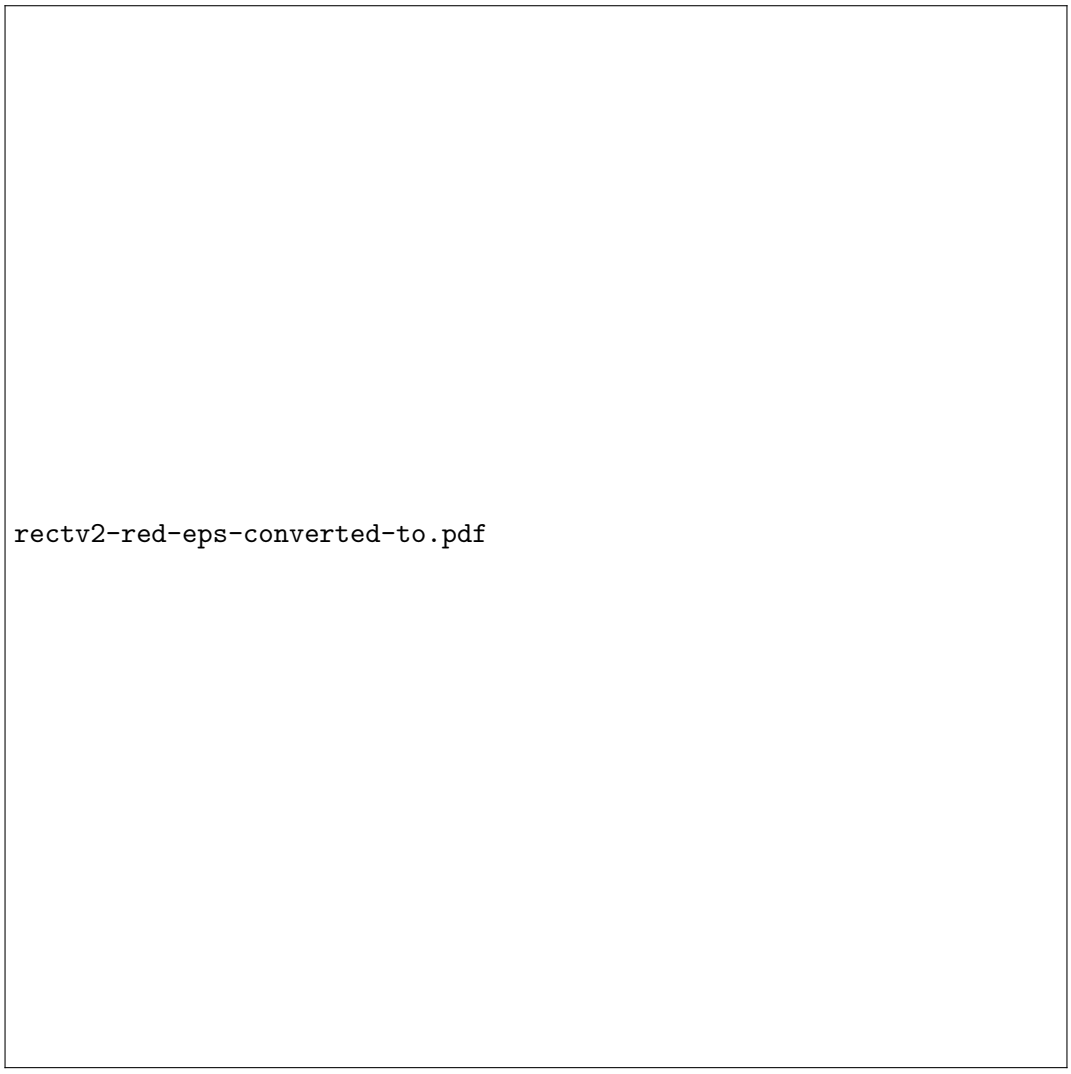


Figure 1.1: Example environment.

environment. All environments we study are flat and have no vertical components. This means we need only focus on building 2D maps to represent the environment. From here on in this paper, we refer to such environments as pipes even though they may not correspond to the even and regular structures of physical utility pipes.

A snake robot is placed in the pipe environment as shown in Figure 1.1. The snake robot consists of regular rectangular segments connected by actuated hinge joints. Each of the joint axes is parallel and coming out of the ground plane. This means the snake

has no means to lift its body off the ground because all of its joints rotate in the plane of the ground. It is only capable of pushing against the walls and sliding along the ground. We choose a reduced capability snake robot because we wish to focus on the mapping problem instead of the more general serpentine control problem.

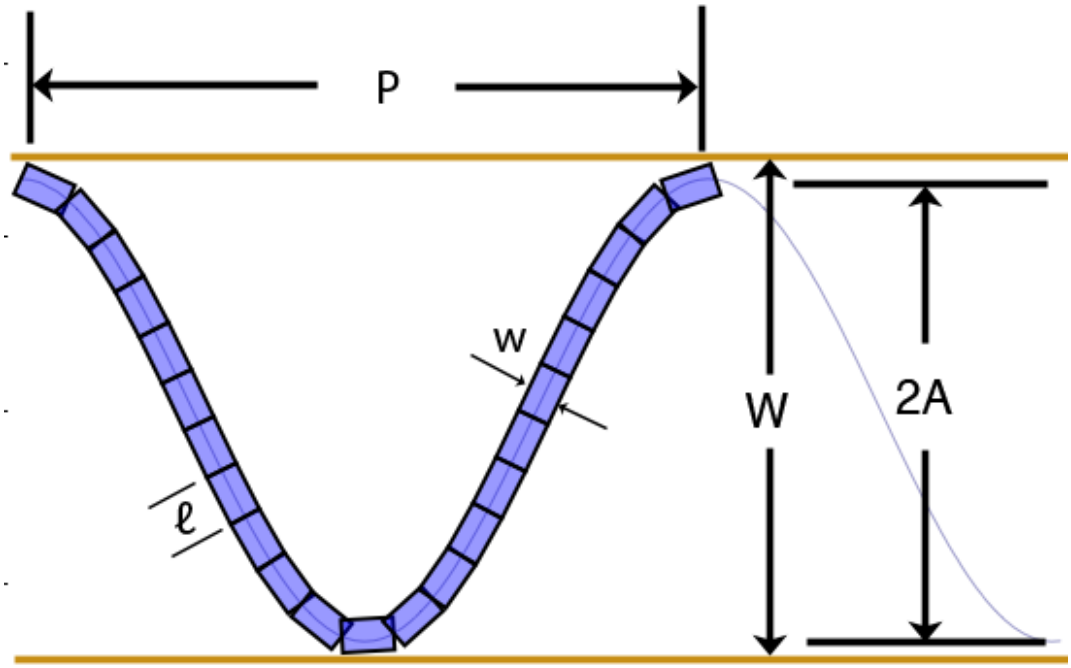


Figure 1.2: Definitions of snake and pipe parameters.

We can parameterize the snake and the pipe environment. For the snake robot, l is the snake segment length, w is the segment width, N is the number of segments connected by $N-1$ joints, and m is the maximum torque capable by each of the joint motors.

[?]

For the environment, we can begin by defining the pipe width W. For a pipe with parallel and constant walls, this is a straightforward definition as seen in Figure 1.2. For non-regular features, we define the pipe width at a given point on one wall to be the distance to the closest point on the opposing wall. This correctly captures the fact that

a robot that is wider than the smallest pipe width W will be unable to travel through that smallest width and will create a non-traversable pinch point. Conversely, a pipe width W that is larger than the reach of a robot will become a void space that the robot will have difficulty completely sensing without the aid of external sensors. For both the minimum and maximum pipe widths of a given environment, we define W_{min} and W_{max} respectively where $W_{min} \leq W_i \leq W_{max}$ where W_i is the pipe width at some point on wall p_i of the pipe environment.

Each joint on the robot is actuated by a motor and PID controller. It can actuate the joint anywhere from ± 160 degrees. It uses the built-in motor capabilities of Bullet that allows us to set the joint angular velocity at run-time. Therefore, all outputs of the PID controller set velocity commands to the joint and the physics engine does its best to satisfy those as velocity constraints.

Algorithm 1 PID Controller

```
1  ε ⇐ (α − φ)
2  if |ε| > tol then
3      ε_sum ⇐ ε_sum + δt × ε
4      δε ⇐ ε − ε_last
5      ε_last ⇐ ε
6      v̂ ⇐ P × ε + I × ε_sum + D × δε/δt
7      if v̂ > v_max then
8          v̂ ⇐ v_max
9      end if
10     if v̂ < −v_max then
11         v̂ ⇐ −v_max
12     end if
13 end if
```

The structure of the PID controller is shown in algorithm 1. Line 1 defines the error as the difference between the target and the actual joint angle. Line 2 prevents the controller from executing if the error falls below an acceptable threshold. This prevents the motor from attempting to perform minor corrections to an already near-correct angle in order to avert oscillations or error-producing compensations. Line 3 is the summation term for error over time while line 4 and 5 is the instantaneous error change from the

previous controller iteration. The actual PID control law is shown on line 6 where P is the proportional term coefficient, I is the integration term coefficient, and D is the derivative term coefficient. The result outputs a command velocity for the Bullet engine. Finally, lines 7–10 limit this velocity to a maximum.

Each of the joints gives angle position information to simulate a shaft encoder or potentiometer. For this study, we do not simulate calibration error, sensor noise, resolution error, or gear backlash. Calibration error has been studied elsewhere /cite and there exists correction mechanisms for it. In the case of sensor noise, the noise on potentiometers is small enough not to affect our algorithms. Gear backlash was encountered and studied in /cite Mazzini. The primary error of concern is resolution error caused by the discretization process of a shaft encoder or an A/D converter. This problem has been studied /cite. Given a sensitive enough A/D converter (really?), this problem can be eliminated. In this study, we assume correct and noise-free joint sensors.

A joint provides an API to the controlling program with 2 read-write parameters and 1 read-only parameter. The read-write parameters are the target joint angle α_i and the maximum torque m_i, with the actual angle ϕ_i being the read-only parameter. α_i is the angle in radians that we desire the joint to rotate to. ϕ_i is the actual angle of the joint that reflects the current physical configuration of the robot. m_i is the maximum permitted torque that we wish to limit the individual motors to. The maximum torque can be lowered to make the joints more compliant to the environment.

Chapter 2

Sensing Space

2.1 Problem

Need to argue here that we are forced to use a contact sensing modality. However, existing contact sensing solutions only return contact point data, i.e. point and location of the boundary. Whereas, with range-based solution, both boundary and void space information are returned. Mostly void space in fact.

Types of sensing can be classified with the following table. The biological senses are grouped with their analogous robotics counterparts.

	Biology	Robotics
Exteroception	sight, hearing, touch, taste, smell	camera, sonar, LIDAR, tactile
Interoception	hunger, hot, cold, thirst, pain	battery sensor, temperature
Proprioception	relative positions of neighboring parts of the body, strength of effort	joint sensor, strain sensor, current sensor, accelerometer, gyros, INS

For the mapping problem, we are going to need a lot of data to build a map of the environment. Existing contact solutions that rely on proprioception such as finger probing and geometric contact detection are too time-consuming to extract data in large volumes. Similarly, other contact sensing solutions such as bump sensors, whiskers or tactile surfaces give more uncertain and noisy contact data while still being time-consuming.

The existing contacts detection approaches that use proprioception are based in the robotic manipulation literature. An end effector as affixed to an arm affixed to ground

in an operational workspace. Contact points are inferred from a combination of the choice of motion, the proprioceptive sensors, and the geometry of the robot. The two major approaches are geometric contact detection and finger probing. The first slides the linkage of the arms along a surface to be detected and reconstruct the contact points from the joint values. The latter infers the contact point from the halting of motion of the end effector along a linear path.

We observe that none of the existing contact sensing solutions emphasize extracting void space as part of their sensing roles. We recognize that in the absence of a range sensor to provide void space data, we can use the body of the robot to extract void space instead. In fact, void space data is extracted in significantly more volumes that it becomes practical to use contact sensing solution to build maps. In this dissertation, we build maps primarily with void space data. We use a rough collision detector to detect dead-ends and is our only instance using any boundary information.

The ratio of the volume of boundary information to void space information can be modeled by the scan angle of a particular range sensor. For a particular scan angle, volume of information can be modeled as number of pixels for some pixel resolution. The number of pixels of void space and boundary can be computed from the area and arc length of a fraction of a circle.

For some scan angle θ and some boundary range r, the arc length L and area A is:

$$L = r\theta$$
$$A = \frac{r^2\theta}{2} \tag{2.1}$$

For some pixel size p_d, the ratio of the volume of void space to the volume of boundary is:

$$\frac{A}{L * d_p} = \frac{2 * r * \theta}{r^2 * \theta * d_p} = \frac{2}{r * d_p}$$

We can see, for any reasonable set of values, a range sensor provides far more void space data than boundary data. For instance, for $r = 10.0$ and a pixel width of $d_p = 0.1$, the ratio of void space to boundary data is $\frac{2}{r*d_p} = 2$. This void space data is often used in many SLAM techniques. Therefore, in a contact-based mapping approach, it would be reasonable to seek out some way to find the comparable void space data for contact sensing and exploit it.

For our particular sensing approach, we chose to use proprioceptive sensors because we want to use the intrinsic geometry of the robot's body as the sensor. For any confined environment, it will be difficult to find the workspace to deploy many of the extended contact sensing approaches such whiskers or probes and use them effectively. Furthermore, a specifically designed external contact sensors such as tactile surfaces or bump sensors will see repeated and near-constant use and may give overly noisy and uncertain data coupled with hastened wear-and-tear. Any approach that uses the existing robot body and does not depend on adding more sensors will be of great use to sensor-challenged and confined environments with existing robots unaugmented.

Our approach is to use the body of the robot to sweep the space of the environment and build up a model of the void space of the environment. From this void space, we can build up a map of the environment without explicitly sensing the obstacles and boundary. We focus on identifying, not where the obstacles are, but where they are not. Through this, we hope to extract the maximum amount of information about the environment with the minimal amount of sensors and action.

2.2 How to Sense

We use a common sense assumption that the robot's body can never intersect with an obstacle. We call this the Obstacle-Exclusion Assumption. If we take the corollary of this assumption, we conclude that all space that intersects with the robot's body must

be free space. If we record the robot's body posture over time, we can use this to build a map of the local environment's free space.

We need 3 key ingredients to build an accurate map of the free space in the local environment: 1) known geometry of the robot's rigid body linkages, 2) accurate joint sensors for computing kinematics, and 3) an accurate reference pose to the global frame. If all of these things are available, we can build a local map such as shown in Figure 2.1.

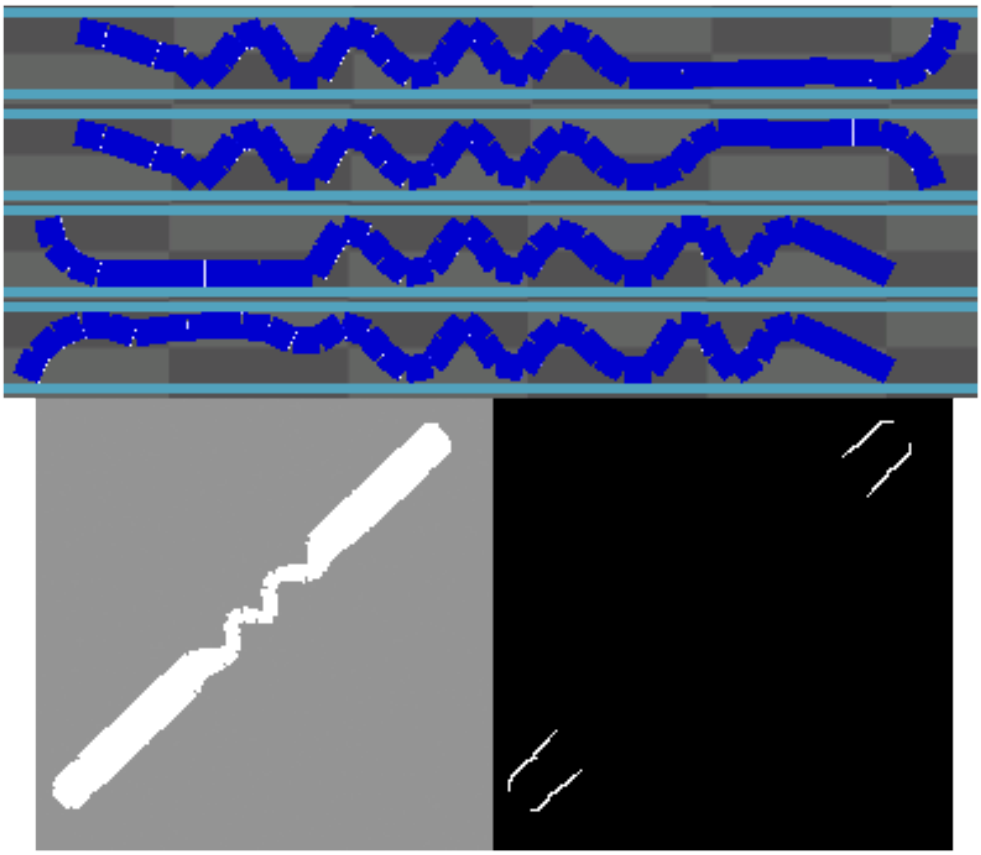


Figure 2.1: TODO: Remove the obstacle map. Free space map from PokeWalls behavior.

The map is achieved by rigidly anchoring to the environment and maintaining a set of stable reference poses. One side of the snake's body is used to sweep the free space, making multiple snapshots of the robot's posture over time and plotting them into a

map. We discuss each aspect of our approach and show the results for a variety of environmental configurations.

The key point here is that sensing cannot be accomplished without action. Action by itself runs the risk of disturbing the environment or causing errors in the reference pose. Though sensing cannot be accomplished without action, action runs the risk of modifying the result. Special care must be taken that the risk of modifying the environment is minimized. Here, we include the robot's body in our definition of the environment so that anchor slippage is a form of environment modification.

As part of the action, at each instance of time t, we produce a posture vector $\bar{phi}_t$, representing the state of the joint angle sensors of the robot such that:

$$\bar{\phi}_t = \phi_1^t, \phi_2^t, , \phi_{M-1}^t, \phi_M^t \tag{2.2}$$

Over a series of time, we produce a series of posture vectors called the posture sequence:

$$\bar{\Phi}_{1:t} = \bar{\phi}_1, \bar{\phi}_2, , \bar{\phi}_{t-1}, \bar{\phi}_t \tag{2.3}$$

Using kinematics and geometry of the robot over time, we can produce the posture image I_k which represents the swept void space at the position and orientation X_k in the global environment.

2.3 Posture Image

Using the control method shown in Control chapter to sweep out the void space of the local environment, we show how we capture sensor data and process it for consumption for our later mapping algorithms.

2.3.1 Data Capture

Once the posture of the snake has stabilized and the anchor points give us good reference poses to the global frame, our objective is take the posture vector, $\bar{\phi}_t$ at time t, and convert it to a 2D spatial representation of free space, M_p at the current pose p. We separate the tasks of spatial representation and positioning the data in the global frame. To do this, we create a local map centered on the robot's body on which the spatial data is plotted while the robot remains in one position.

The local map is an occupancy grid representation where each cell of the grid has two states: *unknown* or *free*. If the body of the robot is present on a cell, this cell is marked *free*. Otherwise, it is *unknown*. It is unknown instead of *occupied* because our approach does not have any means to specifically observe obstacles beyond contact detection heuristics. The heuristics are not accurate enough to plot their positions into the occupancy grid, so we leave the cells unknown for now.

The size of a cell is chosen for the desired accuracy we require. Smaller cell sizes require more computational time, but larger cell sizes will result in blocky maps. We choose our cell dimensions $s_p \times s_p$ to be $s_p = \frac{l}{3} = 0.05$, or one third the length of a snake segment.

Given that we now have our cell dimensions, we can compute the dimensions of our local map to be

$$s_M = l * N + 4 \tag{2.4}$$

where l is the segment length and N is the number of segments. s_M is the maximum length from the origin at the center of our local coordinate system. We want the dimensions to be larger than the worse case scenario of the snake's posture. We also include an extra padding of 4 to ensure that the snake never reaches the boundary of the grid.

The number of cells or pixels in our local map will be $n_p \times n_p$ where

$$n_p = \left\lceil \frac{2s_M}{s_p} + 1 \right\rceil \tag{2.5}$$

This equation ensures that n_p is an odd integer. The $+1$ factor adds an extra cell whose center will act as the origin of our local coordinate system.

Now that we have the dimensions of the grid space and its relation to Cartesian space, we need to know how to convert points from one space to the other. To convert a grid index (i_x, i_y) to Cartesian space point (p_x, p_y) centered within the cell, we compute the following:

$$p_x = \left(i_x - \left\lfloor \frac{n_p}{2} \right\rfloor \right) \frac{s_p}{2} \tag{2.6}$$

$$p_y = \left(i_y - \left\lfloor \frac{n_p}{2} \right\rfloor \right) \frac{s_p}{2} \tag{2.7}$$

Conversely, to find the index of a cell (i_x, i_y) that contains a Cartesian point (p_x, p_y), we compute the following:

$$i_x = \left\lfloor \frac{p_x}{s_p} \right\rfloor + \left\lceil \frac{n_p}{2} \right\rceil \tag{2.8}$$

$$i_y = \left\lfloor \frac{p_y}{s_p} \right\rfloor + \left\lceil \frac{n_p}{2} \right\rceil \tag{2.9}$$

Now that we have the tools for mapping positions in physical space to our grid space occupancy map, we need data to plot into the map. Using kinematics, we compute the geometry of the posture of the snake. We can represent this by a 4-sided rectangle for each segment of the snake. We set the origin of our coordinate system on segment 19 and joint 19 which is the midway point for $N = 40$. We define this to be:

$$O_t = (x_{19}, y_{19}, \theta_{19}) = (0, 0, 0) \tag{2.10}$$

where O_t is the pose of P_{19} in the local frame at time t. This may change which is explained in section 2.5.2.

To compute the segment 19 rectangle, starting from the origin for $k = 19$, $x_k = 0$, $y_k = 0$, and $\theta_k = 0$, we compute the following:

$$\begin{aligned}
x_{k+1} &= x_k + l\cos(\theta_k) \\
y_{k+1} &= y_k + l\sin(\theta_k) \\
\theta_{k+1} &= \theta_k - \phi_{k+1} \\
p_1 &= \left(x_{k+1} - \frac{w\sin(\theta_k)}{2}, y_{k+1} + \frac{w\cos(\theta_k)}{2}\right) \\
p_2 &= \left(x_{k+1} + \frac{w\sin(\theta_k)}{2}, y_{k+1} - \frac{w\cos(\theta_k)}{2}\right) \\
p_3 &= \left(x_{k+1} - l\cos(\theta_k) + \frac{w\sin(\theta_k)}{2}, y_{k+1} - l\sin(\theta_k) - \frac{w\cos(\theta_k)}{2}\right) \\
p_4 &= \left(x_{k+1} - l\cos(\theta_k) - \frac{w\sin(\theta_k)}{2}, y_{k+1} - l\sin(\theta_k) + \frac{w\cos(\theta_k)}{2}\right) \\
R_k &= (p_4, p_3, p_2, p_1)
\end{aligned} \tag{2.11}$$

The result is the rectangle polygon R_k, which represents the rectangle of the segment 19. The next reference pose $(x_{k+1}, y_{k+1}, \theta_{k+1})$ is also computed. Here $k + 1 = 20$. To compute the kth rectangle for $k \geq 19$, we need only compute this iteratively until we reach the desired segment.

To perform this backwards, to find segment 18, where $k{+}1 = 19$ and $(x_{k+1}, y_{k+1}, \theta_{k+1}) = O_t$, we compute the following:

$$
\begin{aligned}
\theta_k &= \theta_{k+1} + \phi_k \\
x_k &= x_{k+1} - l\cos(\theta_k) \\
y_k &= y_{k+1} - l\sin(\theta_k) \\
p_1 &= \left(x_{k+1} - \frac{w\sin(\theta_k)}{2}, y_{k+1} + \frac{w\cos(\theta_k)}{2} \right) \\
p_2 &= \left(x_{k+1} + \frac{w\sin(\theta_k)}{2}, y_{k+1} - \frac{w\cos(\theta_k)}{2} \right) \\
p_3 &= \left(x_{k+1} - l\cos(\theta_k) + \frac{w\sin(\theta_k)}{2}, y_{k+1} - l\sin(\theta_k) - \frac{w\cos(\theta_k)}{2} \right) \\
p_4 &= \left(x_{k+1} - l\cos(\theta_k) - \frac{w\sin(\theta_k)}{2}, y_{k+1} - l\sin(\theta_k) + \frac{w\cos(\theta_k)}{2} \right) \\
R_k &= (p_4, p_3, p_2, p_1)
\end{aligned}
\tag{2.12}
$$

The result is the same polygon as well as the new reference pose for segment 18. To compute the kth segment rectangle for $k < 19$, we need only use this equation iteratively. Computation of all N rectangles results in the set of rectangles $\bar{R}_t$ for the current posture.

Now that we have a set of rectangles which represent the geometry of the robot in its current posture, we want to plot its occupied space into the local occupancy map. To do this, we need to convert polygons in Cartesian space into sets of grid points. To do this, we use a point-in-polygon test algorithm for each of the pixels in the map.

The simplest approach is as follows. For each pixel, convert it to Cartesian space, test if it's contained in any of the polygons, and if it is, set the pixel to *free*. Otherwise, let the pixel remain in its current state. The pseudocode for the point-in-polygon test for convex polygons derived from \cite{orourke98} is seen in algorithm 2.

A single snapshot of the snake posture plotted into the local map is shown in Figure 2.2. The posture $\bar{\phi}_t$ is captured, the rectangles $\bar{R}_t$ representing the body segments are computed from kinematics, each pixel (i_x, i_y) of the map M_p is converted to Cartesian space point (p_x, p_y) and checked by the point-in-polygon algorithm if it is in a rectangle

Algorithm 2 Point-in-Polygon Test

```
R ⇐ rectangle
P ⇐ point
for i = 0 → 4 do
    A_x ⇐ R[i mod 4][0]
    A_y ⇐ R[i mod 4][1]
    B_x ⇐ R[(i + 1) mod 4][0]
    B_y ⇐ R[(i + 1) mod 4][1]
    C_x ⇐ P[0]
    C_y ⇐ P[1]
    if !((Bx − Ax) ∗ (Cy − Ay) − (Cx − Ax) ∗ (By − Ay) ≥ 0) then
        return False
    end if
end for
return True
```

R_k. If it is, $M_p(i_x, i_y)$'s value is set to *free*. This is repeated for each point in each rectangle for a posture snapshot at time t.

While running the PokeWalls behavior, we periodically take snapshots at the conclusion of each Transition step and plot them into the map. A complete sweep with the PokeWalls behavior is shown in Figure 2.3. Furthermore, if we also perform a sweep with the other side of the snake, we can get a map like Figure 2.4.

Notice in Figure 2.4 that there is a gap in the free space data in the center of the body. These segments at the center remain immobilized throughout the sweeping process and never give extra information. This gap in the center is our "blind spot" for this configuration. It is necessary that the center remain immobilized to ensure anchors do not slip so that we maintain correct reference poses to the global frame.

We have no guarantee that there are obstacles at the boundaries of our free space map. We also have no quick way to determine if the boundary of our map is an obstacle or a frontier. While the boundaries at the front and back are usually assumed to lead to more free space, anywhere along the sides could be a missed side passage that the snake failed to discover due to it being too small or being in our blind spot. To combat this situation, we either must perform laborious and time-consuming contact probing of

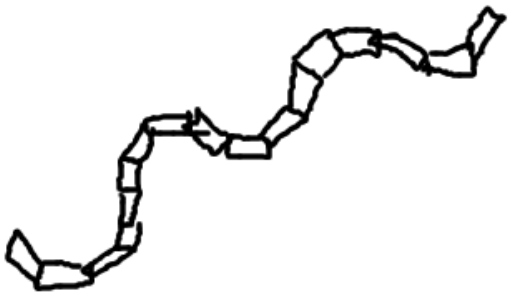


Figure 2.2: Snapshot of $\bar{\phi}_t$ in local coordinates.

every surface, or use quick-and-dirty contact detection heuristics to guide our mapping process.

2.4 Data Processing

Once we have the raw sensor data successfully plotted into a local occupancy map, we must convert this data into a form that is useful for our mapping algorithms. Here, we present three different forms of data processing that we use in our approach.

2.4.1 Convex Hull

One way we process our free space data is by taking its convex hull. In theory, this would create some smooth boundaries and can plausibly represent the true local space under some certain conditions. The definition of the convex hull is, given a set of points P, find the convex polygon H with minimum area that contains all of P.

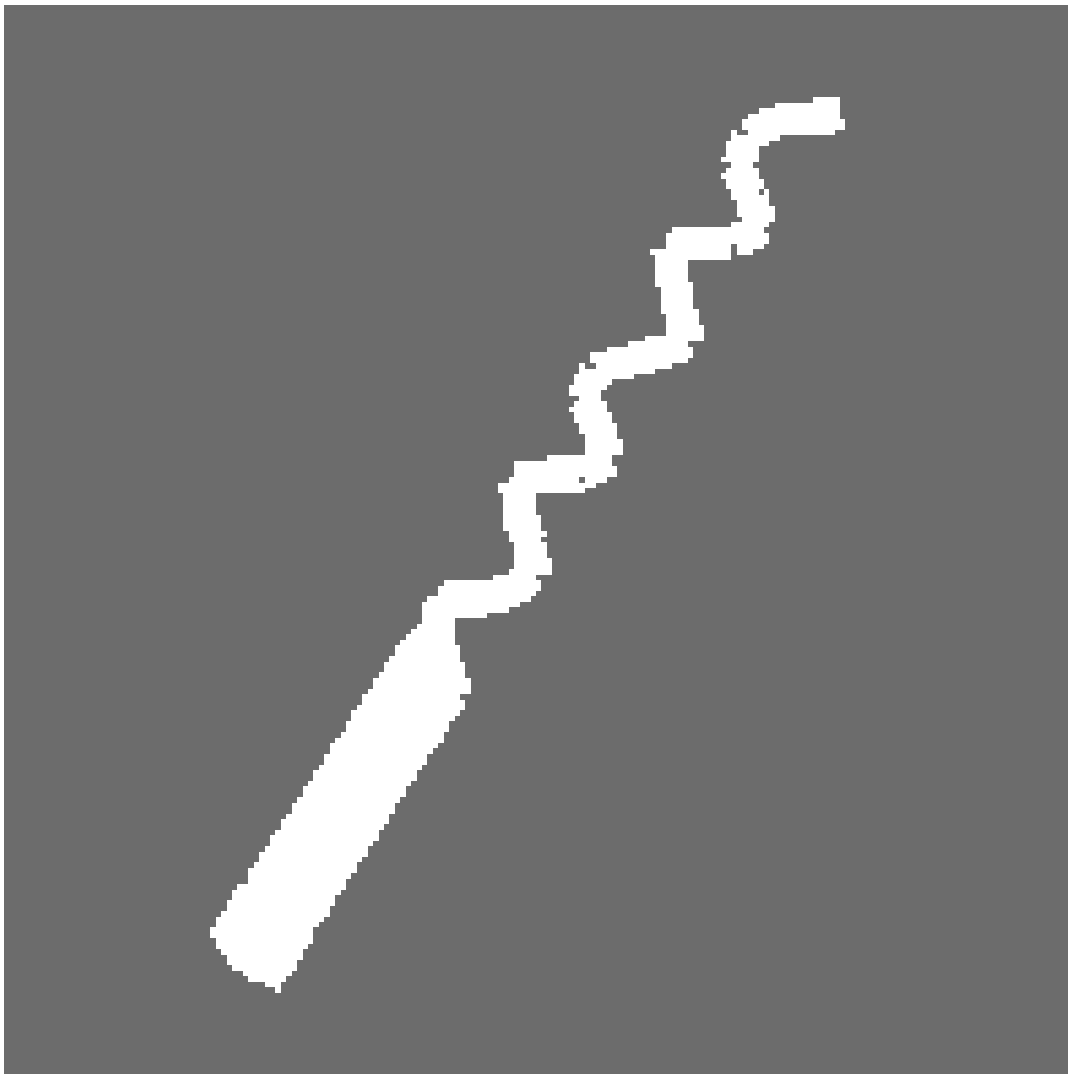


Figure 2.3: Single forward sweep.

To create our set of points P, for each pixel (i_x, i_y) of the occupancy map M_p whose state is *free}, convert the pixel to Cartesian space point (p_x, p_y) using Equation Equation 2.6 and add to P. Using any convex hull algorithm, produce the resultant polygon in Cartesian space. For our work, we use the convex hull algorithm available in CGAL \cite{cgal:hs-chep2–12b}.

An example of the convex hull operation on our free space data set appears in Figure 2.5. We can see for the case that the snake is in a straight pipe, the convex hull

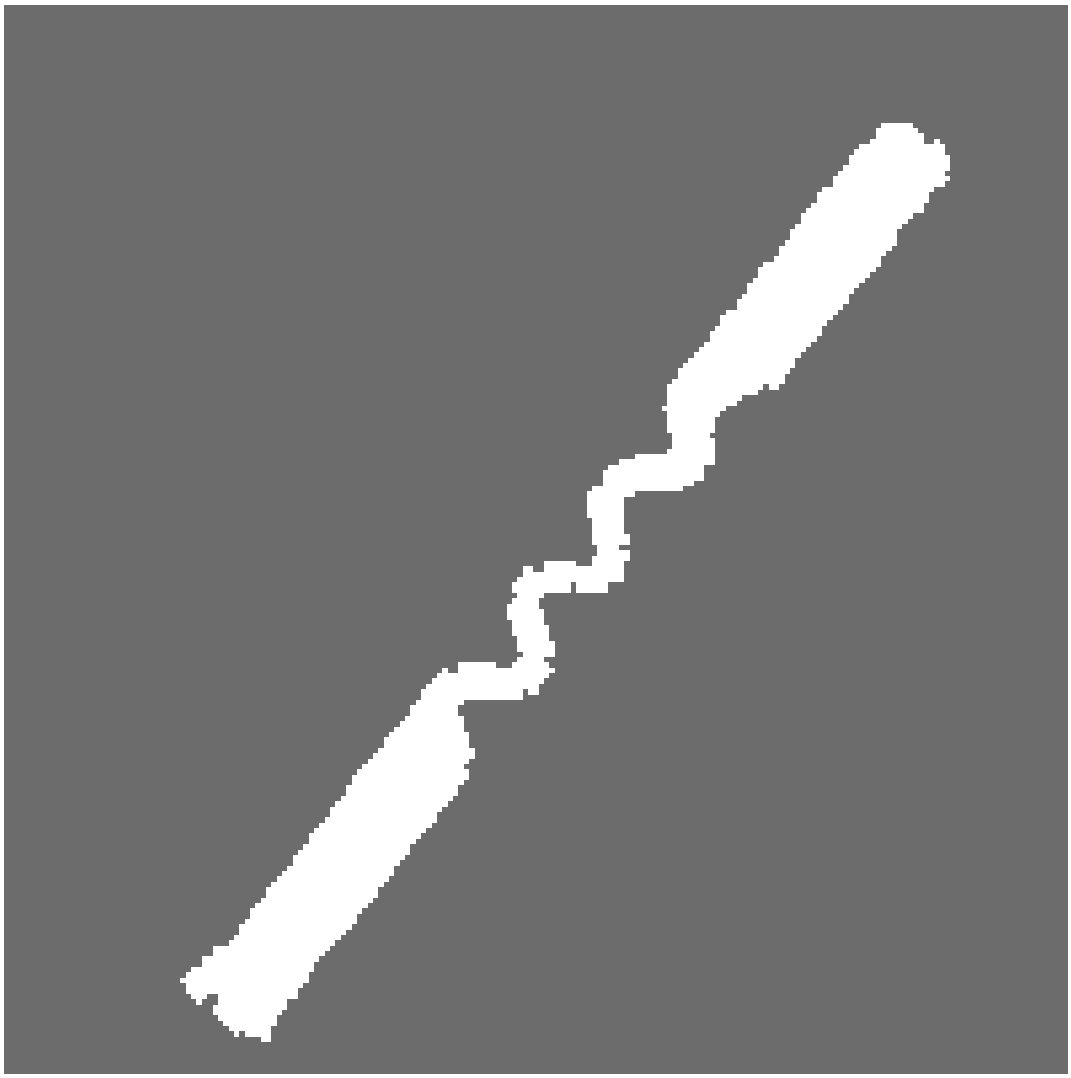


Figure 2.4: Forward and backward sweep.

creates a natural representation of the local pipe environment, filling in the blanks of our blind spot. However, if the pipe is curved as in Figure 2.6, the convex property of the polygon necessarily erases any concave properties of the environment. This also removes any sharp features of the environment such as corners. Therefore, any use we may have for the convex hull will be in limited situations.

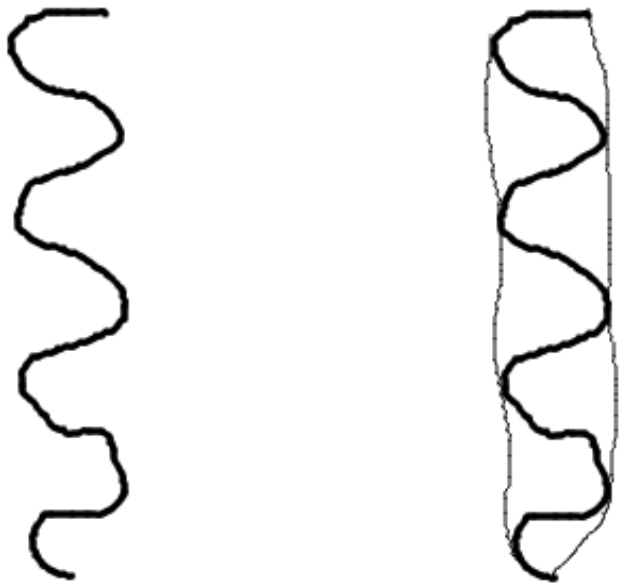


Figure 2.5: Free space data before and after convex hull.

2.4.2 Alpha Shape

An alternative to the convex hull is the alpha shape \cite{Edelsbrunner:1983p772}. Like the convex hull, it creates a polygon or polygons that contain all the points. Unlike the convex hull, it can construct a containing polygon with concave or even hole features.

To construct the alpha shape, first we choose a radius r for a circle C. Next, if a pair of points p_i and p_j can be put on the boundary of C where no other point is on or contained by C, we add an edge between p_i and p_j. The result is seen in Figure 2.7

By changing the radius size, we can tune how much smoothing we want to occur in the resultant polygon, seen in Figure 2.8. If we set the radius to $r = \infty$, the alpha

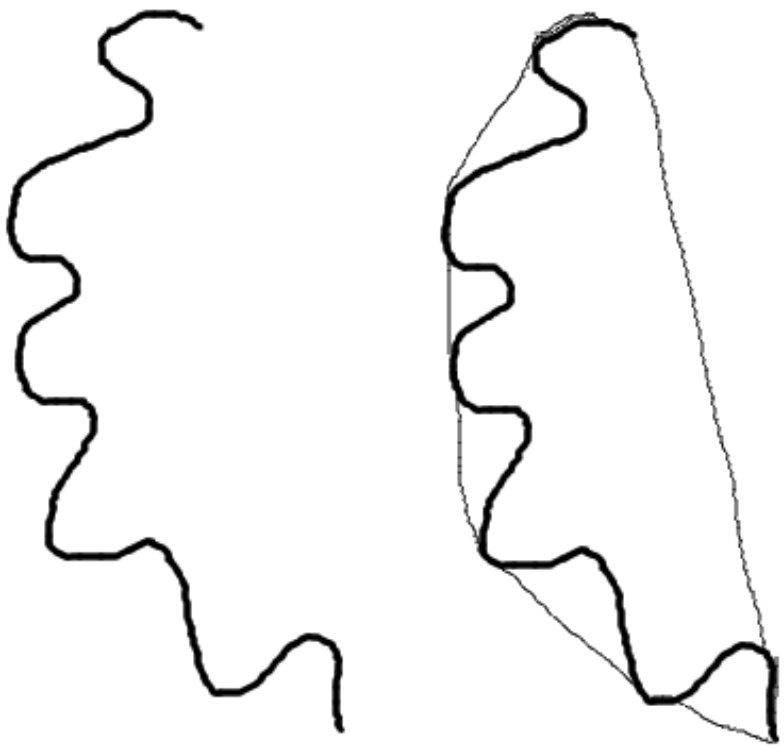


Figure 2.6: Free space data of curved posture before and after convex hull.

shape reduces to the convex hull. Therefore, this is a kind of "generalized convex hull" algorithm.

The benefit to this approach is that we can now capture the concave features of some of our local maps where the convex hull failed. Where Figure 2.6 shows the convex hull, Figure 2.9 shows its corresponding alpha shape. In both cases, the blind spot it smoothed over and filled in. This approach still suffers from the loss of sharp salient corner features, but the gross topology of the local environment has been captured.

For clarity, we refer to the alpha shape of our local free space as the *alpha hull* from here on out.

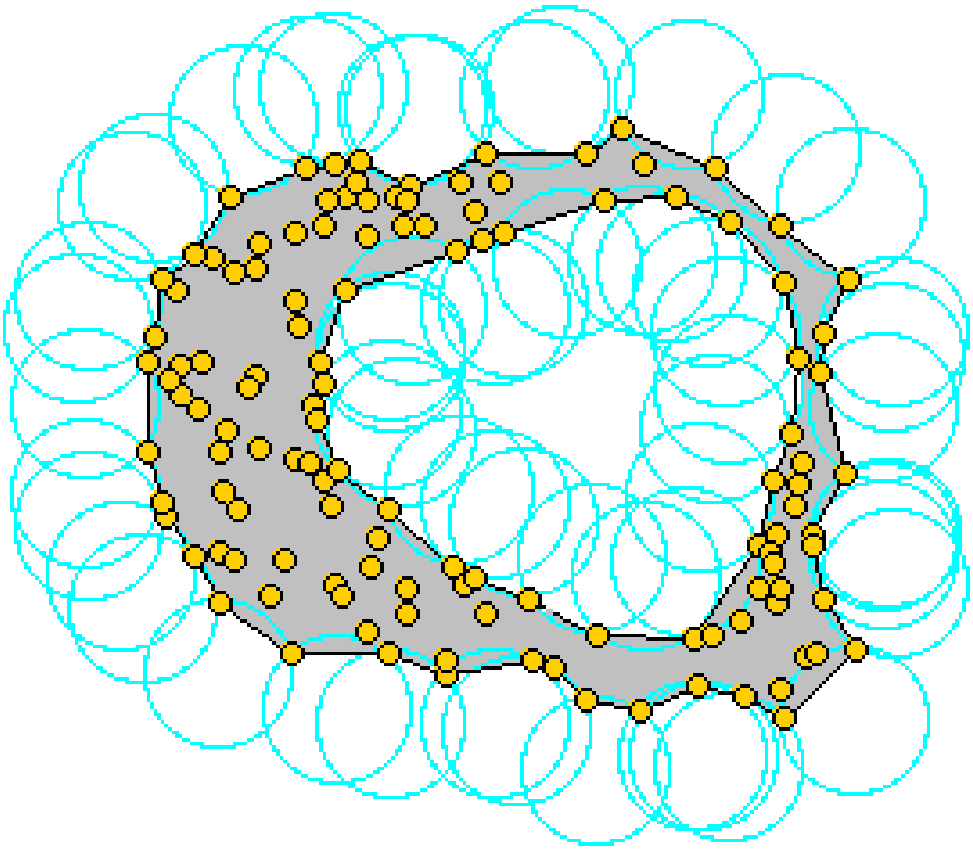


Figure 2.7: Alpha Shape example from \cite[cgal:d-as2–12b]

2.4.3 Medial Axis

Although polygons that represent the sweeped free space are useful, we need something that is more compact that also filters out the smoothing effects of blind spots and sharp features. That is, we want an approach where the negative effects of missing data, erroneous data, and loss of salient features are minimized.

For this purpose, we compute the medial axis, also sometimes known as thinning or skeletonization [cite]. This concept takes a shape and reduces it to a path or series of edges that follow the "spine" of a shape. Approaches range from a strict computational geometry approach such as the Voronoi diagram to image processing approaches of image

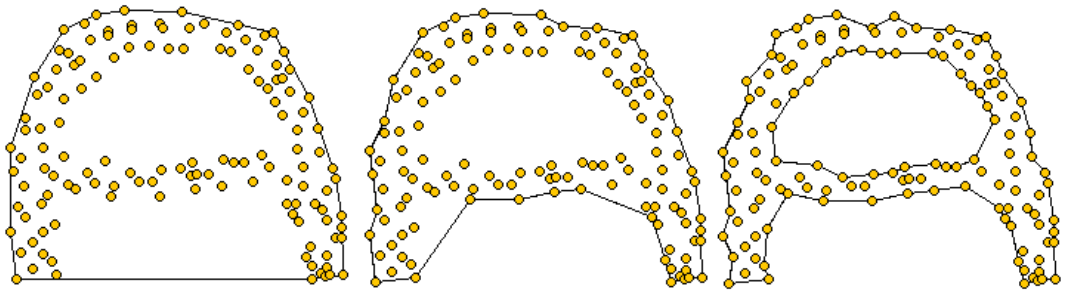


Figure 2.8: Alpha Shape changing radius from \cite(cgal:d-as2–12b)

thinning and skeletonization that erode a pixelized shape until the skeleton is all that remains [cite].

We are interested in extracting the medial axis of the alpha hull generated from our free space map. Since the alpha hull smooths the boundaries of the free space map, generating a medial axis will give us a topological representation of the free space that is resistant to boundary features. Given the free space map in Figure 2.10a, and its corresponding alpha hull in Figure 2.10b, the alpha hull's conversion back to an image in Figure 2.10c, and its resultant medial axis shown in Figure 2.10d. We use the skeletonization algorithm from [ParkerJR] \footnote{C/OpenCV code written by Jose Iguelmar Miranda, 2010}.

sWe can see that the medial axis roughly corresponds to the topological representation of the pipe environment. However, it is crooked, pixelated, and does not extend the full length of the data set. We further treat this data to give a more useful representatation.

Starting with the result from Figure 2.10d shown in Figure 2.11a, we create a minimum spanning tree (MST) graph where each node is a pixel from the medial axis and an edge is added between two nodes if their corresponding pixels are neighbors, shown in Figure 2.11b.

This MST has many leafs due to the artifacts of pixelation. We can simply prune the whole tree by sorting all nodes by degree, and removing the nodes with 1 degree and

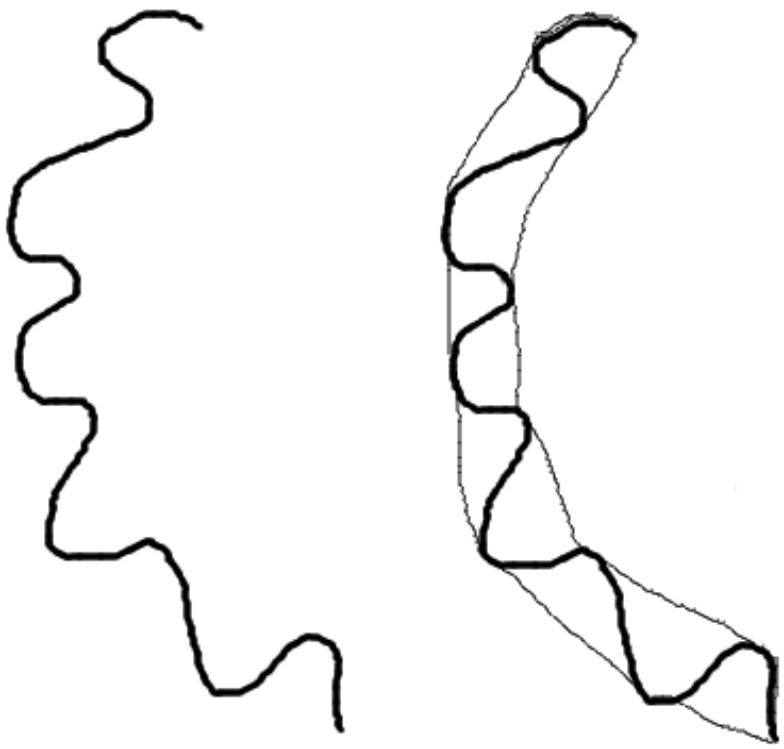


Figure 2.9: Alpha hull of free space data.

their corresponding edges. In most cases, this will leave us with a single ordered path of nodes with no branches. In some cases, we will have a branch if the original free space map shows evidence of a junction.

In the case of a single path, our next step is to extend this path to full extent of the original shape. We begin by fitting a B-Spline curve to the ordered path. With an appropriate smoothing parameter, the spline curve will follow the path of the ordered points but will ignore the jagged edges from pixelation. We then uniformly sample points along this curve to convert it back to an ordered set of points. We extend this path in the front and back by extrapolating more points along the direction specified

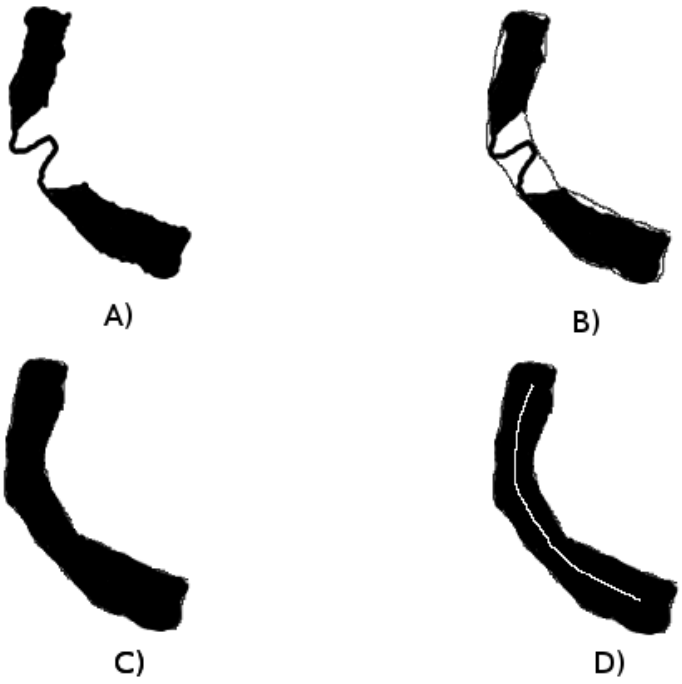


Figure 2.10: Process of generating medial axis.

by the curve tip, as in Figure 2.11c. Multiple tangent vectors are sampled along the tip neighborhood and their direction is averaged to get a direction that is immune to spline curving artifacts at the terminals. Finally, the series of points is cut off once the extrapolated points cross the alpha hull boundary, shown in Figure 2.11d.

In the case that we have a branch, we perform this series of operations for the path between each combination of pairs of terminals. For each medial axis of a free space map, there are $n = 2 + B$ end points where B is the number of branches. The number of unique paths of a medial axis is found by $\binom{n}{2}$ since we are using the MST and there is a unique path between each pair of points.

The final form is a compact representation of the topology of the local free space that removes the noise and uncertainty of the boundary and allows us to only represent

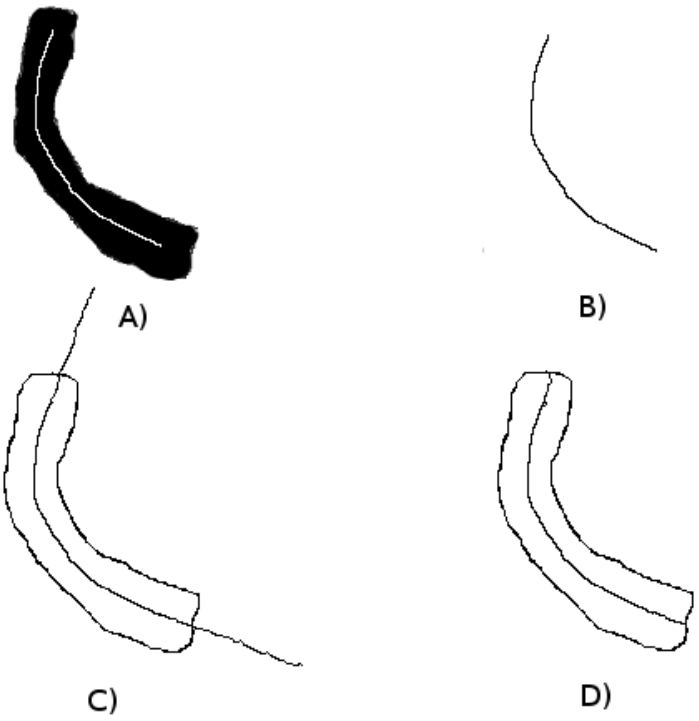


Figure 2.11: Process of generating medial axis.

the area we can move through. This has a number of uses for our map-making that we describe in the next chapter.

2.5 Managing Slip

Since the possibility of error occurring in our free space maps is very real and has the consequences of making the data near-useless, we want to do all we can to prevent and mitigate any failure conditions. As the primary source of error is anchor slip, we do all we can to focus on this issue while probing the environment. We use a number of techniques for both prevention and detection of error. We describe all of our approaches below.

2.5.1 Slip Prevention

We have 6 strategies for preventing anchor slip during the course of probing the environment. These are the following:

1. Prescriptive Stability
2. Local Stability
3. Smooth Motion
4. Averaging Reference Poses
5. Separation of Sweep Maps

The first three we have discussed earlier, with prescriptive stability and local stability in section section 4.2 and smooth motion in section section 5.7. We use prescriptive and local stability to determine which reference poses are available to be activated. Whereas, smooth motion through the use of linearly interpolated steps by using the Transition behavior reduces the chance that our anchors will slip.

The prescriptive stability selection of the PokeWalls behavior requires the robot to only use the anchored portion of the snake body for reference poses. In particular, it is very conservative, where only the segments at the very back and mechanically isolated from any of the sweeping motions in the front will be used for reference. This reduces the chance that any perturbations from the probing motions will have an effect on any of the active reference poses that we are using.

Averaging Reference Poses

For our fourth strategy, when actually using the reference poses to compute the position of the snake in our local free space map, we want to use as many of the reference poses as possible while filtering out any errors that would occur from possible pathological cases. If just one reference pose is erroneous and we use that reference pose to compute position of the snake in free space, it will severely damage our results. We want to

mitigate the possibility of negative effects by taking the average of a set of reference poses when computing a snake's position.

For instance, if we wanted to compute the origin of our local free space map in global space, we would compute the kinematic position of joint 19 with respect to some active reference pose P_k on joint and segment k. Using equation Equation 4.12 or Equation 4.13, we compute the position of P_{19}^k. For every P_k we use, we compute a new P_{19}^k that is possibly different. We then take the average of all of the computed P_{19}^k poses and take that as our best guess for P_{19}.

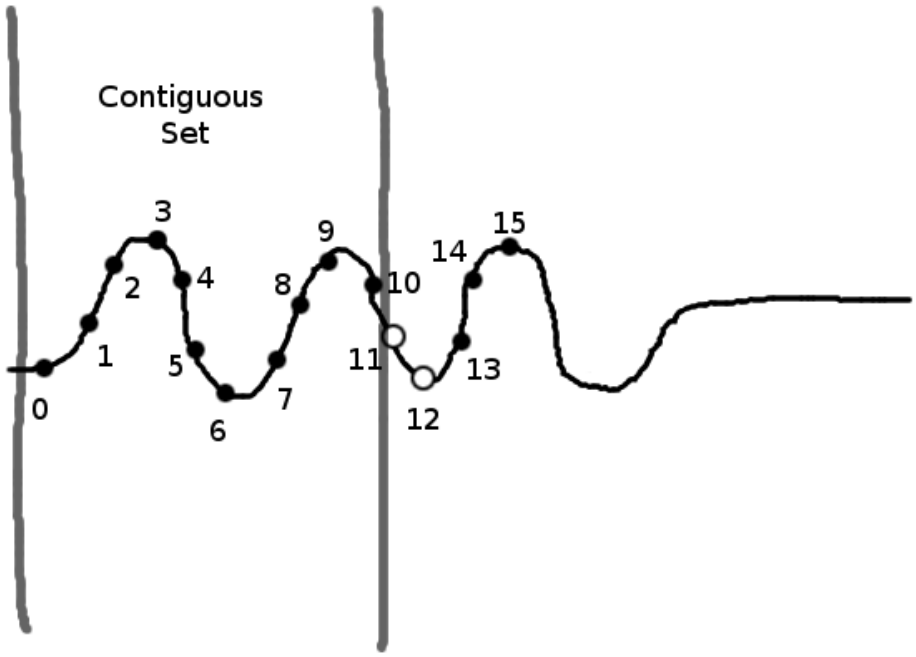


Figure 2.12: Largest set of contiguous reference poses.

How do we select the set of active reference poses from which to compute our estimated snake pose? Of all available active reference poses, our approach is to use the largest set of contiguous poses to compute the average target pose P_{19}. Our assumption is that if we have a series of reference poses from 0 to 15, and 11 and 12 are deactivated, it is highly possible that 13, 14, and 15 are invalid because whatever caused 11 and 12 to be activated will likely have an effect on its neighbors. The larger section from 0

to 9 has less likelihood of being corrupted since more of our sensors indicate stability. Furthermore, if one or two of these reference poses are corrupted, having a larger set reduces the weight of an erroneous value. This example is shown in Figure 2.12.

Separation of Sweep Maps

Our fifth and final strategy for preventing errors in our sensor maps is to divide the forward sweep phase and backward sweep phase into separate maps. From our experiments, we determined that a consistent source of error occurs when switching the anchoring responsibility from the back half of the snake to the front half. The series of events of retracting the front half of the snake to anchors and extending the back half for sweeping tends to cause some discontinuity between the consensus of the back reference poses and the front reference poses. This will show with a very distinct break in the combined free space map.

To avoid this problem, we instead create two free space maps: one for the front sweep and one for the back sweep. What was previously shown in Figure 2.13a will now become the pair of maps shown in Figure b. Not only does this avoid corrupting the free space data in the map, but it allows us to treat the relationship between the two maps as a separate problem for which there are multiple solutions. Our previous sensor processing algorithms work just as well because the alpha shape of the half-sweep free space map will result in an alpha hull from which an equivalent medial axis can be extracted.

We discuss our approach to managing the relationship between the two half-sweep local maps in the next chapter.

2.5.2 Slip Mitigation

In the case that error is introduced into our local free space map, we are interested in mitigating its negative effects. We have developed two approaches for handling error when it occurs during sensing.

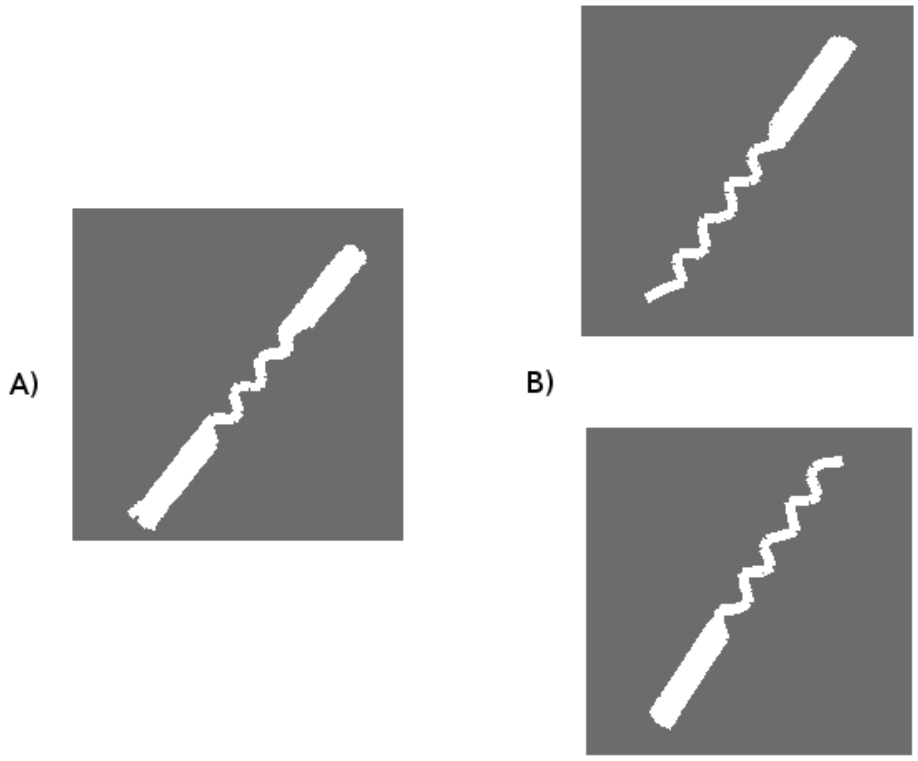


Figure 2.13: Separation of Sweep Maps.

Since we start our mapping process by determining the global pose of the local origin P_{19}, error occurs when P_{19} is no longer in its proper place. Either it moves by translation or more commonly and critically, it experiences rotational error.

Reference Stabilization

A sudden rotation of P_{19} may occur, caused by any of the reasons detailed in Section section 4.2. The consequences of rotational error is that the entire body of the snake will rotate around the origin within the local map. This will result in a map feature shown in Figure 2.14.

We proactively combat this possibility by running a reference stabilization algorithm that continually corrects the local origin O_t for P_{19} in the local map at time t. For $t = 0$,

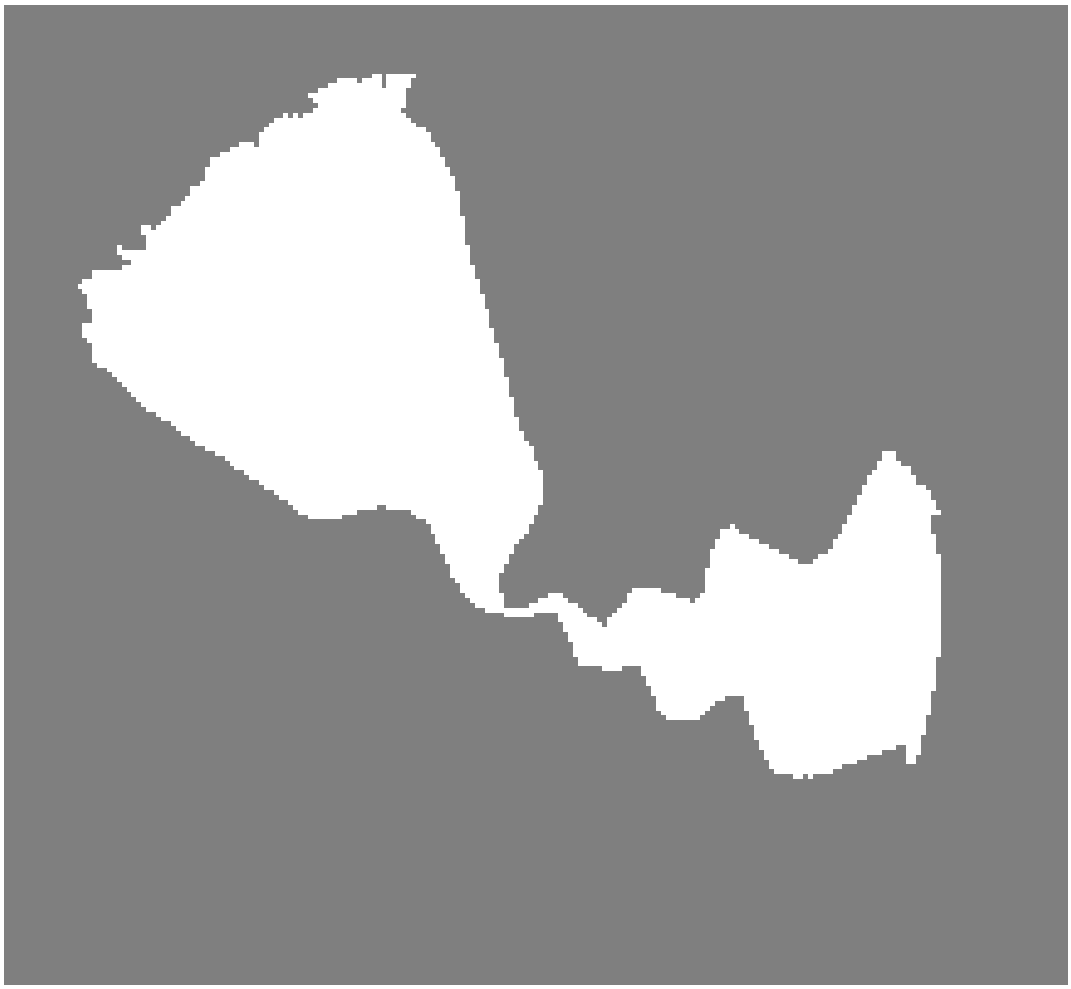


Figure 2.14: Local map rotational error.

the origin is $(0, 0, 0)$ for inputing into the equations Equation 2.11 and Equation 2.12. However, for each time step, we wish to calculate an O_t that is the most "stable" in case P_{19} becomes destabilized.

To do this, we remark that during the probing phase, the back anchored section of the snake as a whole will remainly roughly in the same tight space even if its joints and ssegments should wobble and slip. Short a large amount of translational slipping down the pipe, taken as a whole, the body should remain in the same place. Therefore, in the local free space map, the back anchored portion should of the snapshot time t should also remain in the same neighborhood of the back anchor snapshot at time $t - 1$.

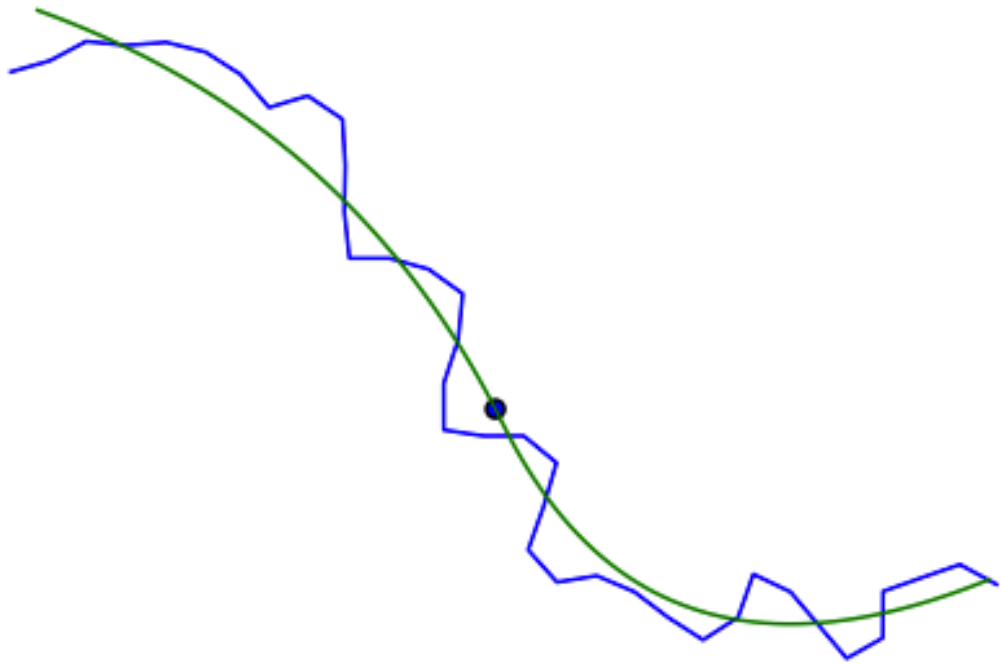


Figure 2.15: Gross Posture Approximation Curve of sample posture.

We enforce this observation by fitting B-spline curve β_t and β_{t+1} along the points that make up the reference poses of the back anchor segments for both the current and previous snapshots. The B-spline curves are calibrated to smooth out the sinusoidal anchoring posture of the snake body and instead reflect the gross posture of the body in the confined environment as shown in Figure 2.15. We then find a $(\delta x, \delta y, \delta\theta)$ for β_{t+1} such that the two curves are aligned and then we update O_{t+1} to reflect this.

This approach prevents sudden rotational errors from occurring and corrects the snake's estimated pose in the local map while we are capturing sensor data by generating a new O_{t+1} that is consistent with past data. So even if the snake's body is wobbling or slipping, we will likely produce consistent sensor data.

Should the data be too hopelessly corrupted from a bow tie effect in the posture image, we can reject the overall data and take the instantaneous shot of the current

posture as the only data. This gives a sparse snapshot of the environment and leaves the exploratory data out, but if will capture the gross shape of the local environments.

Chapter 3

Environmental Landmarks

How to build features and landmarks from void space data? Need to build a map and localize against. Most feature techniques use boundary information as primitives for environmental features. Need approach that uses only void space.

Features to build the map

Boundary features Obstacles/walls Corners

Void space features Shape features Spatial features

3.1 Finding Landmarks

Seeking landmark features for which we can correct the map and localize the robot. Most approaches use point features such as corners, environmental landmarks such as doors, visual features. Our options are very limited because we do not have sensing at a distance capability. All things that we can sense must be within touching distance. Any feature we choose is highly local.

One option is to find corner features by probing the walls. There are two approaches: 1) deliberate contact probing of the wall to produce corner features, 2) serendipitous corner features. Problem: 1 is feasible but slow, see [Mazzini]. 2) is also possible but the false positive rate is too high. Pull some data out of my archives.

Also try and find obstacles and openings . That is, non-obstacle openings. This presupposes that we can distinguish obstacles. Only contact probing methods can truly classify an obstacle boundary. Again, these methods suffer from time-consuming actions. Furthermore, this does not exactly give us a true positive for open space. So door openings cannot be a landmark since we cannot detect them directly.

What about the walls of the environment? The shape of the obstacles? This requires that we fully mark the boundaries of walls and obstacles. Either this is done through laborious contact detection or we make assumptions about what are obstacles based on the data we receive serendipitously. Given a posture image, we could make varying degrees of assumptions about where the boundaries of the environment lie. None of them are likely to be correct, but we can use this information as a feature. However, this approach suffers when our probing actions are deficient for a particular pose. We receive a degenerate posture image that leaves out a lot of data.

Another approach we could take is a sort of approximation and averaging of boundaries of the posture images. This can be accomplished by the use of the alpha shape algorithm developed in [3]. It is a form of generalized convex hull algorithm that is capable of non convex polygons. This can give us our desired approximation effect for identifying the location of walls.

3.2 Corner Examples and Results

Corners are high false positive and too many false negatives.

Requires tuning of parameters, but does not guarantee correction selection.

Corners through serendipitous detection are not seen very often. Will not be seen through consecutive poses or even being in the same location. Not guaranteed to view.

Information too sparse to be useful. False correlation is very costly.

3.3 Wall Detection

Contact detection literature. Reference 2009 paper.

Time spent probing walls. Amount of information is broken up. Example information should walls from contact detection.

Punch probing. Hit the wall and measure the error incurred by the joints in the control loop. Avoids the need for torque sensor. Uses existing control algorithm parameter as error terms.

Broken information, time-consuming. Error prone and often will mark free space as wall. Not detailed enough for a landmark feature.

3.4 Different Macro Features

The possibilities for curves representing the overall macro structure of the local environment are as follows:

1. Curves fitting through the anchor points of the articulated robot. Anchor points are the confirmed contacts and can approximate the walls. However, these are sparse data points, and cannot handle complex environments such as junctions or gaps between the anchor points. Data is sparse and not well-representative of the environment.

2. Polygon created from alpha shape of posture image. Use the edges of the polygon and perform scan-matching of edges to edges. Problem, how to select the points pairs between two polygons. Use angle filter and match two sides together. Is sensitive to missing data. Unsweeped gap will distort the polygon. Performing matches on a distorted polygon will result in poor ICP fitting.

3. Given alpha shape, compute the medial axis of the interior. This is known to be a reduction and representation of the the space. This approach is resistant to noisy edges. Shown are some example of the results with simple single path pipes, and pipes with some junction features. Notice that the information on the width of the environment is lost using this approach.

Shape of the sensed void space in local environment should give us information. How can we extract this information and represent it in an useful way?

Spatial curve ct = scurve(It) where It is posture image Smoothed version gives us a curve representation of the local space

3.5 Macro Feature Extraction

1. curve of anchor points, represents walls. Does not include sensed data

Posture Image -> Alpha Shape -> Medial Axis -> Characteristic Curve

Medial axis is generated from an image and a tree is built. I most cases the tree has only two leafs and is a path graph, but sometimes the tree has more than two leafs and represents more complex space.

Represents local description of space.

Like scan-matching, we can use curves and ICP to find an alignment of the poses. The curves themselves becomes the landmarks.

3.6 Pipe Shape, Macro Features

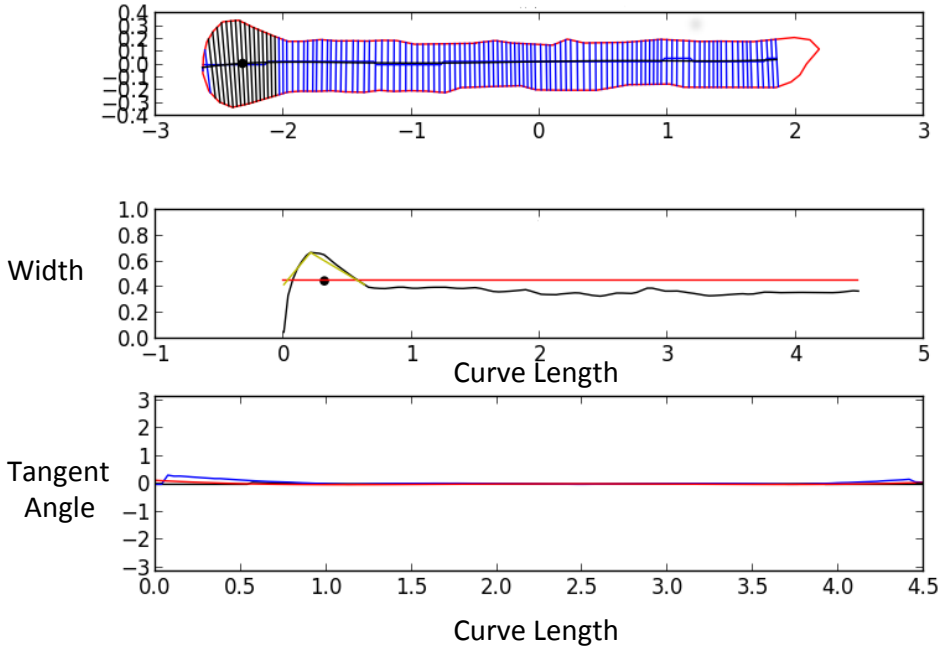
Use the contours and shape of the occupied space as our landmark feature.

Introduces strong lateral correlation, but weak forward-backward correlation. If the local environment is curved in any way, this provides stronger localization.

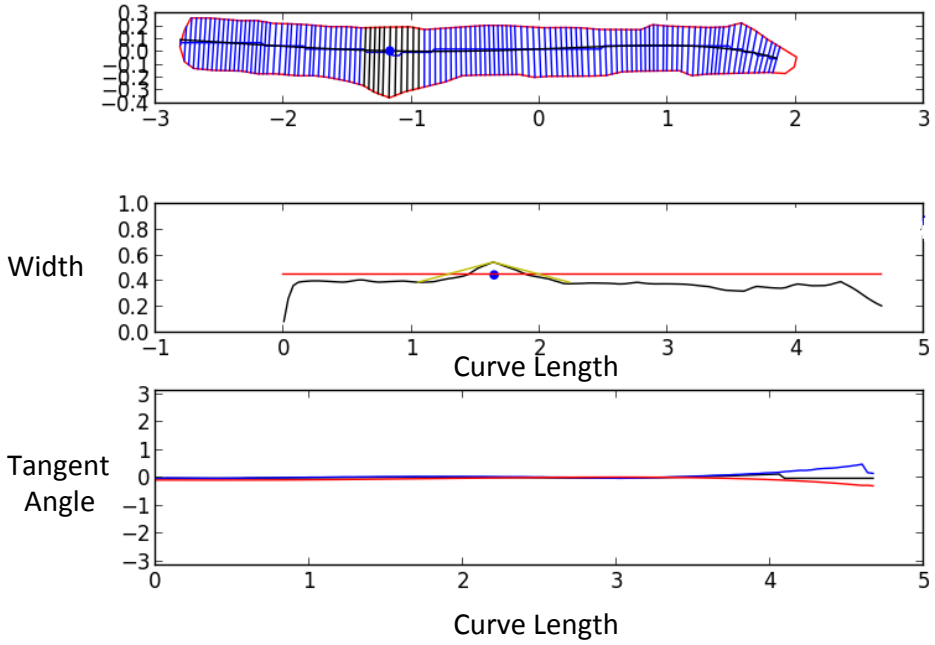
1st order curve is straight line. Only coaxial localization between poses 2nd order curve is a constant curve. Localization to any place with same curve constant. *3rd order curve is changing curvature. Localization to unique point with particular change profile.s

3 different spatial features from posture images blooms arches bends Landmarks usually indicate junctions or gaps in space Can match landmarks together to improve the map

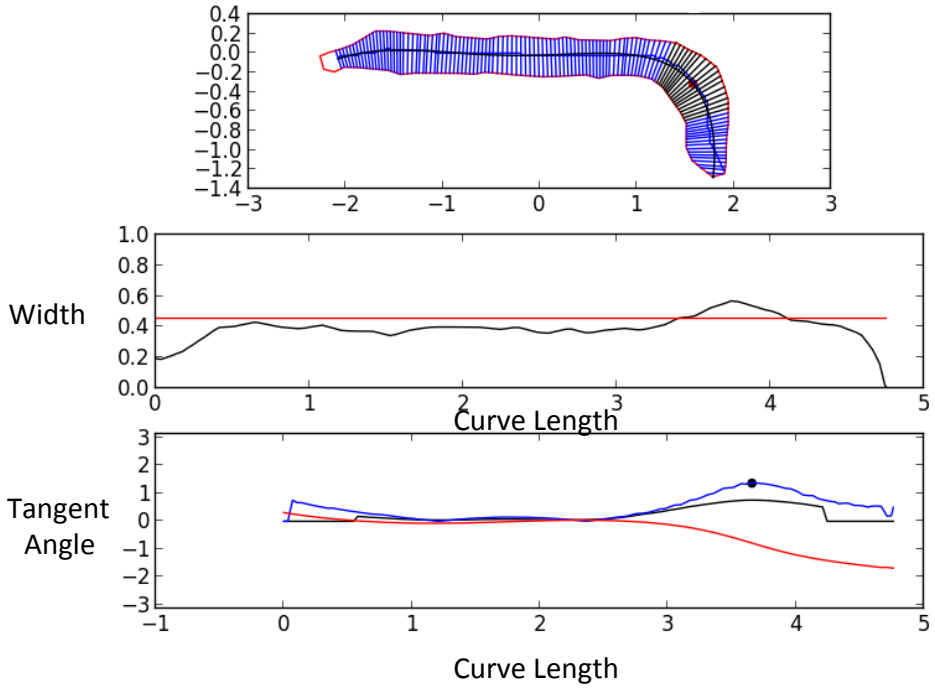
Blooms: swelling at the tip (assumes constant width)



Arches: internal swelling, "arched" anchors (assumes constant width)



Bends: sudden turn of *spatial curve*, crest is the bend point (permits dynamic width)



Chapter 4

Defining Position

4.1 Problem

Now that our snake robot is capable of moving through the environment with the adaptive concertina gait behavior, we need some way of estimating how far it is moving and in which direction.

As we established earlier, we have no external sensors because we assume their failure from environmental conditions. We cannot rely on GPS since it is not effective underground or in pipes. We do not have wheeled shaft encoders since we have no wheels and do not expect to always travel through flat terrain. Finally, although it may indeed help, we cannot expect to make use of an inertial navigation system since they are bulky and expensive. Furthermore, we have not performed experiments to determine if they are capable of handling the high amount of rotation of each of the snake body segments during locomotion.

In absence of a global positioning or odometry, we need some form of alternative motion estimation. Knowing where the robot is located is the basic building block of creating a map. We focus on the problem of finding the relative geometric transform between the two poses at the beginning and end of one step of the adaptive concertina gait.

But we first need to define how we will represent the position and orientation of the robot in space.

Our proprioceptive approach achieves motion estimation by anchoring multiple points of the body with the environment and kinematically tracking the change in distance between the immobilized parts as the snake contracts and extends. This gives us a motion

estimation method that only depends on the joint sensors. We show the effectiveness in a series of experiments.

4.2 Immobilizing the Anchors

A fundamental property of the adaptive concertina gait is that it creates anchors to the surrounding environment. Our goal is to exploit this property to estimate our motion through the environment. In order to do this, we need to ensure that our anchors remain an accurate frame of reference to the global environment.

The primary goal of our previous work on maintaining smooth and stable motion of the snake in Chapter 2 was to prevent any sort of anchor slip that may harm our reference frame to the global environment. It is critical that our anchors remain firmly immobilized with respect to the environment to ensure accurate measurement of motion.

Though we are not able to localize the exact contact point of the anchor with the environment, we can assume that the segments comprising the anchoring curve are themselves immobilized and mechanically isolated by virtue of their static contacts with the environment. We use these isolated bodies as immobilized references to the global environment.

Whether or not we consider the use of a body segment as a reference is determined by our work in Section 2.8. That is, if the joints in the local neighborhood of a segment are locally stable and the behavior's control bit for the attached joint is 0, we assume this segment is immobilized. We can use immobilized segments as reference points to the global frame.

We call the monitoring of local joint variances as local stability and the output of the control masks as prescriptive stability. Though a body segment may be locally stable, it does not mean it is immobile. If we also include prescriptive stability, the behavior's prediction of what joints should be moving and what should not, then we greatly decrease the chances that we will use a slipping anchor as a reference.

If any point the anchor should slip while, our assumption that a body segment is immobilized is violated and could lead to accumulated errors in our motion technique. Therefore, it is critical that we analyze and mitigate all the possible failure modes for anchoring.

We are generally able to classify the types of anchoring and reference errors we encounter during both static and locomotive anchoring. For each, we are forced to either avoid the condition from occurring or detect and correct it. We list the failure modes and explain our mitigation strategy for each.

The *floating reference* occurs when a locally stable segment is made a reference, but it is not part of an anchor. This was particularly troublesome in our early implementations, but we are able to largely avoid this failure mode by having the behaviors explicitly classifying which joints are part of an anchor. The other possibility for a floating anchor would be if an anchoring behavior erroneously reports success in establishing an anchor in free space. This was largely avoided by adding the more careful anchoring process in Section 2.2 that detects impacts to the walls and follows up by doing a jerk test to test whether the anchor is secure.

A *weak contact* occurs when the anchor the reference segment is on has an insecure anchor. This usually happens if an anchor is placed within a pipe junction or if one of the anchors of the Back-Anchor is insecure. The Back-Anchor case is more likely since they are not jerk-tested. Weak contacts usually do not display local instability since the physical contact stabilizes the local joints but does not immobilize them. If it is available, we can use negative prescriptive stability from the behavior to minimize this failure but often occurs in prescriptively stable areas. There is nothing we can do to prevent this error from being added to the pose estimation besides detecting it. Luckily, a weak contact does not move much. Its error is bounded.

Buckling occurs when the anchored joints are frozen in a non-optimal position and external stresses put excessive torque on one or more joints that forces the anchor into a reconfigured position. This is similar to mechanical buckling of structural members

under compression. In this case, it is for an articulated system attempting to hold a rigid pose. Buckling is easily detectable by tripping our local stability conditions and will quickly remove the references and reuse them once the joints have settled down. We have also reduced this occurrence by having our anchoring behaviors search for non-deforming anchors as a natural by-product of algorithm 2.

Impulses occur when the snake robot experiences a sudden collision or a buckling has occurred somewhere else in the robot. Usually this is not a fatal error and the robot will quickly return to its stable pose. However it can trip up the variance calculations and deactivate all the references we have leaving us with none. Recovery is achieved by resurrecting the last best reference, and assuming it is in the same pose. We show how to do this in the next section.

Funnel anchoring occurs when a secure anchor is established between a pair of walls that are monotonically increasing in width. If the anchor receives a force that pushes it towards the direction of increasing width, its anchor can become permanently loose. This is especially problematic in cases of irregular environments where there are many possibilities for this to occur in local conditions. We currently do not have a solution for this yet.

Finally there is a class of *whole body slip* that is completely undetectable with only joint positions. That is, the whole body could be slipping through an environment such as a straight and featureless pipe without causing any joint variance at all. This would only occur if the ground was exceptionally slippery and able to use the robot's momentum to move it large distances or if the robot was on a slope and gravity was exerting a global force. We avoid this error by limiting the environments we explore. Other environments may require extra sensor instrumentation such as accelerometers to detect this failure mode.

Rotational error is the dominant form of error we encounter because, unlike a traditional robot with a stable frame of reference on its chassis, every reference point in the snake robot's body is constantly in rotational motion. Any form of disturbance of

a reference node will manifest itself with a rotational error that will propagate through kinematic computations of new reference nodes. Translational error occurs primarily by anchor slip. Rotational error is easier to correct in our later localization process in Chapter 5.

4.3 Tracking Motion

In order to track motion through the environment, we need to establish our mathematical framework for defining the relative position of things between the robot and the environment. We then define the concept of a *reference pose* to track the position of the environment while the snake is in motion.

4.3.1 Pose and Coordinate Frames

A pose P defines the position and orientation of a rigid body in a 2D plane. Each pose is defined with respect to a coordinate frame O using 3 values: (x, y, θ). A coordinate frame is either affixed in the global environment or affixed to a rigid body on the snake. An example of the global frame O_g and a pose P_k are shown in Figure 4.1.

By convention, we will specify in the superscript which coordinate frame a pose k is defined with respect to. For the global frame, the pose is written as P_k^g. If the pose is with respect to some other coordinate frame O_a, the pose is written as P_k^a. If the superscript is omitted, we assume the global frame.

In a robot system with multiple rigid bodies, we will have a coordinate frame for each body. Often times, we wish to transform poses between coordinate frames. To do this, we first need to define the relationship between coordinate frames, specified by the pose of their origins. In the global frame O_g, the pose of the origin O_g is $P_g = (0, 0, 0)$. For two example coordinate frames attached to rigid bodies, we define the pose of origin O_a to be $P_a = (x_a, y_a, \theta_a)$ and the pose of origin O_b to be $P_b = (x_b, y_b, \theta_b)$ as shown in Figure 4.2.

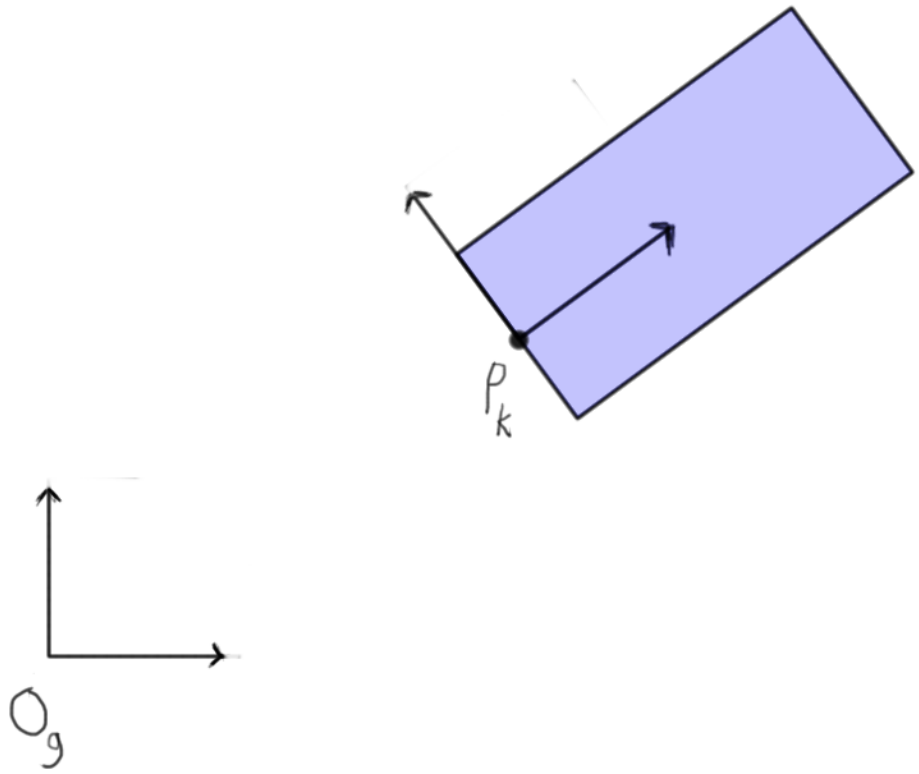


Figure 4.1: Pose of rigid body with respect to global frame.

Suppose we wanted to compute pose b with respect to frame O_a. That is, we wish to find P_b^a and we are given P_a^g and P_b^g. First we focus on finding the Cartesian component of the pose, point $p_b^a = (x_b^a, y_b^a)$. We compute the angle portion θ_b^a separately. From Figure 4.2, we define the following:

$$R_a = \begin{bmatrix} cos(\theta_a) & sin(\theta_a) \\ -sin(\theta_a) & cos(\theta_a) \end{bmatrix} \tag{4.1}$$

This is the rotation matrix for the difference in orientation between the global frame O_g and local frame O_a.

$$d_a = \sqrt{(x_a)^2 + (y_a)^2} \tag{4.2}$$

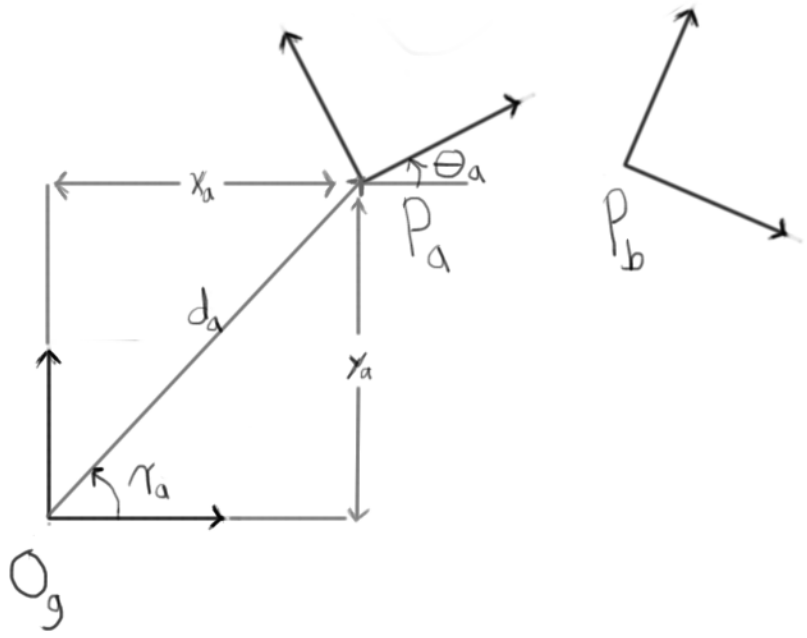


Figure 4.2: Pose of A and B with respect to global frame.

This is the Cartesian distance from the global frame's origin to the local frame's origin.

$$\cos(\gamma_a) = \frac{(x_a, y_a) \cdot (1, 0)}{|(x_a, y_a)||1|} = \frac{x_a}{d_a} \tag{4.3}$$

$$\gamma_a = s \cdot \arccos\left(\frac{x_a}{d_a}\right) \quad \begin{cases} s = 1 & \text{if } y_a \geq 0 \\ s = -1 & \text{if } y_a < 0 \end{cases} \tag{4.4}$$

γ_a is the angle of the vector from the global origin to the local frame origin. The sign of γ_a is determined to be negative if $y_a < 0$. Otherwise, γ_a is positive. This value corresponds to the angle shown in Figure 4.2.

$$G_a = \begin{bmatrix} cos(\gamma_a) & sin(\gamma_a) \\ -sin(\gamma_a) & cos(\gamma_a) \end{bmatrix} \tag{4.5}$$

G_a is the rotation matrix to rotate the local frame's origin onto the x-axis of the global frame.

To find p_b^a in frame O_a from p_b^g in O_g, we compute:

$$p_b^a = \begin{bmatrix} x_b^a \\ y_b^a \end{bmatrix} = R_a {G_a}^{\mathbf{T}} \left(G_a p_b^g - \begin{bmatrix} d_a \\ 0 \end{bmatrix} \right) \tag{4.6}$$

To find the converse, p_b^g from p_b^a, we do the following:

$$p_b^g = \begin{bmatrix} x_b^g \\ y_b^g \end{bmatrix} = {G_a}^{\mathbf{T}} \left(\begin{bmatrix} d_a \\ 0 \end{bmatrix} + G_a {R_a}^{\mathbf{T}} p_b^a \right) \tag{4.7}$$

This approach performs a sequence of rotations to separate translation and rotation of the pose when converting between frames. To compute the angle portion of the pose, we perform the rotation sequence by addition and subtraction of angles. Following the sequence, we get the following:

$$\theta_b^a = \theta_b^g + \gamma_a - \gamma_a - \theta_a$$

This reduces to:

$$\theta_b^a = \theta_b^g - \theta_a \tag{4.8}$$

The converse is achieved similarly using Equation Equation 4.7 and the following rotation sequence:

$$\theta_b^g = \theta_b^a + \theta_a + \gamma_a - \gamma_a$$

$$\theta_b^g = \theta_b^a + \theta_a \tag{4.9}$$

The final result for the pose computation is to put the Cartesian and angular components together.

$$P_b^a = \begin{bmatrix} x_b^a \\ y_b^a \\ \theta_b^a \end{bmatrix} \quad \begin{cases} x_b^a, y_b^a & \text{from Equation 4.6} \\ \theta_b^a & \text{from Equation 4.8} \end{cases} \tag{4.10}$$

And the converse is:

$$P_b^g = \begin{bmatrix} x_b^g \\ y_b^g \\ \theta_b^g \end{bmatrix} \quad \begin{cases} x_b^g, y_b^g & \text{from Equation 4.7} \\ \theta_b^g & \text{from Equation 4.9} \end{cases} \tag{4.11}$$

Now that we have established the mathematical framework for relating the pose of the rigid bodies to the environment and each other, we describe our concept of reference poses to track motion in the environment.

4.3.2 Reference Poses

A reference pose is the position and orientation of a rigid body segment that has been immobilized with respect to the global frame. A reference pose is valuable because it gives a fixed reference point to the global environment and allows us to track the motion of the rest of the robot. Special care must be taken to ensure that when using a reference pose, the rigid body it's attached to is truly immobilized.

A reference pose is either active or inactive. A reference is active once it is created as a result of its rigid body becoming immobilized. We assess whether a reference pose is to be created by checking if it satisfies both our predictive stability and local stability conditions. We discussed predictive and local stability in section section 5.10, where predictive stability is the output of the behavior's control masks and local stability is the result of monitoring local joint variances. Once the reference pose violates either of these conditions, it becomes inactive.

To deactivate a reference pose, if a reference violates either of the stability conditions and it was previously active, we flip a bit to signal it as inactive. Afterwards, we can create a new reference pose on this rigid body once it satisfies the stability conditions again.

To create a new reference pose, its associated rigid body must first have been inactive. Secondly, it must satisfy both prescriptive and local stability conditions. Once the following conditions have been met, the new pose is computed kinematically with respect to another currently active reference pose. If the pre-existing reference pose is correct, the kinematic computation of the new reference pose will be correct for its true position and orientation in the global environment.

Given the arrangement of rigid bodies and their attached coordinate frames shown in Figure 4.3, we need the kinematic equations for computing the pose of a rigid body given its neighbor's pose. That is, if we already have an active reference pose, we wish to compute the pose of neighboring reference in the global frame using only the joint angles from the robot's posture. To compute P_{k+1} if we are given P_k, we do the following:

$$\begin{aligned} x_{k+1} &= x_k + l\cos(\theta_k) \\ y_{k+1} &= y_k + l\sin(\theta_k) \\ \theta_{k+1} &= \theta_k - \phi_{k+1} \end{aligned} \tag{4.12}$$

For the opposite direction, to compute P_{k+1} given P_k, we do the following:

$$\begin{aligned} \theta_k &= \theta_{k+1} + \phi_k \\ x_k &= x_{k+1} - l\cos(\theta_k) \\ y_k &= y_{k+1} - l\sin(\theta_k) \end{aligned} \tag{4.13}$$

If we use the equations iteratively, we can compute the pose of a rigid body segment given the pose of any other segment.

The algorithmic process for creating and deactivating reference poses is shown in algorithm 3, where N is the number of rigid body segments in the snake from which to

create reference poses. The implementation of the function for the kinematic computation of new reference poses is shown in algorithm 4.

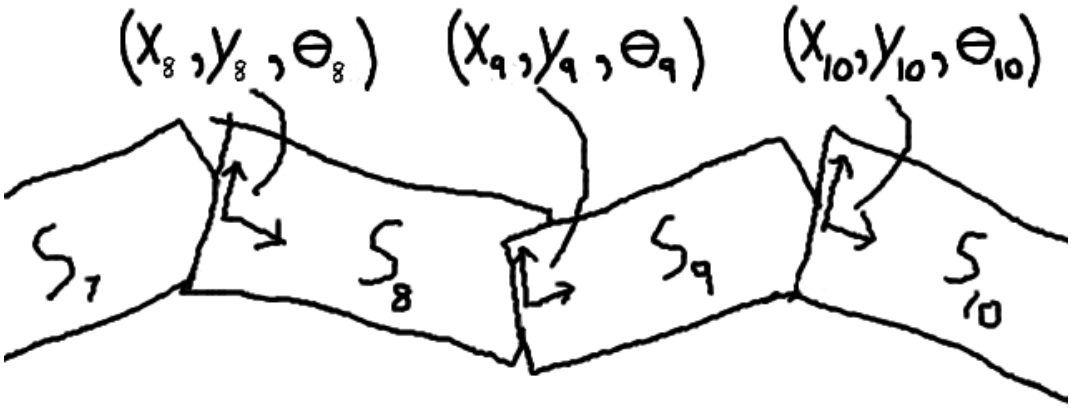


Figure 4.3: Local coordinate frames attached to segments and described by reference poses.

This algorithm assumes that an active reference pose always exists. There are two special cases when no reference poses are currently active. In the first case, the algorithm is initializing and no reference poses have been created yet. In the second case, all of the reference poses violate the stability condition and all have been deactivated. These special cases are not shown in the algorithms, but we explain are approach here.

For the first case, when we initialize, the first reference pose is set to $(0, 0, 0)$. This becomes the origin of the global frame centered at the robot's starting position in the environment. From this first initial reference pose, all new reference poses are derived.

In the second case, no active reference poses are available since no stability conditions have been satisfied. The robot has effectively lost track of its position in the environment. In the event that a new reference pose needs to be created, there is no parent pose from which to kinematically compute. Instead, from the last used inactive reference poses, we

Algorithm 3 Reference Pose Creation and Deactivation

$N \Leftarrow 40$
$activeMask \Leftarrow$ array of size N initialized to 0
$activeRefPoses \Leftarrow$ array of size N initialized to $\varnothing$
$preStabMask \Leftarrow$ behaviorControlOutput()
$locStabMask \Leftarrow$ stabilityCompute()
for $i = 0 \rightarrow N$ **do**
  if $preStabMask[i]$ & $locStabMask[i]$ **then**
    if !$activeMask[i]$ **then**
      $activeRefPoses[i] \Leftarrow$ computeRefPose(i)
      $activeMask[i] \Leftarrow 1$
    end if
  else if $activeMask[i]$ **then**
    $activeRefPoses[i] \Leftarrow \varnothing$
    $activeMask[i] \Leftarrow 0$
  end if
end for

Algorithm 4 *computeRefPose(i)*: Kinematics for Computing Reference Pose

$N \Leftarrow 40$
$\bar{\phi} \Leftarrow$ array of current joint angles
$i \Leftarrow$ target new reference pose index
$j \Leftarrow$ index of nearest active reference pose to i
$k = j$
$(x_k, y_k, \theta_k) \Leftarrow (x_j, y_j, \theta_j)$ ▷ pose of active reference j
if $i > j$ **then**
  while $k < i$ **do**
    $x_{k+1} = x_k + l\cos(\theta_k)$
    $y_{k+1} = y_k + l\sin(\theta_k)$
    $\theta_{k+1} = \theta_k - \phi_{k+1}$
    $k = k + 1$
  end while
else if $i < j$ **then**
  while $k > i$ **do**
    $\theta_{k-1} = \theta_k + \phi_{k-1}$
    $x_{k-1} = x_k - l\cos(\theta_{k-1})$
    $y_{k-1} = y_k - l\sin(\theta_{k-1})$
    $k = k - 1$
  end while
end if
$(x_i, y_i, \theta_i) \Leftarrow (x_k, y_k, \theta_k)$ ▷ new pose for active reference i

select the reference that was contiguously active the longest and use this as the parent pose to compute the new active reference pose. The reasoning behind this is that a long-lived reference pose is more likely to be correct than a short-lived reference pose due to the sometimes intermittent nature of reference poses. This last best reference pose approach is very effective in practice.

So long as the active reference poses satisfy the assumption of no-slip anchors and there always being at least one active reference, the tracking of the robot's trajectory through the environment is correct. Violations of these assumptions introduce error into the motion estimation.

4.4 Robot-Centered Coordinate Frame

Though we now have a local coordinate frame for each segment in the snake, we need a coordinate frame that represents the mobile robot body as a whole. In traditional mobile robots, we usually have a large rigid body chassis with high inertia that is suitable for defining the position and orientation of the robot. In a snake robot, there are multiple rigid bodies, all of which are small in mass, small in inertia, and poor at defining the overall snake's position and orientation. While the snake is executing its locomotion, the body segments are undulating and rapidly changing orientation, making dubious their use as an indication of heading.

In the wake of these phenomena, we found the need to create a dynamically generated coordinate frame that best represents the position and orientation of the snake robot's posture as a whole. We cannot use the center of mass of the robot, because the centroid can be outside of the body if the snake's posture is curved. Instead, we generate a curve called a Gross Posture Approximation Curve (GPAC) that we use to generate a local coordinate frame that is oriented in the direction of travel. This allows us to describe consecutive poses of the snake with local coordinate frames oriented in the same forward direction.

We describe in detail the GPAC and follow with the generation of the local coordinate frame.

Articulated robot has no high-inertia center of mass to serve as its coordinate frame. Require method to establish forward direction of an articulated robot.

Articulated robot has no high-inertia center of mass to serve as its coordinate frame. Individual segments are rotating and undulating Need method to determine direction of snake robot. Information not provided by individual body segments Posture curve generated by smoothed B-spline with joint locations as inputs Generates the posture frame for local coordinate system Different from spatial curve which represents local sensed environment

Establishes stability of robot frame under error of anchoring.

4.4.1 Gross Posture Approximation Curve (GPAC)

The GPAC is a curve that approximates the posture of the robot in the environment without the articulation, as seen in Figure 4.4. The blue lines indicate the posture of the snake and the green line is the GPAC.

The GPAC is derived by taking the ordered points of the snake segment positions and fitting a smoothed B-spline curve to them. To generate the points, we first start from a body frame fixed to one of our body segments. In this case, we use segment 19 centered on joint 19 as our body frame O_{19}. With respect to frame O_{19} and using the current joint posture vector $\hat{\phi}_t$, we compute the segment posture vector $\hat{\rho}_t$ that specifies the N segment poses using the iterative kinematic equations Equation 4.12 and Equation 4.13. If we take only the cartesian components of the segment posture $\bar{\rho}_t$, we set this as a series of ordered points from which we compute a B-spline curve.

We use the Python SciPy library's **splprep** function with a smoothness factor of 0.5 to generate the B-spline curve. This results in the curve shown in Figure 4.4. To use this curve, we define a set of parameterized functions that we use to access the curve. First we show the function provided by the SciPy library:

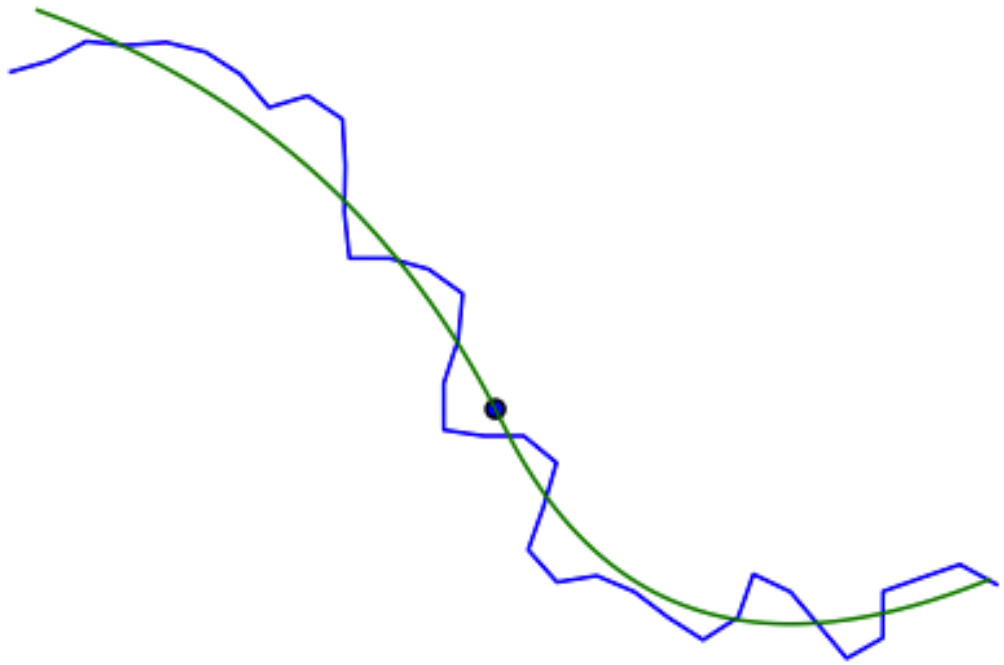


Figure 4.4: Robot Posture, Gross Posture Approximation Curve (GPAC), and Local Frame Origin

$$O(1): \quad \beta_t(u) = (x_u, y_u, \theta_u), \quad 0 \leq u \leq 1 \tag{4.14}$$

This function gives us access to the curve with a normalized parameter u. The function returns the x and y coordinates of a point and the orientation of its tangent vector that correspond to the u value. The drawback to this function is that the difference of arclength between two $\beta(u)$ values is not proportional to the difference in u value. This implies that the following condition exists:

$$\mathbf{dist}(u_1, u_1 + K) \neq \mathbf{dist}(u_2, u_2 + K) \tag{4.15}$$

where the function **dist**() computes the arclength between two u values on the curve, and the constant K adds the same u offset at two different points on the curve specified

by u_1 and u_2. What this means in practical terms is, the u values cannot be relied upon to give us reproducible positions on the curve. In particularly, if we retrieve the point $\beta_t(0.5)$, this will be close to the midpoint of the curve, but not the true midpoint and is subject to change between different curves.

We create a more accurate method that allows us to specify a point of true arclength on the curve. We present it in the form of a parameterized function:

$$O(\log n): \quad \sigma_t(d) = (x_d, y_d, \theta_d), \quad 0 \leq d \leq d_{max} \tag{4.16}$$

Here d is the arclength starting from the $u = 0$ terminator of the curve. d_{max} is the maximum arclength of the curve.

Though this is defined in the form of a function, it is implemented with a search algorithm. If we pre-process the curve by uniformly pre-sampling n points with $\beta_t(u)$ in $O(n)$ time, we can achieve a binary search time of $O(\log n)$ for each invocation of the function. The n parameter here corresponds to the number of samples of the curve we use to search over. The more samples, the more accurate the function will be. Here we use a tractable size of $n = 100$.

To perform the inverse operation for curve functions, we use the same pre-processed n sample points from equation Equation 4.16 and define the following functions:

$$O(n): \quad \beta_t^{-1}(x, y) = u \tag{4.17}$$

$$O(n): \quad \sigma_t^{-1}(x, y) = d \tag{4.18}$$

These inverse operators find the closest sample point on the curve to (x, y) and returns the sample's associated u or d parameter. The correctness of these functions depend on the queried point being reasonably close to the curve, the curve not be self-intersecting, and the point not be equidistant to two disparate points on the curve. They are $O(n)$ because we must compute the distance to all n sample points. Notice that we omit the

angle portion of the curve for inverse operations. We do not define closest point for orientations.

Since these inverse curve functions are very similar, we develop a unified algorithm that produces parallel results. We do this to prevent duplication of search operations when similar information is needed.

$$O(n): \quad c_p(x,y) = (u, d, (x_p, y_p, \theta_p)) \tag{4.19}$$

This function returns the closest point on the curve to (x, y), and its associated u and d parameters.

Finally, we have functions that convert between u and d.

$$O(\log n): \quad \mathbf{d}(u) = d \tag{4.20}$$

$$O(\log n): \quad \mathbf{u}(d) = u \tag{4.21}$$

These can be performed with a binary search on the pre-processed sample points.

4.4.2 Generation of GPAC Local Frame

For each generation of the GPAC, we can create a unique but predictable coordinate frame that represents the gross posture of the snake in the environment. Because the GPAC only follows the general posture of the snake, regardless of its articulation, the difference in curves between generations is small and within reason.

Once we have the new GPAC, to determine the location of our new coordinate frame origin $\hat{O}_t$ with respect to O_{19}, we compute the following:

$$P_d = \sigma_t(d_{max}/2) \tag{4.22}$$

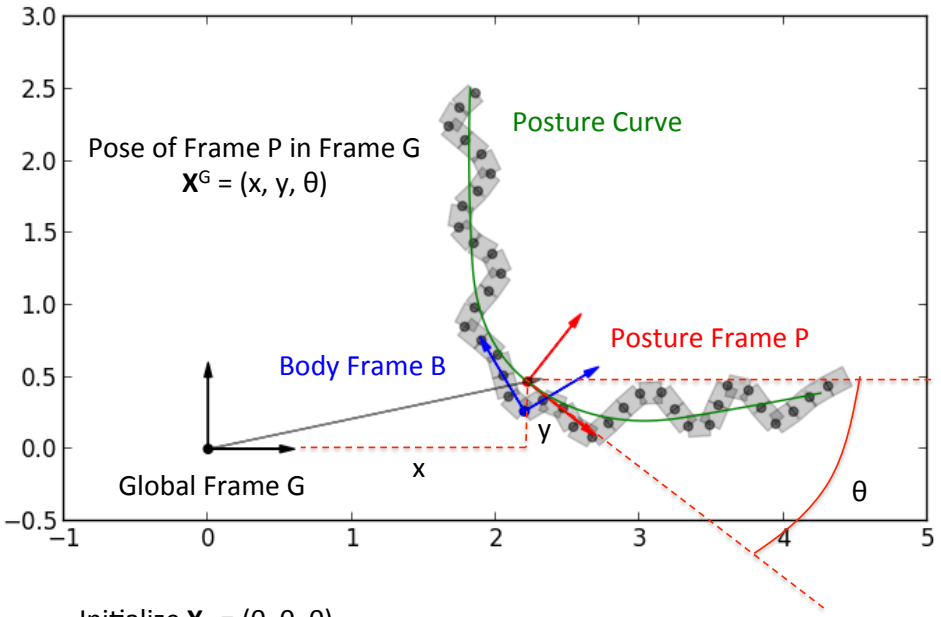
This specifies the halfway point along the curve in terms of arclength with respect to O_t. If we had used $\beta_t(0.5)$ instead, we would have had no guarantee how close to the middle it would be and no guarantees on repeatibility. By using the arclength function $\sigma_k()$, we have more guarantee over the location at the expense of running time.

From our new local frame $\hat{O}_t$, the orientation of the x-axis is directed in the forward direction of the snake, regardless of the articulation of the joints. This approach now gives us the tools to make relations between snake poses where the orientations will have some correlation. If we had insisted on using a body frame O_{19} as a local coordinate frame, the orientations would have had very little correlation. We would have had no means to relate and correct orientations between snake poses.

In the future, we will drop the t subscript of a GPAC and local frame origin and only refer to them as $\beta_k(u)$, $\hat{O}_k$, and $\hat{P}_k$, where $\hat{O}_k$ is the GPAC generated origin, $\hat{P}_k$ is the origin's location in the global frame, and k is the particular pose number in the pose graph. We describe this in more detail in Section 5.

4.4.3 Geometric Transform between Poses

Now that we have defined a local coordinate system for the complete posture of a snake robot, we can compute a geometric transform T_{ij} between two poses of the robot $\hat{P}_i$ and $\hat{P}_j$. We take the snake poses before and after one step of the adaptive concertina gait in the GPAC local frame for each. Each GPAC has a reference pose in the global frame, so it is only a matter of computing a transform between the two.

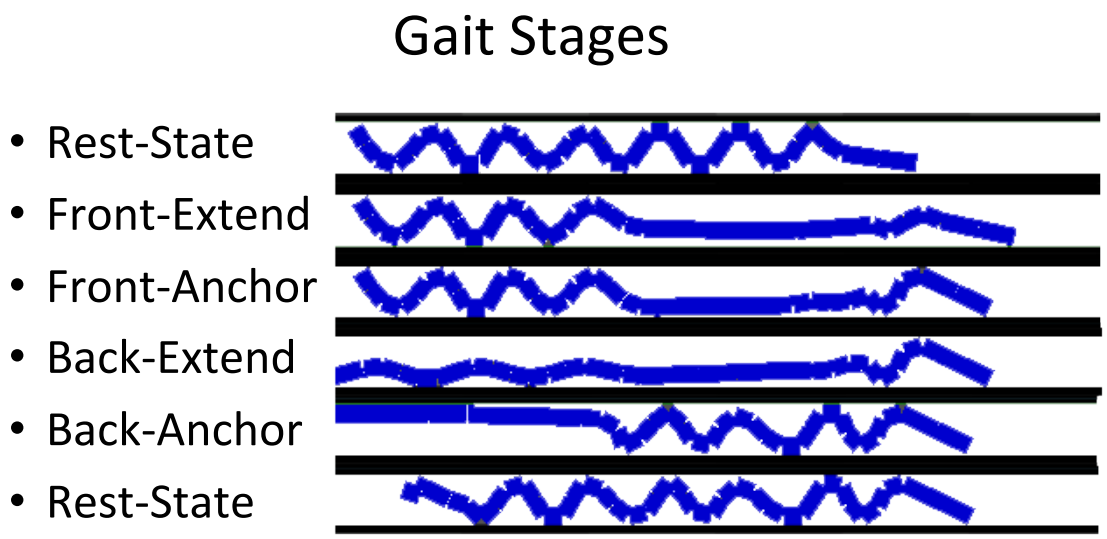


Initialize $\mathbf{X}_1 = (0, 0, 0)$

Chapter 5

Control and Locomotion

Simulated robot control algorithms tuned for simulated environment Adaptive anchoring making strong and stable anchors to the walls with verification Concertina locomotion Practical heuristic for sensor-deprived environment Snake control, backbone curves (Chirikjian) inverse kinematics by fitting to a curve Segmented behaviors sections of linkage controlled by different behaviors Smooth motion and compliance interpolated and compliant motion Stability detection checking for slip and transient disturbances



5.1 Control Methodology

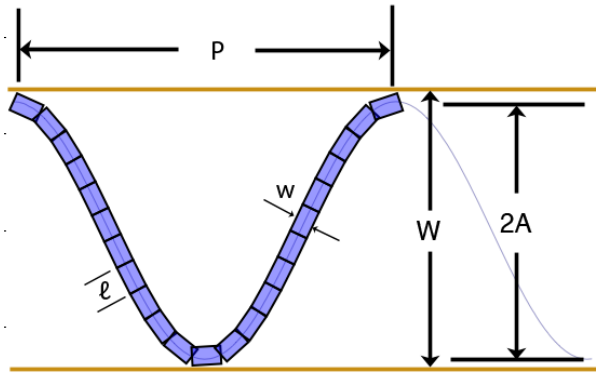
Backbone curve fitting, IK method, segmented behaviors

Anchoring

- Single anchor case
- 2 anchors plus extension require

for W->∞, L = 2(W-w),

so for concertina gait
L = 4(W-w) + e
where e is extension length



$$f(x) = \frac{W-w}{2}\cos\left(\frac{2\pi x}{P}\right)$$

$$L = \int_0^P \sqrt{\left(\frac{-(W-w)\pi}{P}\sin\left(\frac{2\pi x}{P}\right)\right)^2 + 1}\ dx$$

5.2 Behaviors

Locomotion steps, extensions, path-following, anchoring

5.3 Problem

FIXME: Related work and our contrasted difference. What can we achieve that others cannot. Why is our approach better for this particular case?

In order to successfully control a snake robot and have it move through the environment, we need a solution for locomotion, motion planning, and handling collisions. Since we have no exteroceptive sensors, this makes the problem challenging. The robot is unable to sense what is right in front of it and must move blindly. The snake could move into open space or collide with obstacles.

Part of the challenge is having no external sensors and the other part is general task-oriented control of a hyper-redundant robot. There are many biologically-inspired locomotion strategies that live snakes in nature use to move throughout the environment. The closest biological gait solution to moving in confined environments is the concertina gait shown in Figure 5.1.

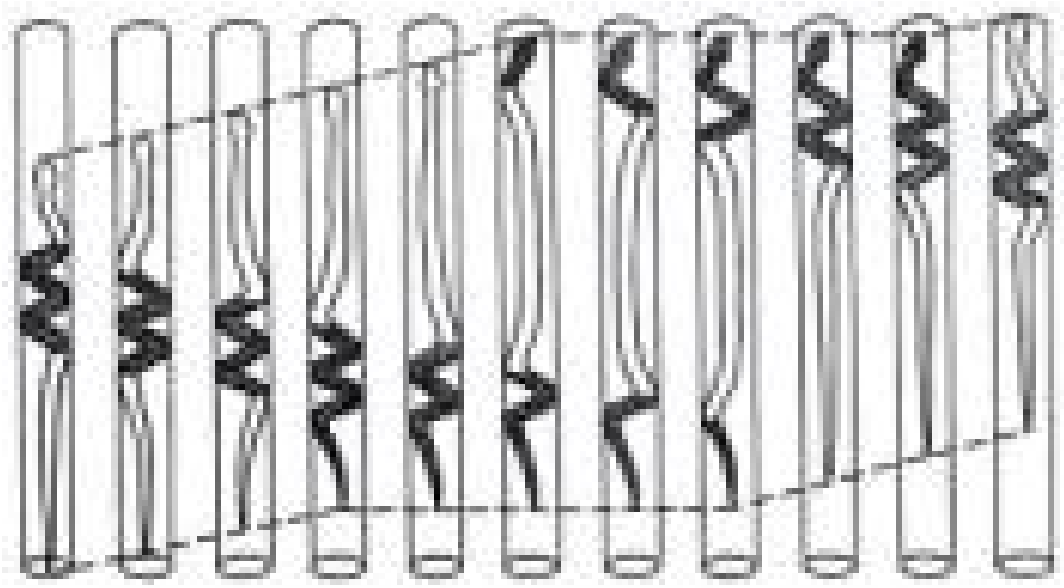


Figure 5.1: Biological Concertina Gait of a Snake in a Confined Space. Image taken from \cite{Gans:1980p775}

This gait is characterized by a concertina motion of alternating extensions and contractions that result in a forward movement. The snake body pushes against both sides of the walls establishing anchors that allow the snake to alternate between pulling and pushing itself forward. Success of locomotion depends on the snake establishing high friction contacts with the environment with which to push and pull.

However, the differences between a real snake and our robot snake make a true implementation impossible. A real snake has the luxury of vision and olfactory sensors to provide early motion planning and a rich tactile sensing surface skin to provide feedback to its control system. Our robot snake has only the benefit of internal proprioceptive

joint sensors. If we knew the exact width of the pipe walls, we could prescribe the perfect concertina motion to move through the environment smoothly.

A simple approach to making the fully anchored concertina posture, we could use the following equation:

$$\alpha_i = A * \cos\left(\frac{2\pi i f}{N}\right) \tag{5.1}$$

where α_i is the commanded joint angle for joint i, f is the frequency or the number of sinusoidal cycles per unit segment length, A is the maximum joint command amplitude, and N is the number of segments on the snake. Since there are N segments, there are $N-1$ joints. Therefore, $i \in [0, N-1)$. This equation requires some tuning to give the right results and can result in bad postures such as that shown in Figure 5.2. In order to effect locomotion, a sliding Gaussian mask should be applied to the joint command output to transition parts of the snake between fully anchored to fully straight to reproduce the biological concertina gait.

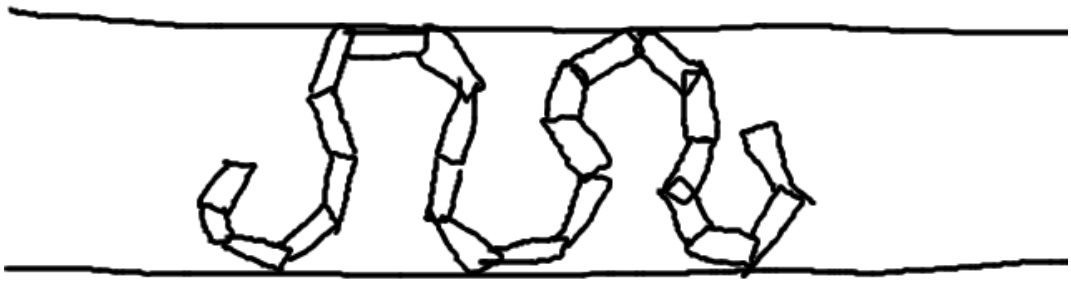


Figure 5.2: Improperly tuned concertina posture using equation Equation 5.1.

The difficulty of this approach is that we need a priori knowledge of the pipe width, the configuration of the snake needs to be tuned by hand for the particular snake morphology, and there is no sensory feedback to this implementation. With no feedback, there is no adaptive behavior possible. We need an approach that will work in environments of unknown and variable pipe width. Our robot needs to be able to adapt its locomotion to the width of the pipe.

Our desire is to be able to prescribe any type of snake posture, for any anchor width and any snake segment length or parameters, and have the snake automatically assume that posture. In order to do this, we need a means of specifying the posture and an inverse kinematics method for achieving that posture.

For both of these requirements, we use the backbone curve-fitting methodology first described by \cite{Chirikijan:1995p774}. This method of control is to produce a parameterized curve that represents the desired posture of the snake over time. Over time the curve changes to reflect desired changes in the snake posture. Snake backbone curve fitting is achieved by finding the joint positions that best fit the snake body onto the curve. This is found either by direct calculation or a search algorithm.

An example of backbone curve fitting can be seen in Figure 5.3. The parameterized curve is shown in light blue in the background. The blue snake body in the foreground is fitted onto the curve using an iterative search algorithm for each consecutive joint.

A typical application would be to specify the posture of the snake using a sinusoidal curve and then applying an inverse kinematics technique to fit the snake to the curve. The problem can be formulated iteratively as follows: for joints $i \in [0, k]$ and joint pose in space $p_i = \langle x_i, y_i, \theta_i \rangle$ are on the curve β with parameter t_i, find the joint angle a_k such that p_{k+1} is on the curve β and $t_k < t_{k+1}$.

For the equation β defined as:

$$y = A \sin(2\pi x) \tag{5.2}$$

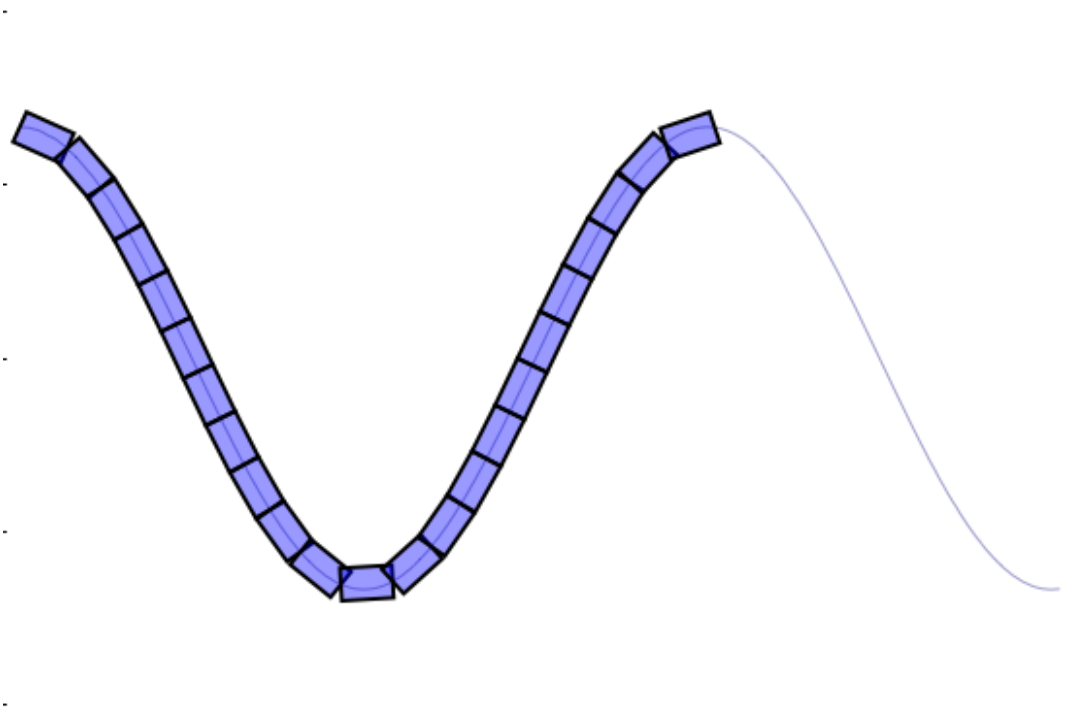


Figure 5.3: One period sine curve

the equation is monotonic along the x axis, so satisfying the criteria $t_i < t_{i+1}$ is the same as satisfying $x_i < x_{i+1}$.

If the length a of snake segment is l, we specify a circle equation centered at the joint position (x_i, y_i) with the following:

$$(x - x_i)^2 + (y - y_i)^2 = l \tag{5.3}$$

Finding solutions for the simultaneous equations Equation 5.2 and **??** will give us possible solutions to x_{i+1} and y_{i+1}. For these two sets of equations, there are always at least 2 possible solutions. We need only select the solution that satisfies the invariant condition $x_i < x_{i+1}$. There may be more than solution. In which case, we select the x_{i+1} where $(x_{i+1} - x_i)$ is the smallest. This prevents the snake from taking shortcuts across

the curve and forces it to fit as closely as feasibly possible to the parameterized curve given the snake robot's dimensions.

The above simultaneous equations do not have closed-form solutions and instead must be solved numerically. This can be computationally expensive using general equation solvers. Instead we propose a more specialized solver for this particular problem.

We choose a series of uniform samples $(q_0...q_k...q_M)$ around a circle of radius l centered at (x_i, y_i) such that all points q_k have $x_k >= x_i$. We are looking for instance of crossover events where the line segment (q_k, q_{k+1}) crosses over the curve β. Given at least one crossover event, we do a finer search between (q_k, q_{k+1}) to find the closest point on the circle to the curve β and accept that as our candidate position for joint point p_{k+1}. As long as we assume that the curve β is monotonic along the x-axis and a point of the curve β can be computed quickly given an x-value, this is a faster method of computing intersections of the circle with the backbone curve than a general numerical solver.

For instance, in Figure 5.23, we show a curve β described by a sine curve and a series of circles intersecting with the curve that indicate candidate locations for placing the subsequent joint on the curve. These are used to determine the appropriate joint angles to fit the snake onto the curve.

Now that we have a means of changing the snake's posture to any desired form given a parameterized, monotonic curve that describes the posture, we now need to form an anchoring approach that will work for arbitrary pipe widths.

5.4 Anchoring

In order to fit the anchor points to a pipe of unknown width, we need some way of sensing the walls. Since we have no exteroceptive sensors, the best way of sensing the width of the walls is by contact. Since we have no direct contact sensor per se, we must detect contact indirectly through the robot's proprioceptive joint sensors.

Using the backbone curve fitting approach, we take one period of a sine curve as the template form we wish the robot to follow. We then modify this sine period's amplitude, at each step refitting the snake robot to the larger amplitude, until the robot make's contact with the walls. This establishes two points of contact with the wall which assist in immobilizing the snake body.

It is clear from Figure 5.3 that as the amplitude of the curve increases, more segments are needed to complete the curve. If the sine curve is rooted at the tip of the snake, this results in the curve translating towards the snake's center of mass. Instead, we root the sine curve on an internal segment, as seen in Figure 5.4, so that the position of the anchor points remain relatively fixed no matter the amplitude and the number of segments required to fit the curve.

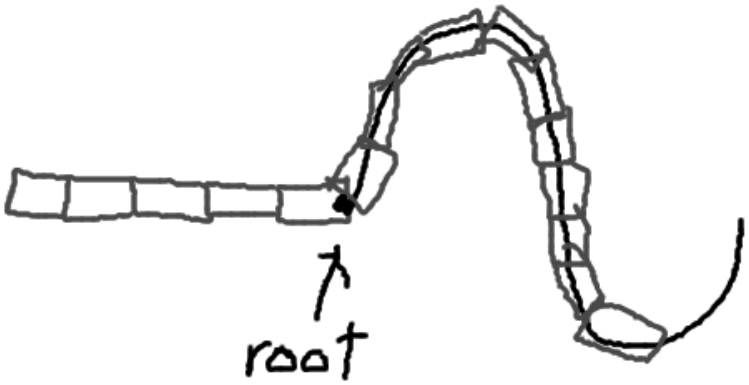


Figure 5.4: Anchor with not enough segments

Two points of contact are not statically stable if we disregard friction. Though friction exists in our simulation and in reality, we do not rely on it to completely immobilize our robot. Our normal forces are often small and the dynamic and transient forces can easily

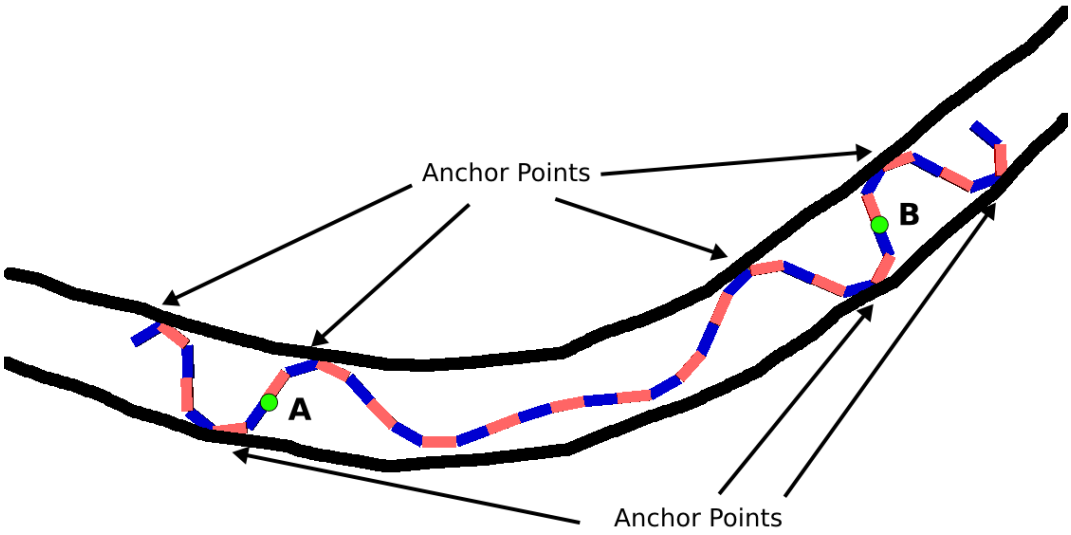


Figure 5.5: Anchor Points

cause a contact slip. Therefore, we need at least 3 contact points to consider an anchor to be statically stable.

One way to achieve this is by adding new sine period sections to create another pair of anchors. The amplitude of the new sine period is separate from the previous anchor's amplitude and is represented by a new curve attached to the previous one. This way the amplitudes remain independent. The equation to describe two sets of sine period curves with independently controlled amplitudes are as follows:

$$y = \mathrm{U}(x)\mathrm{U}(2\pi - x)A_1 \sin(fx) + \mathrm{U}(x - 2\pi)\mathrm{U}(4\pi - x)A_2 \sin(fx) \tag{5.4}$$

where f is the frequency and $\mathrm{U}(x)$ is a unit step function. Or to generalize for N periods and N pairs of anchor points:

$$y = \sum_{i=0}^{N-1} \mathrm{U}(x - i2\pi)\mathrm{U}((i+1)2\pi - x)A_i \sin(fx) \tag{5.5}$$

for each anchor curve amplitude A_i.

So now that we have the means to control the amplitude of our anchor points, we need some means of detecting that a contact is made and another to make sure the anchor is secure.

Our approach is to gradually increase the amplitude of an anchor curve until contact with both walls has been made. We will continue to increase the amplitude until we see significant joint error occur when fitting to the curve. If the amplitude because larger than the width of the pipe, the snake will attempt fitting to a curve that is impossible to fit to and will experience a discrepancy in its joint positions compared to where it desires them to be.

The structure of this error will normally take the form of one or two large discrepancies surrounded by numerous small errors. This reflects that usually one or two joints will seriously buckle against the wall, while the rest of the surrounding joints can reach their commanded position without disturbance. An example of an anchor curve with larger amplitude than the width of the pipe is shown in Figure 5.6.

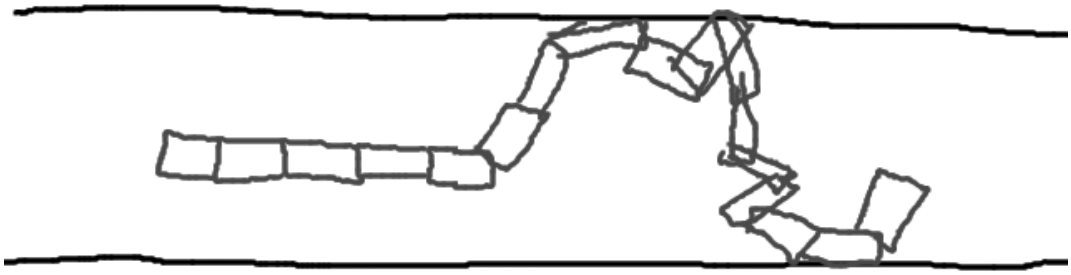


Figure 5.6: Anchor with amplitude larger than pipe width.

The contact is flagged once the maximum error reaches a certain threshold across all the joints of the anchor curve. Of the joints M of the anchor curve and the error e_j, if there exists $e_j > 0.3$ radians, than we flag a contact. It isn't so much that we are detecting a contact but a minimum pushing threshold against the wall. The robot is pushing hard enough to cause joint error over an experimentally determined threshold.

If j_k through j_{k+M} are the joints comprising the fitted anchor curve, the error of one joint is defined by $e_k = |\phi_k - \alpha_k|$. The maximum error is defined by $e_{max} = \max(e_k, e_{k+1} \cdots e_{k+M-1}, e_{k+M})$.

The amplitude is initially changed by large increments to quickly find the walls. Once the threshold has been reached, the current and last amplitude are marked as the max and min amplitudes respectively. We then proceed to perform a finer and finer search on the amplitudes between the max and min boundaries until we reach a snug fit that is just over the error threshold. This is a kind of numerical search(?).

The pseudocode for this algorithm is as follows:

Algorithm 5 Anchor Fitting

```
Â ⇐ 0
A_min ⇐ 0
A_max ⇐ ∞
δA ⇐ 0.04
while (A_max − A_min >= 0.001) & (δA >= 0.01) do
    Â ⇐ Â + δA
    e_flag ⇐ setAnchorAmp(Â)
    if ¬e_flag then
        A_min ⇐ Â
    else
        A_max ⇐ Â
        δA ⇐ δA/2
        Â ⇐ A_min
        setAnchorAmp(Â)
    end if
end while
```

The main objective of this code is to reduce the difference between maxAmp and minAmp as much as possible. minAmp and maxAmp are the closest amplitudes under

and over the error threshold respectively. The function setAnchorAmp() sets the amplitude, performs the curve fitting, and reports back any error condition as described earlier. Once this algorithm is complete, we have made a successful anchor with two contact points under compression without the fitted anchor curve being distorted by joint buckling.

5.5 Curves

When we describe curves for use in backbone curve fitting, they must be assigned to a frame of reference and usually have a starting point. Most often, the frame of reference chosen is one of the local segment frames on the snake body. However, sometimes we could choose the global frame if we wanted to navigate or probe something specific in the environment. The local frame curves that we use all start from a specific body segment and sprout from that segment like a plant. We say that this curve is rooted in segment X.

5.6 Behavior Architecture

Our method of control is a behavior-based architecture that is charged with reading and tasking the servo-based motor controllers of each joint. Each behavior may be a primitive low-level behavior or a high-level composite behavior composed of multiple sub-behaviors. Furthermore, a behavior may have complete control of every joint on the snake or the joints may be divided between different but mutually supporting behaviors. This architecture allows us to achieve a hierarchy of behavior design as well a separation of responsibilities in task achievement. For instance, the back half of the robot could be responsible for anchoring, while the front half could be responsible for probing the environment as seen in the example architecture in Figure 5.7.

Each behavior in the behavior-based architecture is time-driven. It is called periodically by a timer interrupt to compute the next commands for the following time-step.

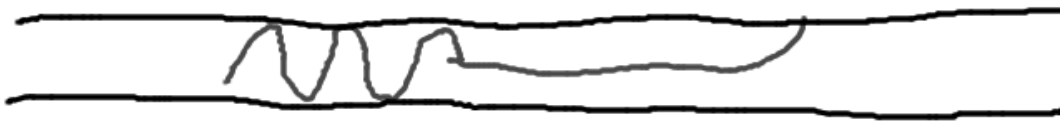


Figure 5.7: Separation of functionality

This could be variable time or constant time, but to make our behavior design simple, we use 10ms constant time steps in our simulation.

At each time step, the state of the robot is captured and driven to the behaviors. The data includes a vector of joint angles ϕ_i, a vector of commanded angles α_i, and a vector of maximum torques m_i. These values were described in the previous section X. The output of each behavior is a vector of new commanded angles $\hat{\alpha}_i$, a vector of new max torques $\hat{m}_i$, and a vector of control bits c_i. The values of $\hat{\alpha}_i$ can be any radian value within the joint range of motion or it can be NULL. Likewise, the new max torque of $\hat{m}_i$ can be any non-negative max torque threshold or NULL. The NULL outputs indicate that this behavior is not changing the value and if it should reach the servo-controller, it should persist with the previous value.

The bits of the control vector, c_i , are either 1, to indicate that the behavior is modifying joint i, or 0, to indicate that it has been untouched by the behavior since its initial position when the behavior was instantiated. This control vector is to signal the

activity to parent behavior that this part of the snake is being moved or controlled. The parent behavior does not need to respect these signals, but they are needed for some cases.

Our behavior-based control architecture follows the rules of control subsumption. That is, parent behaviors may override the outputs of child behaviors. Since we can also run multiple behaviors in parallel on different parts of the snake, sometimes these behaviors will try to control the same joints. When this happens, the parent behavior is responsible for either prioritizing the child behaviors or adding their own behavior merging approach.

Asymmetric behaviors that run on only one side of the snake such as the front are reversible by nature. That is, their direction can be reversed and run on the back instead. All behaviors with directional or asymmetric properties have this capability..

Finally, parent behaviors can instantiate and destroy child behaviors at will to fulfill their own control objective. All behaviors are the child of some other behavior except the very top root-level behavior which is the main control program loaded into the robot and responsible for controlling every joint. The root behavior is responsible for instantiating and destroying assemblages of behaviors to achieve tasks as specified in its programming. We will discuss some of these behaviors and child behaviors in the following sections.

5.7 Smooth Motion

We desire the motion of our snake robot to be slow and smooth in nature. It does us no benefit for the motion to be sudden and jerky. Moments of high acceleration can have unintended consequences that result in our anchor points slipping. Either collisions with the environment or transient internal dynamics can cause the anchor points to slip.

Our method of ensuring smooth motion is to take the initial and target posture of the joints of the snake, perform linear interpolation between the two poses, and incrementally change the joint angles along this linear path. For instance, if the initial joint angles

are $s_0 ... s_n$ and the target joint angles are $t_0 ... t_n$, the interpolated path for each joint j is found by $g_{jr} = s_j + (t_j s_j) * (r/100)$ for $r \rightarrow 0, 100$, for 100 interpolated points on the linear path. The resultant motion is a smooth transition from initial to target posture as seen in Figure 5.8.

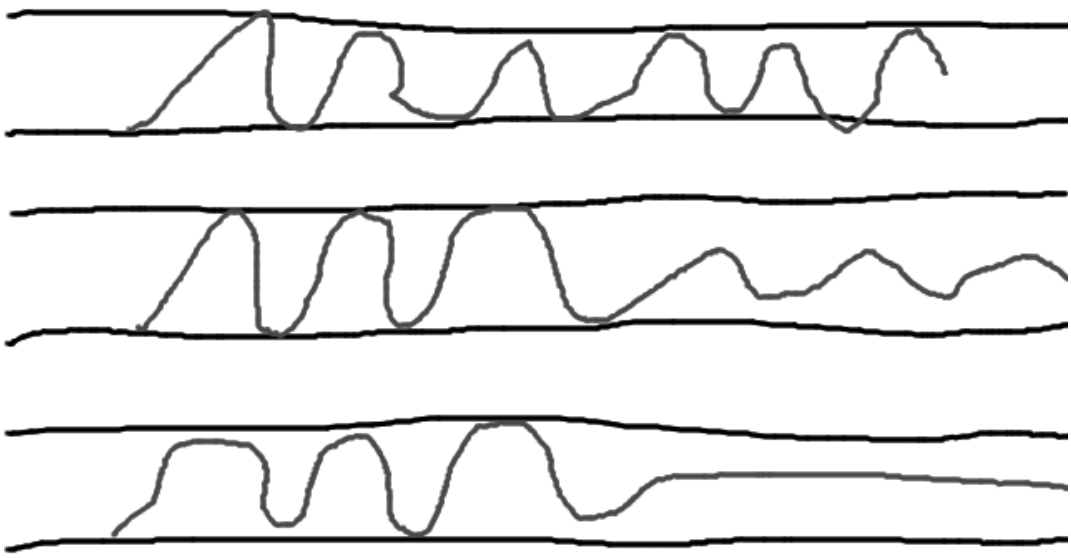


Figure 5.8: Smooth motion from interpolation of posture.

This is the definition of the HoldTransition behavior. When it is instantiated, it is given an initial pose vector S and target pose T, where some values of t_i in T and s_i in S may have NULL for no command. At each step of the behavior, using the equation $g_{jr} = s_j + (t_j s_j) * (r/100)$, r is incremented to compute the new joint command. If t_j is NULL, then $g_{jr} = s_j$. Once $r = 100$, it returns True on step() and on all subsequent steps, $g_j = t_j$.

5.8 Behavior Merging

There are a few methods for merging behaviors that overlap in some way. One way of the previous section is for two behaviors to overlap temporally and using the HoldTransition behavior to handle a smooth transition from one configuration to another.

In the event that two behaviors control adjacent sets of joints or overlap their control in some way, a conflict resolution is method is needed to not only select the proper command for contested joints, but to prevent either behavior from negatively affecting functionality of the other by modifying any positional assumptions or introducing any unintended forces or torques to the other side.

A common occurrence is that two behaviors may try to control the same set of joints at the same time or otherwise. This is a conflict that requires resolution by the parent behavior. In this section we present three different ways to merge the outputs of different behaviors as means of conflict resolution. They are, in the order presented, the precedence merge, the splice merge, the convergence merge, and the compliance merge respectively.

Given that all child behaviors have an ordering, the default resolution method is to take the first behavior's control output over all others. This is called precedence merging. An ordering is always specified by the parent at the time it instantiates the child behaviors. If the first behavior's output is NULL, then it goes to the next behavior and so on until a non-NULL value is found or the last behavior returns a NULL, in which case the parent also returns a NULL unless the parent otherwise specifies.

Precedence merging can be crude when trying to maintain a stable and immobilized position in the environment. Sudden changes at the behavior-boundary can cause jarring discontinuities on the snake's posture and result in loss of anchoring as seen in Figure 5.9. A more finessed approach would disallow these discontinuities and provide a non-disruptive merging. The first approach that does this is the splice merge.

Figure 5.9: Discontinuity in behaviors results in clash of functionality.

In the event that two behaviors overlap, we have the option of having a hard boundary between the behaviors centered on a single joint. Except in the rare case that both behaviors want to command the splice joint to the same angle, it is highly likely that the merge will be discontinuous. In order to ensure that positional assumption is maintained for both behaviors, the behaviors must be spliced so that neither behavior disrupts the other.

If the splice joint is j_s, behavior A controls joints j_0 through j_s, behavior B controls joints j_s through j_{N-1}, S_{s-1} is the body segment connected to j_s in behavior A, and S_s is the body segment connected to j_s in behavior B, we have the situation shown in Figure 5.10a. If behavior A commands j_s to α_a to put segment S_s in the position shown in Figure 5.10b and conversely, if behavior B commands j_s to α_b to put segment S_{s-1} in the position shown in Figure 5.10c, the two behaviors can be spliced discontinuously with no disruption to either behavior by setting j_s to $\alpha_s = \alpha_a + \alpha_b$. The result is shown in Figure 5.10d.

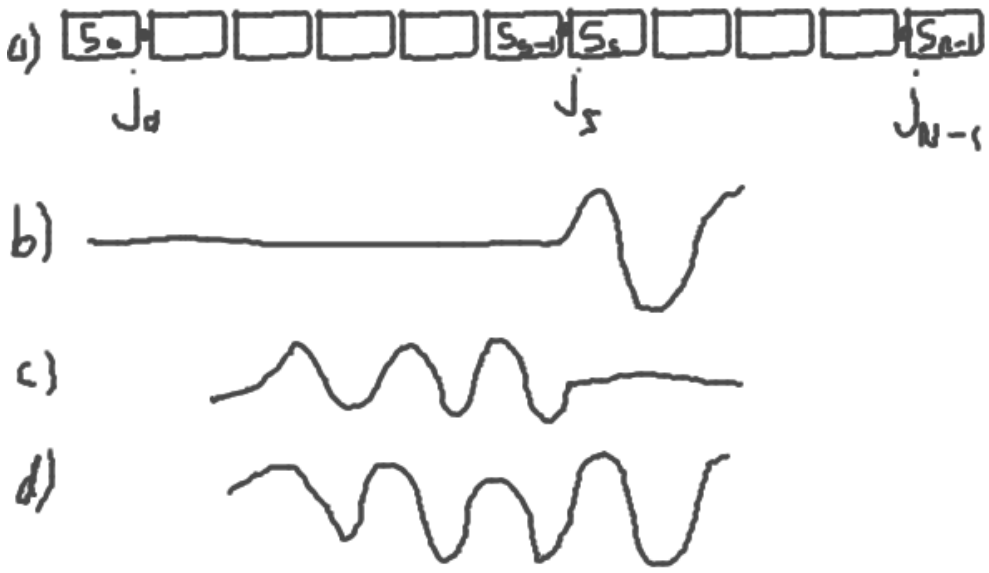


Figure 5.10: Splice joint and connecting segments.

This is the behavior splicing approach because it stitches the behaviors together in a clinical way to ensure proper functionality of both. Its primary role is to connect behaviors with discontinuous conflicts on the behavior boundary and whose positions must be maintained for functional purposes. For more fuzzy continuous boundaries, we use the convergence merging approach.

For the two behaviors shown in Figure 5.11a and Figure 5.11b, the resultant convergence merge is shown in Figure 5.11c. For this approach, there is no discrete boundary but one behavior converging to another over several joints. For a given joint j_i somewhere in this convergence zone, and the commanded values for behaviors A and B being α_a and α_b, the converged command value for j_i is computed by:

$$\alpha_i = \alpha_a + (\alpha_b \alpha_a)/(1 + \exp(i - c)) \tag{5.6}$$

where c is a calibration parameter that controls the magnitude and location of the convergence along the snake. As $c \to +\infty$, $\alpha_i \to \alpha_b$. As $c \to -\infty$, $\alpha_i \to \alpha_a$. In practice,

$|c| < 20$ is more than sufficient for complete convergence to one behavior or the other. The i parameter in the exponent ensures that the convergence value is different for each of the neighboring joints. This produces the effect shown in Figure 5.11c.

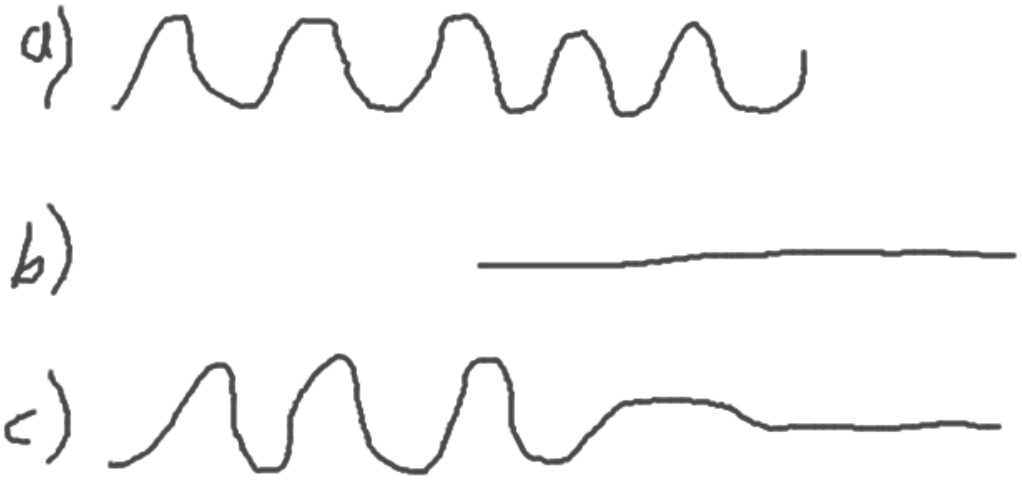


Figure 5.11: Convergence merge.

If we modify c, this gives us a control knob on the merge and can be moved up and down. If we move $c \to +\infty$, behavior B takes complete control of the snake. If we move $c \to -\infty$, behavior A takes control of the snake. c can be changed over time to create a sliding transition effect with a convergence boundary. In fact, this is the basis for a behavior called HoldSlideTransition.

A fourth and final way for two behaviors to be merged is the compliance merge. This is the case where two behaviors are not adjacent but are far away from each other, but still interfere with each other somehow by inflicting unwanted forces and torques between the behaviors.

A compliance merge resolves this by selecting a group of joints between the two behavior's joint sets to be compliant or non-actuated. This has the result of absorbing

any forces or torques transmitted by either behaviors and mechanically isolating one side of the snake from the other. An example is shown in Figure 5.12.

Figure 5.12: Compliance merge.

5.9 Compliant Motion

In the previous section, we mentioned compliance but not our implementation. Compliant motion is a much used technique in robotics and has been discussed widely (cite). The two approaches are either passive or active compliance. Our approach is to use passive compliance since we have no means to detect contact or sense obstacles.

Passive compliance is achieved by changing the settings on the max torque parameter of each joint's servo controller. We use a high setting for active motion, a low setting for compliant motion, and zero for keeping the joint unactuated.

Besides using a compliance merge to mechanically isolate two behaviors, we also use compliance as an exploration strategy. Since we have no exteroception, the only way to discover walls and obstacles is to collide with them. Compliant impacts with obstacles

have a couple benefits. It prevents the force of the impact from transmitting to the body of the snake and possibly causing anchor slip. It also allows the joints of the snake to give out and take the shape of the environment around it.

This latter benefit is one of our primary means of sensing and exploring the environment. By passively letting the snake robot assume the shape of the environment, it guides our mapping and our locomotion. Until we have mapped out open space and boundaries, we have no way of directing our robot towards them beyond colliding with the unknown.

5.10 Stability Assurance

In a previous section on smooth motion, we discussed an interpolation technique to smoothly transition from one posture to another. The primary motive of this was to keep the parts of the snake body that are immobilized from being destabilized by transient forces and causing anchor slip. Static forces from prolonged pushing against an obstacle or wall are a separate matter.

For a pose transition that has a sufficiently large number of interpolation steps of sufficiently large duration, we can reasonably be assured that no transient forces from dynamics or collision will cause anchor slip because we are moving very very slowly. However, sufficiently long transition times for reliability may be impractical for applications, since times for a simple transition can be upwards of 1–2 minutes using a conservative approach. For a robot that performs hundreds of moves that require transitions, this is an unacceptable approach.

If we focus on being able to detect transient forces on the snake body instead, we can use this to control the speed of our movements instead of a one-size-fits-all approach of long transition times for all motion. We have developed a heuristic technique to do just this.

Our approach is to use our only source of information, joint angle sensors, as makeshift stability monitors of the entire snake. That is, if all the joints stop changing within a degree of variance, we can call the whole snake stable. Once we have established that the snake is stable, we can perform another step of motion.

For each, we compute a sliding window that computes the mean and variance of the set of values contained in. If at every constant time step, we sample the value of some joint i to be ϕ_{it}, then we compute:

$$\mu = \sum_{t=j-K}^{j} \frac{\phi_{it}}{K} \tag{5.7}$$

$$\sigma^2 = \sum_{t=j-K}^{j} \frac{(\phi_{it}\mu)^2}{K} \tag{5.8}$$

K is the width of the sliding window or the sample size used for computing the variance. This and the time step size δt, are used to determine the resolution and length of variance computation. In our implementation, with some exceptions, we use $\delta t = 1$ms and $K = 20$. That is, for N joints, we compute the variance of each joint's angle for the last 20 readings once every 1ms. This may be changed and retuned depending on the available computing resources, the resolution of the sensors, and the sensor's sample rate.

Finally, we can determine if the snake is stable if all of the joint variances fall below a threshold S_v and stay that way for over time S_t. In our case, we set $S_v = 0.001$ and $S_t = 50$ms.

We are primarily interested in preventing anchor slip. Most often, it is not necessary to wait for the entire snake to hold still before moving. Instead, we can focus just on the joints in the neighborhood of the anchor points. By computing just the variances of the joints in the local neighborhood of the anchor points, we can determine local stability. Using local stability, we gain some confidence that our anchor points do not slip between motion steps.

In order to use this concept of local stability, we need to have some indication of which joints to monitor and which to ignore. Fortunately, the control vector c, output of the behaviors mentioned in a previous section, is just the kind of information we need to know where to direct our local stability attention. If the values of a control vector indicate '0', this means that these joints are not modified by the behavior and should remain stable before performing another move step. If the values are '1', these joints are actively modified by the behavior and we should not expect these joints to remain stable between move steps. The behavior prescriptively tells us how it should behave and the stability system processes it to accommodate.

5.11 Adaptive Step

Using these tools at our disposable, we can now build a suitable locomotion behavior for an unknown pipe with no exteroception. The entirety of our designed behavior is shown in Figure 5.13 and is called the adaptive concertina gait, a biologically-inspired gait based on the concertina gait but modified for the change in sensors. The behavior is divided into 6 states, and each level of Figure 5.13 shows one of those states. We describe the states as follows:

1) Rest-State (Initial): Rest-State is the initial fully anchored posture of the snake robot. This is our initial pose and it acts as a stable platform from which exploratory or probing behaviors can be executed to sense the environment with little chance of slips or vibration due to the multiple anchor contact points. There is no other action here than to hold the posture that it was already in. The Rest-State is the initial state and the final state of the concertina gait.

2) Front-Extend: In the Front-Extend state, the forward segments are gradually transitioned from their initial position to a straight-forward position. The joints are compliant to the boundaries of the environment to accommodate turns in the pipe.

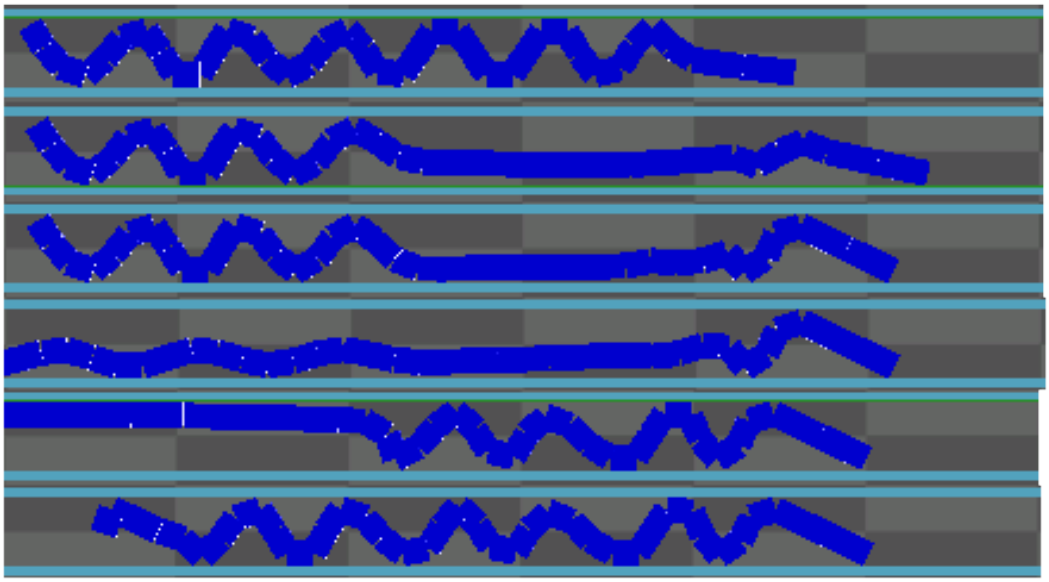


Figure 5.13: Adaptive Step Concertina Gait

3) Front-Anchor: The Front-Anchor state takes the extended portion of the snake and attempts to establish an anchor to the pipe as far forward as possible. The locomotion distance is maximized the further forward the front anchor is.

4) Back-Extend: The Back-Extend state follows a successful Front-Anchor event. All anchors established in the back segments are gradually extended until they are straight. Again, the joints are compliant so that they can conform to the walls of the environment.

5) Back-Anchor: The Back-Anchor state involves establishing multiple anchors with the remainder of the body segments.

6) Rest-State (Final): Upon conclusion of the Back-Anchor state, a single step of the adaptive concertina gait is complete and the snake is now in the Rest-State. From here we can do probing of the environment or perform another step forward.

A successful transition through all the states results in a single step. Multiple steps are used to travel through the environment. We describe no steering mechanism for this locomotion gait because the robot has no means to sense the environment in front of it.

However, the robot's compliant posture will adapt to any junctions or pipe curvature and follow it.

We now describe the implementation of each of these states and their composition of behaviors.

5.11.1 Front-Extend

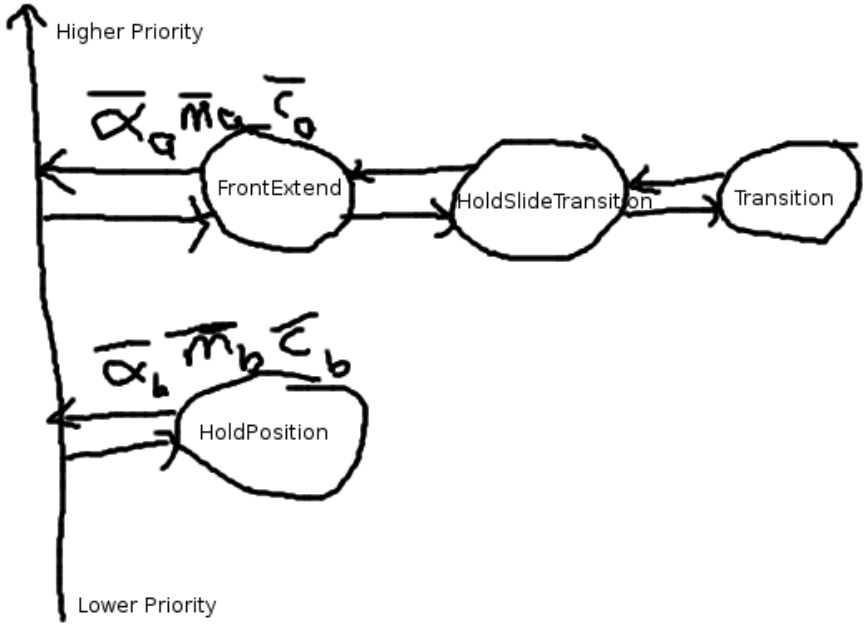


Figure 5.14: Front-Extend behavior assembly

From the fully anchored Rest-State stage, the goal of the Front-Extend stage is to release the front anchors and stretch the front half of the snake as far forward as possible without compromising the rear anchors. The further forward we can extend, the further the snake can travel in a single step.

If the extended snake arm makes a lateral impact, the walls will guide the movement of the snake but not otherwise hinder the behavior. In the event of an actual obstruction, the tip's movement will stop. We are able to track whether the position of the tip hasn't moved overtime and are able to set a flag to abort the behavior at the current posture.

Its implementation consists of 4 behaviors arranged as shown in Figure 5.14. We explain each behavior's functionality.

HoldPosition: The purpose of this behavior is to output the same joint commands indefinitely. It is instantiated with a vector of joint commands, and it continues to output these joint commands until termination. In this case, it is instantiated with the entire initial anchored posture of the snake from Rest-State and continues to output this posture. The behavior's output is gradually subsumed by the growth of the FrontExtend behavior in a precedence merge. HoldPosition is placed 2nd in the behavior ordering at the root level for this reason.

Transition: This behavior's purpose is to take an initial and final pose and interpolate a transition between the two, outputting the intermediate postures incrementally. The number of interpolated steps used to transition between the poses is set by the parent behavior. It is often a function of how much difference there is in the initial and final poses. The clock-rate is dependent on the parent behaviors. Once it reaches the final pose, it returns a flag and continues to output the goal pose indefinitely until reset or deleted. Here it is instantiated by the HoldSlideTransition behavior and reset each time it needs to make a new move command.

HoldSlideTransition: This behavior was first mentioned in section section 5.8 and implements the equation Equation 5.6. Its role is to use a convergence merge to transition from one posture to another. The convergence parameter c is incremented over time to slide the convergence from the front to the back of the snake, to smoothly transition from the fully-anchored posture to the back-anchored and front-extended posture.

For each change in c, the Transition behavior is set up to interpolate smooth motion between the two postures. Once the Transition behavior completes, c is incremented and Transition is reset to the new initial and final postures. c is changed from 8 to 20 while using the joint numbers as the indices in the convergence merge equation shown in X, assuming that the joint numbering starts at 0 at the front tip. An example of the HoldSlideTransition behavior in action is shown in Figure 5.11.

FrontExtend: Finally, the FrontExtend behavior creates HoldSlideTransition as a child behavior and gives it its initial and final posture. It passes through the output of HoldSlideTransition. It also monitors the pose of the snake tip to see if any obstructions are encountered. If the pose remains stationary for a period of time, an obstruction is declared and the behavior is halted. The behavior returns a collision flag.

5.11.2 Front-Anchor

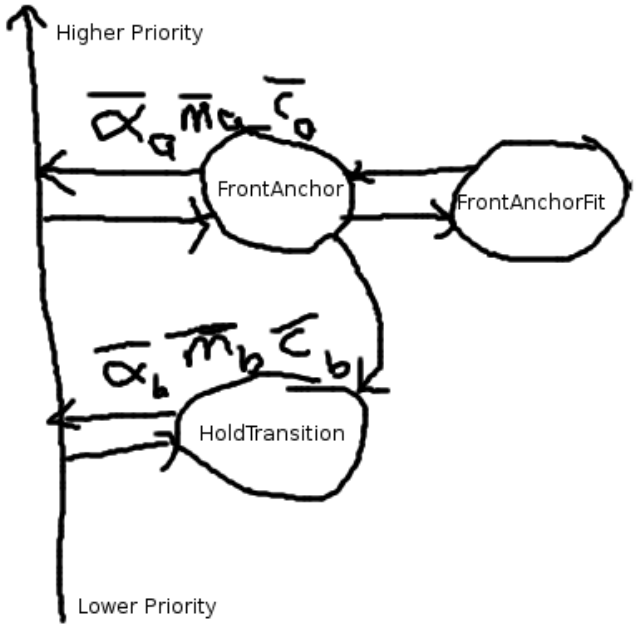


Figure 5.15: Front-Anchor behavior assembly

Once the front-half of the snake has been extended by the Front-Extend stage, the next step is to place the first anchor using the already extended segments. The complete posture of the snake is submitted to a HoldTransition behavior while the front half of the posture is modified by the FrontAnchorFit behavior.

This stage attempts to create a 3-point anchor that is created with the previously extended segments. The anchor is rooted at an internal joint starting from the merge joint 7 giving 8 segments with which to create the anchor. In the event that not enough

segments are available to create the full anchor, the anchoring process is restarted with +2 more segments by moving the root inward by 2 joints.

On successful completion of the 3-point anchor with the available segments, a test is performed to see if the anchor is secure, called a jerk-test. This is accomplished by a sudden rotation of four joints adjacent to the merge joint not on the side of the front anchor. While tracking the position of the anchor segments, if the anchored portion of the snake moves appreciably from the jerk-test, the anchor is deemed insecure, and the anchoring process is restarted with 2 more segments.

Whenever the merge joint is moved inwards, this also results in the contact positions of the anchor points moving inward as well. This can be useful for changing the anchor locations if no stable purchase is found, but also reduces the step distance of the gait by reducing the extension length.

The composition of the behaviors is shown in Figure 5.15 and described as follows:

HoldTransition: This behavior is originally initialized with the complete posture of the snake at the outcome of the Front-Extend stage. The front portion is laterally modified by the FrontAnchor behavior whenever new changes are required to the front anchor shape. The behavior is stepped and drives the robot joints for smooth transitions.

Though the FrontAnchor behavior has priority over the HoldTransition behavior, FrontAnchor almost never drives an output but instead laterally modifies the HoldTransition behavior. Therefore, the HoldTransition behavior is the primary driver of the snake robot.

FrontAnchor: This behavior has complete responsibility for the entire anchoring and testing process. It creates a 3-point anchor for static stability instead of the 2-point anchors previously described in section 2.1. This is because this will be the only anchor to the environment in the next stage. Complete secureness is required to avoid any slip conditions.

A 3-point anchor is created with a cosine curve of period 2.5pi shown in Figure 5.16. The FrontAnchor behavior gradually increases the amplitude of this curve. As the amplitude increases, more and more segments are required to complete the fit. Should the curve be longer than the available number of segments, the anchoring process is restarted with the merge joint moved back 2 joints.

The anchoring process is halted after search algorithm 2 completes. The behavior then performs a jerk-test to determine if the anchor points are secure. If k'th joint is the merge joint, then the joints [k+1,k+4] are rapidly changed in the following sequence: [(0,0,0,0), (–60,–60,–60,–60), (0,0,0,0), (60,60,60,60)]. These joints are directly controlled by the FrontAnchor behavior to create sudden motion and override the parallel Hold-Transition behavior that normally performs smooth motion.

A secure anchor will register little movement in the anchor segments while an insecure anchor will move a great deal. This movement is tracked through kinematics and assumes that the back anchors are secure and stable. Once the deviation in position of the anchor segments is below a certain threshold, than we determine that this is a secure anchor and return success.

FrontAnchorFit: This behavior is responsible for computing the curve-fitting process described at the beginning of section 2. It determines the correct joint orientations for a given curve. The curve is input by the FrontAnchor behavior and modified accordingly. The FrontAnchorFit behavior performs an efficient curve-fitting process assuming the curve starts rooted at the merge joint on the behavior boundary shown in Figure 5.16.

5.11.3 Back-Extend

After the front anchor has been securely established, the next stage's role is to remove the back anchors and extend the back half of the body straight. If the environment prevents the back half from being straight, the joints are compliant to the environment and will extend as far as they are allowed.

The behaviors are shown in Figure 5.17 and described as follows:

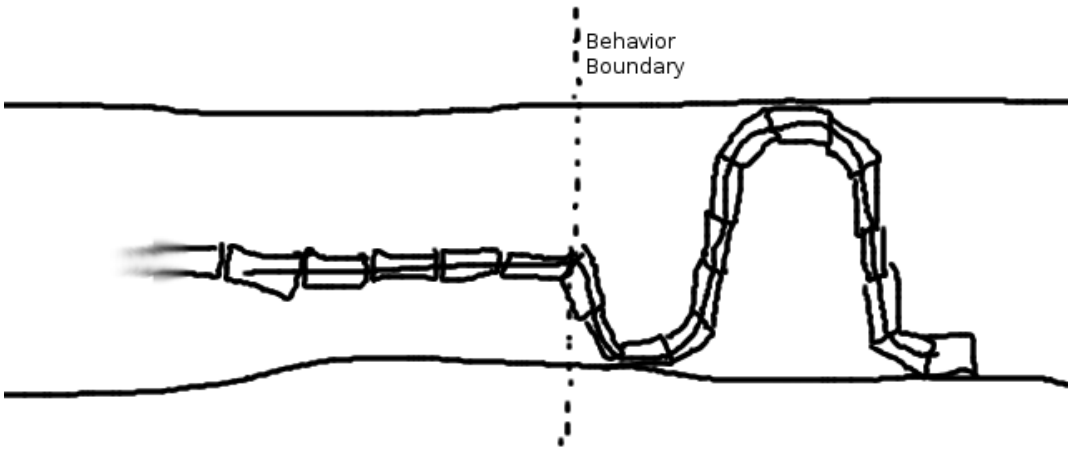


Figure 5.16: Posture that creates a 3-point anchor.

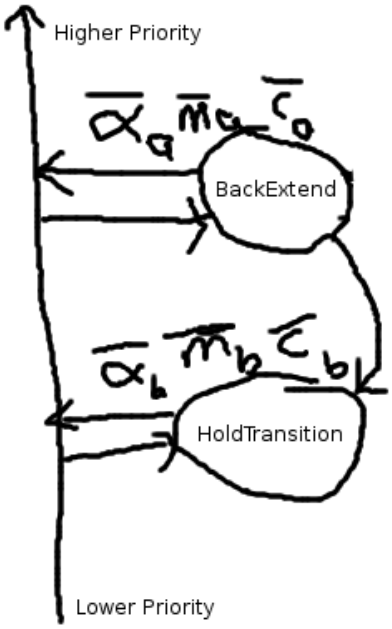


Figure 5.17: Back-Extend behavior assembly.

BackExtend: This behavior simply takes all the joints on one side of the merge joint behavior boundary and commands their values to be 0.0. It also sets all of these joints to have low torque to ensure compliant motion. These values are sent laterally to the HoldTransition behavior.

HoldTransition: This behavior is initialized to the posture from the outcome of the Front-Anchor stage. It receives as input from the FrontExtend behavior, the new straight 0.0 joint positions. It manages the smooth motion from the back anchored position to the back extended position.

5.11.4 Back-Anchor

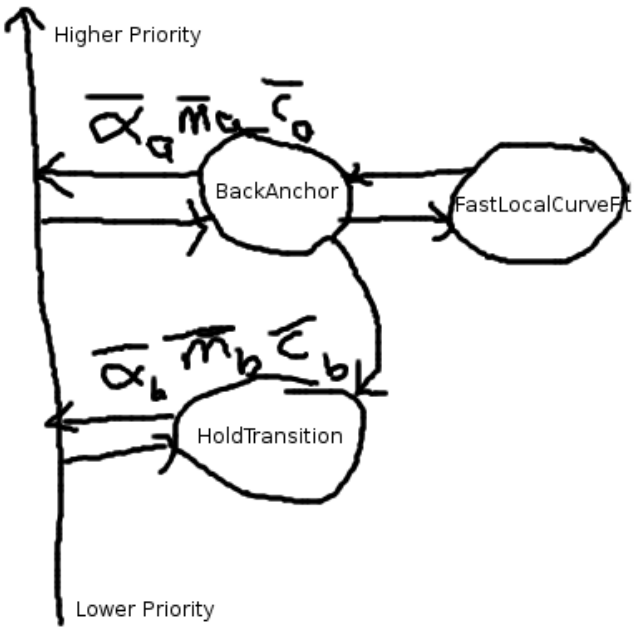


Figure 5.18: Back-Anchor behavior assembly

The purpose of this stage is form the 2-point anchors described in section 2.1 with the remaining extended segments of the back half of the snake. The sine curves are joined together with the front anchor cosine curve to make a continuous splice merge and produce a complete anchored posture of the whole snake.

New pairs of anchor points are created, using one period of a sine curve each, until all of the remaining back segments are used up. At each anchor pair, we use the adaptive anchoring algorithm 2 to detect contacts. However, we do not perform a jerk-test like in the Front-Anchor stage because we only have the front anchor as our reference point and no extra joints to perform the jerking motion. Furthermore, it is not as critical to determine if the back anchors are secure since there are more of them, and we have confidence in the security of our front anchor.

The behaviors are shown in Figure 5.18 and described as follows:

HoldTransition: This behavior fulfills the same role as all of the other instances. It takes the initial posture at the outcome of the previous stage and maintains that posture until modified by the primary behavior, BackAnchor. It manages the smooth transition between postures.

BackAnchor: This behavior performs the anchor contact algorithm 2, modifies the curve parameters, and continues to add anchor pairs until it runs out of segments. It creates the FastLocalCurveFit behavior as a child behavior.

FastLocalCurveFit: This behavior parameterizes the curves representing the series of back anchor curves of varying amplitudes. It is also responsible for efficiently computing the joint positions to fit the snake to the curves. Furthermore, it accepts compliance in the curve-fitting process between each anchor pair. That is, at each merge joint between two anchor pairs, compliance is to find the best anchor posture and the posture is saved for the remainder of the stage.

5.12 Analysis

Our theoretical contribution is a quantitative analysis of the trade-space between environmental characteristics, snake robot morphology, and locomotion control parameters.

That is, given a snake robot morphology and control parameters, what types of environments can we successfully explore. Conversely, given a specific environment, what robot morphology and control parameters are required to successfully explore it.

First we will perform an analysis of the trade-space for forming a 3-point stable anchor in a smooth pipe. A 3-point anchor is shown in Figure 5.19 and is the minimum number of contact points required to form a stable anchor. This posture will hold the body of the snake rigid and prevent it from translating or rotating within a threshold of external force and torque.

We will define and quantify the parameters that describe this system: the environment, the robot, and the control.

Our particular environment of a smooth pipe can easily be described with one parameter, the pipe width W.

The complete snake robot morphology can be described by 3 parameters: segment width w, segment length l, and number of segments n.

Finally, control of the robot is achieved by fitting the body of the snake onto a curve described by a sinusoid. The parameters for control are the values of amplitude A and period P, where $y = A\cos(\frac{2\pi x}{P})$.

5.12.1 Case: Snake as a Curve

We first consider the degenerate case where $l = 0$, $w = 0$, and $n = \infty$. This results in a snake that has no width and an infinite number of joints. Essentially, the snake body equals the curve equation. Since $w = 0$, this results in $2A = W$. This is shown Figure 5.20.

We define L to be the total length of the snake which can be computed from the equation of the curve.

$$f(x) = \frac{W}{2}\cos\left(\frac{2\pi x}{P}\right) \tag{5.9}$$

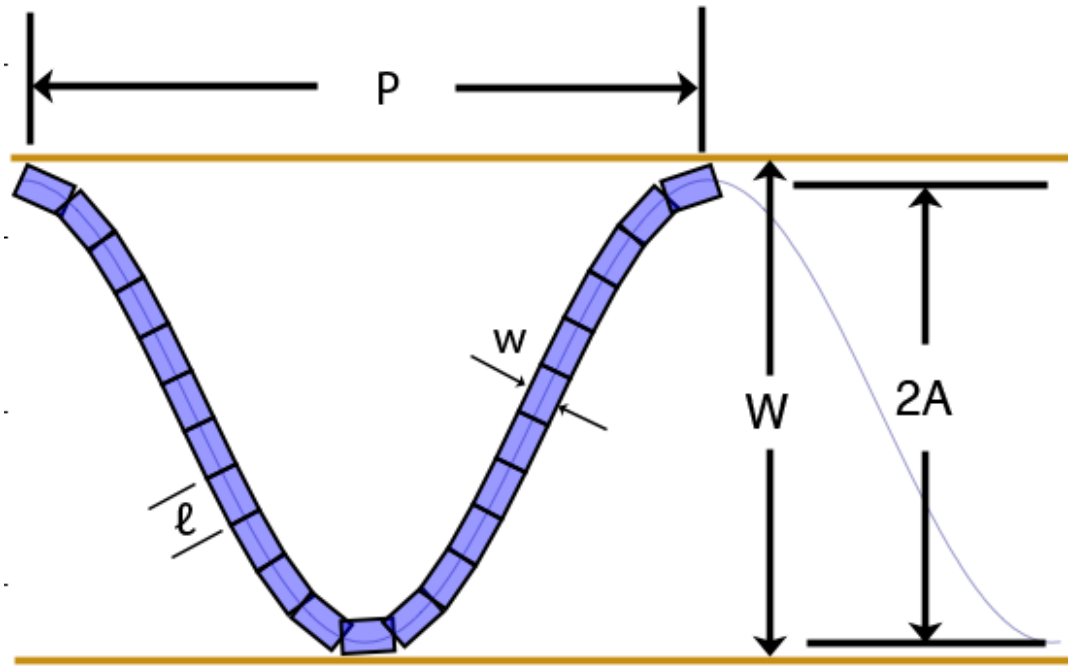


Figure 5.19: Parameterization of 3-point stable anchor in a smooth pipe.

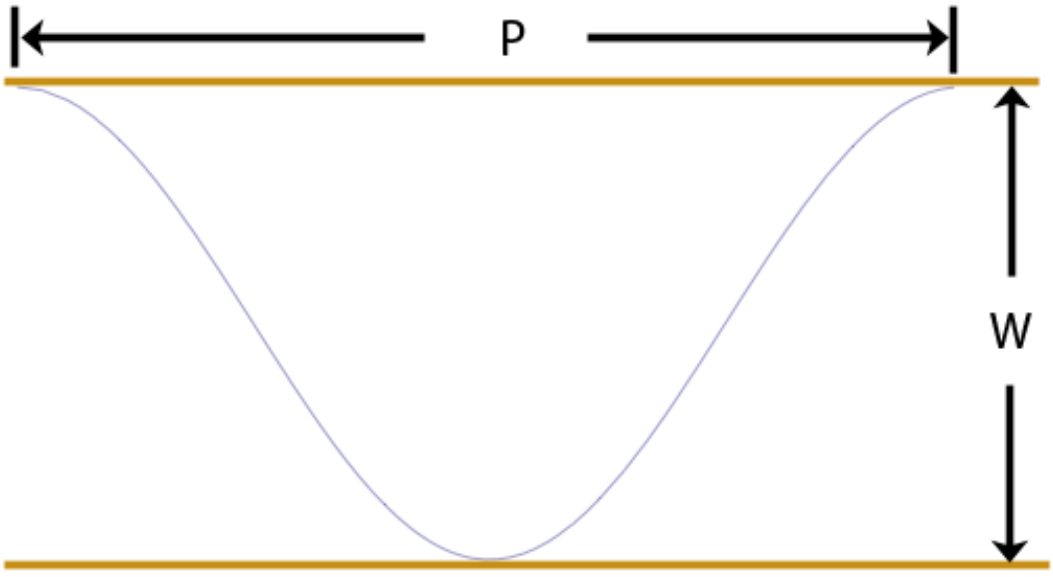


Figure 5.20: 3-point stable anchor with $w = 0$, $l = 0$, and $n = \infty$..

Equation 1 is the equation of the curve with P the period, W the width of the pipe. We can plug $f(x)$ into the equation for computing arc length shown in Equation 2 and 3:

$$L = \int_0^P \sqrt{f'(x)^2 + 1}\ dx \tag{5.10}$$

$$L = \int_0^P \sqrt{\left(\frac{-W\pi}{P}\sin\left(\frac{2\pi x}{P}\right)\right)^2 + 1}\ dx \tag{5.11}$$

This integration can not be solved analytical and must be solved by plugging in values for P and W and computing L numerically. In our control methodolgy, P is usually held fixed and the width of the environment is always changing, so W is always varying. Here we show a plot holding P fixed for various values while changing the value of W in Figure 5.21.

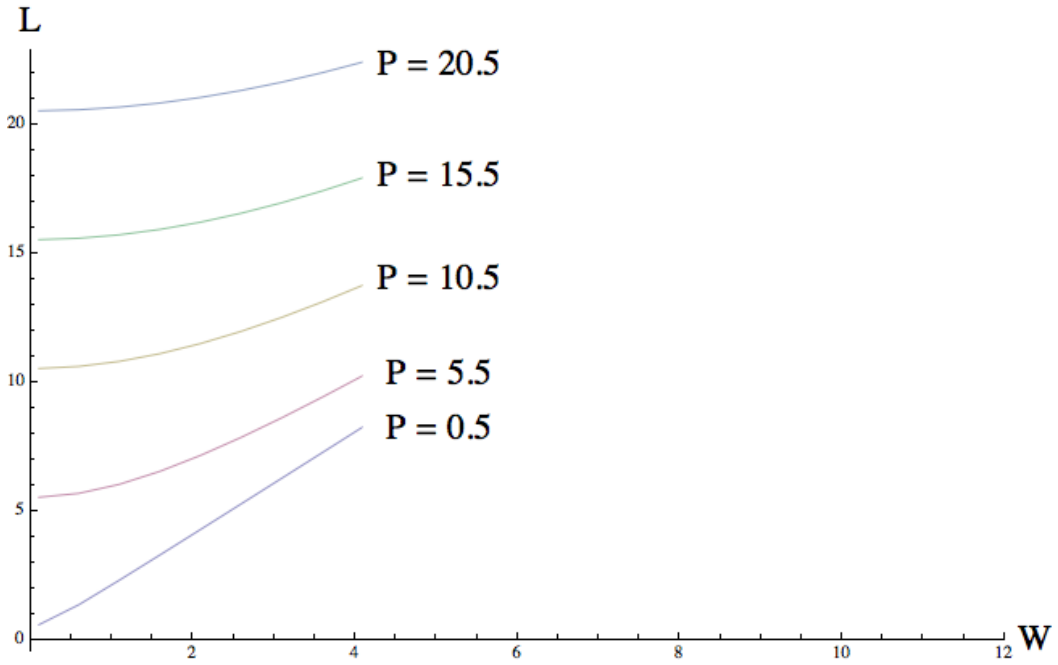


Figure 5.21: Plot of snake arc length L for various values of W and P.

This plot shows that when pipe width $W \to \infty$, the length L converges to a constant slope of $\frac{dL}{dW} = 2$ and $L = 2W\,, \forall P$ as $W \to \infty$. This can be seen by approximating the 3-point anchor and cosine curve as a triangle with a base of P and a height of W, and the vertices on the anchor points. Increasing W will achieve the above results.

The difference in L becomes large for different values of P when W is small. Once W becomes sufficiently large, W dominates and the period becomes less of a factor. Given that our application involves tight quarters, we are likely to find typical values of W to be small and P will vary depending on the context.

5.12.2 Case: Snake with Width

We now consider the case where $l = 0$, $n = \infty$, but $w > 0$. This in effect creates a continuous snake robot with infinite joints but has a thickness to it. To determine the total length of the snake given a pipe width W, a control period P, and a segment width of w, we first determine the equation of the curve:

$$f(x) = \frac{W - w}{2} \cos\left(\frac{2\pi x}{P}\right) \tag{5.12}$$

Since the snake centers its body on the curve, it has space of $\frac{w}{2}$ on either side of it. We start the cosine amplitude at $\frac{W}{2}$ and substract $\frac{w}{2}$ as the width of the snake increases. This results in equation 4.

If we plug the $f(x)$ into the arc length equation, we get the snake length for given W, P, and w:

$$L = \int_0^P \sqrt{\left(\frac{-(W-w)\pi}{P} \sin\left(\frac{2\pi x}{P}\right)\right)^2 + 1}\; dx \tag{5.13}$$

Holding $P = 1$, and increasing the pipe width W for different values of w, we get the following plot shown in Figure 5.22. This plot captures the intuition that a snake robot is not able to explore a pipe that is thinner than the robot's body width. However, once

you enter a pipe where $W > w$, the plot takes on the same curve as Figure 5.21. For large enough W, $L = 2(W - w)$ and $\frac{dL}{dW} = 2$.

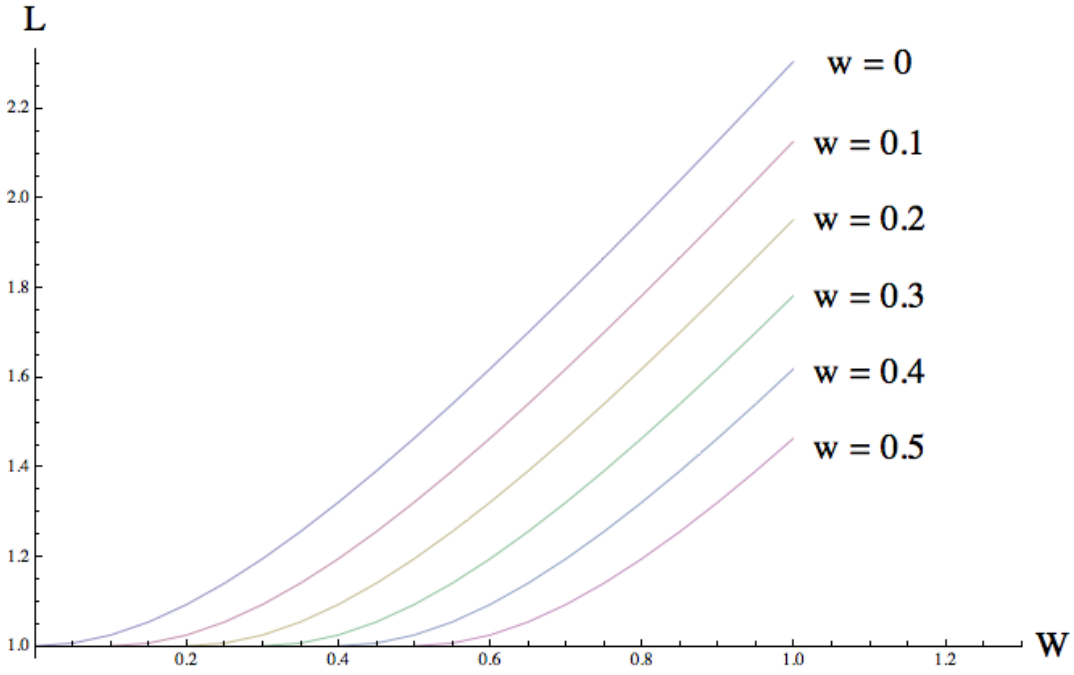


Figure 5.22: Plot of snake length L while $P = 1$, for various values of W and w.

5.12.3 Case: Snake with Segments

We now consider the case where the robot composed of discrete segments of a given length l. Up until now, we have represented the snake as a curve function. However, with segments, the robot needs to fit its body to be placed on the curve. Our current approach assumes a monotonic curve in the x-direction and the joint axes are placed directly on the curve. In this example, we assume that curve fitting is perfect and unique.

Again we look at the 3-point anchor situation determine how the length of the snake changes with different parameters. Instead of length though, we are computing the number of required snake segments.

There are two approaches. The first is to examine the situation analytically, deriving an equation for the number of segments given period P, pipe width W, and segment length l. The second is to run an algorithm that performs the curve fitting given a curve and lets us determine the required number of segments. We do both.

Analytically, we can easily compute an upper bound for the number of segments. If we have already calculated L, we can determine that we will need at most n segments shown in equation 6.

$$n = \left\lceil \frac{L}{l} \right\rceil \tag{5.14}$$

At most n segments will be required in the worst case where the segments lay directly on the curve. However, in most cases, the segments will be cutting corners to put both ends on top of the curve. This will result in the required number of segments being less than the worst case.

Algorithmically, we can determine the actual number of segments required for a given configuration. Given a monotonic curve γ, draw a circle c at the start of the curve with radius l and origin o. Take all the intersection points between c and γ. Take only the intersection points where $p_x > o_x$. Select p with the smallest x value. Place a segment going from o to p. Set $o = p$ and repeat until there is no more curve. This looks like the following in Figure 5.23.

5.13 Sensing Behavior

In order to capture as much information about the environment as we can, we need to sweep the robot's body around as much as possible while trying to cover as much space as possible without losing our reference pose to the global frame. Therefore, we have devised a behavior assemblage that separates responsibility for anchoring on one half of the snake and for probing the environment with the other. This assemblage is shown in Figure 5.24. The resulting behavior is shown in Figure 2.1.

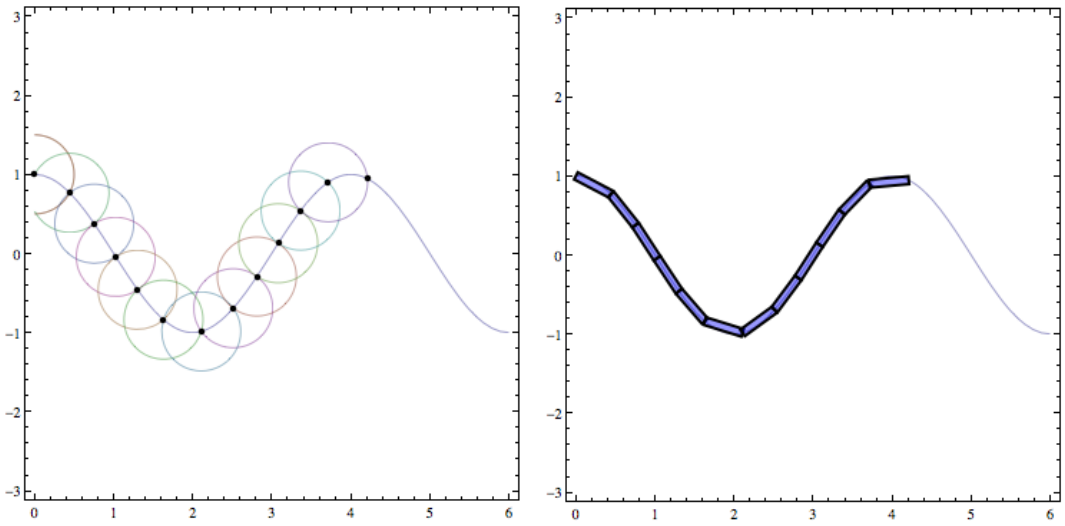


Figure 5.23: Intersecting circles along the line of the curve. Demonstration of curve fitting algorithm.

The resultant behavior sweeps the front end of the snake back and forth, impacting both sides of the pipe. The front is gradually extended to increase the probe reach down the pipe. Once the extension has reached the maximum, the behavior is terminated after returning the snake back to its original fully anchored Rest-State posture.

At regular intervals during the sweeping behavior at the end of each Transition termination, the posture is recorded for later processing. The end result is an ordered list of posture vectors representing each snapshot of the robot during the behavior. These postures are then processed into a map.

We describe each behavior in the assemblage.

Curl: The sweeping behavior is performed by setting a series of consecutive joints all to a 30 degree angle, resulting in a curling motion. In a confined environment, the result of the commanded joints usually results in the snake impacting the walls of the environment. By steadily increasing the number of joints to perform this sweeping motion, the reach of the snake is gradually increased and more space is sweeped out.

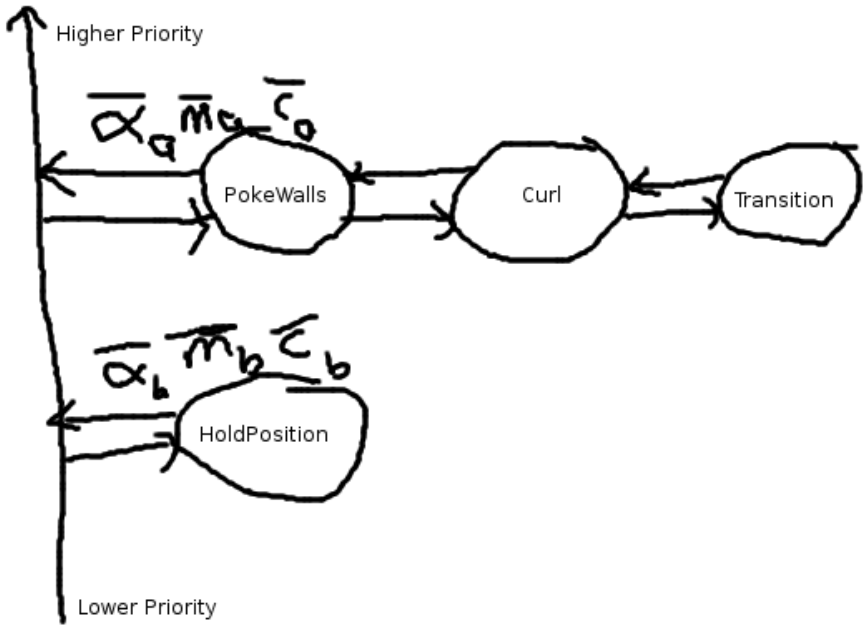


Figure 5.24: PokeWalls behavior assembly.

The Curl sub-behavior is responsible for commanding these joint angles given the specified joint range by the parent. It is also responsible for executing a simple contact detection heuristic in the course of the curling behavior. We describe both here.

Given the top joint j_t specified by the parent, the Curl behavior gives commands to the joints $(j_0...j_t)$, setting $\alpha_i = 0$. All α are set to the following sequence of angles: $(0, 25, 30, 0, -25, -30, 0)$. Between each movement command we use the Transition sub-behavior to manage the motion as a linear interpolation between postures. We also wait for the tip of the snake's position to become stable for executing another movement transition. At the end of the movement sequence and once the tip has become stable, the Curl behavior then terminates and waits to be reset by the parent.

For the contact detection heuristic, once the joints have been set to $\alpha = 25$, the tip of the snake's position p_a is computed kinematically. The joints are then set to $\alpha = 30$ and the tip's position p_b is again computed kinematically. The Cartesian distance between p_a and p_b is computed. If the value is negligible, we assume that the tip of the snake has

made contact with the walls. Otherwise, we do not assume contact has bee made. The repeat this for the negative angle commands on the other side of the pipe.

The way the sweeping behavior is performed is by setting the commanded position of all the joints from the tip of the snake down to a top joint simultaneously from 0 to 30 degrees. This results in a "curling" behavior which gives the subbehavior its name. The joints are set at angles of 0, 25, 30, 0, –25, –30, 0.

PokeWalls: The PokeWalls behavior is responsible for specifying the range of joints that perform the Curl operation, instantiating and configuring the Curl behavior. The PokeWalls behavior is configured with the sweeping direction on the snake, starts the Curl behavior with the joint range specified by $j_t = 6$. It then passes through the outputs of the Curl behavior.

At the termination of the Curl behavior, PokeWalls increments j_t by 2 and restarts the Curl behavior with the new parameters. If $j_t = 10$, we then begin decrementing j_t by 2 until we go back to $j_t = 6$. Once the Curl has been executed for the second for $j_t = 6$, we terminate the PokeWalls behavior.

The PokeWalls behavior is also responsible for setting the maximum torque for each of the curling joints to be compliant. This ensures that the posture of the snake complies to the environment and gives us as much information as possible about the structure of the free space. If we do not have compliant joints, the curling behavior can result in disruption of the anchors and loss of our reference poses.

Transition: This is the same behavior as described in Chapter 2. It takes an initial and final posture and outputs a series of postures that provides a smooth transition between the two using interpolation. Its parent behavior is Curl which instantiates it and resets it whenever it needs to perform a new move.

HoldPosition: This is the same behavior as described in Chapter 2. It is initialized with the complete posture of the snake at the beginning of the behavior. This posture is usually the result of the Rest-State stage of the Adaptive Step locomotion process. It continually outputs the anchored posture. Its output is subsumed by the output from

the PokeWalls behavior in a precedence merge. It is 2nd in the behavior ordering as shown in Figure 5.24.

Chapter 6

Building Maps

6.1 Problem

Now that we have established how to sense the environment, how the robot will be controlled, and how we represent the position and orientation of the robot space, we would now like to synthesize these components into an overall map. We can formalize the definition of the mapping problem as follows.

Given the posture images $I_{1:t}$ and the actions $A_{1:t}$, solve for each $X_{1:t}$ as shown in Figure 6.1. At each anchored position X_k between locomotion actions A_k, we have a posture image I_k for both the front and back probe sweeps respectively. These posture images created while the robot is traveling through the environment need a map representation suited to the robot's experience.

We define the actions of the robot to be $A_{1:t} = f, b, f, f, f, , b, b$ where:

$$A_i = \begin{cases} f, & \text{forward locomotion step} \\ b, & \text{backward locomotion step} \\ \varnothing, & \text{no move (switch sweep side)} \end{cases}$$

The pose $X_i = (x_i, y_i, \theta_i)$, is the position and orientation of the robot's posture frame P with respect to the global frame G. The posture frame is computed from the posture curve, β_i, described in chapter 4.

Each posture image I_i represents the swept void space of the environment, described in chapter 2. Each I_i may have some spatial landmarks, described in chapter 3.

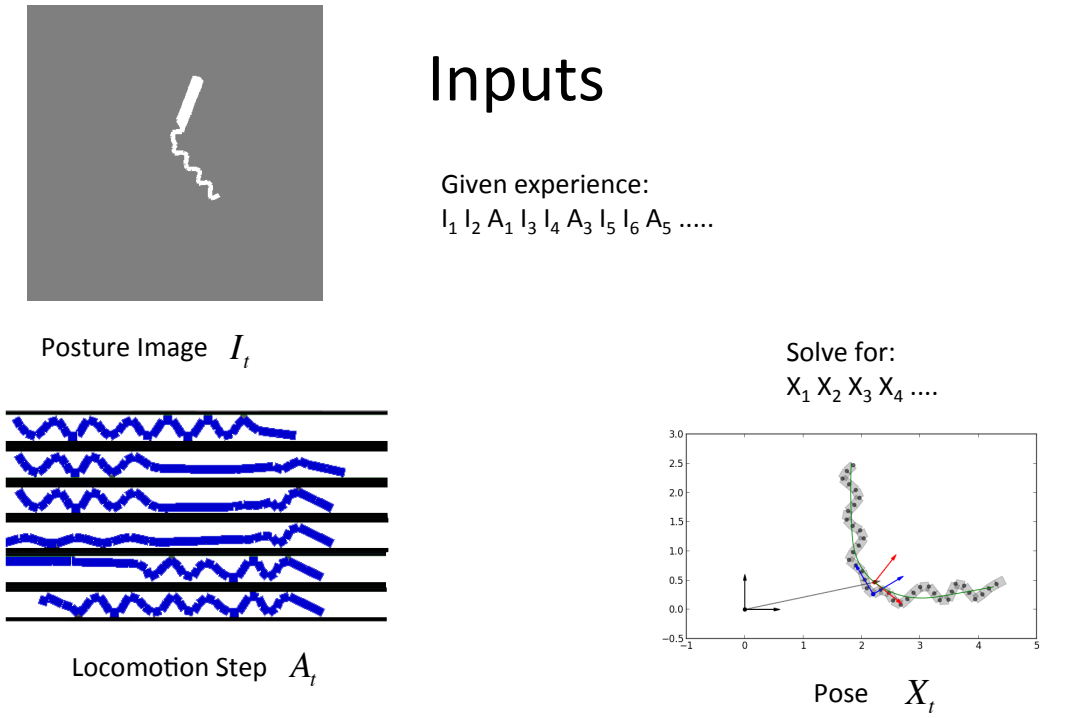


Figure 6.1: Mapping inputs

Figure 6.2: Posture Images to Spatial Curves

For our mapping method, we often use the *spatial curve* C_k, derived from the *posture image* I_k. Though we don't explicitly show it, the computation of $C_k = \text{scurve}(I_k)$, shown in Figure 6.2, is implied throughout the rest of this dissertation.

In order to build a map of the environment, we require sensor features that can be used to build geometric relationships between observations of the environment at different poses. In many existing SLAM algorithms, an abundance of landmark features can be used to build correspondence between different observations.

In our approach, we only have a sparse amount of landmarks corresponding to spatial features which indicate the presence of junctions. There are no landmarks at all for the vast majority of the pipe-like environments. Existing feature-based SLAM algorithms cannot operate on such sparse data. Furthermore, the range of sensing of the robot is very limited, and does not give a very large view of the environment. This causes many locations in the environment to look identical. We need a mapping approach that can map with such limited data, using only spatial curves and spatial features.

6.2 Naive Method

Our first approach is what we will call the *naive method*. In this approach, we apply constraints between the current pose and the previous pose. Specifically, given the previous forward and backward sweep poses of X_{k-3} and X_{k-2}, we apply geometric constraints to the two new poses, X_{k-1} and X_k. This gives us a best guess estimate of X_{k-1} and X_k from the last estimated poses.

The geometric constraint between a pair of poses is achieved by performing an Iterative Closest Point (ICP) fit between each poses' spatial curve. This takes advantage of the fact that consecutive poses of the snake should be partially overlapping as an invariant condition. Therefore, consecutive poses' spatial curves should also be partially overlapping. If we perform ICP on the partially overlapping curves, this should give us a good estimate of the robot's new pose.

The only question that remains is, how much should the curves partially overlap? The amount of overlap can be determined by making estimates on the motion traveled during a locomotion step. If there are any significant curve features that are shared by the two poses, the ICP algorithm should find those and correctly align the two spatial curves together.

There are two constraints we make. The *in-place constraint* between two poses is made when no locomotion occurs, and the *step constraint* between two poses separated

by a locomotion step. The step constraint is made between two poses that are sweeping in the same direction. Specifically, only the forward sweep poses are constrained using the step constraint, X_{k-2} to X_k.

Given the experience, $E = \{I_0, A_0, I_1, A_1, I_2, A_2, I_3\}$, where $A_{0:2} = \{\varnothing, f, \varnothing\}$, $X_0 = (0, 0, 0)$, find the values of $\{X_1, X_2, X_3\}$. Using the constraint function, we can compute the new poses with the series of calls:

$$X_1 = \text{overlap}(X_0, I_0, I_1, A_0) \tag{6.1}$$

$$X_2 = \text{overlap}(X_0, I_0, I_2, A_1) \tag{6.2}$$

$$X_3 = \text{overlap}(X_2, I_2, I_3, A_2) \tag{6.3}$$

We describe each of these two types of constraints and show how a map is formed.

6.2.1 Overlap Function

We will explain in detail about how the **overlap** function works. The objective of the function is to find a geometric transform between the two spatial curves, C_a, C_b, generated from the input posture images, I_a, I_b, given the initial pose X_a and the action A. It then returns the result X_b.

The function includes an ICP search to find the tightest overlap between the two spatial curves, but is agnostic about the amount of overlap that occurs. That is, fully overlapped and partially overlapped curves are equally valid results, so long as the curves have similar tangent angles at each pair of closest points. The pseudo-code for the function can be seen in algorithm 6.

In each spatial curve, we find the point closest to the local frame's origin, $(0, 0)$. For the curve C_a^a in the O_a frame, we find the closest point p_a^a. Likewise, p_b^b in the O_b frame.

Depending on the value of A, we need to displace p_a along the curve C_a forwards or backwards or not at all. This is to give an initial estimate of how far the robot has

Algorithm 6 Overlap Function

Input values
$X_a \Leftarrow$ initial pose
$I_a \Leftarrow$ initial posture image
$I_b \Leftarrow$ current posture image
$A \Leftarrow$ action
Compute spatial curves from posture images
$C_a \Leftarrow \text{scurve}(I_a)$
$C_b \Leftarrow \text{scurve}(I_b)$
Find closest point p_a^a on C_a^a to $(0, 0)$
Find closest point p_b^b on C_b^b to $(0, 0)$
Displace point p_a^a on C_a^a by action estimate
Transform C_b^a such that the point p_b^a is the same as p_a^a
Given two parameters u and γ, for the arc length and orientation of p_b^a on C_a
Perform ICP to find the best u and γ.
Convert u and γ into $(x_b^a, y_b^a, \theta_b^a)$
Find X_b^a
return X_b^g

traveled and the location of X_b. For our purposes, if $A = f$, we displace an arc length of 0.5. If $A = b$, we displace -0.5. If $A = \varnothing$, we don't displace at all.

We then convert the point p_a from cartesian coordinates to a new tuple, (u, γ), where γ is the orientation of the tangent angle of C_b^a at p_b^a, and u is the arc length of p_a on C_a. The ICP problem then becomes a 2 dimensional search space to find the right u and γ to achieve the best overlap.

After performing the ICP search algorithm, the resulting u and γ can then be converted to a new X_b^a. This is then converted to the global frame and returns X_b^g as the result of the overlap function.

6.2.2 Iterative Closest Point

As part of the overlap function, we perform the ICP algorithm to make the two curves overlap. Our ICP algorithm has specific requirements to our particular task. We want to be able to partially overlap two curves, without the ICP algorithm maximizing the amount of overlap.

To accomplish this, we need to constrain the type of closest point comparisons, and tailor the cost function to our desired outcome. Specifically, we want the section of the two curves that are overlapped to be completely aligned and be able to escape local minima to find a better fit.

For the cost evaluation, we used Generalized ICP, originally developed by [**?**]. Generalized ICP allows the evaluation of cost, weighting each of the points by a covariance to describe their orientation. Therefore, lower evaluation costs are achieved by matching points that are similarly aligned. The covariance of the point is computed by finding the tangent angle of the curve at that point. Then a covariance matrix is generated from the direction vector.

Our ICP algorithm is as follows. First, we match closest points between the two curves. For every point on C_a, we find the closest point on C_b. If the closest point is greater than some threshold $D = 0.2$, or the difference in tangent angle between the two points is greater than $\epsilon = \frac{\pi}{4}$, then the match is discarded. Otherwise, the match is accepted. We perform the same closest point search for every point on C_b, and find the closest point on C_a with the above restrictions. Finally, we discard any duplicates.

Now that we have the set of closest point pairs, we can evaluate the cost with a special distance function. If v_p is the normalized direction vector of the point's orientation with an angle of ϵ_p, the covariance of each point is computed by the following equation:

$$C_p = \begin{bmatrix} 0.05 & 0 \\ 0 & 0.001 \end{bmatrix} \begin{bmatrix} cos(\epsilon) & -sin(\epsilon) \\ sin(\epsilon) & cos(\epsilon) \end{bmatrix} \tag{6.4}$$

To compute the optimal transform of a set of matched points, given some transform $\mathbf{T}$ between the two sets of points and a displacement vector d_i between each particular match, the optimized $\mathbf{T}$ is computed in Equation 6.5. The details of the derivation of this cost function can be found in [**?**].

$$\mathbf{T} = \arg\min_{\mathbf{T}} \sum_i d_i^{(\mathbf{T})^T} (C_i^a + \mathbf{T} C_i^b \mathbf{T}^T)^{-1} d_i^{(\mathbf{T})} \tag{6.5}$$

Iterating through the ICP algorithm continues, matching new points and finding the optimal transform **T** until the difference between the current and previous cost value is below a certain threshold. It terminates and returns **T**.

6.2.3 Constraints

Now that we have fully defined our overlap function, there are two types of constraints we can make with this function: the *in-place constraint* and the *step constraint*.

The in-place constraint, shown in Figure 6.3, performs an overlap with two collocated poses that separate the forward and backward sweeping behavior. Given pose X_k, find pose X_{k+1}:

$$X_{k+1} = \text{overlap}(X_k, I_k, I_{k+1}, \varnothing) \tag{6.6}$$

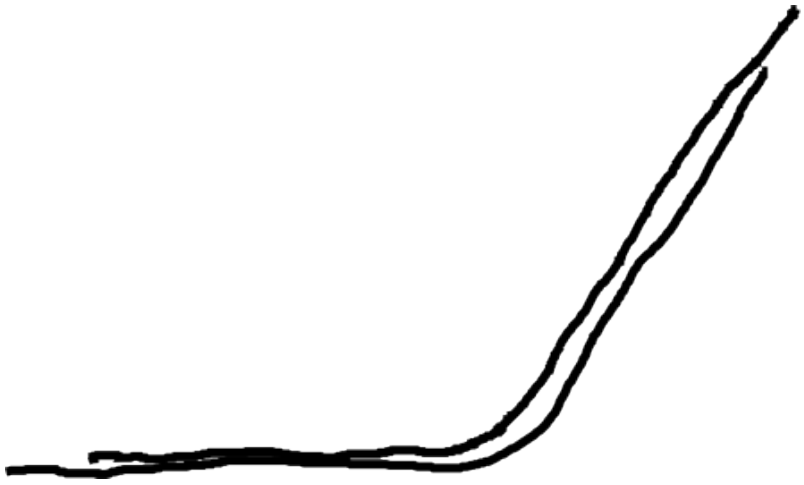


Figure 6.3: In-place Constraint

The step-constraint, shown in Figure 6.4, is performed on two poses where a locomotion step is between them. In particular, we only perform the overlap function between two poses that sweep in the same direction. In our case, we only perform the step constraint between two forward sweeping poses. Given the pose X_k and a forward locomotion step, we find the pose X_{k+2} with the following function call:

$$X_{k+2} = \text{overlap}(X_k, I_k, I_{k+2}, f) \tag{6.7}$$

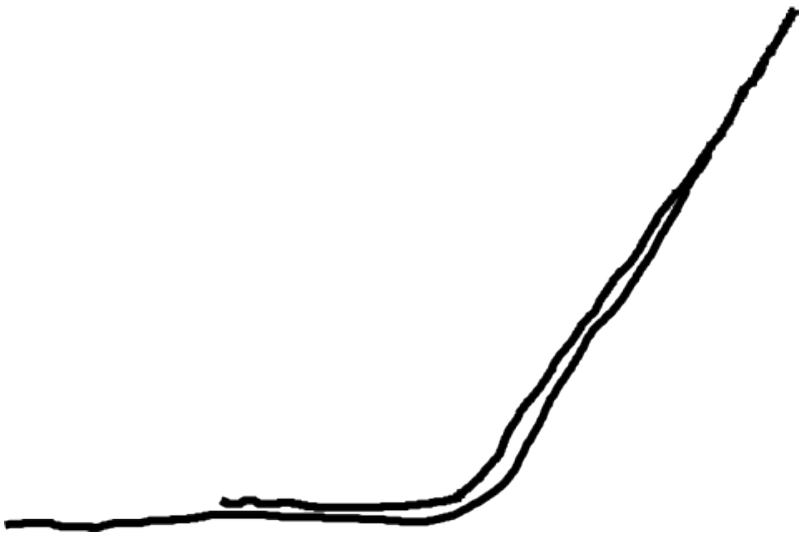


Figure 6.4: Step Constraint

These two function calls give us the tools to find all poses using only the most recent pose.

6.2.4 Results from Naive Method

Some examples of the results from naive mapping method are shown in Figure 6.5.

The pseudocode describing the naive method mapping approach is shown in algorithm 7.

6.3 Axis Method

In the previous section, we discussed the basics of creating a map using constraints between the current and most recent pose. We assumed that our robot was traveling forward through a pipe-like environment while building a map of its traversal.

Each geometric constraint between poses was either an in-place constraint between a pair of poses at the same anchored location, or a step constraint between two consecutive

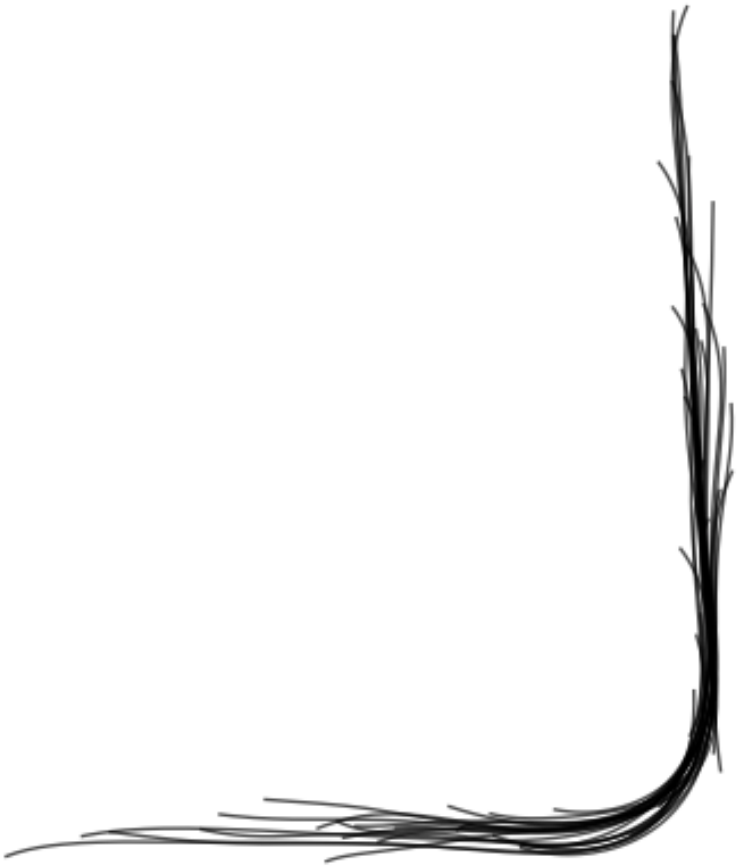


Figure 6.5: Naive Method Results

Algorithm 7 Naive Method

$X_0 \Leftarrow (0, 0, 0)$
$k \Leftarrow 0$
Sweep forward and capture I_k
Sweep backward and capture I_{k+1}
$X_{k+1} = \text{overlap}(X_k, I_k, I_{k+1}, \varnothing)$
while True **do**
    $k \Leftarrow k + 2$
    Locomotion step forward
    Sweep forward and capture I_k
    Sweep backward and capture I_{k+1}
    $X_k = \text{overlap}(X_{k-2}, I_{k-2}, I_k, f)$
    $X_{k+1} = \text{overlap}(X_k, I_k, I_{k+1}, \varnothing)$
end while

forward sweep poses. In the latter case, the step constraints encode the movement of the robot through the environment. In the former case, the in-place constraint encodes the poses being at the same location.

However, if our robot should decide to reverse direction and backtrack through the previously explored environment, using only the constraints we have previously described, the backward path will begin to drift from the forward path as shown in Figure X. We need some approach to constrain the old poses from the forward traversal to the new poses of the backward traversal.

This problem is called both the data association problem and the loop-closing problem. For the data association problem, we wish to be able to identify we are observing something we have seen before. In the loop-closing problem, we want to ensure that the old poses and the new poses of the same location are accurately and consistently integrated into the map. This would fix the problem shown in Figure X and hopefully produce Figure Y.

A standard approach to this problem is to solve the feature correspondence problem by matching feature landmarks between different poses and then adding constraints between the features. Though we have the sparse spatial landmarks, in our approach, they are insufficient to solve this task consistently. We must find a solution that works with just the posture images of each pose.

The simplest manifestation of this problem is the case where a robot travels down a single pipe and then backtracks in the other direction. Our interest is for the new poses created while traveling backward to be correctly constrained with the old poses created by originally traveling forward.

Instead of creating geometric constraints between individual poses, another approach we can take is to make a constraint between a new posture image and all of the other posture images in aggregate. In this way, we can avoid the difficulty of making individual pairwise constraints between poses.

The approach we take is to find the union of all past posture images together and compute the alpha shape from that. From the alpha shape we then compute the medial axis. The result is a much larger version of the spatial curve representation of individual posture images.

To constrain the pose of the new posture image, simply perform ICP between the pose's spatial curve and the past whole map medial axis. The result can be seen in the Figure X. Like the naive method, we have the invariant condition that the new pose's spatial curve partially overlaps the global medial axis. Therefore, there exists a section of the global medial axis curve that the new pose at least partially overlaps. For poses that are backtracking through previously visited territory, the new pose will be completely overlapped by the global medial axis.

This mapping method looks for the best possible result for the current pose. All past poses are fixed and unable to be changed. Should we ever make a significant mistake in the placement of the current pose, this will be permanently reflected in the map for the future.

Even using this simplistic approach, a significant translational error along the axis of travel in a straight section of the environment does not have severe consequences to the map. Along a straight section of the pipe, the current pose can be placed anywhere along that section with little consequence to the structural and topological properties of the map. The only thing that is affected is the current best estimate of the robot's pose.

The mapping system is able to localize the pose much better with respect to curved shape features in the environment. If there are sharp junctions or curved sections, the current pose can be localized with much better accuracy.

Once the current pose's posture image is given it's most likely location, it is added to the existing map. The union of posture images is calculated, the alpha shape is computed, and the global medial axis is computed. If the robot is pushing the frontier of the map, this will result in an extension of the global medial axis. If the posture image's pose is an incorrect position, the resultant global medial axis can be corrupted

and result in kinks (distortions, discrepancies). These kinks can compound future errors and result in even further distorted maps.\

6.3.1 Generating the path

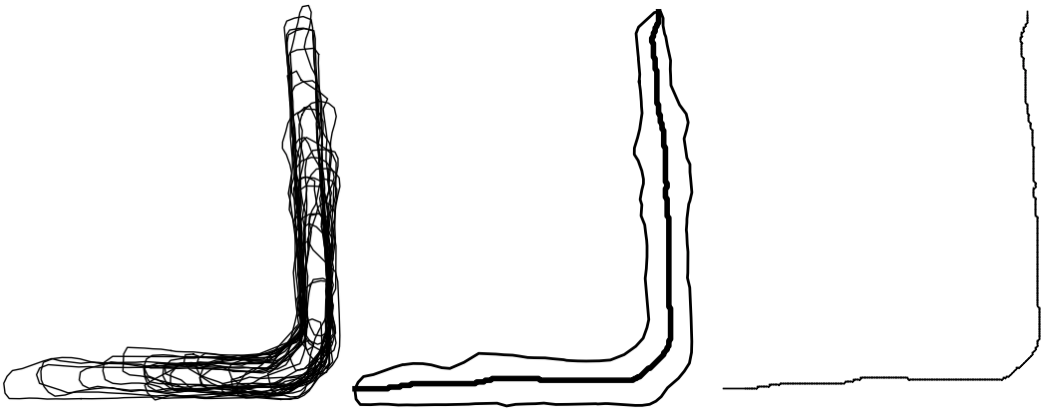


Figure 6.6: Computing Axis from Union of Posture Images

We define the axis to be the medial axis of the union of the globally situated alpha shapes of the posture images. The axis is computed from a collection of poses that represent the past robot's trajectory.

The topology of an axis is computed from the union of all of the poses' alpha shapes. The union of the alpha shapes is populated with points in a grid and the medial axis is computed using the skeletonization algorithm [**?**] .

Given the set of points that are the skeletonization image, the pixels are converted into nodes of a graph, and edges are added between them if they are adjacent or diagonal neighbors. The pixel indices are converted back into global coordinates to give us a set of points that are nodes in a graph, edges connecting to their neighbors.

We reduce the graph first by finding its MST. Then we prune the tree by snipping one node off of leaves. This removes any obvious MST noise artifacts but preserves real branches of the tree.

Paths are computed between each pair of leaves. Each leaf-to-leaf path on the medial axis is extrapolated at the tips to intersect and terminate at the boundary of the alpha shape polygon. This adds further usable information to the produced curve and in practice is a correct extrapolation of the environmental topology.

We select the longest leaf-to-leaf path. This becomes the axis for purposes of our axis method mapping.

6.3.2 OverlapAxis Function

Now that we have axis computed from the union of alpha shapes of posture images, new poses can now be constrained to the axis. In the event that the robot backtracks, it will utilize the full history of robot's poses to localize the robot's new position.

We define a new function similar to **overlap**, called **overlapAxis**, that functions the same way as the overlap function. However, instead of overlapping spatial curves with spatial curves, it overlaps a single pose's spatial curve to the existing map axis.

The objective of the function is to find a geometric transform between the spatial curves, C_k, and the map axis, C_g. Unlike the **overlap** function, **overlapAxis** must have an initial guessed pose for X_k. The guess of X_k, $\hat{X}_k$, is computed from the original step constraint in the previous section. Therefore, the procedure to find the new X_k and it's companion X_{k+1} is as follows.

$$\hat{X}_k = \text{overlap}(X_{k-2}, I_{k-2}, I_k, f) \tag{6.8}$$

$$X_k = \text{overlap\_axis}(\hat{X}_k, I_k, C_g) \tag{6.9}$$

$$X_{k+1} = \text{overlap}(X_k, I_k, I_{k+1}, \varnothing) \tag{6.10}$$

The same ICP point matching, cost and search algorithms are run in **overlapAxis** described in the previous section.

6.3.3 Results

Our results show that various examples of single path environments will produce usable maps at all stages of the mapping. The primary source of error occurs from coaxial translational error along the path. Poses can fall somewhere along the path that is off by some amount.

Regardless if there is a severe positional error of a pose, the resultant map will still be topologically correct. The axis still indicates the travel trajectory of the robot. We can also see that the more curve features in the environment the better because this gives salient local minima for the ICP search algorithm to find. Environments that are straight will produce more error in where the pose is localized since all locations will look equally likely.

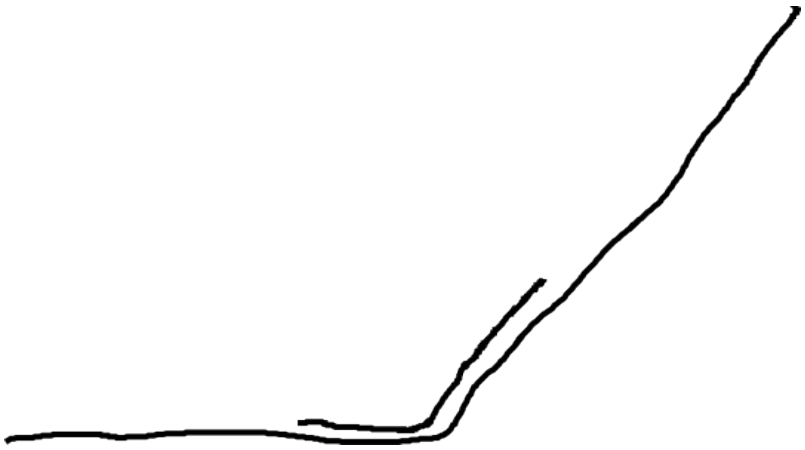


Figure 6.7: Axis Method Results

The pseudocode describing the axis method mapping approach is shown in algorithm 8.

Algorithm 8 Axis Method

$X_0 \Leftarrow (0, 0, 0)$
$k \Leftarrow 0$
Sweep forward and capture I_k
Sweep backward and capture I_{k+1}
$X_{k+1} = \text{overlap}(X_k, I_k, I_{k+1}, \varnothing)$
generate C_g from $I_0 : k+1$ and $X_0 : k+1$
while True **do**
    $k \Leftarrow k + 2$
    Locomotion step forward
    Sweep forward and capture I_k
    Sweep backward and capture I_{k+1}
    $\hat{X}_k = \text{overlap}(X_{k-2}, I_{k-2}, I_k, f)$
    $X_k = \text{overlap\_axis}(\hat{X}_k, I_k, C_g)$
    $X_{k+1} = \text{overlap}(X_k, I_k, I_{k+1}, \varnothing)$
    generate C_g from $I_0 : k+1$ and $X_0 : k+1$
end while

Chapter 7

Mapping with Junctions

7.1 Problem

The axis method approach works well for environments that are just a single continuous path. However, it breaks down in the event the environment has a junction as shown in Figure 7.1. The axis method assumes that a single continuous curve fully describes the environment. The only localization problem is fitting the new pose's spatial curve onto this existing axis with ICP.

If ICP is run with a spatial curve that branches off of the existing axis, it will result in finding a local minima that is incorrect as shown in Figure 7.2. Clearly, this is a fundamental break down of the axis method approach. We need some method that can handle the case of arbitrary types of junctions.

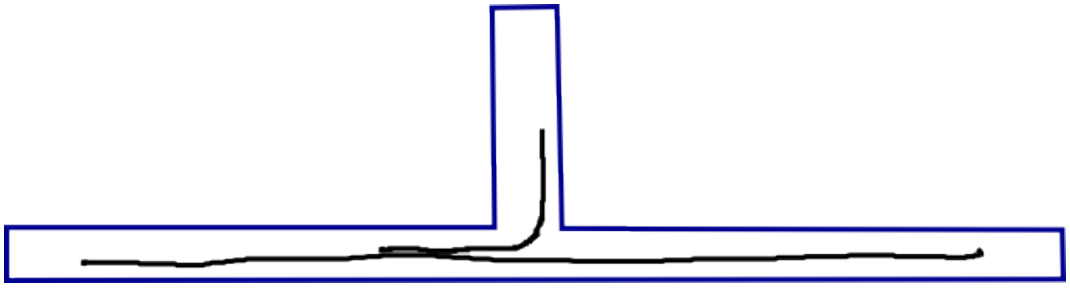


Figure 7.1: Existing Axis with Newly Discovered Junction

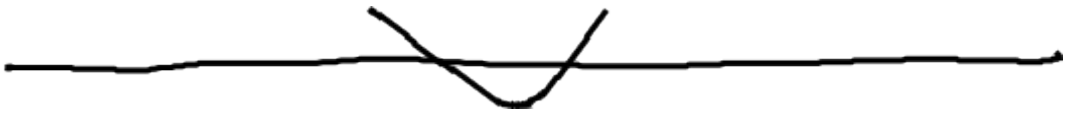


Figure 7.2: Axis Method Failure with Junction

One of the unique challenges of mapping sensor-challenged environments with junctions is that they are only partially observable. That is, there is no direct observation using void space that will give a compete representation of the properties of the junction. The robot may even be passing through a junction without detecting it as such. Only by passing through all arms of the junction over time are we able to infer the location and properties of the junction.

Though we can reason and safely map long tubes with hard turns or curved and undulating paths, if we add junctions to the mix such as Y and T junctions, recognizing the encounter and the parameters of the junction becomes especially challenging.

We define a junction to be the confluence of over two pipes at various angles. For instance, an L bend we do not consider a junction, but a T is a junction. To see how this is challenging, imagine the Figure [X] where the robot's body is within the L junction. Similarly, the robot's body is in the T-junction in the same configuration. Both position's give us the shape of the environment correctly. However, we can only partially view the junction at any given time.

In order to sense and completely characterize the T-junction, the robot would be need pass through or back up and make an effort to crawl down the neglected path. If we knew that the unexplored pipe existed, this would be an easy proposition. As it stands, it is very difficult to distinguish an opening from an obstacle.

There are deliberative ways to completely map and characterize the environment as we travel. The work of Mazzini and Dubowsky [cite] demonstrate a tactile probing approach to harsh and opaque environments. Though this option is available to us, this has it's own challenges and drawbacks.

For one, tactile probing is time consuming. A robot would be spending all of its time touching the walls of the environment in a small range and not enough time exploring the environment in the larger range. We decided very early on in our research [Everist2009] that although contact detection approaches have proven their effectiveness in producing

detailed environmental maps, they are not a practical solution to exploring complex environments that won't take multiple days to complete.

Similarly, the act of probing and sweeping the environment has the possibility of disrupting the position of the robot and adding defects to the contact data locations. Tactile probing is particularly vulnerable to this since it involves actual pushing against obstacles and possibly disrupting the robot's anchors.

For all of these reasons, it is desirable to find an approach for mapping environments with junctions when the junctions are only partially observable. That is, at best, we do not know if we are in a junction until we come through it a second time. Often times, junction discovery is serendipitous because openings look identical to obstacles in a free space map. Without deliberate tactile probing, junction discovery will always be serendipitous for rapid mapping.

Evidence for the existence of a junction is given by a spatial curve partially overlapping a section of the global medial axis and then diverging from it at a sharp angle. In fact, the relationship between a spatial curve and the axis it's being fitted to gives us clues on whether or not there is a junction. If we examine the different types of overlapping events of a curve on a larger curve, we see there are three different possibilities as shown in Figure 7.3.

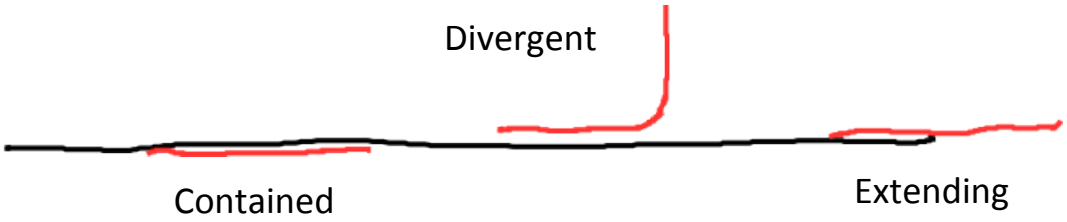


Figure 7.3: Curve Overlap Events

We must recognize that there are three possible ways for a new pose's medial axis to fit onto an existing global medial axis. It can either 1) completely overlap, **contained** on the axis, 2) partially overlap by **extension**, pushing forward past the termination

point of the axis, or 3) partially overlap by **divergence**, curving off the axis at a sharp angle signifying a junction-like feature.

A fundamental primitive problem we have is determining when two partially overlapping curves diverge from each other. Divergence between two curves is the point at which the overlapping curves separate. It is not a complete overlap but a partial overlap of two noisy curves. The challenge is defining under what conditions two curves are considered to be diverging and distinguish that from just contained or extending. We define the function to be COMPUTEDIVERGENCE().

The difficulty associated with this task is that there is no discrete event to determine a divergence. It is measured by passing a threshold that must be calibrated to the task. The thresholds are both sensitive to cartesian distance and angular difference between the diverging curve and the divergence point on the host curve.

The algorithm begins by finding the closest point distance of each point on the candidate diverging curve to the host curve. We examine the closest point distances of both tips of the candidate curve. If the closest point distance of the tip to its companion is above a threshold, DIV_DIST, and the difference of tangent angles is also above a threshold, DIV_ANG, then we classify this curve as diverging from its host. For our work, $DIV\_DIST = 0.3$ and $DIV\_ANG = \frac{\pi}{6}$.

If the curve is not diverging, then it is either extending or contained. To determine which, we compare the identity of the matched points of both of tips of the candidate curve. If either matched point is the tip of the host curve, then it is extending. Otherwise, it is contained.

7.2 Junction Method

The approach we use to handle junctions in the environment is called the **junction method**. We first need to introduce a new map representation.

7.2.1 Skeleton Maps

To represent and keep track of junctions and any instance of branching off from the main global medial axis, we introduce a new map representation called a skeleton map. First we define the **skeleton** to be the medial axis of a union of posture images that permits multiple junction features. In the previous section, our medial axis of the entire environment represented a single path where there were no junctions. Once a divergence event is found, a new skeleton is created that represents the branching off. The coordinate frame origin is placed on the original skeleton and becomes the *parent* of the new skeleton.

Corresponding to the three cases of how a spatial curve overlaps an existing global medial axis, 1) either the pose is contained within an existing skeleton, 2) the pose extends the skeleton, or 3) the pose diverges and is added to a new skeleton that branches off. Given the nature of the skeleton structure, their is a parent-child relationship between skeletons, with the stem being the root skeleton. All skeletons except for the root, have an origin placed on its parent skeleton.

The skeleton is generated from the same computation as computing the global axis. However, in the axis computation, we selected the longest path from leaf to leaf. In the case of the skeleton, we keep the whole tree. This allows the representation of junction features if they are needed. We create new skeletons whenever a new spatial curve diverges from the parent skeleton and does not fit on to any existing skeleton. An example of a skeleton map is shown in Figure 7.5 and Figure 7.6.

We say a pose is a member of a skeleton T_p if it has some overlap with T_p but does not overlap a child, T_c, of T_p. That is, if any portion of a pose overlaps a child T_c, the pose is a member of T_c and not T_p. A pose can be a member of more than one skeleton if there are more than one child skeletons. The member skeletons can be siblings or any other combination where the skeletons are not direct ancestors or descendants.

7.2.2 Generating Skeletons

To build the skeleton has the same computation as in subsection 6.3.1. However, in this case, we do not find the longest path. We instead keep the whole tree.

Algorithm 9 Generate Skeleton from Posture Images

function SKELETONIZE($X_{0:k}, I_{0:k}$)

$I_{0:k} \Leftarrow$ posture images
$X_{0:k} \Leftarrow$ global poses

for $i = 0$ to k **do**
  $H_i \Leftarrow$ALPHASHAPE(I_i) ▷ alpha shapes from posture images
end for

Uniformly distributed points inside alpha shape polygons
$M_{0:k} \Leftarrow$ POINTSINPOLYGON($X_{0:k}, I_{0:k}$)

Compute alpha shape of union of points
$\hat{H} \Leftarrow$ ALPHASHAPE($M_{0:k}$)

Fill points into alpha shape and make image
$\hat{I} \Leftarrow$ POLYGONTOIMAGE($\hat{H}$)

Skeletonize the image
$\hat{T} \Leftarrow$ SKELETONIZE($\hat{I}$)

Compute set of leaf-to-leaf paths
$\hat{C}_{1:j} \Leftarrow$ COMPUTESPLICES($\hat{T}$)

return $\hat{T}, \hat{C}_{1:j}$

Shown in algorithm 9, we compute the union of the posture images by first finding the union of their alpha shape polygons, filling in the polygons with uniformly distributed points, and then finding the alpha shape of the union of points. From this we find the skeletonization of the alpha shape polygon converted to image format. We find all the possible paths between leafs and return them as a set of j "splices".

7.2.3 Adding New Poses to Skeleton Map

Now that we have described how to generate a skeleton given a set of data, we need to explain how we go about adding new poses to a map. In the base case, the first two poses are added to the initial skeleton. The initial skeleton then becomes the root skeleton and ancestor to all future skeletons. A skeleton map can have one or more skeletons. We explain how to add a pose to a single skeleton map and then explain how to add a pose to a multi-skeleton map.

Given one existing skeleton, to add a new pose, we select all splices for that particular skeleton. For each splice, we attempt to fit to that curve using the existing ICP techniques from the OVERLAPAXIS() function. For each splice, we have a fitted result. We evaluate the best fit according to some evaluation criteria. We select the fitted pose for the best splice as the robot's new position and orientation after a locomotion step. This is the motion estimation phase similar to subsection 6.3.2.

After the motion has been estimated for the new poses, then we perform the divergence test described in section 7.1. After the test, the new pose's spatial curve is either contained, extending, or diverging from the selected splice of the current skeleton. If it is either contained or extending, we add it to the existing skeleton. If it is diverging, we create a new skeleton with this pose as its only member and set the current skeleton as its parent.

That is the single skeleton case. Now if the skeleton map has more than one skeleton, instead of the splices from a single skeleton, we find the splices of all the skeletons merged together. That is, for the m trees, $\hat{T}_{1:m}$, merge them together by adding edges between nodes in the global frame that are less than a distance threshold. In our case, the threshold distance is 0.2.

Then we find the global terminals. Though we can take the terminals of each of the individual skeletons, often times they are contained within another skeleton or they are similar to an existing terminal. We only collect terminals that are not contained within

another skeleton. If a terminal is similar to another terminal, we only take one. This results in a sparse set of unique terminals of the entire map.

Given these global terminals, we then find the terminal-to-terminal path of the merged graph of skeletons. These paths become our new global splices shown in Figure 7.7 and Figure 7.8. For n terminals, there are $\binom{n}{2}$ possible splices. We then perform the same set of motion estimation ICP fitting and best splice selection as the single skeleton case. The final result is the best location and orientation for the robot given the map.

Now that the best estimated location is chosen for this pose, it needs to be added to the map. This means we need to determine which skeleton the pose should be added to, and whether or not a new skeleton needs to be created if it is diverging because of a new junction.

The first thing we do is performance the divergence test. Unlike in section 7.1 where divergence is determined with respect to a curve, here we wish to determine if the pose's spatial curve diverges from a set of skeletons. We accomplish this by defining a new function call GETPOINTSOUPDIVERGENCE(). It is similar to the earlier divergence function, but instead we convert all of the global skeletons into a set of points and compute the closest skeleton point to all of the locations on the spatial curve.

The algorithm begins by finding the closest point distance of each point on the candidate diverging curve to the set of skeleton points. We examine the closest point distances of both tips of the candidate curve. If the closest point distance of the tip to its companion is above a threshold, DIV_DIST, then we classify this curve as diverging from its host. For our work, $DIV\_DIST = 0.3$.

If the curve is *not* diverging, then it is either extending or contained. To determine which, we compare the identity of the matched points of both of tips of the candidate curve. If either matched point is a skeleton terminal, then it is extending. Otherwise, it is contained. If the curve is diverging, then we need to create a new empty skeleton and this pose as its first member.

In the event of no divergence, we have to determine which skeleton this pose should be added to. First, we need to determine which skeletons the spatial curve is overlapping. The spatial curve could be overlapping a single skeleton or a combination of overlapping skeletons. We perform an overlap test for each individual skeleton to determine if any of its points are within range of the spatial curve. If they are, then we consider this skeleton as a candidate for the new pose.

Given the set of all overlapping skeletons, we select the most junior skeletons such that, none of the selected skeletons are ancestors of each other. This means that we select single children or we select two siblings to add the pose to. The reason for this will be discussed later in chapter 8.

7.2.4 Algorithm

The pseudocode describing the axis method mapping approach is shown in algorithm 10. The portion to estimating the motion of all possible splices on the skeleton map is in algorithm 11. The portion devoted to adding the pose to the right skeleton is in algorithm 12. The portion regarding generating the skeleton is in algorithm 13.

Algorithm 10 Junction Method

$X_0 \Leftarrow (0, 0, 0)$
$k \Leftarrow 0$
$m \Leftarrow 0$
$S_0 \Leftarrow \emptyset$
$C \Leftarrow \emptyset$ ▷ set of global splices
Sweep forward and capture I_k
Sweep backward and capture I_{k+1}
$X_{k+1} \Leftarrow \text{OVERLAP}(X_k, I_k, I_{k+1}, \varnothing)$

Members of skeleton set $S_m \Leftarrow S_m \cup \{0, 1\}$
$\hat{T}_{0:m}, \hat{C} \Leftarrow \text{GENSKELETONS}(m, S_{0:m}, X_k, I_k)$

while True **do**
    $k \Leftarrow k + 2$
    Locomotion step forward
    Sweep forward and capture I_k
    Sweep backward and capture I_{k+1}
    $\hat{X}_k \Leftarrow \text{OVERLAP}(X_{k-2}, I_{k-2}, I_k, f)$
    $X_k \Leftarrow \text{MOTIONONSKELETONS}(m, T_{0:m}, C, \hat{X}_k, I_k)$
    $X_{k+1} \Leftarrow \text{OVERLAP}(X_k, I_k, I_{k+1}, \varnothing)$

    $m, S_{0:m} \Leftarrow \text{ADDTOSKELETONS}(m, S_{0:m}, T_{0:m}, X_k, I_k)$
    $m, S_{0:m} \Leftarrow \text{ADDTOSKELETONS}(m, S_{0:m}, T_{0:m}, X_{k+1}, I_{k+1})$

    $\hat{T}_{0:m}, \hat{C} \Leftarrow \text{GENSKELETONS}(m, S_{0:m}, X_k, I_k)$
end while

Algorithm 11 Motion Estimation

function MOTIONONSKELETONS$(m, T_{0:m}, C, X_k, I_k)$

$X_{best} \Leftarrow X_k$
for $\forall c \in C$ **do**
    $X_k \Leftarrow \text{OVERLAPAXIS}(\hat{X}_k, I_k, c)$
    evaluate X_k and set to X_{best} if best
end for
return X_{best}

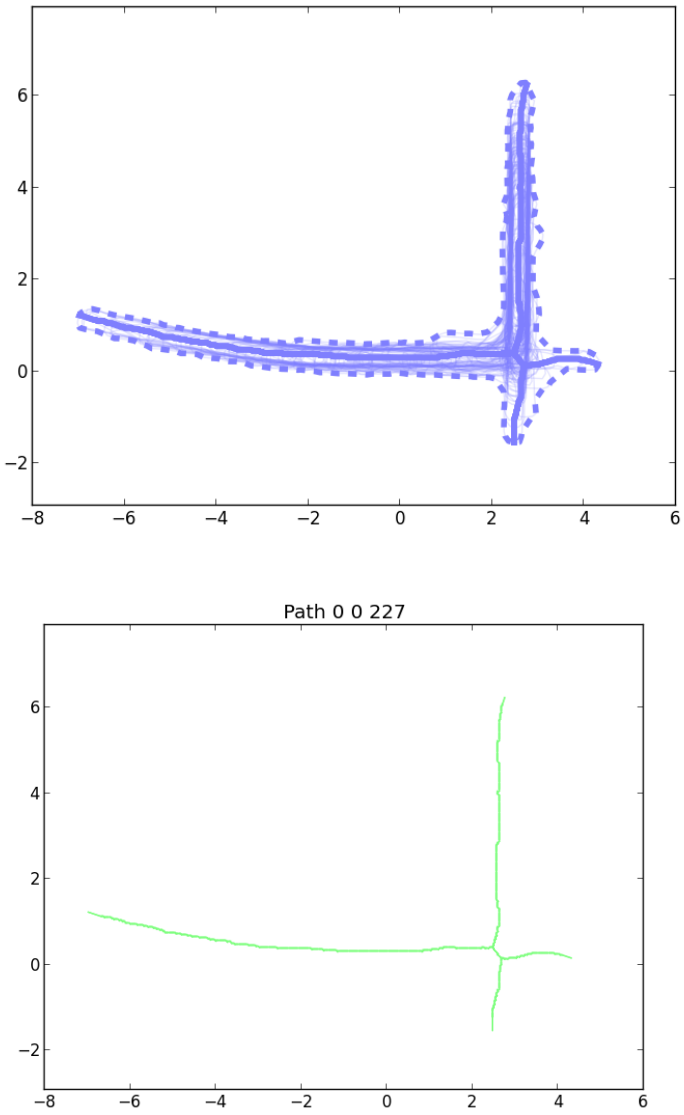


Figure 7.4: Skeleton Example

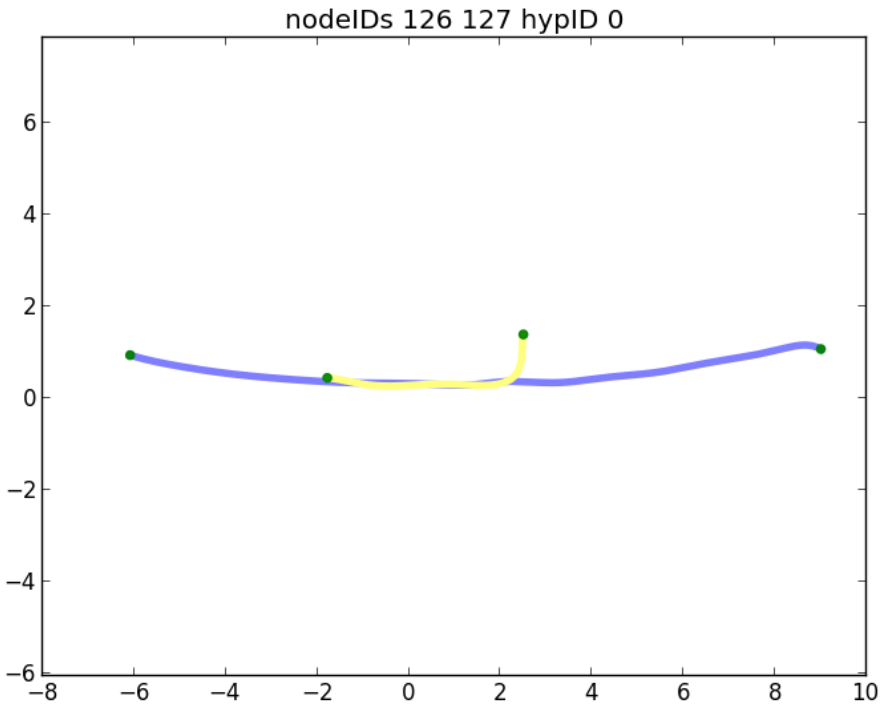


Figure 7.5: Skeleton Map 1

Algorithm 12 Add to Skeletons

function ADDTOSKELETONS($m, S_{0:m}, T_{0:m}, X_k, I_k$)

▷ this is a comment

$div_k \Leftarrow$ GETPOINTSOUPDIVERGENCE($\hat{T}_{0:m}, X_k, I_k$)
if div_k **then**
    $m \Leftarrow m + 1$
    $S_m \Leftarrow S_m \cup \{k\}$
else
    find member skeleton S_p
    $S_p \Leftarrow S_p \cup \{k\}$
end if
return $m, S_{0:m}$

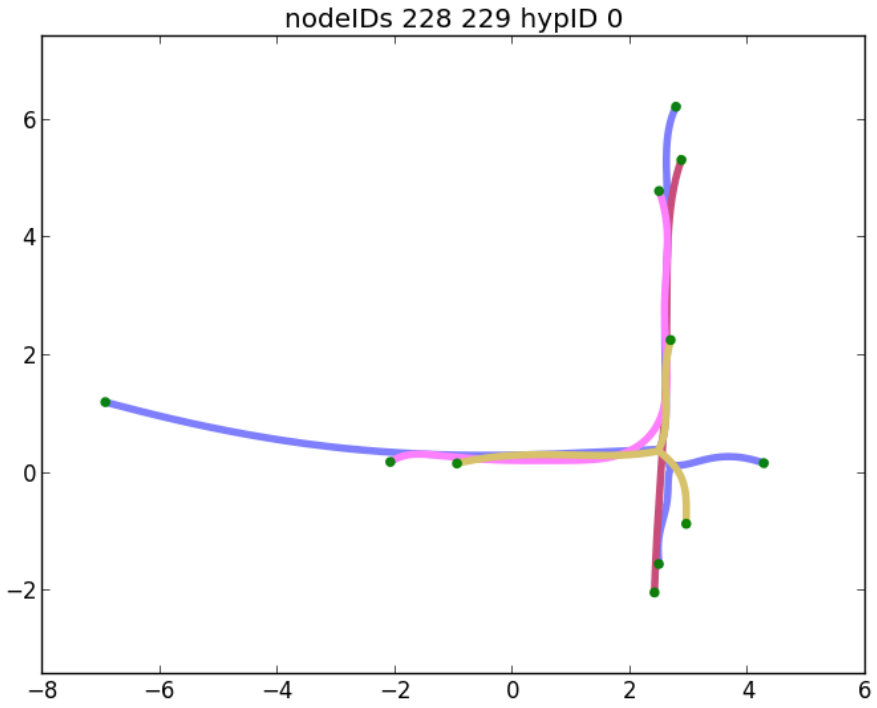


Figure 7.6: Skeleton Map 2

Algorithm 13 Generate Skeletons

function GENSKELETONS($m, S_{0:m}, X_k, I_k$)

for $n = 0$ to m **do**
  $\hat{X}_n \Leftarrow \emptyset$
  $\hat{I}_n \Leftarrow \emptyset$
  for $\forall i \in S_n$ **do**
    $\hat{I}_n \Leftarrow \hat{I}_n \cup I_i$
    $\hat{X}_n \Leftarrow \hat{X}_n \cup X_i$
  end for
  $\hat{T}_n, \hat{C}_{1:j} \Leftarrow$ SKELETONIZE($\hat{X}_n, \hat{I}_n$)
end for
return $\hat{T}_{0:m}, \hat{C}_{0:m}$

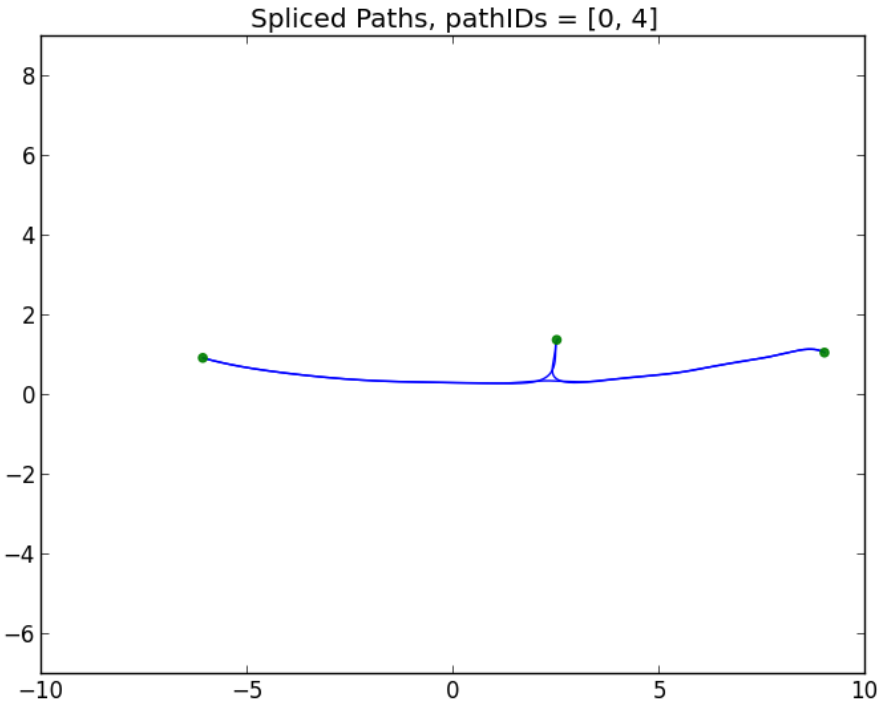


Figure 7.7: Skeleton Map Splices 1

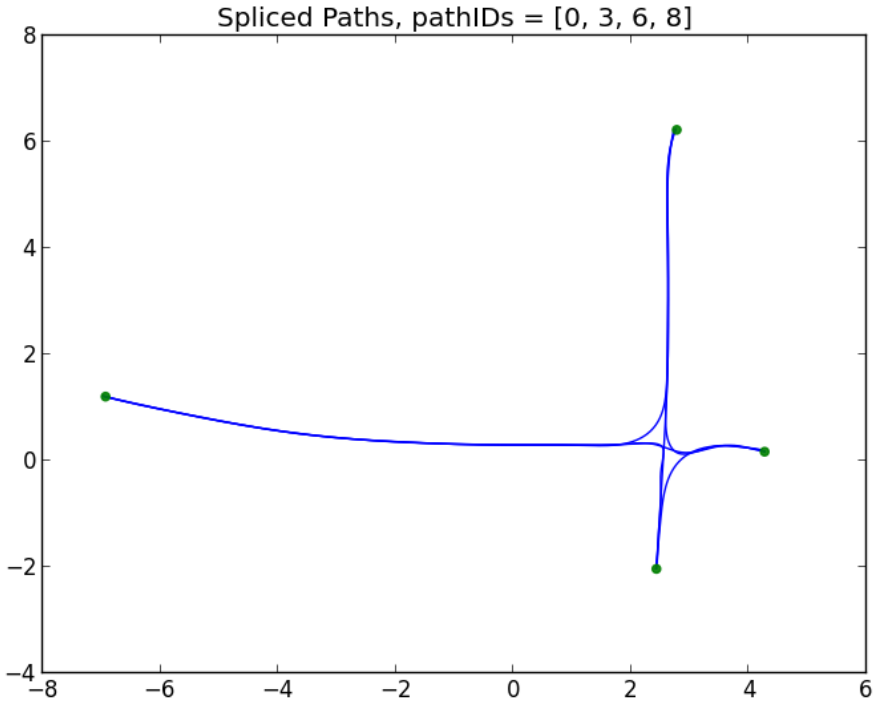


Figure 7.8: Skeleton Map Splices 2

Chapter 8

Searching for the Best Map

8.1 Problem

Though we have shown how to build a map with junctions in the environment, there are a number of ways for error to be injected into the map. In particular, we identify three possible ways for error to be added.

The first is that there is error in the pose of the robot. In this case, a combination of the last best pose and the current estimated pose gives a pathological estimation of the location and orientation of the robot. This could have innocuous results in straight and featureless environments or very severe errors in the case of the robot taking a wrong turn down a different branch of the map.

The second form of error is whether or not a divergence is indicative of an actual branch. Though a spatial curve is diverging from the skeleton map at the current pose of the robot, it does not necessarily mean that we have found a new junction. This could mean that the spatial curve would have a more proper *contained* fit somewhere else in the skeleton map. It may also indicate a junction that we have already encountered before. We need some way to take in account all three possibilities of a divergence: 1) a new junction, 2) an existing junction, or 3) no junction at all.

The third form of error that we consider is the actual location of a junction after the detection of a divergence event. After a divergence event, a new skeleton is created with respect to its parent based on the estimated pose of the diverging spatial curve. However, after the initial placement, it is useful to consider the spatial relationship between the two skeletons independent of a single pose. Since junctions are only partially observable, we cannot reconstruct the entire junction without multiple views and different

branching events. The spatial relation of the skeletons that compose the junction need to be changeable so that they can be later optimized into the correct configuration. For example, in the cross environment map, it is common to discover the arms of the junction in an unaligned fashion. Only some post-hoc optimization process given enough data can align them into their true configuration.

In this chapter, we introduce the **search method** which provides some way to handle this error and search for the best possible solution. Our approach discretizes the state space and uses ICP and evaluation functions to find the most consistent map given the observational data. Just like in the *junction method*, we use the whole history of the sensor data to make mapping decisions. However, this approach has the capacity to defer mapping decisions or remove branches that are unlikely.

8.2 Search Method

The *search method* is trying to estimate and optimize the consistency of the map defined by the following parameters: 1) $X_{0:t}$, the global poses of the robot, 2) $S_{0:m}$, the skeletons and their member poses, and 3) $b_{1:m}$, the spatial transform between parent and child skeletons. Finding the best map would involve making sure the poses are classified to their appropriate skeletons, the poses are placed so that their spatial curves are mutually overlapping, and that the relative spatial transform between the skeletons is made to produce a configuration of branches consistent with the observed data.

There are 5 steps in the *search method*. They are as follows:

1. Motion Estimation
2. Add to Skeleton Map
3. Overlap Skeletons
4. Localization
5. Merge Skeletons

We describe each step in detail after we first describe the parameterization of the discretized map space.

8.2.1 Parameterization

In order to perform a search of the best map, we need to define the parameters of the map we will be searching over. The ultimate goal is to search for the best possible values of $X_{1:t}$ to produce the most consistent map given the posture image observations of $I_{1:t}$ and the actions $A_{1:t}$.

To restate the experience of the robot, we have a series of actions:

$$A_t = \begin{cases} f, & \text{forward locomotion step} \\ b, & \text{backward locomotion step} \\ \varnothing, & \text{no move (switch sweep side)} \end{cases} \tag{8.1}$$

The robot has a series of observations that create posture images:

$$I_t = I_t(i,j) = \begin{cases} 1, & \text{if void space} \\ 0, & \text{if unknown} \end{cases} \tag{8.2}$$

$$0 \leq i \leq N_g, \quad 0 \leq j \leq N_g \tag{8.3}$$

where N_g is the number of pixels on a side of the posture image. The posture image can be reduced to a spatial curve representation:

$$C_t \Leftarrow \textsc{scurve}(I_t) \tag{8.4}$$

We want to solve for all poses in the global frame for each posture image:

$$X_t^g = (x_t^g, y_t^g, \theta_t^g) \tag{8.5}$$

However, it is easier to break the problem down and reduce the search space with some assumptions. The first way we break the problem down is grouping the poses into sets that represent the skeletons $S_{1:m}$. Each pose is then represented in the local frame of the skeleton whose origin is placed on some point on the parent skeleton.

$$S_i = \{\exists k \quad \text{s.t.} \quad 0 \leq k \leq t\} \tag{8.6}$$

The global pose of each of member of the skeleton is computed by using the relative skeleton transforms found by local frame origin placed on the parent skeleton:

$$b_i = (x_i, y_i, \theta_i) \tag{8.7}$$

For instance, for the root skeleton S_0, the origin is placed at the global origin $b_0^g = (0, 0, 0)$. So the global pose X_k^g and the local pose X_k^0 are identical. However, for a child skeleton S_1, whose parent is S_0, the location of its origin is some pose b_1^0 in the local frame of the parent. Therefore, the global pose of some X_k^g from X_k^1 for $k \in S_1$ is $X_k^g = T * X_k^1$ where T is a transform derived similarly to Equation 4.11.

The problem of global registration of branches to form junctions becomes the problem of finding the transform between skeletons. Since a skeleton is created when a divergence occurs, finding the correct transform between two skeletons becomes finding the correct location of a branching arm. This gives us a solution to the hard problem of correctly registering the branches of a cross junction.

Since the transform is a 3 degree of freedom search space, we can reduce to one dimension by finding the child skeleton origin placed on some curve of the parent skeleton as seen in Figure 8.1. We call the location on the parent skeleton the *control point* b_i and serves as the origin of the skeleton's local frame. The search space remains one dimension so long as we assume that the angle remains constant. Any amount of angular transform results in the parent and child skeleton no longer overlapping cleanly, so this is a reasonable assumption in practice.

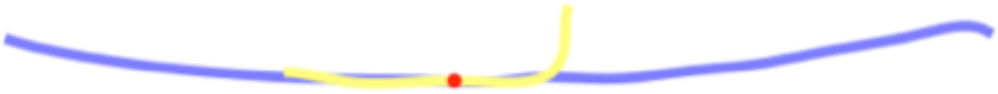


Figure 8.1: Control point on parent skeleton indicates location of child frame with respect to parent.

Since the robot is in a confined space and all the sensed area is void space, it stands to reason that any possible location of the robot will be close to or on one of the skeleton splices. With this observation, it is possible to reduce the dimensionality of the possible pose locations and make the localization problem much simpler. To reduce the search space of optimizing the pose, we consider a uniform distribution of initial guesses that are located on the splices of the global skeleton map as seen in Figure 8.2.

The pose is originally represented by 3 values: (x, y, θ). We can reduce the dimensionality of the pose by restricting the pose to be a point on a skeleton splice. Specifically, given a curve, a pose on the curve can be represented by two variables, the angle θ and the arc distance u. Therefore, the current pose of the robot given the current map is represented by the vector (u, θ) and its designated skeleton splice curve c.

This approach assumes that any possible pose will be located on a skeleton splice curve. This approach can encounter difficulties if this is not the case. Any situation where the pose would not be on the skeleton curve is either, the skeleton is distorted by improperly localized previous poses, or the robot is not in the center of the pipe. The latter case is possible if the pipe is too wide for the robot to anchor to the edges. One of our original assumptions is that environment is not too wide for the robot to anchor.

The spacing between the initial samples indicates how deeply the search will be formed. With a sparse distribution, the search will be quick, but chances are the correct pose will not be found. With a dense distribution, search time will be longer because each pose needs to be evaluated. The distribution constitutes the initial guess of the

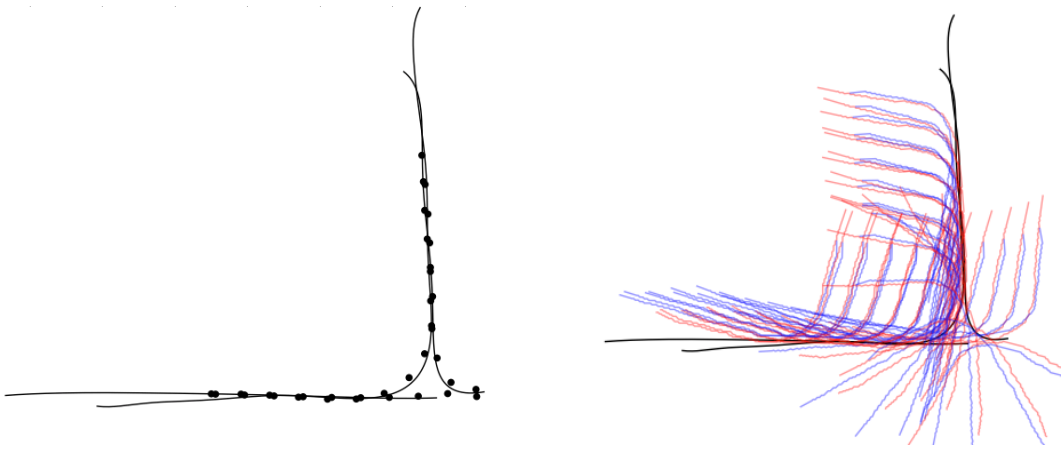


Figure 8.2: Uniform distribution of initial pose guesses on global skeleton splices.

pose and does not indicate a likely final result. Later use of ICP will allow us to find the correct pose in a continuous search space. However, ICP requires an initial value to start the search, so this distribution is our source for initial values.

8.2.2 Motion Estimation

Given the initial distribution of poses shown in Figure 8.2, we want to first model the movement of the robot given the current action A_t. For every candidate pose shown in Figure 8.2, motion is achieved by displacing the pose by a fixed distance along the curve of each of the splices the pose is contained on. The fixed distance displaced is an empirically determined value from the average locomotion distance of the robot. For our current configuration, we use the value 0.8.

If a candidate pose is on 3 different splices, we will have 3 different displacement locations for the given candidate. This multiplies the total number of possible locations. Given all of these candidates, we then need to evaluate them based on some fitness function. We tailor our fitness function for consistency between the current poses and the previous poses, as well as a Gaussian forcing function that biases selection for a pose that is displaced from the previous pose.

The evaluation function we use includes a number of metrics that we will need to define individually. They are as follows:

1. Landmark Cost
2. Angular Difference
3. Overlap
4. Contiguity Fraction

Landmark Cost

This metric finally uses the spatial features derived in chapter 3. Our goal is to compute the sum of the distances of every feature to every other feature. Therefore, a low sum would indicate the features are all very close to each other and would likely mean that all of the poses that have evidence of junction are close together and building the junction properly.

However, each spatial feature has a different reliability in its location indicating a junction. For instance, the bloom and arch features have similar quality and indicate the swelling inside of a junction. The bend feature is more difficult because the point of maximum angular derivative does not necessarily indicate being within the junction, just very close to it. So the bend feature is more prone to error and any cost sum should reflect this lower reliability.

We assign the covariance for the bloom and arch features, C_a, and the bend features, C_b, are defined respectively:

$$C_a = \begin{bmatrix} s_a^2 & 0 \\ 0 & s_a^2 \end{bmatrix} \quad C_b = \begin{bmatrix} s_b^2 & 0 \\ 0 & s_b^2 \end{bmatrix} \quad \text{where} \quad s_a = 0.3 \quad s_b = 0.9 \tag{8.8}$$

The distance between an arch feature $\bar{p}_a$ and a bend feature $\bar{p}_b$ can be found by the following equation:

$$\textsc{landmarkDist}(\bar{p}_a, \bar{p}_b, s_a, s_b) = \sqrt{\frac{|\bar{p}_a - \bar{p}_b|}{s_a^2 + s_b^2}} \tag{8.9}$$

To compute the sum of all the distances from every landmark to every other landmark, there are $N * (N-1)$ calculation if there are N landmarks:

Algorithm 14 Compute Landmark Cost

```
function computeLandmarkCost(p_{0:N}, s_{0:N})
sum ⇐ 0
MAXDIST ⇐ 7.0
for i = 0 to N do
    for j = i + 1 to N do
        d ⇐ |p̄_a − p̄_b|
        if d < MAXDIST then
            sum ⇐ sum + landmarkDist(p_i, p_j, s_i, s_j)
        end if
    end for
end for
return sum
```

Here, $MAXDIST$ is a maximum cartesian distance for including the distance between two points as part of our final cost. This is designed such that two junctions that are spaced apart do not erroneously try to align the spatial features of both and merge the two junctions into one. This restricts the types of environments we can map such that the junctions should be sufficiently spaced apart. However, it allows us to map multi-junction environments.

Angular Difference

Angular difference is a very intuitive metric. For the previous poses, X_{k-3} and X_{k-2}, and the current poses, X_{k-1} and X_k, what is the difference in orientation of the poses and their spatial curves $C_{k-3:k}$. Essentially, we want the consecutive poses to be similarly oriented because they should be overlapping.

For each pose and its spatial curve, we consider three different angles. The first is the angle from the pose θ_k. Then we consider the tangent angle of the tips of the spatial curve, ϕ_k and β_k for the forward tip and backward tip angles respectively.

To compute the angular difference, we compare forward sweep poses together and backward sweep poses together. Therefore, we compare X_{k-3} to X_{k-1} and X_{k-2} to X_k. We define the angular difference in algorithm 15.

Algorithm 15 Compute Angular Difference

function COMPUTEANGDIFF($\phi_i, \phi_j, \theta_i, \theta_j, \beta_i, \beta_j$)

$\phi_d \Leftarrow |\arctan(\sin(\phi_i - \phi_j))|$
$\theta_d \Leftarrow |\arctan(\sin(\theta_i - \theta_j))|$
$\beta_d \Leftarrow |\arctan(\sin(\beta_i - \beta_j))|$

$sum = \phi_d + \theta_d + \beta_d - \text{MAX}(\phi_d, \theta_d, \beta_d)$

return *sum*

We compute the sum of the two minimum difference angle. We do this because usually at least one of the differences is very large due to the robot traveling through a junction or turn of some kind. By filtering out the maximum difference angle, we get an angular consistency measure of two consecutive poses.

For the current poses X_k and X_{k-1}, we can compute the angular difference metric with the following calculations:

$$\text{angDiff}_k = \text{COMPUTEANGDIFF}(\phi_k, \phi_{k-2}, \theta_k, \theta_{k-2}, \beta_k, \beta_{k-2}) \tag{8.10}$$

$$\text{angDiff}_{k-1} = \text{COMPUTEANGDIFF}(\phi_{k-1}, \phi_{k-3}, \theta_{k-1}, \theta_{k-3}, \beta_{k-1}, \beta_{k-3}) \tag{8.11}$$

Overlap

The overlap is a measure of how closely a pose's spatial curve overlaps the assigned skeleton splice. If off by a fixed offset, the overlap sum would be really high. If it closely follows the splice curve, then the overlap sum would be near zero. It uses a similar cost

function as an ICP evaluation function, but instead it just uses the some of cartesian distances and only uses the maximum contiguously overlapping section.

In order to compute this measure, we need to find the maximum contiguous section of the spatial curve. If there are N uniformly spaced points composing the spatial curve, we are looking for the maximum contiguous sequence of points that all have closest distance to the splice curve less than some distance threshold D_c. Once we find that section, then we sum the closest distance of each of those points and divide by the number of contiguous points. This gives us our overlap sum sum_k as seen in algorithm 16.

Algorithm 16 Compute Maximum Overlap Contiguity Section and Sum

function COMPUTEMAXCONTIG(C_{splice}, C_k)

$D_c \Leftarrow 0.2$
$N \Leftarrow |C_k|$
$m \Leftarrow 0$
$sum \Leftarrow 0$
$m_{max} \Leftarrow 0$
$sum_{max} \Leftarrow 0$

for $i = 0$ to N **do**
    $p \Leftarrow C_k(i)$
    $D_{min} =$ CLOSESTDISTANCE(p, C_{splice})
    if $D_{min} < D_c$ **then**
        $m \Leftarrow m + 1$
        $sum \Leftarrow sum + D_{min}$

        if $m > m_{max}$ **then**
            $m_{max} \Leftarrow m$
            $sum_{max} \Leftarrow sum$
        end if
    else
        $m \Leftarrow 0$
        $sum \Leftarrow 0$
    end if
end for

$sum_k \Leftarrow \frac{sum_{max}}{m_{max}}$

return sum_k, m_{max}

Contiguity Fraction

This metric gives us the fraction of the spatial curve that is part of the maximum contiguous section. Taking the value algorithm m_{max} from algorithm 16, for a pose X_k and its spatial curve C_k, and $|C_k|$ is the number of points on the spatial curve, we can compute the contiguity fraction from the following equation:

$$\text{contigFrac}_k = \frac{m_{max}}{|C_k|} \tag{8.12}$$

This metric is not used for motion estimation, but will be used in later localization.

Evaluation

Given a whole series of candidate poses pairs X_{k-1} and X_k, with an associated skeleton splice C_{splice}, we compute the fitness evaluation with the following equation:

$$F(k-1,k) = -2*sum_{k-1} - 2*sum_k - \frac{\text{angDiff}_{k-1}}{\pi} - \frac{\text{angDiff}_k}{\pi} + 2*\frac{LM_{max} - LM}{LM_{max}} \tag{8.13}$$

This evaluation function minimizes the angular difference so that the two poses are angularly consistent with the previous poses. In addition, it seeks to minimize the amount of "play" in the overlap of the spatial curve with the skeleton splice. Finally, the equation also seeks to minimize the landmark cost. LM_{max} is the maximum landmark cost out of all the candidate poses. We select the pose with the highest cost evaluation as our selected pose after motion estimation as shown in Figure 8.3.

We can see a breakdown of evaluation of each metric for each pose on every splice shown in Figure 8.4. Each color curve on the graph indicates a particular splice. The x-axis indicates the arc length on the splice that the particular pose is located. Each splice is shown twice because we are evaluating a pair of poses together representing both

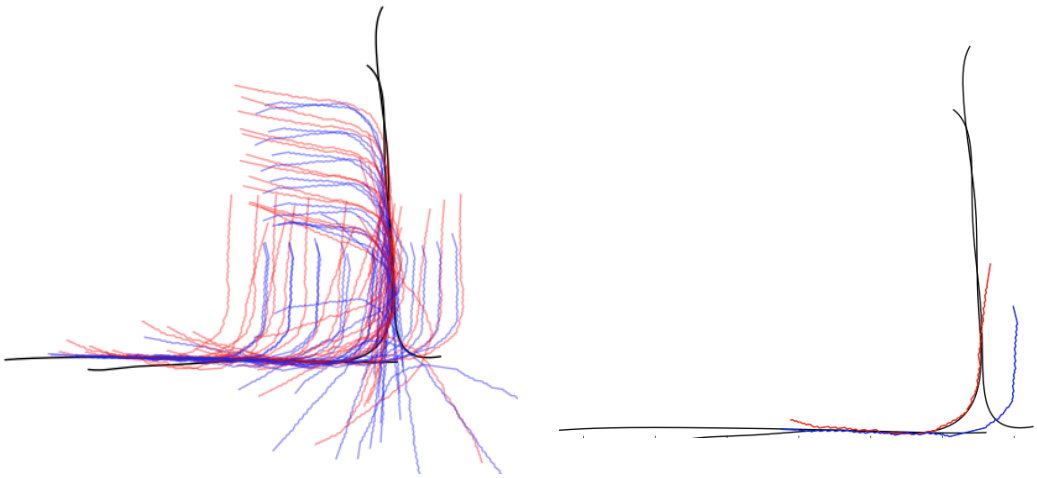


Figure 8.3: Best evaluated pose after motion estimation.

the front and back sweeping posture images. The evaluation function result is shown in the fifth row.

A Gaussian bias is applied with respect to the previous pose that acts as a forcing function to represent the motion shown in sixth row. The final result is shown in the bottom row and is used to select the best pair of poses.

Though we have selected the best pose out of all candidates, we still keep around these other candidates for later. We will reuse them when we perform localization.

8.2.3 Add to Skeleton Map

This phase of the *search method* is identical to subsection 7.2.3. In this case, we use the best pose selected from the motion estimation phase. If the spatial curve is diverging, we create a new skeleton. If it is not, we add it to an existing skeleton.

8.2.4 Overlapping Skeletons

The next phase of the *junction method* is to maximize the overlap of the skeletons. That is, we will adjust the transform between parent and child skeletons such that they are aligned and create junctions that are consistent with the observations.

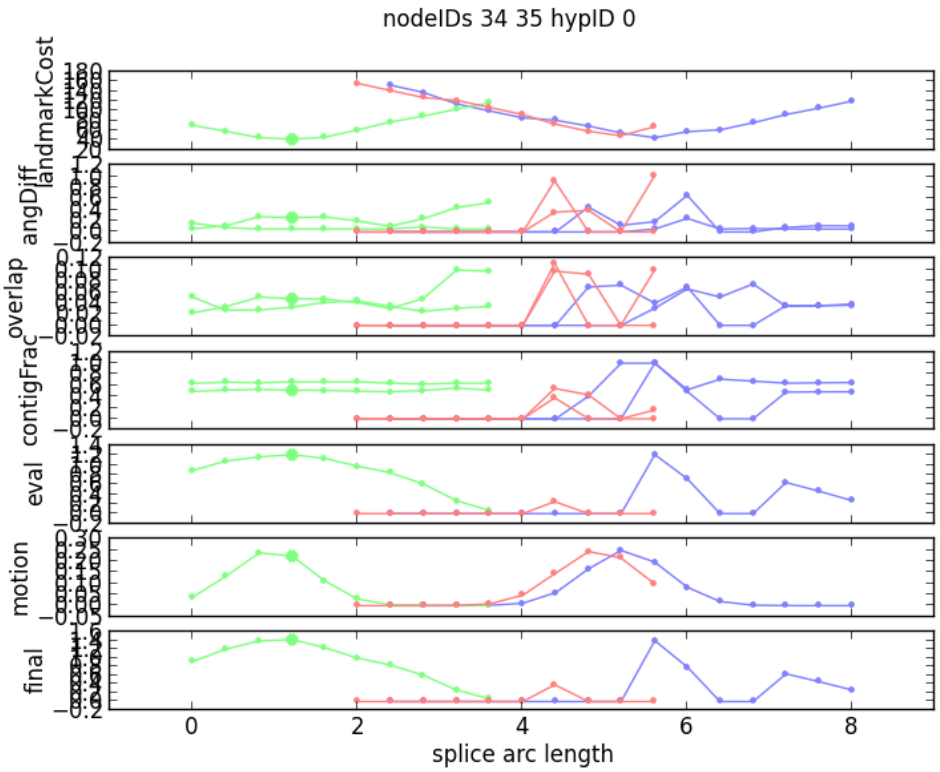


Figure 8.4: Evaluation function and metrics. Each color curve is a different splice. The x-axis indicates the arc length of the splice where the pose is located. In order from top to bottom: 1) landmark cost, 2) angular difference, 3) overlap sum, 4) contiguity fraction, 5) motion evaluation function, 6) motion gaussian bias, 7) sum of bias and eval

Given a parent-child skeleton pair, like the motion estimation phase, we discretize the possible locations of the control point on the parent. That is, we select a handful of possible locations for the child frame to be located on the parent that are uniformly spaced. Given this handful of states, we evaluate each one with a cost function.

If the map is more complex and includes more than two skeletons, then we evaluate every combination of control point location for each parent-child skeleton pair. This creates an exponential explosion in combinations for each skeleton that is added. In the future we can create smart strategies for pruning the search space, but for simple environments, this is sufficient.

We begin by describing our evaluation function. We use two metrics. The first is the result, LM, of the landmark cost function, computeLandmarkCost(), shown in **??**. The second is the amount of overlap, MC, between all of the skeletons which is shown is shown in 17.

The skeleton overlap function computes the number of points on a skeleton that are close enough to any other point on the other skeletons. Therefore, skeletons that are mutually overlapping will maximize their match count. A match is made if a point is within D_c distance to a point on any other skeleton. The algorithm for the computation is shown in 17.

Algorithm 17 Skeleton Overlap Evaluation

```
function overlapSkeleton(T̂_{0:m})
D_c ⇐ 0.2                                          ▷ match distance threshold
MC ⇐ 0                                             ▷ number of point matches
for i = 0 to m do
    T' = T̂_{0:m} − T̂_i                  ▷ points of skeletons minus current skeleton T̂_i
    for ∀p ∈ T̂_i do
        D_min = closestDistance(p, T')             ▷ closest point to other skeletons
        if D_min < D_c then
            MC ⇐ MC + 1                ▷ increment if made a match under threshold
        end if
    end for
end for
return MC
```

Given the global configuration of skeleton trees, $T_{1:m}$, we compute the evaluation of a particular state with the following equation:

$$G(T_{1:m}) = \frac{LM_{max} - LM}{LM_{max}} * \frac{MC}{MC_{max}} \tag{8.14}$$

LM_{max} is the maximum landmark cost over all of the states and MC_{max} is the maximum match count over all of the states. The state with the highest evaluation becomes our selected spatial transform between all of the skeletons. This maximizes the mutual overlap between all of the skeletons and increases the consistency of the junctions.

Since in the previous phase, we may have had a divergence and created a new skeleton, the skeleton overlap phase may have found a state that causes this new skeleton to completely overlap an existing one. This indicative that the new branch is not in fact new, but re-observation of an existing one. This mutual overlapping of skeletons gives us the opportunity to solve the loop-closing problem when revisiting existing junctions.

8.2.5 Localization

ICP localize all previous samples across all splices Select maximum evaluated pose H(x) H(x) = normLandmarkSum * contigFrac0 * contigFrac1 Correct the map with best fit pose

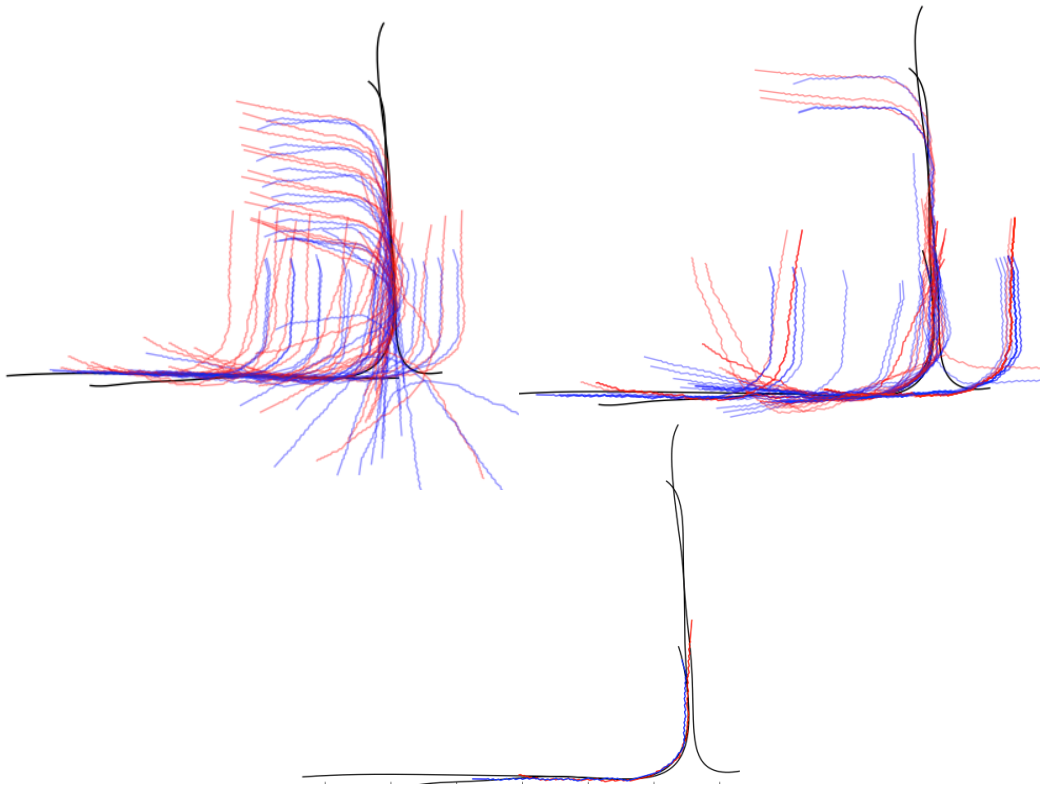


Figure 8.5: Localization: ICP of initial poses and selection of best pose.

Immediately after motion estimation, branch detection, and skeleton categorization, our next task is to find the best likely location for the current pose. We measure the

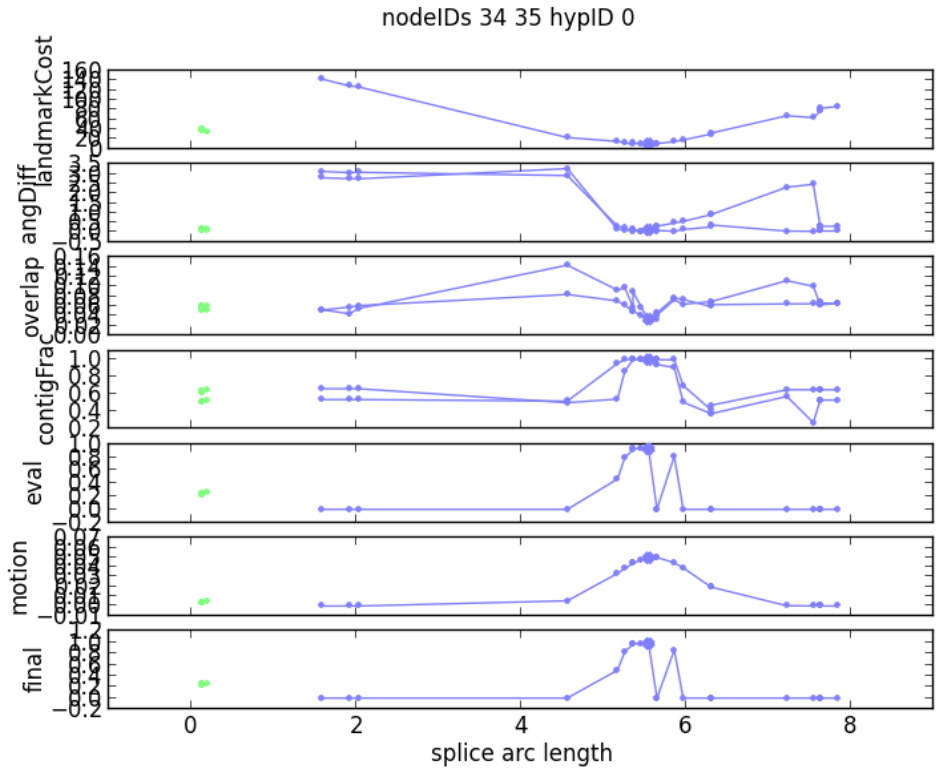


Figure 8.6: Localization: ICP of initial poses and selection of best pose.

likelihood in terms of the best possible fit of the current posture curve to the current skeleton map. To do this, we ag

8.2.6 Merge Skeletons

Merge skeletons together if they are fully constrained

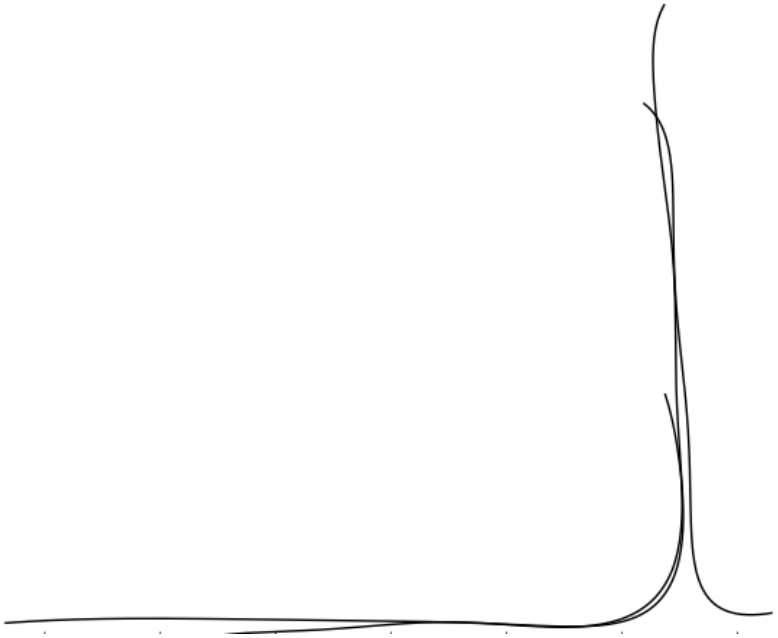


Figure 8.7: Skeletons before merge.

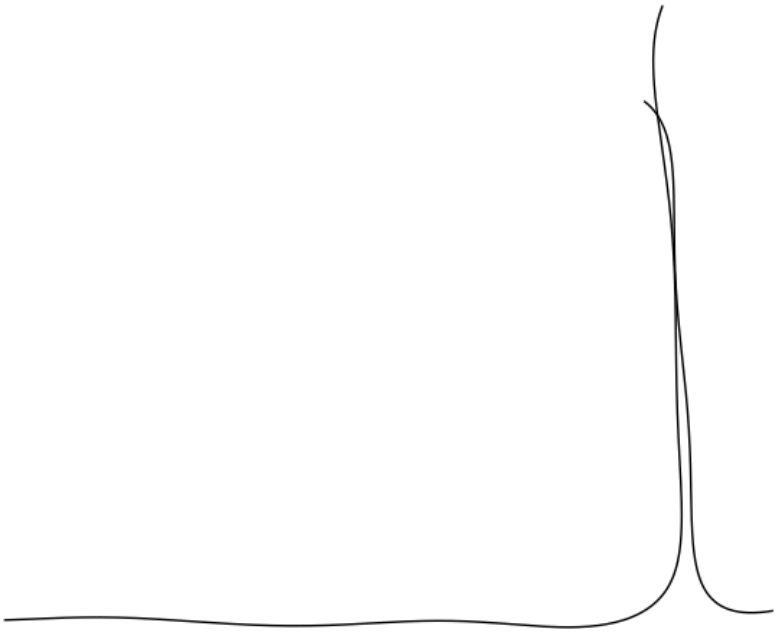


Figure 8.8: Skeletons after merge.

Chapter 9

Experiments

9.1 Experimental Setup

9.2 Results

The results as they stand. WE earlier presented videos showing one-off examples of successful mapping. Each of the maps was built in controlled conditions with parameters that were specifically tuned to the particular feature of the map.

In our desire to make a more unified mapping approach, a generalized algorithm is not as successful for individual maps, but produces mediocre results over all. Frequently the robot fails to map the environment correctly because it erroneously eliminated the correct map. We differentiate between eliminating the correct map possibility and the maximum likelihood map being the incorrect map. A maximum likelihood map may be incorrect, but the possibility for the correct map still exists. If the correct map is eliminated, there is no chance for recovering the correct map.

The results for mapping single path environments is very successful. With the exception of sharp 90 degree turns, the robot can map any single path with very few problems. One of the major challenges of this type of the map is creating phantom branches. This occurs when the robot backs up and returns to a junction feature that diverges from the existing path. The challenge is to determine whether or not the map should have a new shoot branch or whether the robot should be localized to an existing location. This is an example of the data association problem which is a hard problem. Our current approach is to consider both possibilities and use criteria for elimination.

Sharp turns like in the 90 degree junction are not always successfully mapped. This is primarily because robot turning is compliant and exploration of turns is serendipitous. A robot that fails to map a 90 degree junction is a robot that has not yet discovered the turn. In the event of that the robot discovers a piece of the junction, it can continue further mapping through navigation to the undiscovered point.

The 60 degree and curved path environments are successfully mapped with no errors because there are few poses that appear to be new branching events. Nor are there chances for mislocalization of the robot.

The Y junction, cross junction, and T junction environments are capable of taking advantage of spatial landmark features. If there is a bulge in the spatial image created by sweeping the environment, a detector is used to create a landmark that is indicative of the center of a junction. For the multi-arm junctions in question, sweeping inside the junction will show a "bloom" feature.

Challenges: - pose particles don't always cover the correct location so the pose is localized incorrectly - branch shoots are sometimes "reversed" when moved around because the branch characteristics change too much as it's moved around and shoot computation algorithm doesn't recognize the correct direction. - branch location utility is not currently used in evaluating pose particles - bend features are important indicators of junctions, but have high error which can jumble the representation of the junction

9.3 Conclusion

Foo