Autonomous Navigation through Obstacle Filled Environments for Quadrupedal Robots

Jeremy S. Morgan

May 2018

1 Abstract

The objective of this project is to build a system capable of autonomously navigating quadrupedal robots through complex and obstacle filled environments. To achieve this goal, three separate planners of different abstraction levels were designed and implemented - a high level trajectory planner, a step sequence planner, and a real time, full body motion planner.

The current system, while still in development, has produced positive results. When tested in a cost free and low cost world the system generated stable, self collision free, and kinematically valid motion although when tested in a high cost world the system produced self collisions and in feasible configurations.

With the addition and improvement of specific subsystems the project will be robust and reliable enough to for non-simulated robot motion.

2 System Design and Components

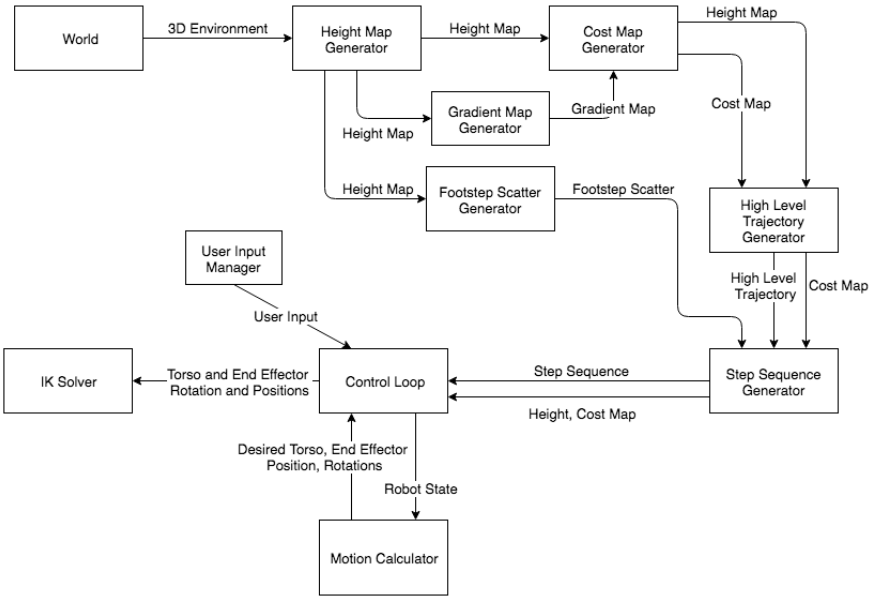

Figure 1: High Level System Design

The system is designed with an emphasis on modularity and conformity to encourage subsystems to be replaceable, discrete and testable. For example, the height, cost, and gradient generator subsystems save their respective outputs to a standardized 'Map' object that maintains a standard interface for retrieving

data. Stored objects are saved and loaded via the cPickle python package for inspection and transfer between subsystems.

2.1 Height Map

The height map stores the environment's height as a 2d array within a specified boundary and with a specified granularity. The height is calculated using Klampt's model.collide library. The code for calculating the height at a specified (X, Y) coordinate is as follows:

```
def height_at_xy(x,y):
  geometry_list = [self.world.terrain(j).geometry() for j in xrange(self.world.numTerrains())]
  collide_rc = collide.ray_cast(geometry_list, [x,y,3], [0, 0, -1])
  z = collide_rc[1][2]
  return z
```

2.2 Footstep Cost Map

The foothold cost map stores the cost to place an end effector at each (X, Y) coordinate in the search grid. The cost at each point is the sum of two weighted heuristic functions, namely a slope heuristic and a cumulative slope difference heuristic.

The slope heuristic calculates the average slope in the x and y direction in the rectangle of width equal to the radius of the robots end effector at the specified (X, Y) coordinate. If the x or y slope is greater than a specified maximum allowable slope, the function returns a cost linearly proportional to the difference of the calculated slope and the max allowable slope. If the slope is greater than a predefined value, it is presumed to be as a result of a large height discrepancy in the world, cause by example, the edge of a platform, and is thus ignored as such a feature falls outside the scope of the heuristic.

The cumulative slope difference heuristic returns the cumulative difference between the x and y slope at all points inside the square of width equal to the robots end effector at the specified (X, Y) coordinate versus the averaged x and y slopes in the same region. To find the slope at a point the gradient is first calculated. The calculation is given as:

$$grad_x(x, y) = \frac{h(x + \Delta x, y) - h(x - \Delta x, y)}{2\Delta x}$$

$$grad_y(x, y) = \frac{h(x, y + \Delta y) - h(x, y - \Delta y)}{2\Delta y}$$

$$\text{where } h(x, y) \text{is the height at } (x, y)$$

The slope, measured in degrees, in the x and y direction is then calculated as:

$$slope_x(x, y) = a * \arctan\ (b * grad_x(x, y))$$

$$slope_x(x, y) = a * \arctan\ (b * grad_x(x, y))$$

$$\text{where } a = 57.352, b = 99.36, \text{found with regression tools.} \quad R^2 = .9999$$

Once the cost map has been built it is normalized by dividing all costs by the maximum cost. This is done to keep weighting schemes standardized. A new cost map is then created by taking the normalized cost map, and setting the value at each point (X, Y) to the average cost in a square of predefined size centered at the point (X, Y). This is done to add knowledge of a points surrounding to that point. If a particular location has a low cost but is surrounded by unsuitable terrain, the cost map will reflect this. A averaging area that is too large will cause the cost map to loose information, as each point will be too 'averaged', where as a small averaging area will prevent information of a points surrounding area from being transfered to the point.

2.3 Scatter List

A scatter-list is a list of potential foot step holds and associated foot-step hold costs which is provided to the step-sequence planner. The costs are referenced from the foothold cost map. Each (X, Y) scatter point is spaced by a predetermined granularity away from other points.

$$\text{Scatter List: } [(X_0, Y_0, Z_0, C_0), ..., (X_n, Y_n, Z_n, C_n), ..., (X_n, Y_n, Z_n, C_n)]$$

2.4 High Level Trajectory Planner

The high level trajectory planner creates a route for the robot to follow in order to arrive at its target destination. The planner accepts a starting and end coordinate describing the position and heading of the torso, $(X_{t0}, Y_{t0}, \Psi_{t0})$, and $(X_{tf}, Y_{tf}, \Psi_{tf})$, respectively, and returns a list of coordinates, given as $[(X_{t0}, Y_{t0}, \Psi_{t0}), ..., (X_{ti}, Y_{ti}, \Psi_{ti}), ..., (X_{tf}, Y_{tf}, \Psi_{tf})]$, for the torso to follow, representing the calculated route. The coordinates are calculated by applying the A* search procedure. The characterizing features of the search are found in below. w_g, and w_g, the traveling and heuristic weights were experimentally determined to be 1.0and1.0 respectively. A too high w_g to h ratio will result in the plan avoiding obstacles to too great an extent, which manifests itself in a contorted and twisted path that does not approach the target quickly. A h too large for the given w_g will result in the plan approaching the target quickly, without considering obstacles.

$$\text{State: } (X_t, Y_t, \Psi_t)$$

Algorithm 1 High Level Trajectory Planner A* Successor Function

function SUCCESSOR(q) ▷ Where $q = (X_t, Y_t, \Psi_t)$
    return-states = []
    return-costs = []
    for for every combination of $\pm\Delta x, \pm\Delta y, \pm\Delta\Psi$ **do**
        new-state = $(q_X \pm \Delta x, q_Y \pm \Delta y, q_\Psi \pm \Delta\Psi)$
        return-states.add(new-state)
        return-costs.add(Get-Cost(new-state))
    end for
    return return-states, return-costs
end function

Algorithm 2 High Level Trajectory Planner A* GetCost Function

function GETCOST(q) ▷ Where $q = (X_t, Y_t, \Psi_t)$
    $X_{fl,rotated}$ = 2d-rotation($x_{basestate}$,$y_{basestate}$,q_Ψ) ▷ Where $X = (x, y)$
    $X_{fr,rotated}$ = 2d-rotation($x_{basestate}$,$-y_{basestate}$,q_Ψ)
    $X_{br,rotated}$ = 2d-rotation($-x_{basestate}$,$y_{basestate}$,q_Ψ)
    $X_{bl,rotated}$ = 2d-rotation($-x_{basestate}$,$-y_{basestate}$,q_Ψ)
    $X_{fl} = X_{fl,rotated} + (q_x, q_y)$
    $X_{fr} = X_{fr,rotated} + (q_x, q_y)$
    $X_{br} = X_{br,rotated} + (q_x, q_y)$
    $X_{bl} = X_{bl,rotated} + (q_x, q_y)$
    return w_g(CostMap(X_{fl})+CostMap(X_{fr})+CostMap(X_{br})+CostMap(X_{bl})+step-cost)
end function

Algorithm 3 High Level Trajectory Planner A* Heuristic Function

function HEURISTIC(q) ▷ Where $q = (X_t, Y_t, \Psi_t)$
    err = $\sqrt{w_x(q_X - X_f)^2 + w_y(q_Y - Y_f)^2 + w_\Psi(q_\Psi - \Psi_f)^2}$
    return w_h*err
end function

Algorithm 4 High Level Trajectory Planner A* Is-Goal Function

function HEURISTIC(q) ▷ Where $q = (X_t, Y_t, \Psi_t)$
    err = $\sqrt{w_x(q_X - X_f)^2 + w_y(q_Y - Y_f)^2 + w_\Psi(q_\Psi - \Psi_f)^2}$
    if err ¡ goal-err-threshold **then**
        return True
    else
        return False
    end if
end function

2.5 Step Sequence Planner

The step sequence planner generates a list of foot step holds the robot will take as it travels along the high level trajectory. The holds are generated by applying the A* search procedure. The characterizing features of the search are as found below.

Other notable functions include State-Is-Kinematically-Valid(q), Get-Next-Leg-To-Move(leg$_{current}$), and Get-Estimated-Torso-XY-Yaw(q). As the name suggests, State-Is-Kinematically-Valid filters kinematically invalid states. This is done by ensuring the line segments defined by the left and right end effectors, and front and back end effectors do not cross. It also returns false if the distance between end effectors does not fall into specific, precomputed acceptable ranges. Get-Next-Leg-To-Move($leg_{current}$) returns the next leg to move in the step order. The provided step order is [3,2,4,1], indicating that the back right, then front right, then back left, then front left steps. This order was found to provide large support areas in all stages of the gait, although the order has yet to be rigorously tested and there may be more stable step orders to follow. The function Get-Estimated-Torso-XY-Yaw(q) returns the torso's estimated (X, Y, Ψ) from a given state. This is performed by averaging the X and Y coordinates of the end effectors foot holds, and taking the Ψ commanded by v, where $v = v_{l,n} + v_{r,n}$. The vector $v_{l,n}$ points from the back left to the front left end effector, and $v_{r,n}$ is the vector from the back right to front right end effector.

The planner is quite efficient, and typically calculates routes on a matter of seconds, if not shorter. The run time is largely determined by the length of the provided scatter-list. To this end the x and y granularities of the scatter-list are set to .1(m). This value was experimentally determined to result in the generation of around 100 successors from any given state, which is in the desired range for a successor function $(60 - 120)$.

As is seen in algorithm [6], the planner finds end effector positions by simply shifting the robot, when posed in its base state, along the high level trajectory by a predefined distance dL. This is a largely naive procedure, as it does not account for either 1) the position of the other end effectors, or 2) the gait implemented by the motion planner. This procedure can lead to over stretched and comparatively unstable robot poses if one leg is staggered, or especially far to the side. In complex environments with numerous obstacles this can be and issue however enerally this method effectively generates a feasible and stable step sequence pattern.

Another important input to the planner is the dL parameter, the end effector step distance. The distance defines the speed of the robot gait, and greatly impacts the margin, and existence of the stability region of the robots torso. It is currently set to .175, which creates suitably long steps without significant loss of stability in the gait although this parameter remains to be optimized. A potential method for optimization would involve creating a cost function that takes as input the dL parameter and returns the weighted sum of heuristic costs measuring the relative stability of a step pattern generated with the provided dL input as well as the speed of the resulting motion.

$$\text{State} = (i, j, k, l, \text{leg-to-move} \in 1, 2, 3, 4)$$
where i,j,k,l refer to indices in the provided scatter-list
$$\text{Goal-State} = (-1, -1, -1, -1, -1)$$

Algorithm 5 Step Sequence Planner A* Successor Function

function SUCCESSOR(q) ▷ Where $q = (i, j, k, l, \text{leg-to-move} \in 1, 2, 3, 4)$
    return-states = []
    return-costs = []
    for i in length(scatter-list) **do**
        new-state = copy(q)
        new-state.index-of($q_{leg-to-move} - 1$) = i
        new-state$_{leg-to-move}$ = Get-Next-Leg-To-Move($q_{leg-to-move}$)
        if State-Is-Kinematically-Valid(new-state) **then**
            return-states.add(new-state)
            return-costs.add(Get-Cost(new-state))
        end if
    end for ▷
    return-states.add(Goal-State)
    return-costs.add(Get-Cost(Goal-State))
    return return-states, return-costs
end function

Algorithm 6 Step Sequence Planner A* GetCost Function

function GETCOST(q) ▷ Where $q = (i, j, k, l, \text{leg-to-move} \in 1, 2, 3, 4)$
    if Is-Goal-State(q) **then**
        return 0
    end if
    $(X_{desired}, Y_{desired}, Z_{desired})$ = base state location of end effector when torso is shifted along high level trajectory from closest point along trajectory to current state by dL
    $(X_{actual}, Y_{actual}, Z_{actual})$ = scatter-list[$state.leg_to_move - 1$]
    function HYBRID-COST-PARABOLA((X, Y)) ▷ See Figure 2
        return $w_x(X_{desired} - X)^2 + w_y(Y_{desired} - Y)^2 + w_c$Cost-Map-At$(X, Y)$
    end function
    return w_xCost-Parabola(X_{actual}, Y_{actual})
end function

Algorithm 7 Step Sequence A* Heuristic Function

function HEURISTIC(q) ▷ Where $q = (X_t, Y_t, \Psi_t)$
    return $w_h\sqrt{w_x(q_X - X_f)^2 + w_y(q_Y - Y_f)^2 + w_\Psi(q_\Psi - \Psi_f)^2}$
end function

Algorithm 8 High Level Trajectory Planner A* Is-Goal Function

function HEURISTIC(q) ▷ Where $q = (i, j, k, l, \text{leg-to-move} \in 1, 2, 3, 4)$
    if $q[0] = -1$ **then**
        return True
    end if
    return False
end function

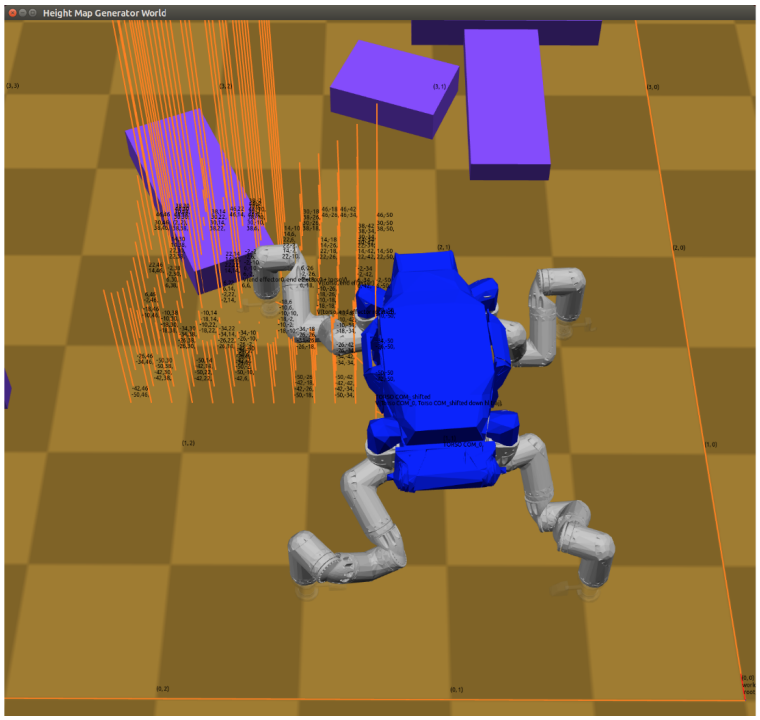

Figure 2: Hybrid Cost Map

2.6 Control Loop

The control loop runs in real time, and is tasked with calculating robot configurations which result in a stable, continuous, non-colliding and kinematically valid gait.

It receives as input a footstep-scatter, a footstep-sequence, a height and gradient map of the world, and a start and end position and heading.

The control loop implements a statically stable gait composed of two stages. In the first, the torso shifts to the centroid of the region formed by the intersection of the current support triangle, and 5 end effector range circles. The support triangle is formed by the three end effectors which will not move in the preceding step. A range circle is a geometric object defined as a circle centered about and end effectors current or future position, with a radius equal to the distance the torso, when set with its current heading can move without over exerting its range. The inverse kinematic solver responsible for generating configurations is only provided the torso's $(X_{t,desired}, Y_{t,desired}, Z_{t,desired})$. The rotation of the torso is left as an open constraint. This is problematic when the currently occupied feasible region of the robots configuration space manifold disappears. The inverse kinematics solver must then move relatively large distances through the configuration space to enter a new feasible manifold subset. This results in sudden jumps in the torso's pose. This problem can be solved by manually optimizing the torso's torso rotation. The optimizer would find a torso rotation that maximizes the cumulative range of the end effectors while disallowing sudden jumps. Such a system will be implemented and included in future development of the control loop.

In the second stage of the gait the moving end effector follows a parabolic arc to its final position. The calculation for a end effectors mid motion (X, Y, Z) at time $t \in [0, 1]$ is found as:

$$X = X_0 + t(X_f - X_0)]$$

$$Y = Y_0 + t(Y_f - Y_0)$$

$$\Delta_{xy} = \sqrt{(X - X_0)^2 + (Y - Y_0)^2}$$

$$\Delta_{xy,total} = \sqrt{(X_f - X_0)^2 + (Y_f - Y_0)^2}$$

$$Z = Z_0 + t(Z_f - Z_0) + \frac{h_{step}\Delta_{xy}(\Delta_{xy,total} - \Delta_{xy})}{\Delta_{xy,total}}$$

End effectors are posed such that they are normal to the ground. A rotation matrix is generated to this end. This calculation is given below.

$$U = [0, 0, -1, 0, -1, 0, 1, 0, 0] \textit{ where } U \textit{ is the upright rotation matrix}$$

$$\vec{n} =< \text{Gradient-At}_x(X, Y), \text{Gradient-At}_y(X, Y), 1 > \textit{ where } n \textit{ is the normal to the surface at } (X, Y)$$

$$\vec{n} = \frac{\vec{n}}{|\vec{n}|}$$

$$\vec{v} =< 0, 0, 1 >$$

$$\text{axis} = \vec{n} \times v$$

$$\phi = \arccos(\frac{\vec{n} \times \vec{v}}{|\vec{n}||\vec{v}|})$$

$$\text{R}_{rotation} = \text{From-Axis-Angle}((axis, \phi))$$

$$R = U * \text{R}_{rotation}$$

When the end effector is in motion, we interprolate a mid motion rotation matrix for it to follow, so that starting from its initial rotation it rotates smoothly to its final rotation as the step is completed. This is a commonly studied calculation, formally called Spherical Linear Interpolation, or Slerp for short. The calculation is given below.

$$q_0 = \text{Rotation-To-Quaternion}(R_0)$$

$$q_f = \text{Rotation-To-Quaternion}(R_f)$$

$$q_{0,n} = \frac{q_0}{|q_0|}, q_{f,n} = \frac{q_f}{|q_f|},$$

$$\theta_0 = \arccos(q_{0,n} \cdot q_{f,n})$$

$$\theta = t\arccos(q_{0,n} \cdot q_{f,n}) \textit{ where } t \textit{ is the provided time constant } \in [0, 1]$$

$$\text{sin-}\theta = \sin t\arccos(q_{0,n} \cdot q_{f,n})$$

$$\text{sin-}\theta_0 = \sin \arccos(q_{0,n} \cdot q_{f,n})$$

$$S_0 = \cos\theta - \frac{(q_{0,n} \cdot q_{f,n}) * \text{sin-}\theta}{\text{sin-}\theta_0},\ S_1 = \frac{\text{sin-}\theta}{\text{sin-}\theta_0}$$

$$q = S_0 q_0 + S_q q_1$$

$$R = \text{Quaternion-To-Rotation}(q)$$

The control loop is still in development. As of currently, the control loop calculates configurations with an inverse kinematics solver seeded with a non-colliding ideal base posture, which encourages non colliding poses, however provides no guarantee that the robot will not collide. It will, when finished, be built around a PRM or RRT configuration space sampler responsible for finding collision free paths. A configuration space sample based planner was built but not included because it was unable to find configurations in between the start and end configurations of a given torso motion or end effector step. This is a fixable issue and will be resolved shortly.

3 Methods

The system was tested in three different worlds, flat world, obstacle world, and fractal world (see Figures [3,4,5]). In each world, the robot was given a start and end position and heading. The time to perform each of the two A* searches, and the number of inverse kinematics failures, as well as the total amount of self collision time during motion was recorded. The number of collisions between the robot and the environment was not recorded. This is was an intentional choise motivated by the combination of two factors: 1) The collision detection system recorded collisions whenever an end effector was in contact with the ground and not moving creating countless falsely recorded collisions and 2) The configuration space sample based planner has not been implemented, thus the number of collisions created by the current system are of little importance as they will not reflect the performance of the final system. The same argument can not be applied to the number of inverse kinematic errors and self collisions as these quantifiers are indicative of the fundamental stability and feasiblity the step sequence planner's generated paths.

The system was tested with largely idealized parameters. Low terrain cost positions were chosen as start and end points and the robot was not tasked with traversing difficult terrain. Once the system is finished it will be tested more rigorously. Future testing will include large numbers of trials each with a randomized start and end position, heading and environment. This testing will bring to light unforeseen scenarios which lead to any combination of inverse kinematics errors, self collisions, kinematically in-feasible commanded robot poses, or loss of static stability.

[][a]

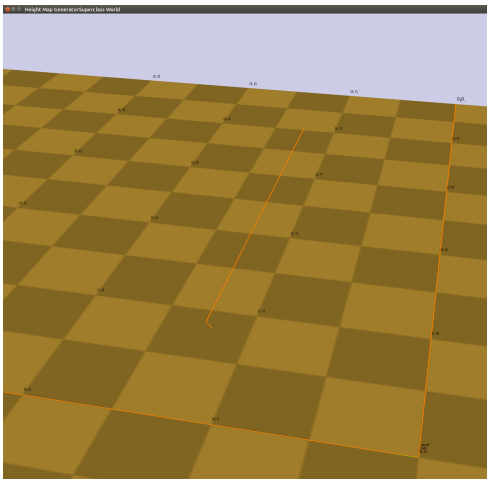

Figure 3: Flat World

[][a]

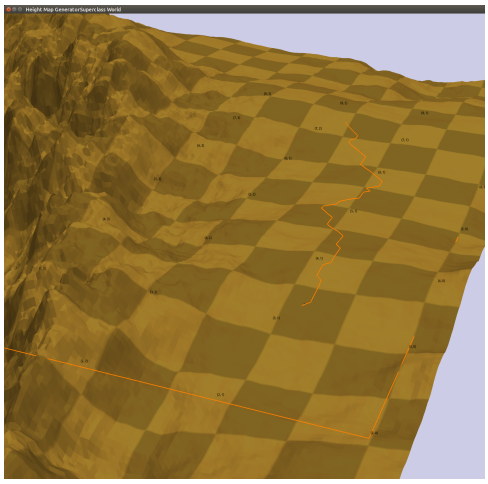

Figure 4: Fractal World

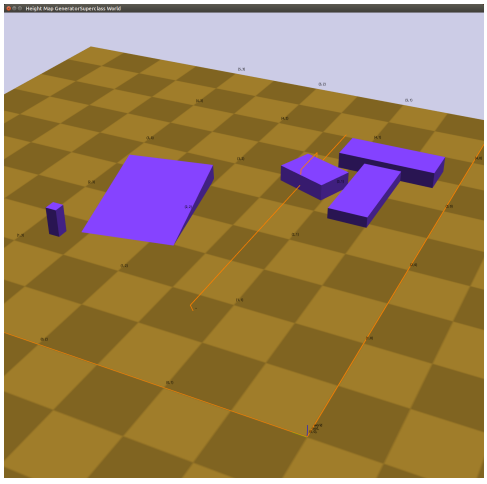

Figure 5: Obstacle World

4 Results

Results				
World	HL. Traj. A* Search Time(s)	Step Seq. A* Search Time(s)	IK Failures	Self Collision Seconds(s)
Flat World	1.585	3.165	0	0
Obstacle World	1.752	2.989	0	0
Fractal World	1.904	3.123	113	8.2

5 Discussion

Initial tests of the system have produced promising results. In cost free and low cost environments, such as the Flat and Obstacle World respectively, the realized gait is stable at all times without any inverse kinematics errors or self collisions. The system is shown to not be robust however, as there are numerous inverse kinematics errors and self collisions in more difficult environments, namely the fractal world. With the implementation of the torso rotation optimizer and the configuration space sample based planner, the robustness and reliability of the system will dramatically increase, ideally making the system suitable to control the robosimian outside of simulation.

The next significant addition to the system will be the ability to input, analyze and act on 3D point cloud data. The system is currently given complete certainty of the world, which is unavailable in practical situations.

6 Conclusion

The system described is able to efficiently plan through environments and generate stable, self collision free, and kinematically valid motion through low cost environments. With the addition of a torso rotation optimizer and a configuration space sample based planner the system's robustness will significantly improve allowing motion through difficult terrains. Furthermore, with the construction of a subsystem capable of interpreting 3D point cloud data, the system will be capable of controlling the robosimian quadruped in real life, thereby achieving the overall goal of this project.