AMCL Localization and Move-Base Navigation with Two Different Robot Models

Arnold Kompaniyets

Robotic Software Engineering, Term 2

Project 2

Abstract:

Two robot models (both implementing a camera and laser scanner, structured in URDF format) were individually interconnected with “move_base” and AMCL nodes in a ROS environment to produce a virtual simulation of robotic localization and navigation. Gazebo and Rviz were used to visualize the robot model and a pre-made navigation map, the latter of the programs used to also visualize sensory data and observe the success of robot navigation and localization. Parameters for both nodes were tuned to allow the robot to both navigate towards a distant goal in a linear and time-effective path, as well as provide an accurate localization point-array. The ROS Navigation package was used to accomplish the above tasks. A brief discussion is provided afterwards to analyze the importance of model simplification, sensor fusion, and balancing costmap/model accuracy to achieve optimal performance times.

Introduction:

Robotics, at its core, is a science of action. Computers can perform mind-boggling calculations and serve as a platform for any and all electronic creative ventures, but robots are the ones that act in the physical world. They roll, fly, pick things up, etc., which ultimately serves as the inspiration and drive for roboticists. Few things can compare to the pleasure and excitement of building a mechanical creation that is a robot and seeing it interact with the world around it. This is where one of the biggest, if

not the biggest, challenges with robotics arises. That same aforementioned robot, when set into the world, doesn't necessarily know what this "world" is. Presumably, the robot does have a means of "sight", be it a camera or laser scanner, but these merely serve as eyes to an animal – without a functional mind to interpret all the incoming signals, the eyes are all but useless. The normal adult human mind, being the incredible computer it is, has little issue performing sensor fusion (combining sight, hearing, olfaction, etc.) and giving the individual an extremely accurate prediction of location. A robot, virtually regardless of its intended actions, needs to have this capability as well, and figuring out how to program this localization capability into a robot's mind has been a matter of large debate and effort for decades.

This particular project aims to use a ROS interface, implementing navigation and particle filter nodes (Adaptive Monte Carlo Localization, or AMCL for short) to allow two robot models to navigate a pre-structured map and visually illustrate successful localization. The specific robot models are as such: a basic model provided by Udacity for this project and a personally-designed "Gorilla-bot".

Background:

As mentioned above, the human mind exhibits an incredible ability for localization. In fact, virtually all localization takes place subconsciously…and incredibly fast too. Only in complex situations, involving matters of large scale localization (Which country am I in? Which state is this?) does the subconscious mind begin to wonder. In matters more concerning robotics, such as knowing where the subject is in a room, or which side of the road a subject is on, and so forth, the average human mind doesn't have any issues, even in entirely novel environments. How does such quick and accurate localization happen? The full series of pathways active in the mind during localization is certainly yet to be mapped and seeing something of this magnitude achieved may yet be a long ways away. It can likely be assumed that the mind's incredible capacity for storage of sensory data, combined with vast computational ability, is what ultimately allows an individual to localize incredibly well even as a child.

Computationally speaking, the feat described above is perhaps quite reasonable. After all, from the very first time a spark vitalizes the interconnected web of neurons that is the brain, it never ceases to work. Therefore, given such incredible computational prowess, continually training to localize should allow the mind to be rather proficient even after just a few years. Ultimately, though, how it learns and actually executes this localization, remains a mystery.

In the field of robotics, the goal remains to try and catch up to the capabilities of the human mind as quickly as possible, and then surpass it. One factor of this is the aforementioned eyes to the outside world. There are now plethora sensors available for use in robotics, many which go above and beyond the capabilities of the human body. Sensor noise is still certainly an issue, but rapid progress continues to be made every year. The other major factor is what to do with all this incoming data.

Two popular methods exist for processing sensory data into location probability, and they are the Kalman filter and particle filter (sometimes also called Monte Carlo localization). Both filters focus on providing an array of probabilities that correlate to localization estimates. The Kalman filter does this by using Gaussian distributions and matrix transformations. Visually speaking, the localization filter starts out with a plane of location probabilities covering the entire sensory map. This initial plane is quite flat, with perhaps a few hills here and there, illustrating a fairly uniform set of guesses as to location. However, once measurements of location begin streaming in, the filter can combine the Gaussian distribution of the initial estimate and the measurement, producing a much more accurate guess of location, illustrated visually by the appearance of a mountain in the probability plane. The more measurements and less noise, the higher the peak of said mountain (illustrating less variance). As the position measurements of a Kalman filter stream in, the matrices of calculation allow the filter to also guess the robot's direction and estimate future position (equivalent to velocity). This estimate of motion decreases the peak of the location mountain, which must then be supplemented by more measurements. This exact cycle of measurements and estimates is how a Kalman filter runs.

A simpler means of going about the issue is via a particle filter. This filter removes the Gaussian distributions and the interconnected plane of location probabilities, instead filling the entire sensory map space with any and all conceivable guesses for location and orientation of a robot, up to the allowed maximum of particles allowed. Once measurements begin to flow in, the particle filter then asks the same question of each particle it has generated, “How likely is this particle to be the robot’s actual position and orientation?” Naturally, the further away the particle is from the measurement in terms of x-y coordinates, or radians in orientation, the less likely that the particle is equivalent to the true position. The particle filter itself accomplishes this by assigning a weight to each particle – the closer the particle is to the measurement, the higher the weight. Next comes the unique step of the particle filter. All the available particles are aligned along a circle, each particle being given a “piece of the pie” equivalent to its weight. Then, visually speaking, a point is randomly placed on the circle and allowed to go around at random lengths, picking particles along the way until the same number of particles is back in the overall pool. Of course, the particles with the larger weights are more likely to be selected (especially multiple times), essentially eliminating many of the unlikely particles. After many rounds of this process repeated, only the particles most closely reflecting the true position will remain “alive”. The continuation of this filter is adding an adaptive component, which simply allows the filter to change the number of random particles it has at a time. This feature allows the filter to “restart” its localization process at various times, which is very helpful in situations when the robot is randomly placed in a new location.

No matter what a robot is doing, there is little hope of the task being done correctly without proper localization, meaning that proper localization will continue to grow in importance as the field of robotics improves and advances. Both of the filters described above are frequently used in this task, but differences do exist between the two. Kalman filters take in many assumptions when predicting position, such as linear motion and Gaussian probability distributions. The real world certainly doesn’t function this way, so innately the Kalman filter accepts the realization that even its finest guess will be made with lots of simplifications. Particle filters do not have this as an issue, but this advantage comes at the cost of greater computational requirements. While Kalman

filters may be more difficult to program, position estimates remain in the form of a matrix, with measurement updates and position estimates being only matters of matrix manipulation (something computers are incredibly well-suited for). Particle filters are less of actual particles and instead represent their own instances of the robot in question. So, with 1,000 particles, that's 1,000 instances of the robot being initiated. Another interesting difference is the fact that the number of particles the particle filter works with each round differs – usually with less particles than the previous round. The Kalman filter, conversely, always functions on the same sized probability landscape, it is merely the location and size of the probability "mountains" that differ.

Nonetheless, it is a particle filter used in this particular project, simply because the robot motion (with the navigation node used) is incredibly non-linear. Additionally, since the robot used here is virtual, computational power and space is not as much of an issue, and the particle filter allows much greater visualization of the success of the filter, since the particles themselves can be directly visualized.

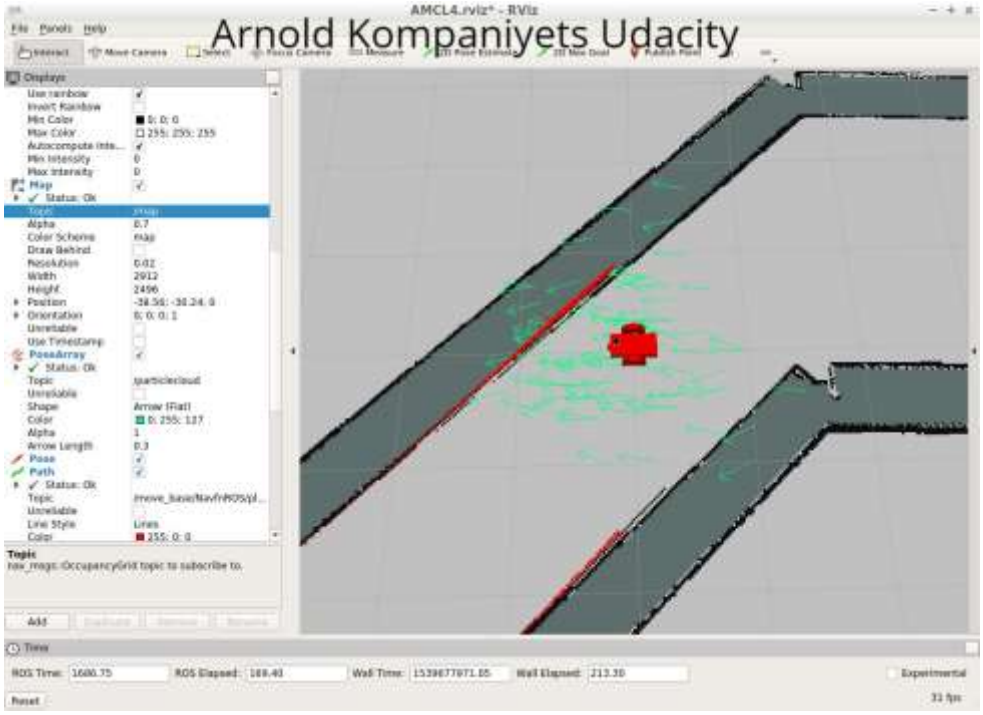
Results:

Udacity Bot:

Once the AMCL node was successfully tuned and the navigation node installed, the provided robot was, after multiple attempts and further fine-tuning, able to successfully navigate to the desired destination.

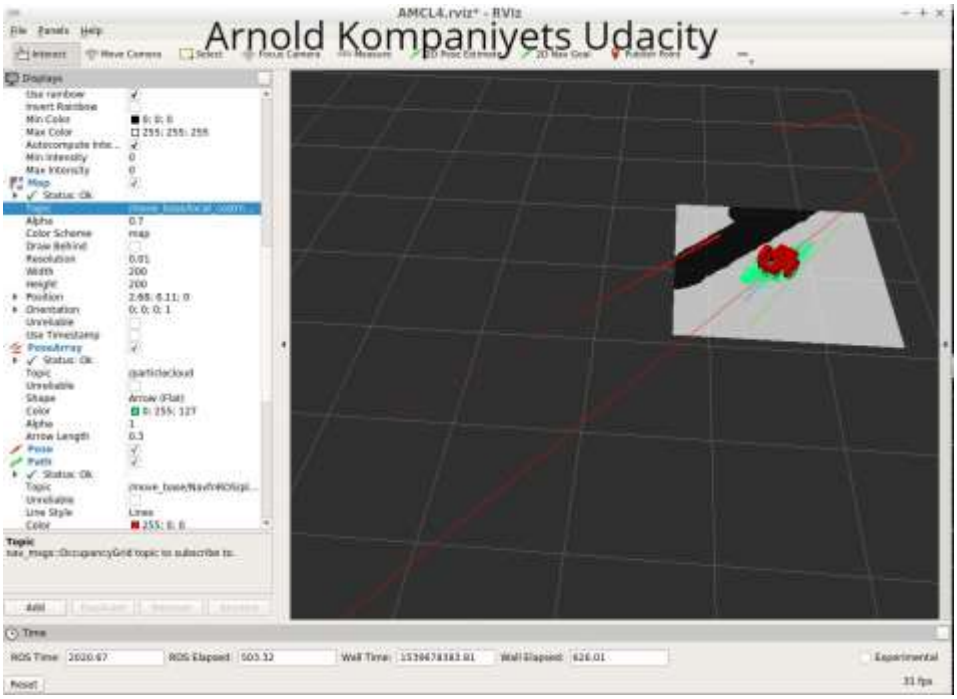
At its starting position, the robot appeared as such in Gazebo:

In Rviz, the robot was also visualized within the map, but also with the position/orientation particle estimates:

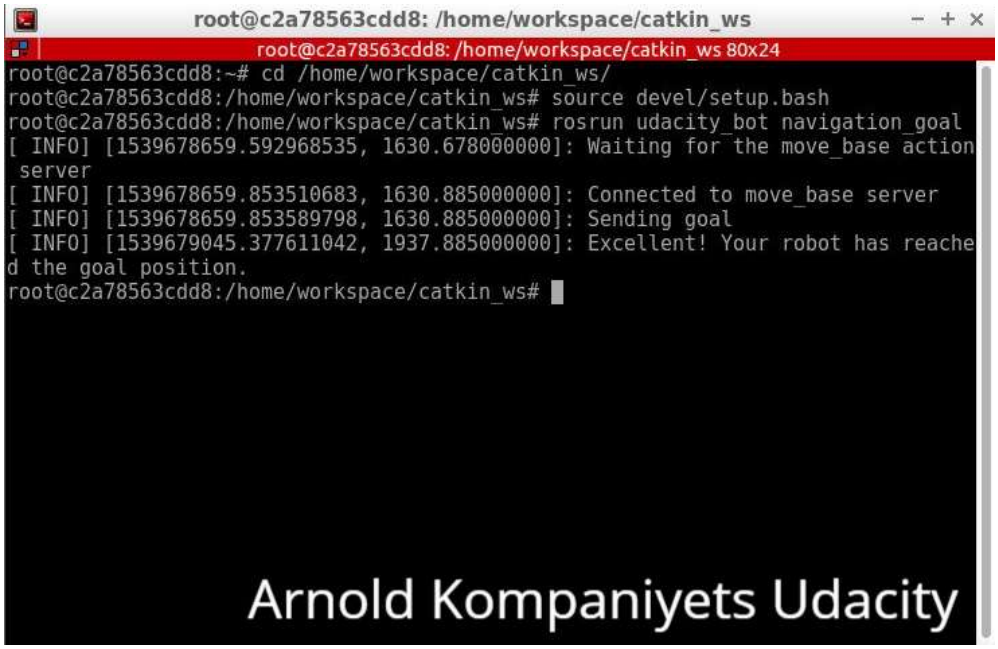


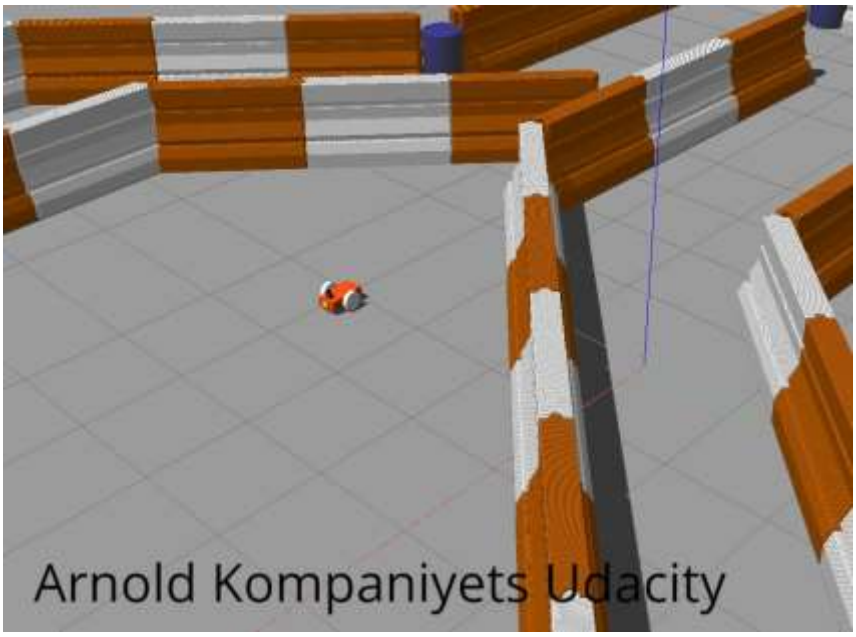
As can be seen, the particles show a large radius of uncertainty (up to four times the size of the robot). The orientation variability is not as great, but still does show 30 degrees or so of variability.

While moving towards the goal, the robot's localization guesses achieved much greater certainty:

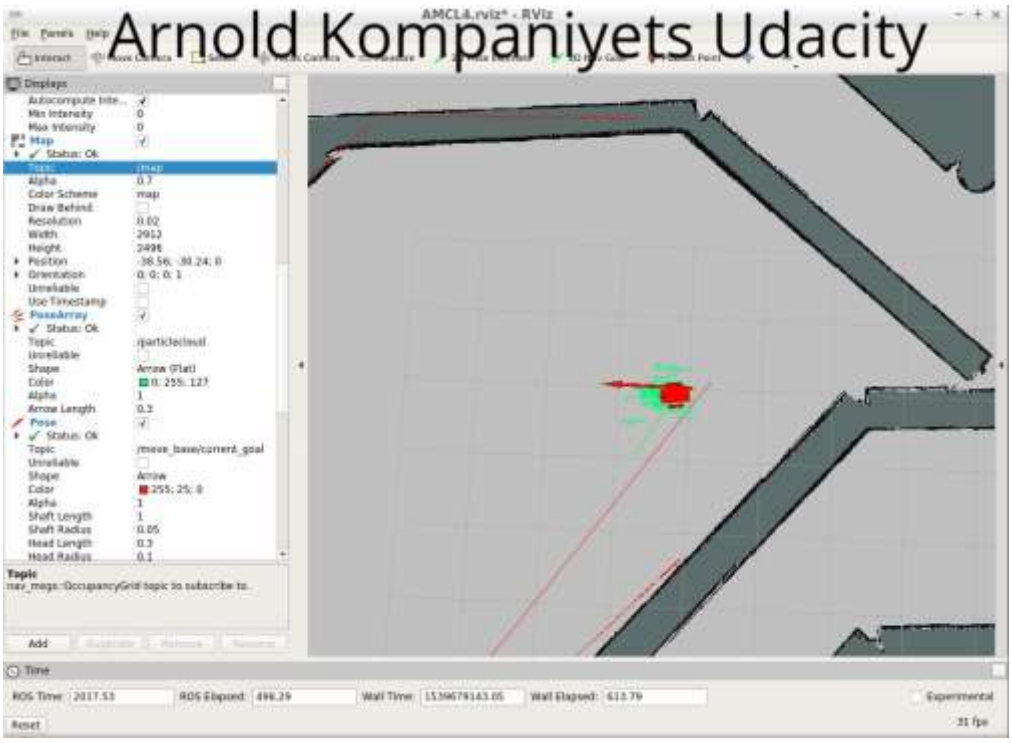


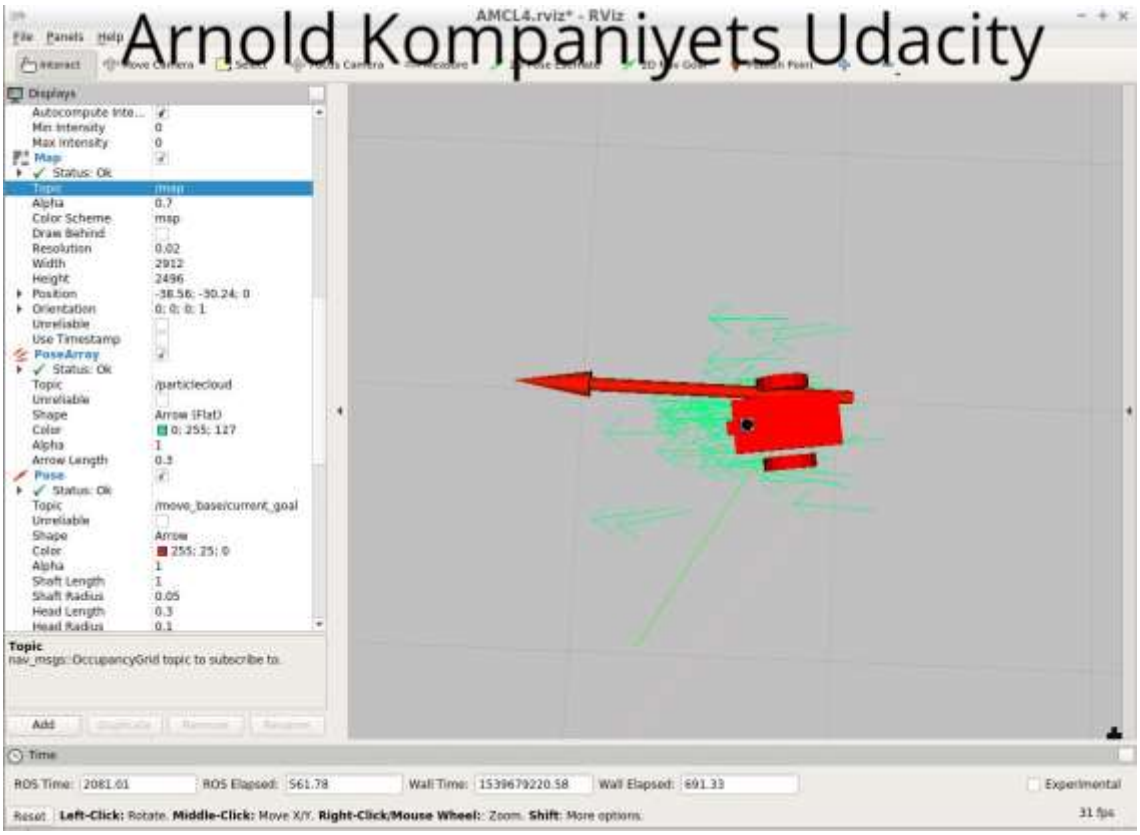
After a few minutes of travel, the robot successfully reached its goal:

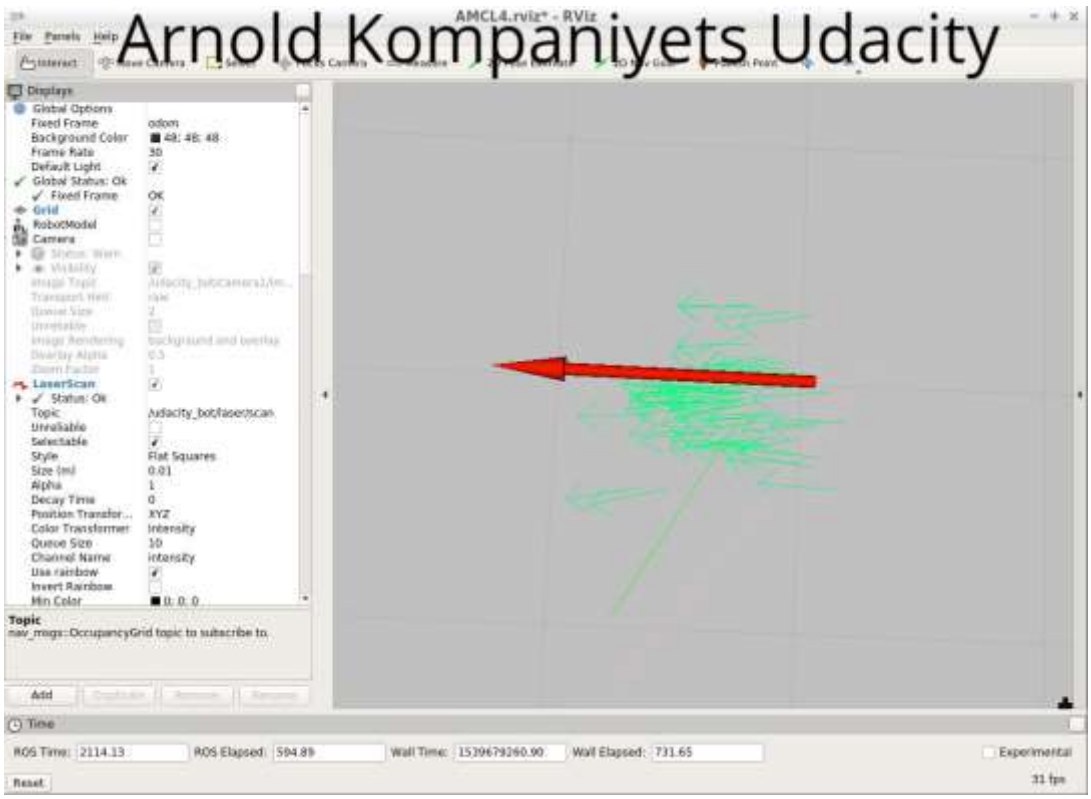




In Rviz, the robot appeared as such:



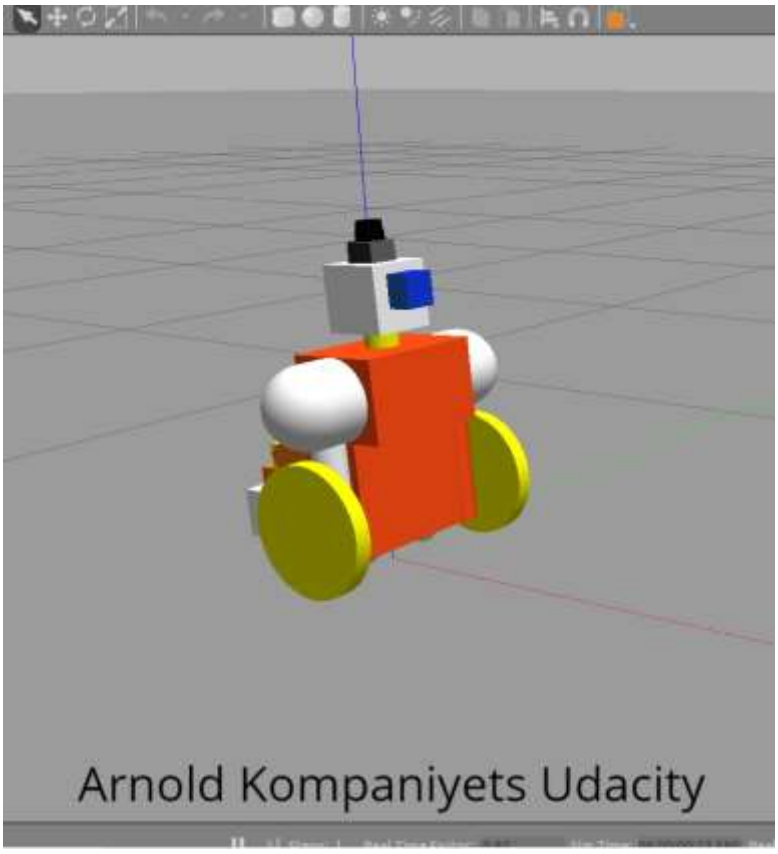

Arnold Kompaniyets Udacity
AMCL4.rviz* - RViz
File Panels Help
Interact
Displays
Autocompute Inte...
Min Intensity 0
Max Intensity 0
Map
Status: OK
Topic /map
Alpha 0.7
Color Scheme map
Draw Behind
Resolution 0.02
Width 2912
Height 2496
Position -38.56; -30.24; 0
Orientation 0; 0; 0; 1
Unreliable
Use Timestamp
PoseArray
Status: Ok
Topic /particlecloud
Unreliable
Shape Arrow (Flat)
Color 0; 255; 127
Alpha 1
Arrow Length 0.3
Pose
Status: Ok
Topic /move_base/current_goal
Unreliable
Shape Arrow
Color 255; 25; 0
Alpha 1
Shaft Length 1
Shaft Radius 0.05
Head Length 0.3
Head Radius 0.1
Topic
nav_msgs::OccupancyGrid topic to subscribe to.
Add
Time
ROS Time: 2081.01
ROS Elapsed: 561.78
Wall Time: 1539679220.58
Wall Elapsed: 691.33
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
31 fps

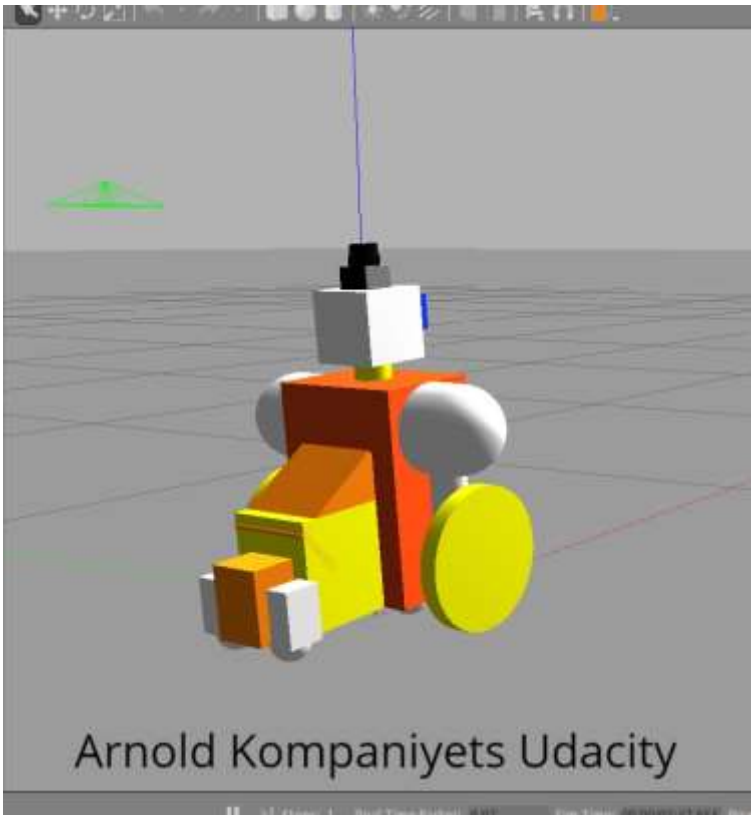

Arnold Kompaniyets Udacity
AMCL4.rviz* - RViz
File Panels Help
Interact
Displays
Global Options
Fixed Frame odom
Background Color 48; 48; 48
Frame Rate 30
Default Light
Global Status: Ok
Fixed Frame OK
Grid
RobotModel
Camera
LaserScan
Status: Ok
Topic /udacity_bot/laserscan
Unreliable
Selectable
Style Flat Squares
Size (m) 0.01
Alpha 1
Decay Time 0
Position Transfor... XYZ
Color Transformer Intensity
Queue Size 10
Channel Name intensity
Use rainbow
Invert Rainbow
Min Color 0; 0; 0
Topic
nav_msgs::OccupancyGrid topic to subscribe to.
Add
Time
ROS Time: 2114.13
ROS Elapsed: 594.89
Wall Time: 1539679260.90
Wall Elapsed: 731.65
Experimental
Reset
31 fps

As can be seen above, the final position estimates (given just a few outlier particles) are essentially at the actual position of the robot, having a radius of variation less than the size of the robot itself and the orientation varying only 5 or so degrees from the correct orientation.

“Gorilla-Bot”:

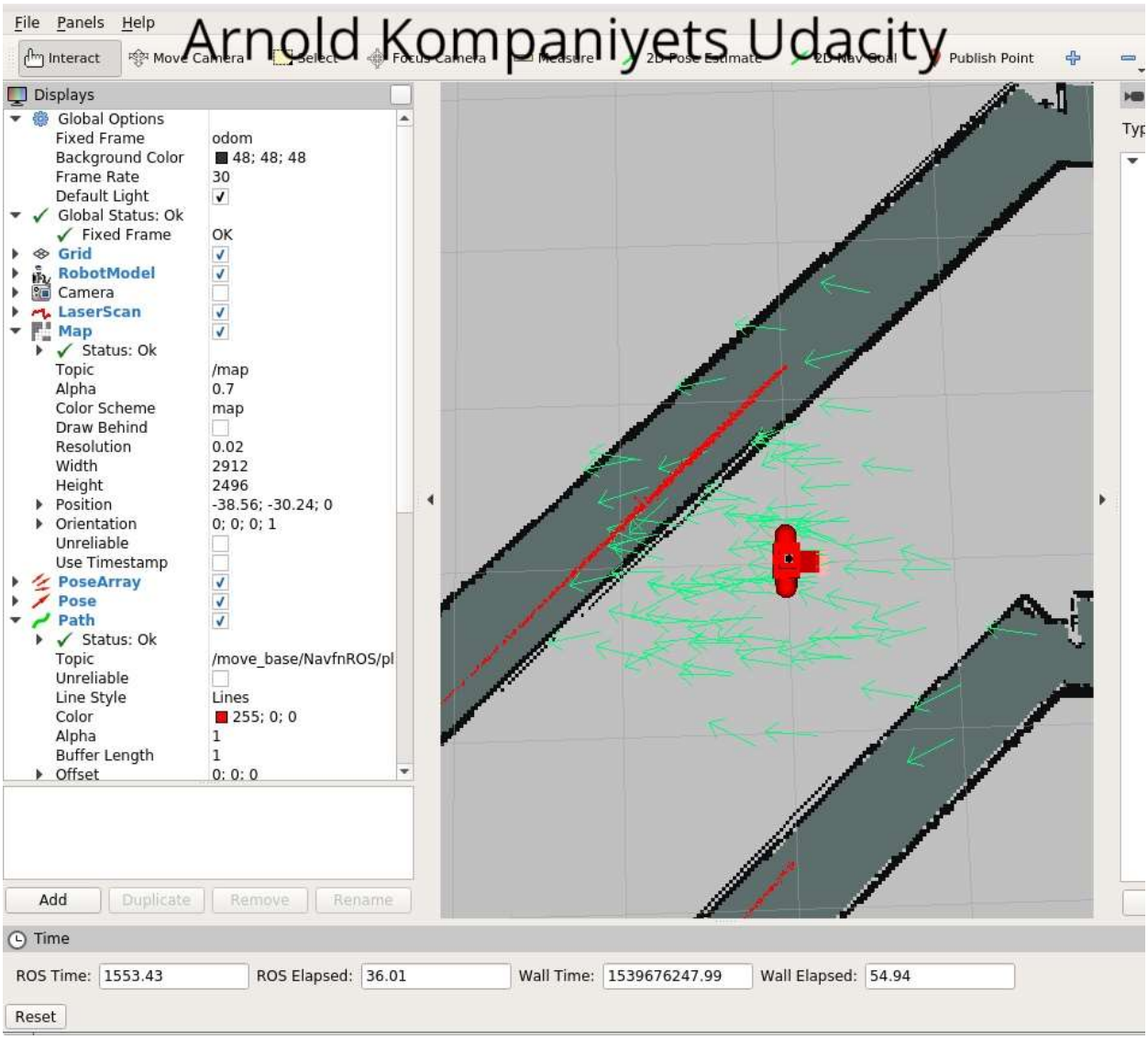
This robot was designed using four boxes for the main body (front torso, back torso, butt, and chest) along with two small back-leg boxes, a set of large shoulders and two cylinder arms. The arms have wheels connected to them laterally, the wheels being 0.2 meters in diameter and having 0.34 meters of separation. Additionally, the robot has a small cylindrical neck and a head, the latter of which has a front-facing camera attached and the Hokuyo laser scanner on top of the head. In order to off-set potential robot tilting, five caster wheels were attached to the main body of the robot. The final product looked as such:




Arnold Kompaniyets Udacity

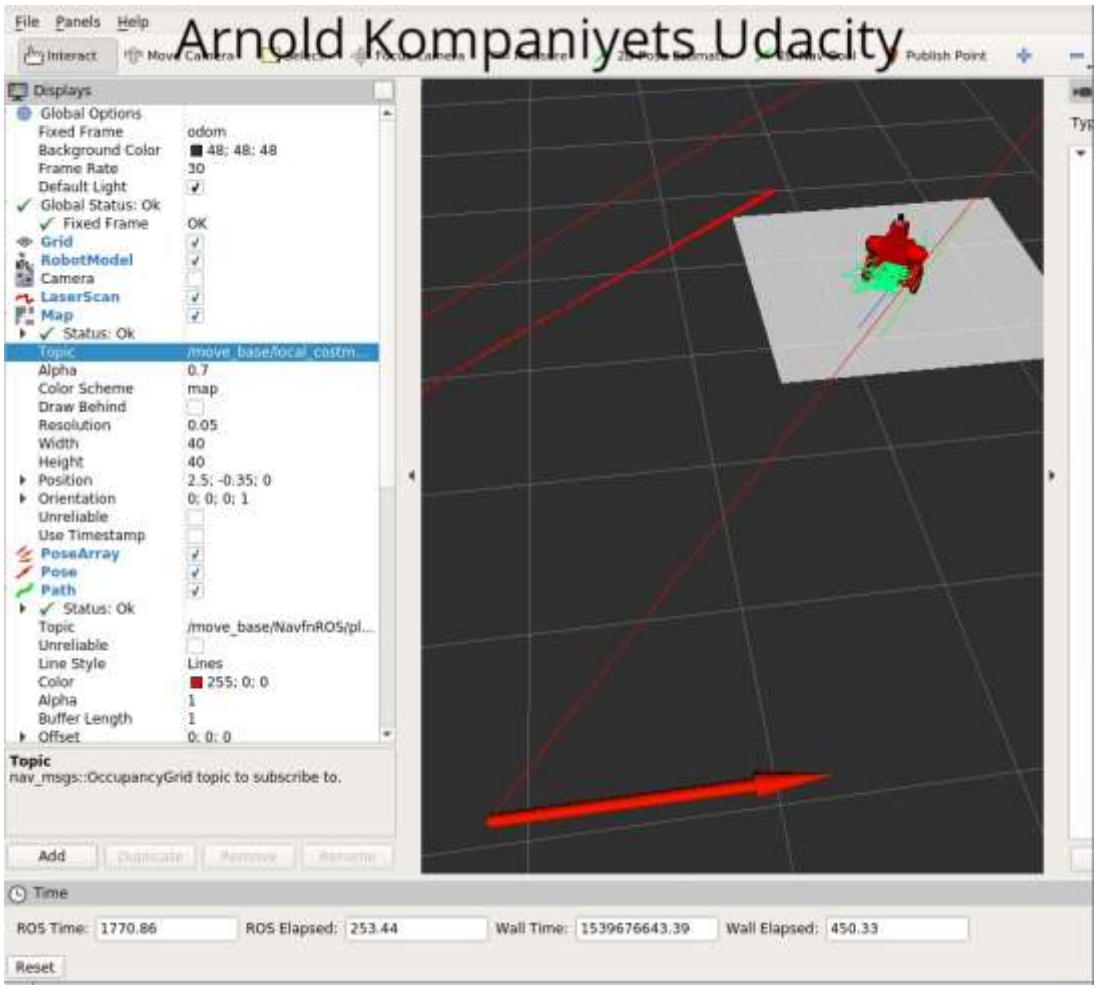

Arnold Kompaniyets Udacity

As with the Udacity Bot, the “Gorilla-Bot” also started off its journey with quite a bit of uncertainty in its particle filter:

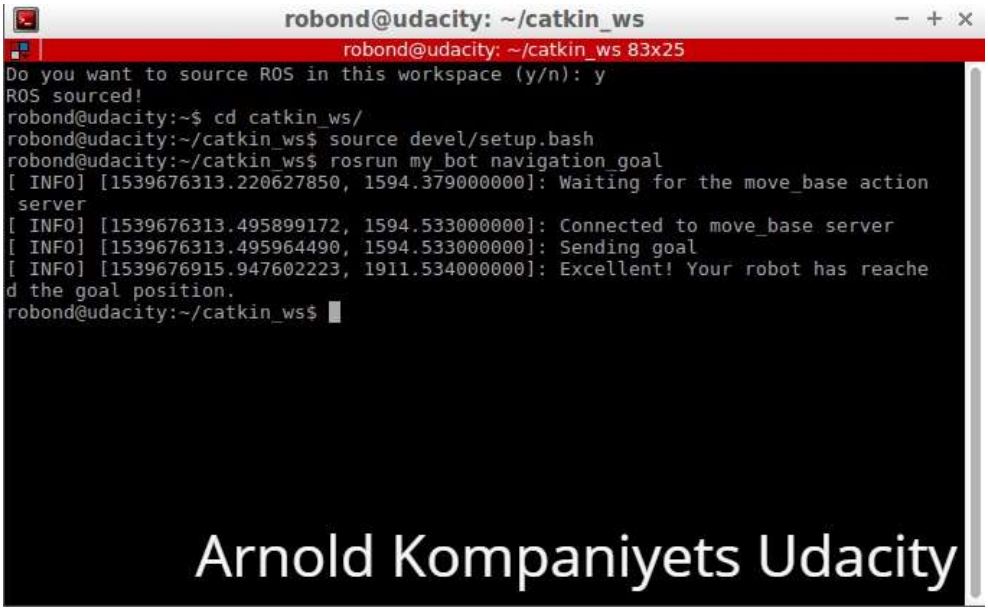


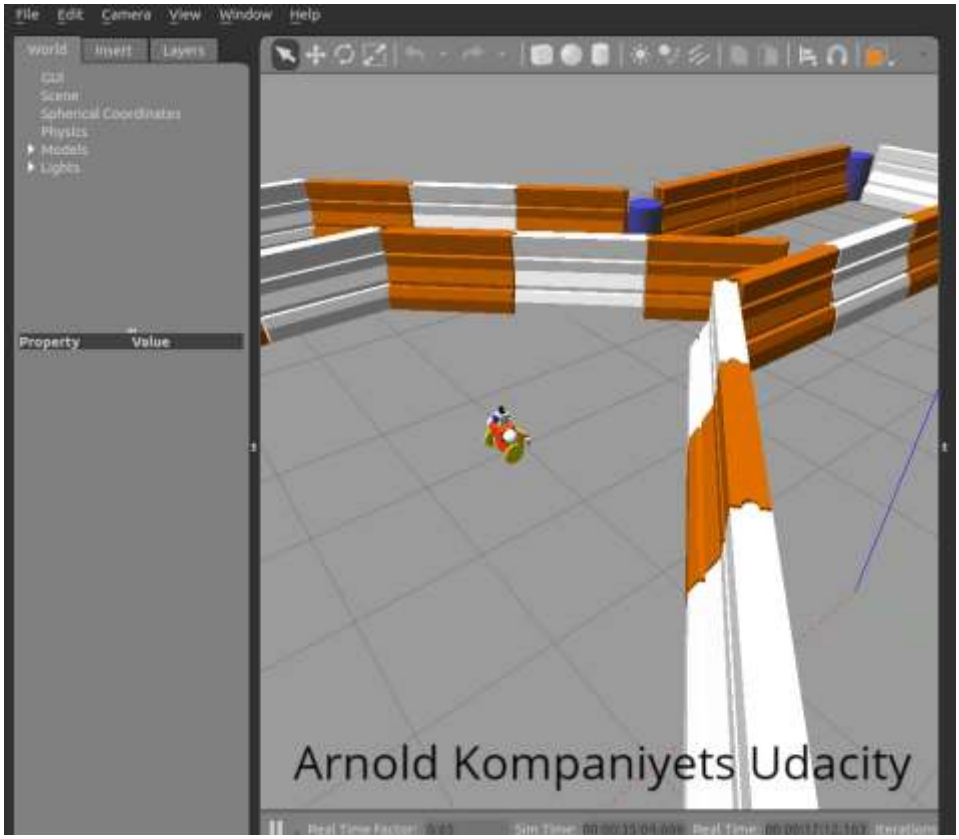
This initial uncertainty was similar to the Udacity Bot, showing positional variability of up to four times the size of the robot and approximately 30 degrees of variability in the robot’s estimated orientation.

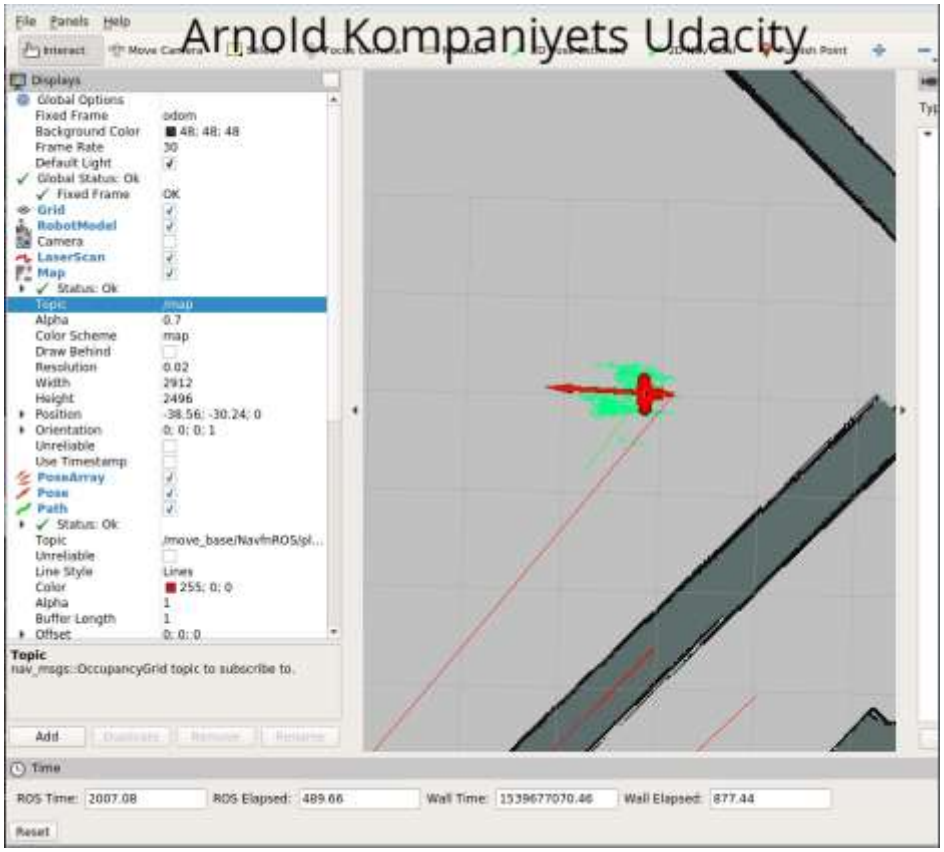
On its journey to the goal, the location estimation improved tremendously:

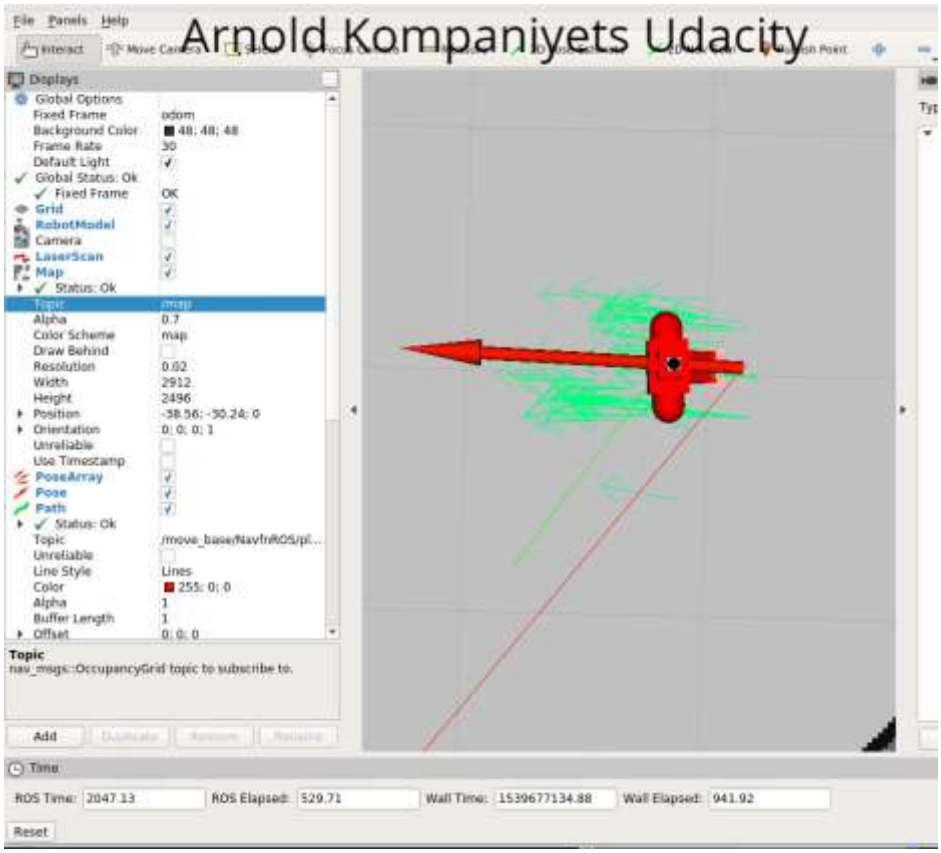


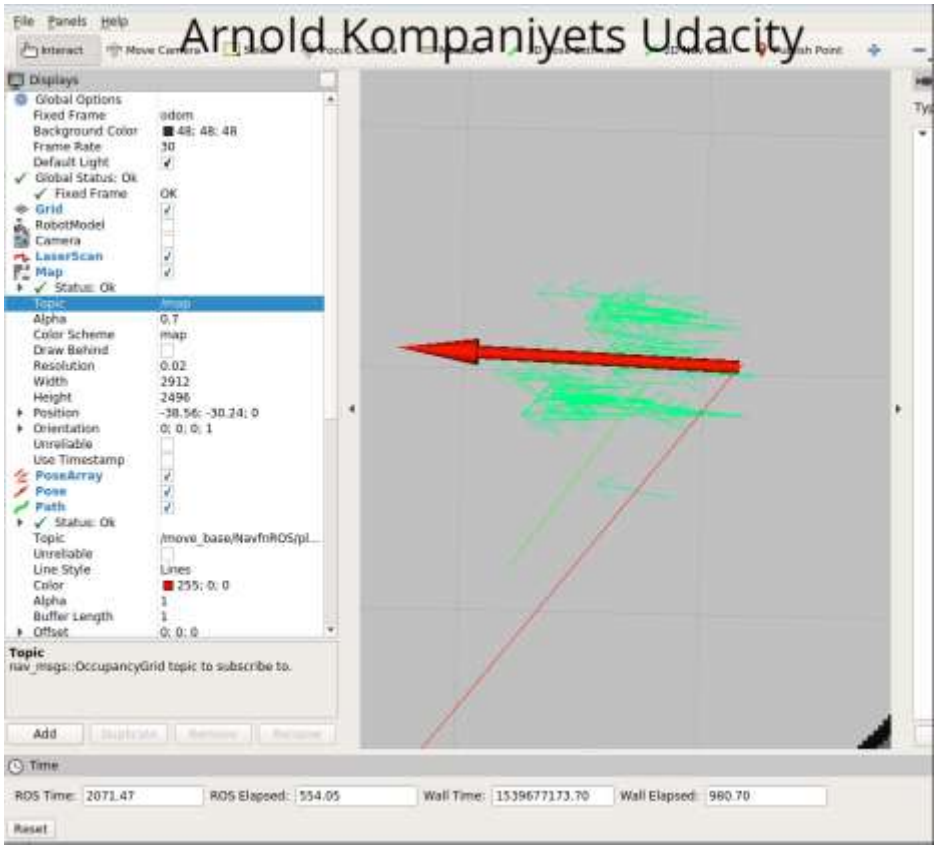
At its goal, the results were as such:




File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Models
Lights
Property Value
Arnold Kompaniyets Udacity


Arnold Kompaniyets Udacity
File Panels Help
Displays
Global Options
Fixed Frame odom
Background Color 48; 48; 48
Frame Rate 30
Default Light
Global Status: Ok
Fixed Frame OK
Grid
RobotModel
Camera
LaserScan
Map
Status: Ok
Topic /map
Alpha 0.7
Color Scheme map
Draw Behind
Resolution 0.02
Width 2912
Height 2496
Position -38.56; -30.24; 0
Orientation 0; 0; 0; 1
Unreliable
Use Timestamp
PoseArray
Pose
Path
Status: Ok
Topic /move_base/NavfnROS/pl...
Unreliable
Line Style Lines
Color 255; 0; 0
Alpha 1
Buffer Length 1
Offset 0; 0; 0
Topic
nav_msgs::OccupancyGrid topic to subscribe to.
Add
Time
ROS Time: 2007.08
ROS Elapsed: 489.66
Wall Time: 1539677070.46
Wall Elapsed: 877.44
Reset


Arnold Kompaniyets Udacity
File Panels Help
Interact
Displays
Global Options
Fixed Frame odom
Background Color 48; 48; 48
Frame Rate 30
Default Light
Global Status: Ok
Fixed Frame OK
Grid
RobotModel
Camera
LaserScan
Map
Status: Ok
Topic /map
Alpha 0.7
Color Scheme map
Draw Behind
Resolution 0.02
Width 2912
Height 2496
Position -38.56; -30.24; 0
Orientation 0; 0; 0; 1
Unreliable
Use Timestamp
PoseArray
Pose
Path
Status: Ok
Topic /move_base/NavfnROS/pl...
Unreliable
Line Style Lines
Color 255; 0; 0
Alpha 1
Buffer Length 1
Offset 0; 0; 0
Topic
nav_msgs::OccupancyGrid topic to subscribe to.
Add
Time
ROS Time: 2047.13
ROS Elapsed: 529.71
Wall Time: 1539677134.88
Wall Elapsed: 941.92
Reset


Arnold Kompaniyets Udacity
File Panels Help
Interact
Displays
Global Options
Fixed Frame odom
Background Color 48; 48; 48
Frame Rate 30
Default Light
Global Status: Ok
Fixed Frame OK
Grid
RobotModel
Camera
LaserScan
Map
Status: Ok
Topic /map
Alpha 0.7
Color Scheme map
Draw Behind
Resolution 0.02
Width 2912
Height 2496
Position -38.56; -30.24; 0
Orientation 0; 0; 0; 1
Unreliable
Use Timestamp
PoseArray
Pose
Path
Status: Ok
Topic /move_base/NavfnROS/pl...
Unreliable
Line Style Lines
Color 255; 0; 0
Alpha 1
Buffer Length 1
Offset 0; 0; 0
Topic
nav_msgs::OccupancyGrid topic to subscribe to.
Add
Time
ROS Time: 2071.47
ROS Elapsed: 554.05
Wall Time: 1539677173.70
Wall Elapsed: 980.70
Reset

As can be seen, the final localization estimates were much tighter in grouping. The guesses were no wider than the shoulders of the robot itself, with some greater variation in forward/back estimation (although always tightly hugging the goal line). The orientation guesses of the vector are nearly identical to the goal line, with only a few degrees of variation in some particles.

Model Configuration:

Udacity Bot:

Attaining the correct model parameters proved to be quite the journey. In following the lessons provided, the **transform_tolerance** was changed first. From 0.0, the parameter was first changed to 0.1. This allowed for the localization node to function, although when working with later tuning of parameters (such as resolution), there were times when the transform tree had a difficult time keeping up with only 0.1 seconds of tolerance. As such, this variable's value was raised to 1.0 and eventually 5.0 seconds. Realistically, 5 seconds of tolerable delay is quite excessive and wouldn't be suggested in a real-life robot, but in this virtual scenario it didn't lead to difficulties and kept the parameter from being a nuisance.

Then, the **update_frequency** and **publish_frequency** had to be changed in the global and local config files. Both of these were initially set to 50 Hz, a pace much too fast for the computational abilities of the system provided. As such, the frequency was lowered to 5.0 Hz for both parameters. Once again, this allowed for functionality, although later tweaking of other parameters resulted in a few instances of update loops taking longer than the allowed time, so both parameters were slowed down to 1.0 Hz.

After this, error messages began to appear regarding control loops taking longer than allowed. This was tied to the **controller_frequency** parameter. Initially, this parameter was updated in both the local and global config files, but appeared to have no effect on the allowed control loop times. Therefore, the parameter was instead changed in the **amcl.launch** file, under the **move_base** node. The frequency was

lowered from 20 to 5.0 Hz, and eventually down to 1.0 Hz, to allow for greater tuning of other parameters without the control loop time becoming an issue.

Next, the three common costmap parameters were modified: **obstacle_range**, **raytrace_range**, and **inflation_radius**. A number of different values were used for the range parameters, from 0.5 meters to 20.0 meters. All in all, altering these parameters appeared of little value beyond a certain point. 0.5 meters was certainly too short, since the robot wouldn't have enough time to recognize an obstacle before hitting it, but 5.0 meters proved plenty for each of the range values, so these parameters were left as such. In regards to the **inflation_radius**, many values were also experimented with. Anything beyond 0.7 meters proved to be far too large, but 0.1 meters was not quite enough cushion for the robot to avoid constantly hitting the walls. In testing, 0.4 – 0.5 meters proved to be the most effective for fine-tuning other parameters, but in the final runs, 0.2 meters was chosen to allow for the most free space of movement for the robot.

At this point, the robot was able to move fine without stopping due to timed-out calculation loops or constantly running into walls. However, the robot's ability to navigate to a goal was still atrocious. Therefore, attention was turned to the **base_local_planner** for better navigation. It was observed that a warning would pop-up on the terminal suggesting to turn on **meter_scoring**. This was experimented with, but ultimately proved to show little benefit, so it was eventually left as "false". To allow the robot enough time to calculate paths in tough situations, the **sim_time** was raised to 15.0 seconds, which would most likely be very excessive for a real-life robot, but allowed, in simulation, for the robot to plan better paths when very close to walls. **pdist_scale** and **gdist_scale** proved to be incredibly helpful to alter. The former determines the weight placed on costmap tiles relative to the path chosen, while the latter places weight on reaching the local goal. **pdist_scale** was lowered from its default of 0.6 to 0.1, and with **gdist_scale** raised to the maximum of 5.0, the robot was given the most freedom to stray from its pre-determined path and assure its short-term path was properly completed. With these parameters, the robot was still able to keep a sufficient perspective of its eventual path and goal, but focused most on getting over local challenges (which proved to be more problematic, such as rounding corners).

Another parameter called **occdist_scale** does exist, which places weight to avoid obstacles, and while it was initially thought that this would be very useful, raising this variable up proved to not alter the frequency of the robot hitting walls, so it was simply left at default. Ultimately, as can be seen in the provided ROS package, many other variables were experimented with, but showed to be inconsequential to successful navigation. The only other variables that proved useful in changing were the **max_vel_theta** and **min_vel_theta**, which set the maximum turning velocities of the robot. It was found that the robot tended to turn a bit too rapidly for the local path planner to keep up, so the values were slowed to 0.1 and -0.1, respectively.

Having altered the base local planner, the robot did begin to navigate more appropriately, but only when tasked with reaching close goals. In trying to reach far away goals, such as the one asked for by the navigation.cpp file, the robot would seem to keep becoming confused and run itself back and forth, with no successful trajectory in sight. This challenge didn't seem to initially have a source, since the local planner was well-tuned, and the global path was set appropriately (since it could be visualized in Rviz). The solution came after reading through the Slack forums and realizing that the path planner was basing its immediate trajectory decisions based on the total local costmap. The global costmap could be allowed any size, since it served merely as a means to plan the overall path from the provided map, but the local costmap was the source of trouble. Initially set to the same size as the global costmap, it was realized that the local costmap was placing far too much emphasis on the direction of the eventual goal or other parts of the total path, should either appear on the local map. This led to the robot turning endlessly, being pointed in directions of walls or away from the actual short-term path. Upon making the local costmap smaller, first to 5 X 5 meters and then eventually to 2 X 2 meters, the local planner was only able to see its immediate path and make short distance plans accordingly.

With the navigator taken care of, attention was paid to the localization. Within the **amcl.launch** file, a few changes and additions were made to the amcl node. First, the range of allowed particles was lowered to 20-100. The default range cluttered the screen with far too many vectors to properly visualize the success of the localizer,

hence the lowering of total particles allowed. Having around 100 particles to start with didn't appear to significantly alter the success of the localizer, so it was kept at that level. Next, the **laser_max_beams** was raised from 30 to 100. This was done in the hopes of the Hokuyo laser being allowed greater accuracy in sensing the local environment, although it is not certain if any benefit was actually derived from this. Following, the **kld_err** and **kld_z** were altered from default to 0.0001 and 0.9999, respectively. The two parameters determined the amount of allowed error in the estimated particle distributions, and since the map of the environment was fully known and the sensory noise was assumed to be zero, the allowed errors were lowered significantly from default, by a factor of 100 actually. Next, the **update_min_d** and **update_min_a** were changed. These parameters determined the minimum distance and turn, respectively, the robot could take before an update in localization was made. The default values were a bit too large (leading to issues especially at the final goal, where the robot would be turning too rapidly to attain the correct desired angle), so they were arbitrarily lowered, allowing for much more frequent array updates. After this, a few other parameters were altered, all relating to the laser sensor, but appeared to have little to no effect on the eventual success of localization.

"Gorilla-Bot":

Having figured out the big issues in navigation and localization, setting the parameters for the custom robot proved significantly easier. First, the Gazebo URDF folder was altered to reflect the new robot's wheel diameter and separation, in addition to body part colors.

In the common costmap parameters, the range values were both left at 5.0 meters, and the **transform_tolerance** was left at 5.0 seconds. The latter could have probably been lower, but it was nonetheless kept as such due to the much higher number of components present in the robot. The **inflation_radius** was raised to 0.4. The "Gorilla-Bot" proved to be a bit more maneuverable yet more affected by wall collision, so a bit more cushion was allowed for in terms of wall proximity.

In the global and local costmap parameters, the parameter values remained largely the same. The **update_frequency** and **publish_freqency** parameters were kept at 5.0 in both global and local files. The global costmap size was allowed to remain at default, but the local costmap was once again lowered to 2 X 2 meters.

Following the lessons learned from tuning the Udacity Bot, the **base local planner** had fewer parameters needed to alter. The **pdist_scale** and **gdist_scale** were altered as with the Udacity Bot (to 0.1 and 5.0, respectively). The **sim_time** was adjusted similarly, to 15.0 seconds. The acceleration limits for the x-direction (i.e. forward movement) and theta (i.e. rotation) were lowered, as were the theta max velocities allowed (from default to 0.2 and -0.2), once again to prevent the robot from turning too rapidly for the local path planner to catch onto.

With the navigation component tuned and performing successfully, the localization was once again focused on. The **amcl.launch** file was edited as before, starting with lowering the allowed particle range to 20-100, as with the Udacity Bot. The **laser_max_beams** were once again raised, but this time to only 50, as 100 seemed too unnecessary and only adding to computational cost. The **kld_err** and **kld_z** were once more lowered to 0.0001 and 0.9999, respectively, and the **update_min_d** and **update_min_a** were lowered again to 0.05 and 0.1, respectively.

Discussion:

All in all, the two robots used were able to perform quite well in their localization capabilities. From the initial range of uncertainty, both robots were able to attain significant improvements in localization accuracy through minimal movement, and while a few other parameters were altered, it seems that the largest effect on localization accuracy was derived from changing the **kld_err** and **kld_z** parameters. Forcing the localization node to abide by more strict error allowances kept only the most accurate particles alive, which allowed for a more aesthetically-pleasing array cloud.

Navigation proved to be a much more stressful component to alter. The base local planner parameters and the common costmap parameters were the logical

components to alter, and many hours were spent making all sorts of changes and trying out different permutations, attempting to logically figure out what was preventing the navigation node from simply telling the robot to follow the path towards the goal so clearly set by the global planner. It was only after maximum frustration was reached that the Slack forums were explored and the issue with the local costmap size was discovered. To see how such a minor change could suddenly make the navigation work fine (especially with the silly reasoning that the local navigator couldn't parse the difference between the part of the path closest to it compared to a later part, as well placing unnecessary weight on the goal itself) was, in a way, even more frustrating, but nonetheless ended up being the all-important change.

In regards to the kidnapped robot problem, the AMCL package should work perfectly fine. In instances when the robot would become confused during various testing phases of this project, the localization node would simply increase the number of particles, adding new vectors in a greater range of uncertainty. With minimal movement, the AMCL node was able to once again eliminate the unlikely particles and only leave the most accurate vectors going. Should the robot have been picked up and placed randomly at another location in the map, I presume it would have gone through the same pattern of adding a new set of particles at a wide radius around the robot and then re-narrow the set of particles after additional movement and measurements.

Considering the industrial uses of particle filters, said filters can be applied to virtually any field requiring extensive robot movement in a non-controlled environment (i.e. not the factory floor). Self-navigating cars are definitely an example of this, as are rovers designed to explore areas inaccessible to human exploration. There's no reason why this couldn't be scaled up to help large drones/airplanes navigate the skies or ships navigate the seas. Of course, the same could be said about scaling down, using particle filters to help very small robots navigate complex structures. In essence, the particle filters could be used in any instance where a map of a given environment is accessible, suiting this filter for use in any robotic application that requires movement outside the very streamlined and predictable factory floor.

Conclusion/ Future Work:

In discussing accuracy versus processing time, the classic logic still certainly holds true – the greater the desired accuracy, the higher the processing time. With this project specifically, there weren't necessarily too many instances of this, but an easy example to mention is with the variable of "resolution". Altering the resolution for the global costmap proved to not have any effect, since the pre-made "jackal-race" map resolution took over, but this did not apply to the local costmap. Therefore, increasing the resolution even slightly (for example from 0.5 to 0.1), which raised the amount of detail present in the local map, also led to significantly longer processing times. Prior to the resolution increase, allowing 5.0 seconds for the **transform_tolerance** and 5.0 Hz for the update parameters was far above necessary, but afterwards appeared not nearly enough. This presented a pressing issue, because while a more detailed map is indeed beneficial for more accurate robot movement and localization, it is also potentially quite detrimental, since taking over 5 seconds to fully transform the robot's odometry measurements into map coordinates can result in severe enough delays to make accurate localization and motion planning difficult. Therefore, the true problem lies in correctly balancing desired accuracy with the limitations of the hardware used. High accuracy and detail is important, but not to the point where the resulting computational delay leads to excessively long processing times, because the overall accuracy ends up falling off. Sure, the robot could proceed to move incredibly slow, allowing for high processing time to be cushioned and keep the desired accuracy. In the real world, though, unless the utmost accuracy in needed (such as perhaps surgical robots), it may not be plausible or wanted to have such a slow-moving robot. In properly figuring out the balance of the highest accuracy without heavily straining the hardware, the optimum balance can be achieved to allow a robot to run at reasonable speeds. Ultimately, superior hardware would be necessary to further increase accuracy.

Continuing, it was discovered that the processing time dilemma becomes all the more prominent with increasing robot model complexity. The Udacity-Bot had only a single box as a torso, along with two wheels. The "Gorilla-Bot", however, had 14

different components in its model design (without the sensors), which made transformation times much longer, even without efforts to increase accuracy. This was an important lesson to learn, illustrating the benefit of following Occam's razor – in building a robot, make the simplest model to perform the task correctly, as that will allow for the lowest performance time requirements.

As a sidenote to the above point, adding additional sensors would also apply. The fewer sensors necessary to accomplish the task would lead to the fastest performance times, but there is certainly something to be said about the benefit of sensor fusion. By adding additional sensors (versus having the laser scanner doing all the work) would allow each of the individual sensors to have lower accuracy needs. With a "the whole is greater than the sum of its parts" mindset, each sensor could work less intensely, yet still provide for superior overall sensory data for navigation and localization. It would certainly be interesting to see just how the performance demands vary in increasing sensor numbers and if there is an ideal ratio of sensor number to required sensor accuracy.

Besides simplifying the robot model base and adding more varied sensors, it would most likely be beneficial to add a set of four omnidirectional wheels, versus just a set of two normal wheels. The differential drive controller worked adequately for the navigation task provided, but there were many instances where the robot, upon running into a wall, needed to back up, turn around its base, and then once again proceed forward in a new direction. This made reaching the goal a much more time-consuming task. With omnidirectional wheels, the robot could navigate past obstacles much quicker and perform the desired task faster.

In trying to bring the implemented robot models into the real world, a few considerations would need to be made. First, the caster wheels wouldn't quite work, since no material used would actually be able to have zero friction. Therefore, the caster wheels would need to be replaced with ball bearings, which would allow for the simulation of non-friction better. The materials used would also need to be placed into more careful consideration, especially if building a small robot akin to the robot models used in this project. Due to the small size, an immediate limitation would be placed on

the size of motors that could fit inside the body to drive the wheels. Therefore, if using anything other than thin plastic, or perhaps molded carbon fiber, the motors might not be strong enough to drive the wheels forwards (especially if the wheels are relatively heavy themselves). As such, the wheels would most likely need to be of a lightweight material as well. The camera and laser scanner could still certainly be installed, but there would be the additional task of supplying a power source, meaning a battery would need to fit inside the robot's main body. Furthermore, there would be the matter of handling the actual processing task. In simulation, the program running the virtual robot could directly receive sensory data, interpret it, and then act on the robot. In a real-life implementation of this, the interpretation component would need to be accounted for as well. This would mean either having something like a Bluetooth dongle to send sensory data to a nearby computer (and motor commands back to the robot), or fully integrating something like the Jetson TX2 inside the robot model itself (which can directly connect to the sensors and the motors, albeit placing a greater demand on the battery power source). Lastly, unless the robot was to be used in a fully mapped environment, additional coding would need to be performed to allow for the robot to do simultaneous mapping as well.