Lightweight Visual Positioning System

by

Matt Calhoun

Maryville University

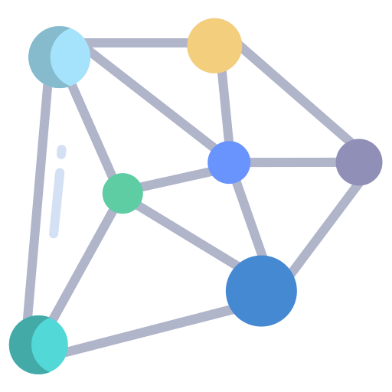
December 2023

Summary

For my A.I. master’s degree capstone project, I have built a working reference implementation of a Single Board Computer (SBC)-deployable vision-based positioning system that utilizes inexpensive hardware a DIYer would typically already be using, to overlay a cartesian coordinate system onto any area. This will allow hobby-level machines to perform higher-level navigation-intensive tasks efficiently. Given a map of an area (JSON file) describing a few known and recognizable objects, as well as their position in this imaginary overlayed coordinate system, the positioning software can estimate a robot’s current position and heading within the area.

Motivation for Building

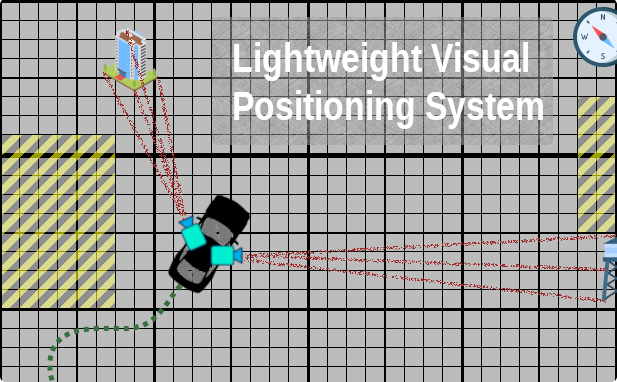
Creating autonomous robots that can accomplish useful tasks is extremely challenging. If the target task requires movement within a space, effective navigation involves an entirely distinct set of challenges. By *effective navigation*, I mean traversing a physical space in such a way that intelligent, efficient movement is possible and minimized. Whether a goal is to cover an entire area thoroughly or to arrive at a specific destination in the shortest time, many tried-and-true classical computer science algorithms are available to assist. However, these algorithms typically require calculating distances and knowing one’s current position relative to other known positions.

For example, to apply one of my personal favorite algorithms, the A\* Algorithm (Russel & Norvig, 2022), to navigation, one would need to know locations and their relative distances, the robot’s starting position, and how to physically move to each location. Such an application would require a map and the ability to determine location and heading. This presents a difficult technical gap for a DIYer to bridge. A GPS-based system could be useful, but when working on small-scale (and typically indoor) projects, accuracy within inches is desired.

Alternative Solutions

Some impressive solutions to the problem exist, but they do not suit the needs of a DIYer. For example, Sewio Networks (2023) offers an industrial-grade indoor positioning system. The system utilizes ultra-wideband (UWB) radio frequencies, measuring UWB pulses between permanently placed transmitters and portable receivers. The receivers are mounted to the item to be tracked, and determining location involves making API calls to a centralized server, which monitors the positions of all tracked items. While this solution seems perfect for inventory tracking, utilizing it to drive a robot through a maze would not be cost-effective. The system is also not portable, which might be a desirable feature for a DIYer.

Other solutions typically require the purchase and installation of permanently fixed hardware, such as Bluetooth beacons or other signal emitters. Another interesting-looking option from Mapsted Corp (2023) utilizes a variety of available sensors on a mobile phone to determine position. This is not suitable for hobby-level autonomous robot navigation, as it requires constant access to a feed of crowd-sourced location/sensor mapping data and is only accurate to within 1 meter. The cost is also out of range for a typical DIYer.



Introducing Lightweight Visual Positioning System

LVPS is a system comprised of several software components that, in combination, allow a robot (or other device) to determine its current position and heading based entirely on what is currently within the robot’s field of view. After downloading any required vision models and maps, a host robot can determine its current position and heading, regardless of network connectivity, without using typical positioning sensors other than vision (no GPS, compass, Wi-Fi, or Bluetooth). A few features of the system are:

|  |  |
| --- | --- |
| Zero Infrastructure | No installation of emitters, no server to setup and maintain |
| SBC Capable | Ability to run entirely on SBCs typically used by DIYers (Raspberry Pi, Nvidia Jetson Nano, etc.). |
| Indoor Capable | No reliance on GPS or other electronic positioning signals. |
| Off-Grid Capable | No reliance on internet or Wi-Fi connectivity |
| Accuracy | Accurate position within inches |
| Easy Integration | At any time, the software controlling the robot can ask, ‘What is my position and heading,’ and get a quick and accurate answer (x, y, degrees). |

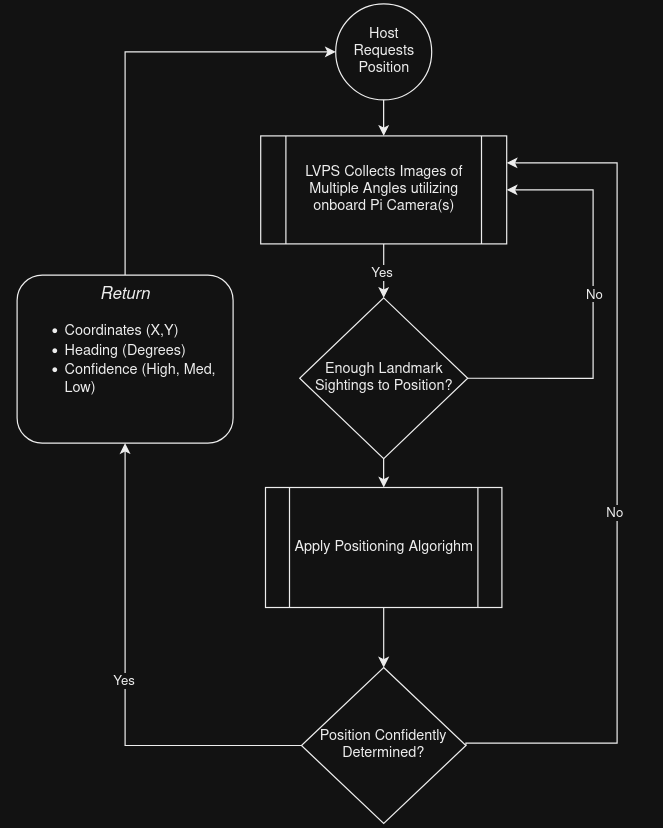
LVPS Components

Although only the *Pilot* software is required at run-time, multiple supporting applications and services have been developed to assist in testing, visualization, and other aspects of working with a multi-robot positioning system.

|  |  |  |
| --- | --- | --- |
| Component Name | Description | Technologies |
| Vision Models | The following vision models are deployable to mobile devices: An infrared light object detection model. A second detection model that can locate random objects that I will set around the area. | TensorFlow  TFlite |
| LVPS Maps | A simple text-based mapping format overlays a cartesian coordinate system on any area, blocking out certain areas, setting boundaries, and defining where 'landmark' objects can be found to assist in positioning. | JSON |
| Pilot and Config | A code library that can retrieve live images from multiple cameras and use these images with a map to determine the current location and heading from the cameras' point of view. For a host system using the positioning system, a simple method invocation to *get position* should *return a* set of coordinates, a heading, and a confidence indicator. This library also provides a simple framework for controlling and directing autonomous vehicles. | Python |
| Nav Service | A simple rest service that hosts the maps and models in a way that makes them readily available to robots within the system. The service also provides a centralized place for robots to log their locations to be tracked and easily retrieved later. This service can be hosted by one of the robots if desired. Ideally, the robots should be able to navigate fine without this but would need to connect to it initially to download maps/models/etc. | Python  Django  PostgreSQL |
| Bot Captain | A mobile app that provides visibility into the maps, locations, and movements of robots or devices using the system and the ability to assign tasks to any robot. | Android  Kotlin |
| Reference Robots | A few reference robots that implement a small set of capabilities host the positioning system and perform some basic actions. | Arduino  C++ |

Positioning Flow

At run-time on a host robot, the flow of requesting and receiving live position and heading is as follows:



Vision Models

The vision models are TensorFlow (Google, 2023) object detection models. Since the object detection models will be running on computationally limited devices, I have utilized the *lite* version of TensorFlow. Positioning should be possible with a variety of models and objects. Based on my goals testing, I created two models. One of the models recognizes infrared lights, while the other recognizes assorted items I purchased from a local Hobby Lobby store. I trained the object detection models as follows:

1. I took around 2,000 photos (using Pi cameras) of the IR lights and Hobby Lobby objects, ensuring the photos were of varying resolution and quality.
2. Using Label Studio (HumanSignal Inc., 2023), I selected bounding boxes and labeled objects on each image.
3. I exported the IR light images and labels as the *lights* data set and the Hobby Lobby object images and labels as the *basement* data set.
4. Following the process documented on the TensorFlow site, I started with an SSD MobileNet V2 model and further custom-trained the model using each of the two data sets (separately).
5. The result was two different object detection models, which I converted and exported as TFLite models.

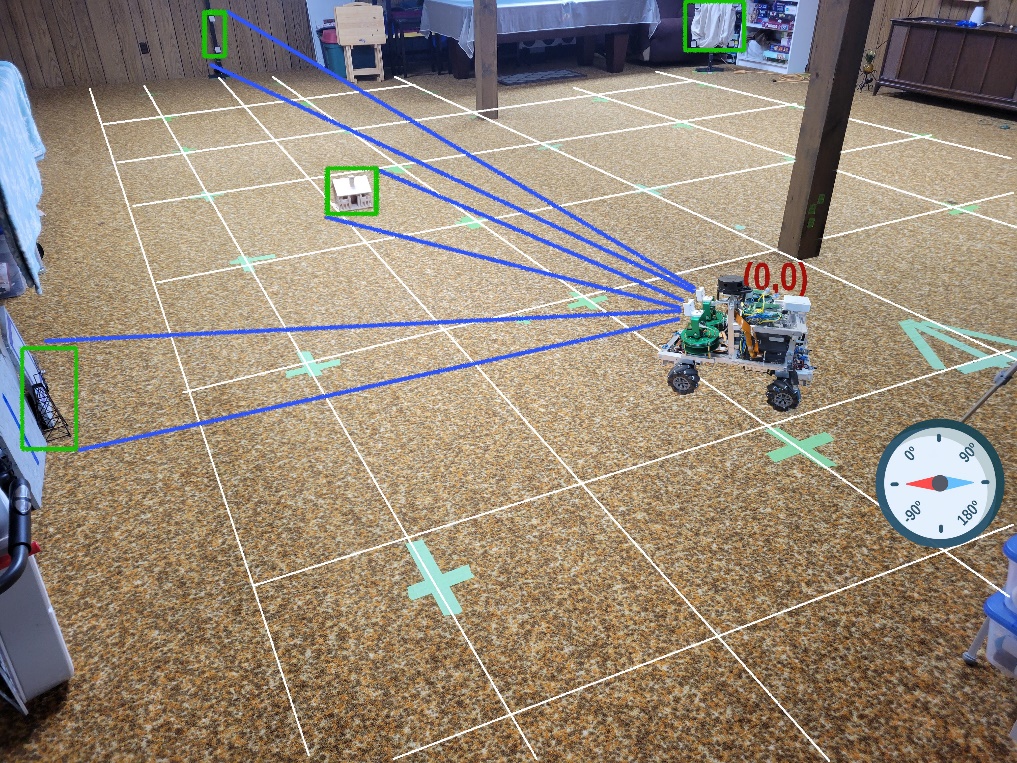
|  |  |
| --- | --- |
| **IR Lights** | **Sample Hobby Lobby Items** |
| Horizontal Angles |  |

Maps

Within LVPS, a map is a simple JSON document that describes where landmarks can be found in atarget area. The same area could be mapped in many ways. For example, I have a map of my basement that only considers IR lights as landmarks, while another considers IR lights *and* Hobby Lobby objects as landmarks.

When constructing a map, I arbitrarily chose a location in the basement to be the origin (0,0) and an orientation of the axes*.* This origin location and orientation do not appear anywhere in the map document. However, they become the frame of reference when taking measurements to determine the coordinates of the objects chosen to represent landmarks.

|  |  |
| --- | --- |
| Component | Description |
| Landmark | Any number and combination of landmarks can exist on the map. Each landmark will include the object type and the object detection model that knows how to recognize the object, its height and width (used for gauging distance by looking at the visual size vs. known size), whether the object might be visible to LIDAR, and a few other optional configurations. |
| Priority / Tier | Landmarks can be prioritized so that specific landmarks are preferred for positioning purposes over others (improving accuracy in some cases). |
| Dead Zones | Zones where positioning may be impossible due to obstructed view, etc. |
| Boundaries | The min/max (x,y) coordinates that are considered to be *on-map* or *in bounds*. |
| Near Boundaries | A 'gray zone' that is considered out-of-bounds but still within the realm of possibility that a robot may find itself there |
| Obstacles | Areas where the robot cannot or should not travel. |
| Search | For search tasks, a set of objects considered searchable on the current map. |



In addition to providing the necessary inputs for LVPS to triangulate position, the map also serves as a visualization aid within the Bot Captain app. The app merges the map document with Pilot self-reported positions to help users visualize what is happening with robots.

|  |  |
| --- | --- |
| Example Map JSON Snippet | Bot Captain Map Visualization |
|  |  |

Nav Service

This rest service enables robots using the LVPS system to download maps, vision models, and assignments. It also allows the Bot Captain app to have visibility of all robots within LVPS and assign tasks. If desired, the robots can also see one another’s positions and travel history, which may be helpful during cooperative tasks.

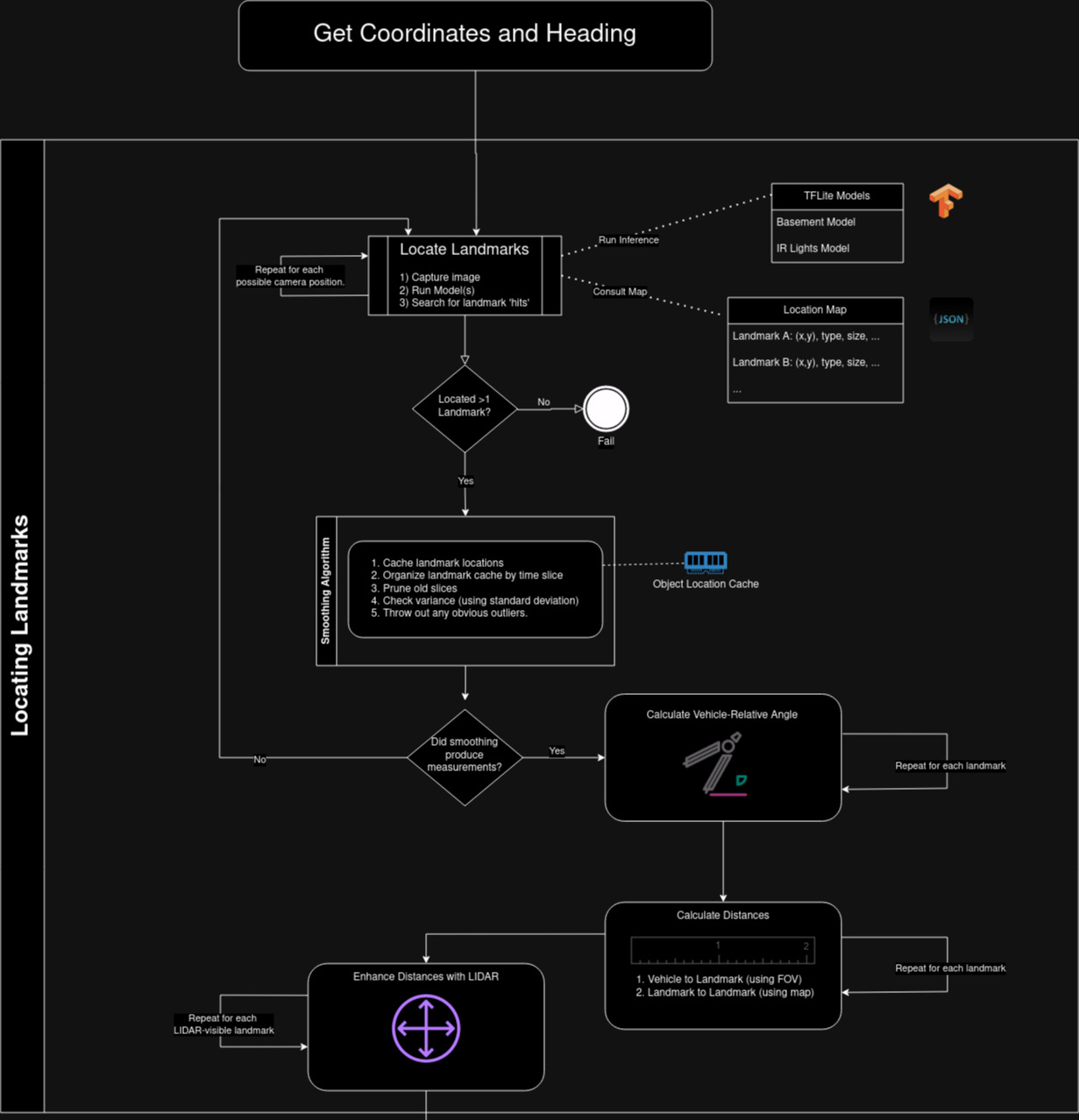
During testing, the Nav Service is used as a logging system for LVPS Pilot, so images used for positioning can be logged for further analysis. Only intermittent connectivity to this service would be needed for robots to perform autonomous tasks. No connectivity would be needed *during* tasks.

The Nav Service is a simple REST service built on Python/Django that uses PostgreSQL (2019) for persistence.

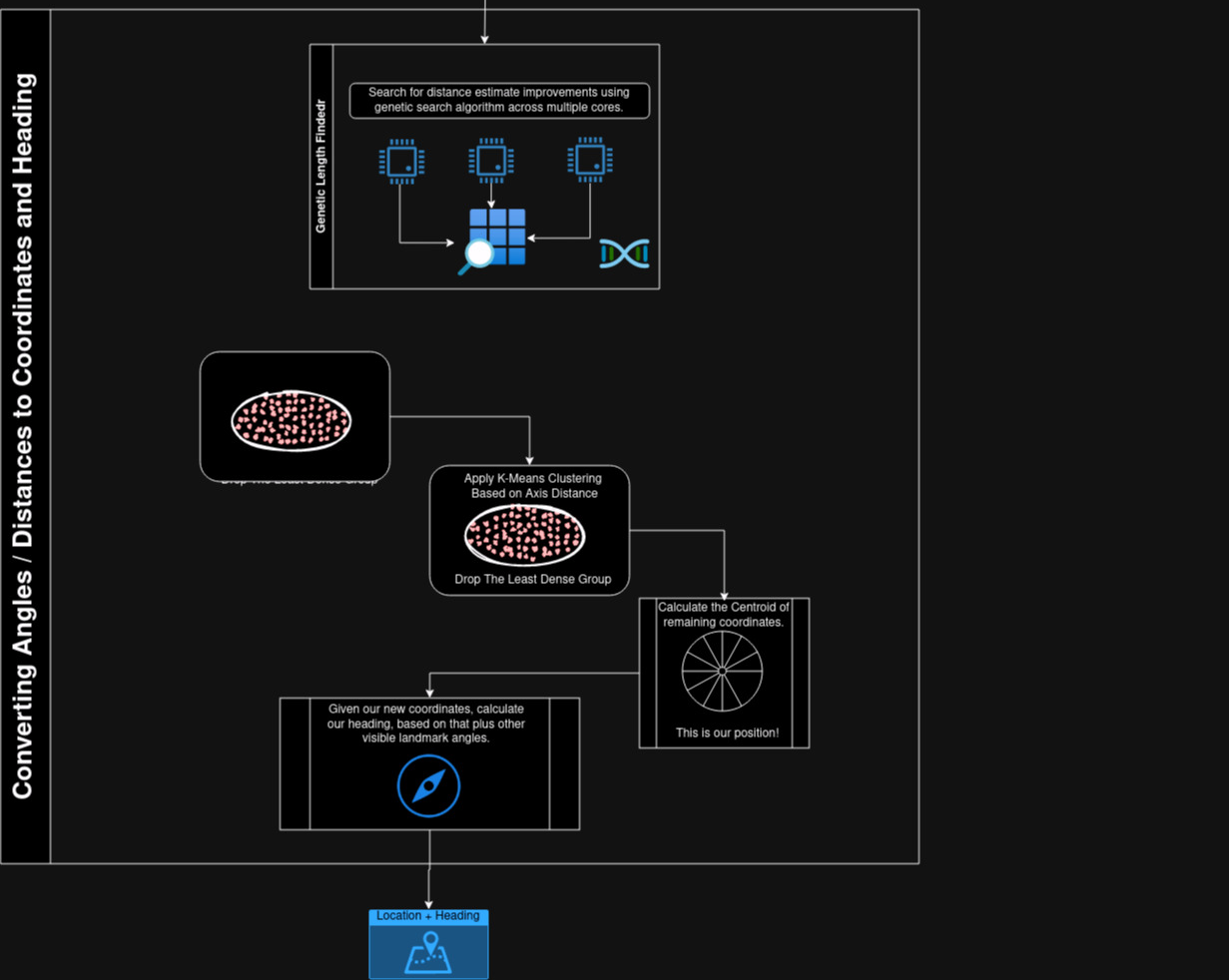
LVPS Positioning Algorithm

While the positioning algorithm has many steps, it follows the flowcharts below, which can be summarized as follows:

1. Use onboard cameras and TFLite models to locate any landmarks.
2. Use the Arduino-reported camera angles, known camera field of view, and object locations on images to estimate the distance from the object and vehicle-relative angle of the object.
3. If configured, repeat 1&2 as many times as desired to “smooth” the distance and angle values.
4. If configured, attempt to improve distance values using LIDAR.



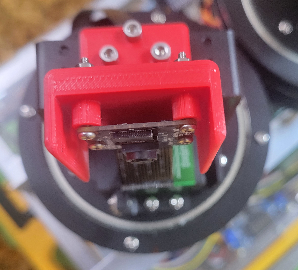
1. Assign a confidence value to each measurement, indicating the expected margin of error. For example, visually estimated distances are highly affected by bounding-box accuracy, so these measurements will have a lower confidence. Lidar measurements and visual angles (angles between two sighted landmarks) will have higher confidence values.
2. Improve the lowest confidence values using a genetic search algorithm (Hurbans, 2020). The output will be a potentially large set of values that serve as the final measurements needed to complete triangulation.
3. Convert each of the triangulated value sets to distinct coordinates and headings. The output will be a potentially considerable number of coordinates and headings. Use various techniques to remove impossible coordinates, given the map data.
4. Apply K-Means clustering (Heaton, 2013) to group remaining coordinates and analyze where predicted coordinates are most dense.
5. From the densest group of coordinates, locate the centroid of heading and centroid of coordinates. This is the final selected position.



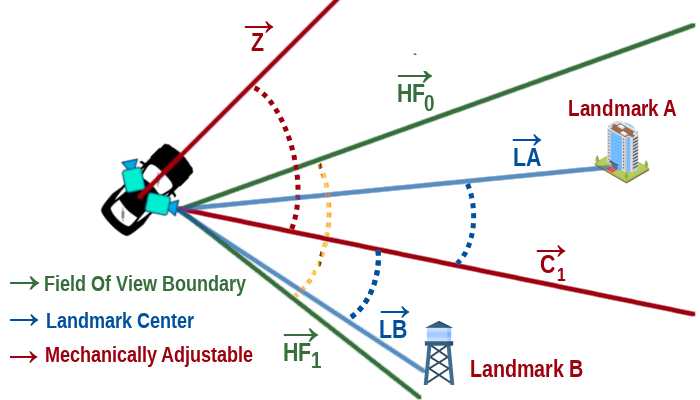
A more detailed description of the calculations involved in the process follows in the next section.

Calculating Direction

In contrast to estimated distance, relative directions can be calculated much more reliably. The cameras are mounted on calibrated servos, which are mounted on fixed and known positions and orientations onto the vehicle. Given that we know the angles involved there, and we trust the Arduino-reported servo positions, we can have high confidence that when an image is captured, we know the vehicle-relative angle of the image.

Also, we know the horizontal field of view of the camera, as well as any cropping and scaling that was done, so for any given pixel or set of pixels, we can determine the vehicle-relative angle of that pixel. By then looking at two sighted landmarks and calculating the angle between them, we have some useful information in determining the direction we are facing, as well as another piece of the position calculation.

The following illustration shows an example of measurements available to us, given that two landmarks have been sighted. In the illustration, Z is a vector representing the ‘zero’ position of the vehicle (where it is pointed). HF0 and HF1 are vectors representing the horizontal limits of the image field of view. C1 is a vector representing the center of the camera’s horizontal field of view. LA is a segment between the camera and the sighted landmark “A,” and LB is a segment between the camera and sighted landmark “B.” By calculating the angles between the landmark segments and the centered camera and combining that with the other available measurements, we can accurately determine the vehicle-relative angle of each landmark.



Estimating Distance

Although the system will use LIDAR if configured, it is not required. To visually estimate distance (looking at the second example below), we imagine looking from the side at a right triangle involving the camera and the landmark, with the 90° angle vertically above the landmark. One of the triangle legs becomes the "top" segment *PT*. The hypotenuse, *LB*, is the segment between the camera and the landmark bottom. We use a second oblique triangle, with the sides being *LB* (landmark bottom), *LT* (landmark top), and *LH* (landmark height). *PT* length is the value we are solving for, as it is the ground distance to the landmark.

|  |  |
| --- | --- |
| *Same Altitude Example*  This example shows the angles involved when estimating distance for a landmark at the same altitude as the camera. |  |
| *Differing Altitudes Example*  When the camera is at a higher altitude, the calculation is different, and the illustration to the right shows how the angles look in that case. |  |

***Beginning with the oblique triangle...***

1. **Calculate ∠ *LH* , or the visual height, *in degrees*, of the landmark**. We use knowledge of the image resolution, the resolution of the down-scaled copy we fed to the object detection model, the current vertical field of view of the camera, and the boundary box height of the landmark.
2. **Find the degree/height ratio in our current view of the landmark**. By this, I mean how many degrees per inch, mm, or the unit the height is mapped in.
3. Use the degree/height ratio to **calculate how many degrees were added to the visual height of the landmark** when we overlaid the projection. In the diagram, this is highlighted by the green dashed arc. It is the visual height (degrees) of the *projection extension if* we could observe it.
4. **Find the visual projection height ∠ *PH* (in degrees)** by adding the visual degree height of the landmark to the *projection extension* degree height, and we now have a second angle in our right triangle (identified by the blue dashed arc).

***Now, back to the right triangle...***

1. **Calculate the projection height *PH*** (camera altitude - landmark bottom) in units, not degrees. We know these values because the altitude is available on the vehicle at any time, and the unit height of the landmark is available by consulting the map.
2. Find the hypotenuse: *LB* = *PH* / sin(∠ *PH* )
3. Finally find our ground distance, *PT* = √( *LB* ² - *PH* ²)

“Fuzzy” Trigonometry

Apologies to mathematics if I am butchering the term *fuzzy* here, but this is how I think of it. After repeating the calculations above for every landmark found on every camera at one point from a single location, we have many measurements with *varying degrees of accuracy*. Some measurements are based entirely on the bounding boxes' accuracy, which makes everything *fuzzy*. To try and account for this fuzziness, I assign a confidence level to each value, which will later be used by a genetic algorithm that will attempt to “improve” the questionable values.

Let us assume we have found two landmarks, which we will call *LA* and *LB*. Let us also assume the object detection model *correctly identified* these landmarks (which is not an entirely safe assumption, but we will ignore that for now). Let us also assume *the vehicle is level*. In the table below, I use the following ad hoc notation in the interest of conserving space:

* **↑** A ray from the front of the vehicle, or 0° (Z on the diagram).
* **∠↑ *LA*** Angle between 0° and where *LA* visually appears. If you draw a ray from the vehicle center to *LA*, this represents the angle between **↑** and the *LA* ray you just imagined.
* **※** Represents the current (x,y) coordinates of the vehicle's center.

|  |  |  |
| --- | --- | --- |
| Measurement | Confidence Level | Description |
| || *LA* - *LB* || | Very High | As defined on the map, we can calculate the exact distance between *LA* and *LB.* |
| ∠↑ *LA* | High | Assuming our camera view has not been altered somehow, we can be quite confident in the measurement that tells the degrees between the front of the vehicle and ↑ *LA*. |
| ∠↑ *LB* | High | Again, we can be quite confident in the degrees where *LB* appears, relative to vehicle 0°. |
| || ※ - *LA* || | Medium | Distance between vehicle center and *LA*. This measurement is based on a few uncertain values mentioned in the distance section above. Since this measurement has significant room for error, we can only have medium confidence. |
| || ※ - *LB* || | Medium | Distance between vehicle center and *LB*. |

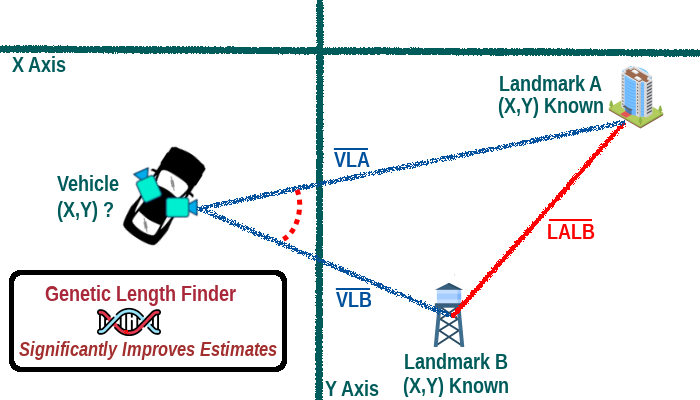
LIDAR

Although LVPS can estimate position and heading based entirely on vision, it can improve the estimates using LIDAR if configured and available on the host robot.

|  |  |
| --- | --- |
| For each landmark where the map indicates LIDAR detection may be possible, a second pass checks the 360° LIDAR data at the vehicle-zero relative angle believed to currently contain the landmark (within .5 degrees in either direction). Supposing we get a reasonable-looking LIDAR hit (within a configured range of the visually estimated distance), we substitute that LIDAR measurement for the previous distance and *boost the confidence* value of that distance measurement.  The result of that confidence bump is a shrinking of the search space for that value when the Genetic Length Finder algorithm takes over. |  |

Genetic Length Finder

After all the calculations and passes that occurred up to this point, a genetic algorithm takes over. In the diagram below, examples of low confidence values are shown in blue, while examples of higher confidence values are shown in red. The genetic *length finder* is so named because it searches for *lengths,* helping to arrive at a valid triangle when minor (or no) adjustments are made to the high confidence values and larger adjustments are allowed on the low confidence values. This finder returns a large set of potential solutions.



Extracting Coordinates and Heading

The length search algorithm will return many potential candidates for each vehicle-to-landmark distance. We convert angles to slopes from each returned and use geometry to plot an (x,y) coordinate. Due to the way the potential positions were calculated, the set of possibilities will include coordinates for which the vehicle is upside down or out of bounds. LVPS assumes the vehicle is right-side up and will remain inside or at least *near* the map-defined boundaries, so any coordinates that do not follow those assumptions are thrown out. By this point, the coordinates tend to look quite similar, which is good.

Clustering

In one last attempt to narrow down and improve the position candidates before making a final decision, we apply K-Means Clustering in two passes. The Scikit Learn package (Scikit Learn developers, 2019) is used to accomplish this.

|  |  |
| --- | --- |
| One pass calculates the map-relative heading (degrees relative to the y-axis) for each position candidate. We look at the mean/standard deviation among the headings. If there is significant variance (configurable), we generate two clusters, based on heading, and choose the densest cluster. If clustering *was* done, the less dense set of coordinates is thrown out.  On the second pass, assuming we still have enough potential coordinates to look for density, we again check the mean and standard deviation to see if there is a significant variance. This time, we use Euclidian distance from the origin as the metric. If distance variance is high enough to trigger clustering, we divide candidates into two groups and throw out the lowest density set. |  |

Final Coordinate Selection

After clustering is applied, we are left with a significantly smaller set of potential coordinate values (sometimes none). Whatever is left, we find the centroid (x,y) coordinate and calculate the heading if the vehicle were sitting in that position. The algorithm returns the centroid x, centroid y, calculated heading, plus a calculated confidence value. The confidence value is derived from the number of landmarks used and the combined confidence of base values.

Testing

Although LVPS should be considered highly experimental, a preliminary round of testing has been performed on the system. A reference ‘robot’ was built with a basic set of components. This ‘robot’ is called the observer, whose only function is to collect observations. A few images from the assembly of the observer are shown below. Note that LIDAR is available onboard the observer but can be turned on or off for testing purposes. More documentation on the build process can be found at: https://github.com/mattcalhoun1/LVPS/wiki/Vehicle:-The-Observer.

|  |  |
| --- | --- |
|  |  |
|  |  |

Test Process

When testing, I turn up the logging features of the Pilot software so I can later (and in real-time) view a position and heading log, positioning images, a map of the path taken, and all values and estimates used in each position estimation. By collecting all of this during a single test, I can later *simulate* the entire test, along with any improvements I have made post-test.

With logging fully enabled, I choose a predefined pattern to push the observer through the mapped space. A common pattern I use is an *N* pattern, as it tests the vehicle in corners and through the center of the mapped space. I alter vehicle orientation throughout, so I cover a large variety of coordinates and headings.

Recent Test Results

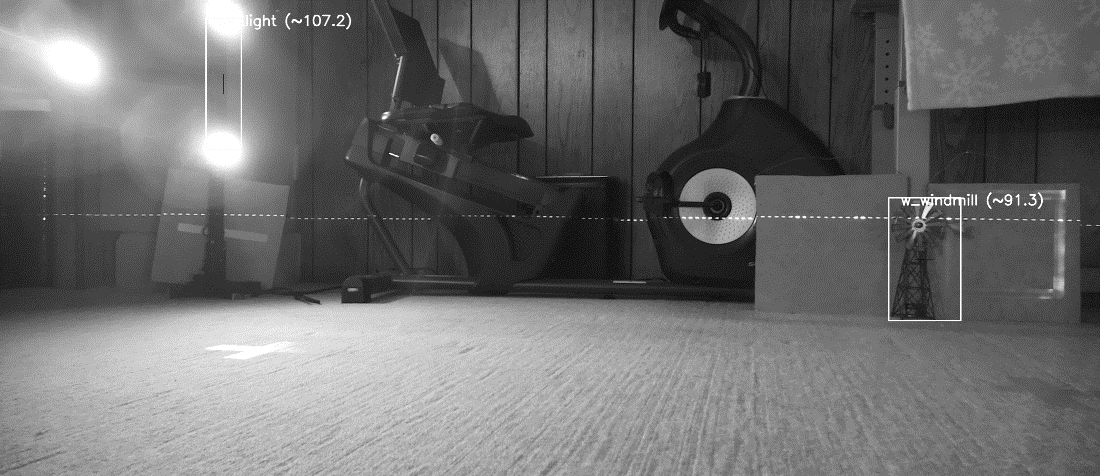
Video from a recent narrated test session can be found in the references section of this paper. During a recent test, I placed carefully selected landmarks throughout a 500 ft2 area of my basement. I created a JSON map of the area and made it available to the Nav Service. When I powered up the observer and started the Pilot software, it immediately downloaded the basement map over my home Wi-Fi and was ready to start accepting commands.

|  |  |  |
| --- | --- | --- |
| Issuing Commands | Sample Position Log | Recent Testing Map |
|  |  |  |

The map screenshot above shows a recent successful test, where the observer was moved through the mapped portion of the basement in an ‘N’ pattern. In addition to the coordinates being accurate, the headings also proved to be reasonably accurate. ￼

Positioning Image Examples

A couple of positioning images from recent test sessions are shown below. You can see the landmark IDs and estimated distance for each landmark spotted by LVPS in these images. The dotted line going across the center of the images is the reflection from LIDAR. As you can see, the shape made by the groups of IR lights helps to determine which (if any) landmark they represent.





A Note About IR Light-Based Landmarks

The TensorFlow model provides the location of each individual IR light, after which a follow-up method converts the individual IR light groupings into a single landmark when appropriate. Using the centers of the individual IR lights to identify the “height” of an IR-based landmark as an input to distance calculation has proven to work very well. In fact, the bounding boxes returned by TensorFlow for individual IR lights can be significantly off, but by centering each one and disregarding the rest of the bounding box, the relevant information derived from the landmark becomes quite accurate, even with inaccurate bounding boxes.

Movement and Search Testing

|  |  |
| --- | --- |
| Another series of recent tests involved placing a recognizable location in an obscured area and assigning the robot the task of traveling to that area, searching for the target object, and (if found) reporting the location of the target object to the NavService.  For these tests, I built a second robot host for the Pilot software. This second robot is named MecCar, in reference to its mecanum wheels, which allow for strafing and other unusual movements. The pilot software for this robot runs on a Raspberry Pi CM4. |  |

Documentation of the MecCar build can be found at: https://github.com/mattcalhoun1/LVPS/wiki/Vehicle-:-MecCar.

|  |  |  |
| --- | --- | --- |
| *Pineapple hidden beside a stereo* | *The robot’s view of the found pineapple, with the estimated distance noted in the image.* | *Map of travel, with the robot-estimated position of the pineapple marked as a green dot. Blue indicates LIDAR bouncebacks.* |
|  |  |  |

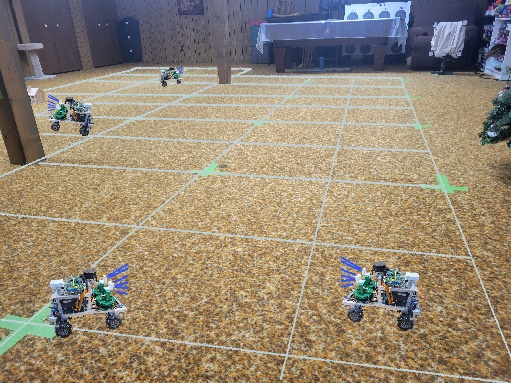
A short video showing portions of these sets of tests can be found in the references section of this paper.

Further Testing

More testing needs to be done for LVPS to become a production-ready (or DIYer-ready) positioning system. It has been accurate in my basement, but I have yet to test it in any other environment. Many permutations of objects, map shapes, and landmark positions should be tested.

I would also like to test the system on a larger scale. Over a much larger area (several acres+), it might be possible for a flying drone to utilize this positioning system in areas that have easily recognizable features, such as water towers, lakes, cell towers, buildings, or nearby well-defined skylines.

Potential Improvements

All current reference robots have camera movement capabilities, which allow the cameras to be rotated at almost any angle. However, these rotations take time, and by small robot (drone) standards, they can be quite heavy. It should be possible to collect enough images using a set of 3 or 4 fixed-position Pi cameras (which are small, light, and cheap). LVPS will support any number of cameras, provided the SBC can support them (Pi CM4 and Jetson Nano are currently limited to 2). Additionally, utilizing a 360° camera could be explored as another option.

Although the system does not require Wi-Fi or network access for autonomous robots to determine their position at any time, having the ability to track movement and issue commands in real-time has proven to be quite desirable when possible. Given that, if the NavService was deployed as a cloud-based service, DIYers could have access to all the same app features I have without having to do any special installations or setup. I think there might be significant interest in such a service.

Final Thoughts

Building LVPS was a highly rewarding experience despite the many challenges it presented. Over the past several months, it is no exaggeration to say that hundreds of hours of imagining, designing, coding, and brushing up on my math skills have gone into this project. This project has allowed me to thoroughly explore, exercise, and demonstrate all the A.I.-related techniques I learned at Maryville University.

I am looking forward to (and have already started) working on a simulated LVPS environment and an accompanying Gymnasium environment for applying reinforcement learning to applications that will use LVPS for positioning in multi-robot cooperative or competitive tasks.

All the code for this project is publicly available on my GitHub project site at:

https://github.com/mattcalhoun1/LVPS/wiki/Positioning-Algorithm

References

Calhoun, M. (2023). *Third Round of Testing: Vision-Based Positioning for Autonomous Robots*. Www.youtube.com. <https://youtu.be/eWgWZX0g05k?si=-5zw8lBauglW2TIe>

Calhoun, M. (2023b). *Vision-Based Positioning for Autonomous Robots - Simple Search Testing*. Www.youtube.com. <https://youtu.be/4Bum-ED4Oyw>

Farama Foundation. (2023). *Gymnasium Documentation*. Gymnasium.farama.org. <https://gymnasium.farama.org/>

Google. (2023). *Object Detection with TensorFlow Lite Model Maker*. TensorFlow. <https://www.tensorflow.org/lite/models/modify/model_maker/object_detection>

Heaton, J. (2013). *Artificial intelligence for humans volume 1: fundamental algorithms*. Heaton Research, Inc.

HumanSignal Inc. (2023). *Label Studio – Open Source Data Labeling*. Label Studio. <https://labelstud.io/>

Hurbans R. (2020). *Grokking Artificial Intelligence Algorithms*. Manning.

Mapsted Corp. (2023). *Best Indoor Positioning Systems - Local Positioning System for All Businesses | Mapsted*. Mapsted Technology. <https://www.mapsted.com/indoor-positioning-system>

Raspberry Pi Foundation (2023). Raspberry Pi. <https://www.raspberrypi.com/products/camera-module-v2/>

Russell, S., Norvig, P. (2022). *Artificial intelligence, A Modern approach*. (4th ed.). Boston: Pearson.

Sewio Networks (2023). *Real-time location system (RTLS) for indoor tracking*. Sewio RTLS. <https://www.sewio.net/>

Scikit learn developers (2019). *sklearn.cluster.KMeans — scikit-learn 0.21.3 documentation*. Scikit-Learn.org. <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>

The PostgreSQL Global Development Group. (2019). *PostgreSQL: The world’s most advanced open-source database*. Postgresql.org. <https://www.postgresql.org/>