Chase Devitt

https://github.com/CJdev99

Autonomous Snowplow Navigation System

Odometry Sources & Sensor Interface:

Wheel odometry from motor controller:

- We have two 1024 ppr encoders mounted to both wheels. We use these to calculate the velocity of each wheel. However, with the encoders being mounted to the wheels and not directly on the motors, the motor controller's PID control is not accurate (from what we have been able to do), so we run motors in open-loop mode (send voltage input rather than desired speed output).
- To interface ROS and the motor controller, we use the roboteq_diff_driver ROS package. This uses the motor controller's C++ API to send motor commands, receives the encoder count stream, and also publishes motor controller currents, temperature, etc. This package converts encoder data to a full odometry message: estimated (x,y) distance traveled, orientation, and linear & angular velocities (based on wheel circumference, distance between wheels). This data is prone to noise & drift, which is accounted for in the Kalman filter.

LPMS IMU:

- This sensor provides heading via magnetometer, angular velocities, and linear accelerations. We can access this data using the openzen C++ ROS driver (my updated branch to work w/ Kalman filter here)
- Using the LPMS control tool, we can modify the imu itself by changing filtering parameters and calibrating the sensors. Calibration is important, as if the sensor goes through a lot of vibration or sits around for a long time, the MEMS sensors can change and have biases. Also, in different environments, the magnetic field changes. This means it is important to calibrate the imu's magnetometer in different settings it will be used (we ran motors while calibrating to accommodate for noise).
- Our current IMU setup reports orientation from fused accelerometer+gyro data, and a low pass filter. To work with our EKF, The IMU orientation needs to be stated absolute to earth (orientation real value = 0 when facing east or including mag data in orientation calculation). This requires fusing magnetometer data in the IMU's Kalman filter, but through testing this exhibited strange behavior. This is why we use another filter on our IMU data called the *"IMU Madgwick filter".*
- This filter fuses data from the linear accelerations, angular velocities, and magnetometer readings, and filters noisy data. This allows us to have an absolute-to-earth orientation, which is important for gps conversion later.

Swiftnav+Trimble GPS

- Our GPS system consists of a swiftnav antenna and a Trimble BD990 GPS receiver. The receiver is configured to our router I/O so we can configure it over wifi, as well as

receive NMEA streams over a TCP port. There are many different NMEA sentences we can use, but we just use GGA and GST sentences.
    - GGA => latitude, longitude, altitude, etc.
    - GST => corresponding covariance (or expected accuracy) of paired GGA sentence
- In order to access the data, we need to parse nmea sentences, build ROS messages, and publish the data to our ROS system. For this, I wrote a ROS driver called nmea_socket_pub.
    - This code connects to the GPS' TCP port & parses the stream of NMEA sentences as they come in and publishes the parsed data to the trimble_gps/fix topic. This contains the latitude, longitude, and resulting *covariance* matrix. The *GST* sentence contains estimated 'covariance' or accuracy of the prior *GGA* (lat,lon) data that was sent at that time. With that data, we must convert it to the robot's coordinate frame (mentioned later).

Sensor Fusion Using Extended Kalman Filter

- Kalman Filters are commonly used in 'state estimation' when there are multiple sources of data at play.

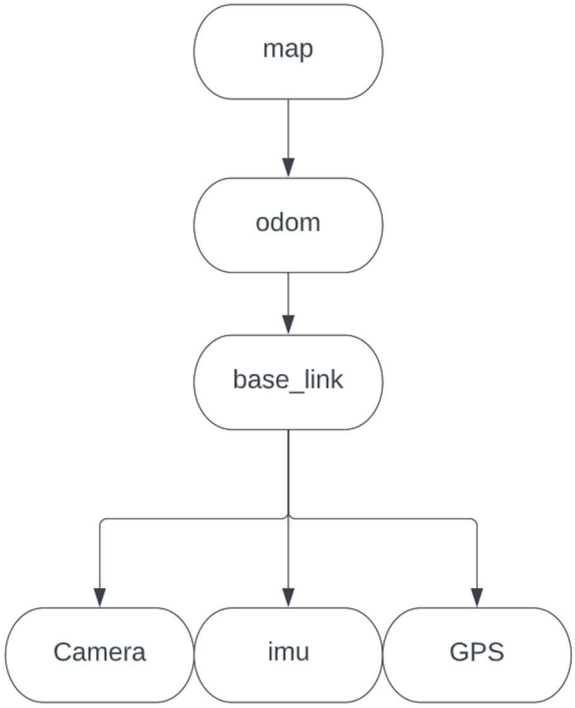
'The Extended Kalman Filter (EKF) is an algorithm used for estimating the state of a dynamic system from noisy measurements. It extends the basic Kalman Filter by allowing for non-linearity in the system model and measurement models, making it suitable for a wider range of problems. The EKF works by iteratively updating the estimated state of the system, using the current estimate and the latest measurement, and then using linearization techniques to approximate the non-linear system as a linear one, which can then be treated using the Kalman Filter. The output of the EKF is the best estimate of the true state of the system, taking into account both the estimated state and measurement errors (covariance) [1].'

- Another important use-case is the ability to estimate an unknown variable from other given data. For example, if we fuse linear & angular velocity from our wheel odometry and linear accelerations, angular velocity, and heading from our imu, the Kalman filter will provide fused data for all of these variables (taking into account expected noise of the measurement), along with an absolute position (x,y, heading) estimate from the resulting differential equation from known and unknown variables.
- Along with this data, the filter also involves a covariance matrix, which, put simply, is the estimated noise/std deviation of the filter's fused orientation at that time. From my experience, these require quite a bit of tuning of the data being fused in the sensor.
    - Each data source subsequently requires its own covariance matrix when being fused in the filter. The covariance matrix of each data source *should* be an accurate representation of the data source's accuracy, or how prone it is to noise. So, for an EKF that is fusing data from our IMU and wheel odometry, we can fuse angular velocity from both sources. However, if you have an inaccurate covariance for either sensor, the filter will oscillate and not converge on a value. It is often better to overstate a sensor's covariance than understate. For the most

part, manufacturers will provide covariance values, but our IMU did not, and the covariance was different in our environment. This resulted in a lot of hand-tuning of IMU covariances and modification to the IMU driver's source code (linked above).

- Fusing multiple sources of data also means they need to be stated in the same coordinate frame. We use 'transform broadcasters' (built-in ROS tool) to transform sensor data, in the sensor's coordinate frame, to the 'base' frame of the robot, what we call the base_link. All data must adhere to REP-103 of ROS standards for them to be used in the robot_localization EKF (z up, x forward, y left, angular velocities consistent with RHR).

- Since we are fusing multiple sources of data, one being GPS, it is important to adhere to these coordinate frame conventions & standards.
- [include tf tree]



- Map Frame: This coordinate axis is fixed to earth. It is where our global 'costmap' is held and where RTAB-map generates its map. However, the point of using the EKF is knowing the robot's current pose with respect to both where it started (where our 'odom' coordinate frame originates) and the other earth-fixed frame 'map'. Both odom and map frames are world-fixed, but they have two separate uses. The robot's pose w.r.t. the map frame should be the most accurate pose, with no drift over time. The robot's state w.r.t. the map frame can have discrete jumps over time (i.e. when corrected using SLAM or GPS), or in other

words, incorporate discontinuous data. This is typically used to correct the robot's absolute pose in the odom frame as it drifts.

- The odom frame is the source of continuous state estimate. This coordinate frame is prone to drifting over time (as it has two noisy data sources – wheel odometry and IMU), but since it is continuous, the robot's state in this frame will not jump at all. This is why it is used for all path planning and obstacle avoidance. The relationship between odom and map frames is that the robot's pose with respect to the odom frame will drift, but the robot's absolute pose will be corrected in the map frame as it receives updated state estimates. This is why in certain cases, it is useful to use 2 EKF's running in parallel, or the dual EKF method.
- Note: For sensors mounted to our robot, we use a static transform for the data to the base_link, since that transform is constant.

Dual EKF Method:

- GPS data is inherently discontinuous and will have an occasional jump when a more accurate gps fix is achieved. This creates an issue when it comes to fusing the wheel odometry, imu data, and GPS data, as the first two are continuous and the GPS is not. To accommodate, we use two EKFs running in parallel. The first instance of our EKF is for continuous data: fusing IMU and wheel odometry for state estimation with respect to the 'odom' frame (as mentioned before, this will drift over time). Alongside the first instance, we can run another EKF in parallel that estimates the robot's pose with respect to the 'map' frame. In this instance, we fuse imu, wheel odometry, and GPS data. Again, since this estimate is prone to jumps, we only use it to correct for drift, and run all instances of path planning and obstacle avoidance in the continuous 'odom' frame, and in the background, the absolute location of the robot w.r.t to the global map is being corrected by the second 'map' frame EKF instance. As good as this sounds, it complicates matters quite a bit, and the RTAB-map section below mentions the tradeoff between this and other methods. Through testing, our implementation of RTAB-map is typically good enough where the robot is localized properly in the map frame. This means we don't necessarily need the gps in our current setup, but it helps.
- Using the dual ekf method means we probably aren't using a SLAM algorithm that works well outdoors and need a better means of correcting odom drift. This method uses lower computation compared to using SLAM but is harder to integrate and not as robust.

Converting GPS data to robot frame:

- It is important to mention how we convert GPS data to the robot's base frame. Since GPS data is given in a global frame, there are a few conversions that take place, which all are done using the navsat_transform node.
  - First, the GPS data is converted to the UTM grid, or another cartesian format rather than latitude, longitude. This requires accurate heading in the UTM frame, which comes from our filtered IMU. It first receives a 'datum' that includes the first accurate gps (lat,lon) fix, and using imu heading, it sets that value to the

'zero' point for the rest of incoming GPS info. From that point on, we will have an odometry stream based on our robot's initial state.

- Between these concepts, we have tuned our system to the point where we have a pretty accurate state estimate of the robot's base. For the competition, we actually didn't use any GPS for state estimation, and only ran the odom EKF w/ SLAM, as the drift was minimal in a small map such as the one in the competition. Also, the use of RTAB-map almost removes the need for a GPS in smaller maps, as it uses landmarks and meshed RGB-D frames to localize itself in the global map.

RTAB-map, SLAM, And Obstacle Detection:

RTAB-map:

There are multiple layers in our mapping/navigation system. From the highest level, we use SLAM to build a 'global' map of the robot's surroundings and store obstacles. For this, we use Real-Time Appearance-Based Mapping (RTAB-map). RTAB-map is a unique implementation of a couple different SLAM methods.

"**RTAB-Map** is an RGB-D, Stereo, and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector. The loop closure detector uses a bag-of-words approach to determinate how likely a new image comes from a previous location or a new location."

More info on rtabmap here

This means that we can drive the robot around an area via RC to build the map and save the generated grid map to a database on our computer. Then when we run RTAB-map again, it uses the database to determine where it is located in the map (achieving loop closure). A similar and more popular approach is used by Adaptive Monte Carlo Localization (AMCL) or ICP, but these only work with LiDAR and do not work well in outdoor use (limited to 2D). There are also implementations of RTAB-map that use GPS data to assist with loop-closure on startup (in case one area has similar features as another) by adding a GPS constraint to the map. In initial testing, it was not required as we were using a smaller map, but as we test at larger scale, this will be useful. Our limitation with this right now is not having a LiDAR. The camera only gives us a 90* FOV, where if we had LiDAR data, we could have nearly 360* and expand path planning functionality & reliability.

Intel D435i RGB-D camera:

For consistent obstacle detection, we need a proper pipeline of both processing and filtering data from the RGB-D camera. Below is an outline of the methods used.

Note: In ROS, Nodes are defined simply as an executable that uses ROS for communication, or it sends information using the ROS master communication protocol to make its published information available. For large amounts of data being transported (like images), it's inefficient to have some information available to the entire system, thus we have nodelets. These allow us to run multiple nodes that communicate a lot of information in a single process and run each node

in a separate thread without using network protocol used by a typical ROS node. This improves efficiency, and it is good to use nodelets when possible. For our camera pipeline, we run all image post-processing and filters under the same nodelet as the camera to reduce latency & unnecessary communications.

Pipeline from camera -> SLAM

- RGBD_Sync nodelet used to synchronize image frames & depth camera data stream from realsense driver into a single message, improves RTAB-map performance dramatically.
- points_xyzrgb: Nodelet that converts RGB-D images into 3D pointcloud2 datatype (used for obstacle detection, also throttles data to reduce cpu usage by obstacles_detection node).
- obstacles_detection nodelet: used to differentiate ground & obstacles. The segmented 'ground' pointcloud is used to clear areas in the costmap, while the segmented 'obstacle' pointcloud is used to mark high-cost areas in the costmap. This works well with uneven ground because it allows us to set a minimum ground height, so we can ignore snow in our costmaps but still mark a noticeable obstacle & avoid rough terrain.
- RTAB-map_ros: mentioned above, subscribes to rgbd_image from synced RGBD data in RGBD_Sync nodelet. Optionally, can include GPS input and IMU input to assist with loop closure and add gravity constraints to map.

Navigation & Path-Planning

ROS comes with a built-in motion control package called move_base. Move_base uses 2D graph-based search algorithms for path planning, using 'costmaps' built from pointcloud2 and laserscan data.

Costmaps are a key component of robotic navigation that provide a representation of the environment around a robot in the form of a grid or 2D map. These maps are used by robots to plan their paths and avoid obstacles.

In costmaps, each cell in the grid is assigned a cost that represents the difficulty of traversing that area of the environment. This cost can be based on a variety of factors such as the distance to obstacles, the presence of other robots or moving objects, or the terrain type.

There are two main types of costmaps used in robotic navigation: static costmaps and dynamic costmaps. Static costmaps are pre-generated and do not change unless the environment changes significantly. Dynamic costmaps, on the other hand, are continuously updated based on the robot's sensor data and other information about the environment.

Using costmaps, a robot can plan a safe and efficient path through an environment by selecting cells with lower costs and avoiding cells with higher costs. This allows robots to navigate complex environments with obstacles, narrow passages, and other challenging terrain.

To navigate these costmaps, there are a couple different "planners" we can implement using plugins to modify the behavior of the package. These planners mostly use graph-based search algorithms. The three main local planners used are Time Elastic Band (TEB), Dynamic Window Approach (DWA), and Trajectory Rollout. From other papers, it is apparent that TEB local planners are better suited for larger robots as it includes the kinematics of the robot itself into the planning, but it comes with many different parameters to tune. It also can incorporate dynamic obstacles, so if an obstacle has a velocity in one direction, the planner will bias the opposite direction.
However, we have not tuned TEB well enough for it to suit our needs. DWA and trajectory rollout planners are both very similar as they take multiple samples of different linear and angular velocities and simulate the resulting path using the current costmaps and determine the path with the lowest cost. DWA and trajectory rollout both require less cpu usage as they are simpler.

Global planning is the simplest planner typically, as it only takes global costmaps into account and isn't dynamically updated. Once it's generated, the local planner will stick to that path unless it recognizes it can't.

There are also a few packages that implement coverage planners (cover entire area of a map based on robot (plow) width. These are typically used by cleaning robots indoors but are also useful in the case of our robot as it is clearing snow from a parking lot or sidewalk.

Improvements:

- Path planning: right now, we use the simplest form of path planning. It works well, but TEB would probably work better. There are a lot of jolts and pauses, probably due to computation or overflowing messages over TCP
- LiDAR: Including LiDAR in mapping will give us 360* FOV, instead of 90.
- Camera: Camera gets too cold and stops working when outside for too long
- Convert this entire system to ROS2: Improved performance & modernized.

Comments to add later:

- There are two ways we can go about GNC with what we have. If we want to minimize computation, we can use a pre-generated [occupancy grid] map from sidewalk blueprints rather than SLAM-generated map. We can load that using a map server and it will serve as our global ('static') costmap. This would mean SLAM is not necessary and we can use the dual EKF method to properly localize and simplify obstacle avoidance to just using the obstacle_avoidance nodelet, along with LiDAR, if we want. This has complications, as if the robot is improperly localized for whatever reason, it will have a lot of trouble correcting compared to using SLAM.
- The other way, and in my opinion the best option, is using RTAB-map along with GPS assistance, with fused continuous data from EKF. The biggest reason is not needing absolute IMU data. If we use the GPS in an EKF, we need pretty accurate magnetometer readings, which isn't necessary in the SLAM approach. Another plus of this method is it will update the map each time the robot drives through it, so if there is a new landmark,

the map will be updated permanently. If we want this to work at all times of day, we need to build the map in different lighting settings for it to work, as it will have troubles with loop-closure when there is different lighting (which is why the gps-assisted loop-closure is helpful).

- Both dual ekf and RTAB-map: I have seen an implementation of SLAM and the dual EKF, but have not figured it out yet, as using both would require two odometry frames. Clearpath Husky's package uses this, and the GPS EKF broadcasts map->odom frame, and their SLAM package broadcasts map->gmapping_odom frame. Something to look into later.

TODO for next year:

1. Convert system to ROS 2 (currently in ROS1)
    - Better performance & versatility
    - IMU and Camera already have compatible ROS2 drivers, GPS driver can easily be converted to ROS2. Motor controller driver will be the hardest driver to convert, as there is no equivalent ROS2 driver.
        - Update: I converted it to ROS2 in this repo: https://github.com/CJdev99/roboteq_ros2_driver
2. Get system to work with Microcontrollers
    - ROS allows use of multiple controllers in the system, Use the Jetson Nano for image processing & raspberry pi for other nodes to make system more reactive.
3. Figure out PID motor control with motor controller
    - Either fix the internal PID control of the motor controller or implement it in the roboteq_diff_driver package.
4. Look into SMACH python library and behavior tree c++/python libraries for more complex robot behaviors (state machines and behavior trees). Test both and determine better option.
5. Improve navigation behavior
    - Right now, it can avoid obstacles, but issues arise in tight spaces and randomly lags
    - Test different navigation algorithms and determine best implementation
6. Implement LiDAR with the rest of system using SiCK Appspace, if we don't this semester
    - The Tim881p is not made for ROS and has its own CPU. The SiCK Appspace API does not integrate well when used in a system similar to ours. Need to figure out how to use LiDAR data with the ROS system. Sick is also looking into how to do this so we might have a better idea towards end of semester.
7. Test implementations of LiDAR, RGB-D SLAM implementations, determine the best route for this project. Optimize for both reliability and computation.

- We use RTAB-map right now, which allows use of both LiDAR and Camera, but it is very computation-heavy. Research other methods that could work.

Tutorial for website:

https://husarion.com/tutorials/ros-tutorials/5-running-ros-on-multiple-machines/

https://github.com/husarion/route_admin_panel

https://github.com/zengzhen/progress-web-vis/tree/master/3d

Host private website with husarion:

https://husarnet.com/blog/host-private-website/

State Machine:

https://wiki.ros.org/move_base_flex/Tutorials/MoveBaseFlexSmachForParallelPlanning