WARP

Marc Benedí

Autonmous driving.
Everywhere.

WARP

SLAM

https://gitlab.com/paverobotics/vehicle/vehicle-core

vehicle-core/localization/warp_slam

Autonmous driving.
Everywhere.

Configuration

- A class that holds all the parameters
- **Singleton pattern**
  - Can only be accessed through the method *GetInstance()*

```
config::Config::GetInstance().submap.size += 1;
```

- Parameters can be dynamically changed using [ROS dynamic reconfigure]

```
// warp_slam/include/configuration/config.h

class Config {
  public:
    // Structs containing the parameters of each component
    Imu imu;
    // ...
    ScanMatch scanMatch;

    static Config& GetInstance(const std::string& path = "");

    // Singleton pattern restrictions
    Config(Config const&) = delete;
    void operator=(Config const&) = delete;


  private:
    Config();

};
```

Configuration

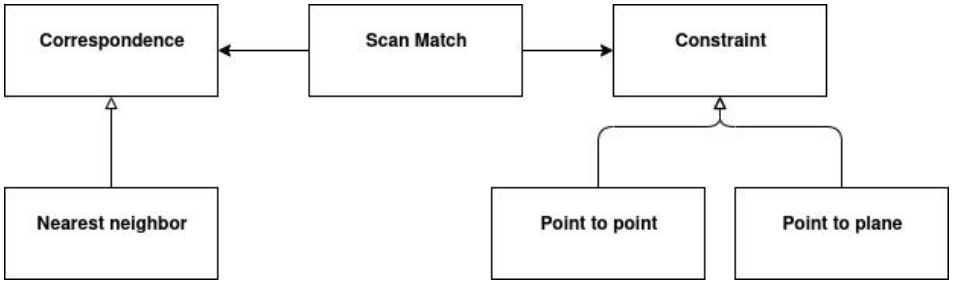
- Parameters can be restored from a **YAML** file *(example in warp_slam/test/assets)*

```cpp
Config& Config::GetInstance(const std::string& path) {
  static Config instance;

  if (!path.empty()) {
    YAML::Node config = YAML::LoadFile(path);

    // IMU config
    // ...
    instance.imu.gravity_time_constant = config["imu"]["gravity_time_constant"].as<double>();
    // ...
  }

  return instance;
}
```

```yaml
submap:
    num_scans: 30
    size: 30

# ...

submap_creator:
    max_queue_size: 0
    max_scans_in_submap: 0
```

Optimization / Scan Match

- Returns a transformation that aligns two point clouds
- Transform optimize(submap, scan)
  - Initialize a transformation as identity
  - Loop until end of optimization
    - Apply transformation to scan
    - Calculate correspondences
    - For each correspondence create a constraint (point to point)
    - Add the constraints to the optimization problem



Optimization / Correspondence

- Base class for any kind of correspondences
  - Returns the index of the corresponding point to *point* in *pointCloud*

```
class Correspondence {
  public:
    virtual int calculate(const pcl::PointXYZ & point,
      const pcl::PointCloud<pcl::PointXYZ>::Ptr pointCloud) = 0;

    std::vector<int> calculate(const pcl::PointCloud<pcl::PointXYZ>::Ptr pointCloudA,
      const pcl::PointCloud<pcl::PointXYZ>::Ptr pointCloudB);
};
```

Optimization / Constraint

- Builds a cost function for the Google Ceres Solver
- At the moment there is only one child: point_to_point.cc
- Open-closed principle: It is very easy to extend to other constraints such as point_to_plane thanks to inheritance

```
class Constraint {
  public:
    virtual ceres::CostFunction* build() = 0;
  protected:
  private:
};
```

VPN

https://www.notion.so/VPN-7a5e8696ebe44846995865c34696d12d

VPN

- Deployment of a VPN for Warp


https://hub.docker.com/r/kylemanna/openvpn/


https://github.com/kylemanna/docker-openvpn

Current configuration:

- warp-cloud-dev/vpn-server
- IP: 34.107.86.221

GPS

https://gitlab.com/paverobotics/vehicle/vehicle-core

vehicle-core/sensors/gps
deployment/pulumi/gps

GPS

https://github.com/KumarRobotics/ublox.git

- Docker image that deploys the ublox package
- Pulumi deployment to deploy the GPS into the cluster

A word of advice:

Do not try to setup a GPS indoors

(even if gps is hanging out from the window)

grid2occupancy

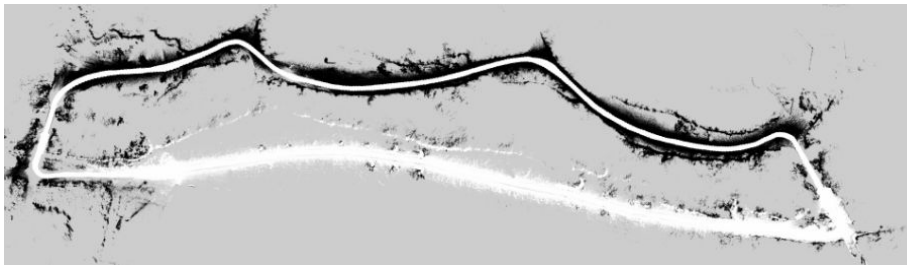
https://gitlab.com/paverobotics/cloud/pointcloud-to-grid/

pointcloud-to-grid/src/grid2occupancy

Autonmous driving.
Everywhere.

Description

- A ROS package that generates a **costmap** for the planner
- Two use cases
  - From a point cloud (PCD)
  - From a grid map
- It can use **trajectory** information to free the path
- It uses the segmentation **labels** of the points to adjust the cost
- It is also available as a Docker Image



Parameters

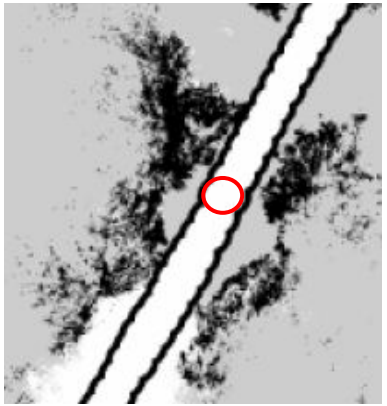
- A class that holds all parameters for the package
- Singleton pattern
- The parameters are loaded from a **YAML** file
- Full description of the parameters in *grid2occupancy/README.md*

```yaml
grid_map_pcl:
  parameters_file: .../grid_map/grid_map_pcl/config/parameters.yaml

costmap:
  cost_per_label:
    # Label -1 should not be used as it is used for unknown
    # Cost should be between 0 and 254
    -
      label: 1 # Grass
      cost: 100
    -
      label: 2 # Asphalt
      cost: 0
    -
      ...
  thresholds:
    lethal: 2
    inscribed_inflated: 1
    free: 0

labels_to_grid:
  deviation: 0.05

trajectory:
  use_trajectory: false
  csv_file: .../trajectory_filtered.csv
  radius_free: 1.0
  radius_block_max: 2.0
  radius_block_min: 1.0
```

Trajectory extraction

- A Jupyter notebook used to extract the coordinates of the trajectory
- Three steps:
  - Convert the pbstream into a bag file
  - Convert the bag file into a csv
  - Extract x,y,z fields from the csv

- A circle of radius *trajectory/radius_free* is marked as free
- The area between two circles with radius *trajectory/radius_block_min* and *trajectory_radius_block_max* is marked as blocked



pbstream_trajectories_to_rosbag_main.cc

https://github.com/AtsushiSakai/rosbag_to_csv

BaseNode

- Contains the common behaviour between the two use cases
  - Initialize ROS
  - Get a grid map
  - Adjust values of unknown cells
  - Shift all cells such that the minimum is zero
  - Free trajectory if required
  - Generate a costmap
  - Save the costmap using ROS map server

```
class BaseNode {

public:

    int main(int argc, char** argv);

protected:

    virtual gm::GridMap getGridMap() = 0;
    virtual void saveGridMap(gm::GridMap gridMap);
    std::unique_ptr<ros::NodeHandle> nh_;

};
```

pointcloud2costmap node

```
roslaunch grid2occupancy pointcloud2costmap.launch folder_path:=$(pwd) pcd_filename:=pointcloud.pcd
```

- Node that converts a PCD into a costmap
- Convert the PCD into a grid map using the **GridMapPclLabelLoader** class
- The grid map is saved into a **bag** file in *folder_path/preprocessed_gridmap.bag* in the topic */gridmap*

- Input
  - folder_path
    - parameters.yaml
  - pcd_filename
- Output
  - folder_path
    - costmap (yaml and pgm)
    - preprocessed_gridmap.bag

GridMapPclLabelLoader

- A custom class that inherits from GridMapPclLoader
- It adds support for points with the fields x,y,z,Label
- getGridMap()
  - It gets the grid map generated by the parent class (elevation)
  - It creates a new layer (labels). For each cell (x,y), it assigns the most common label of the points contained in that cell at that elevation. (± *labels_to_grid.deviation*)

```
// grid2occupancy/include/grid_map_pcl/GridMapPclLabelLoader.hpp

class GridMapPclLabelLoader: public GridMapPclLoader
{

 public:
  void loadCloudFromPcdFile(const std::string& filename);
  const grid_map::GridMap& getGridMap() const;

 private:
  pcl::PointCloud<pcl::PointXYZL>::Ptr rawLabelInputCloud_;
  void setRawLabelInputCloud(pcl::PointCloud<pcl::PointXYZL>::ConstPtr rawLabelInputCloud);
  std::unique_ptr<grid_map::GridMap> gridMapPtr_ = std::make_unique<grid_map::GridMap>();

};
```

gridmap2costmap node

```
roslaunch grid2occupancy gridmap2costmap.launch folder_path:=$(pwd) bag_filename:=preprocessed_gridmap.bag
```

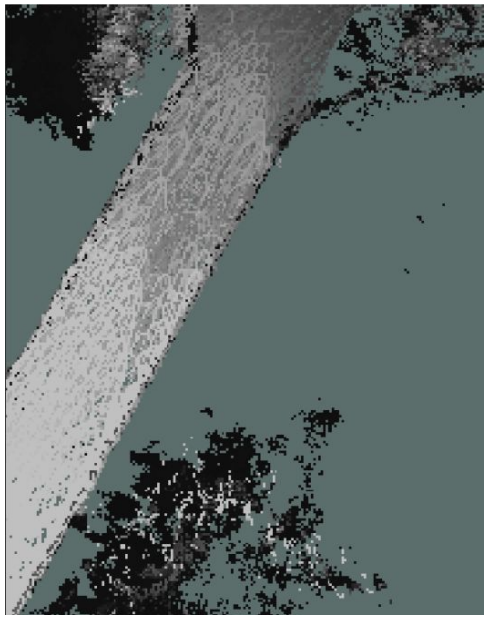
- Inherits from BaseNode
- getGridMap() loads a bag file containing the preprocessed gridmap stored by the previous node

```
class GridMap2CostmapNode: public BaseNode {
  protected:
    gm::GridMap getGridMap() {

      std::string inputBagName, folderPath;
      nh_->param<std::string>("folder_path", folderPath, "");
      nh_->param<std::string>("bag_filename", inputBagName, "bag_cloud");
      std::string pathToBag = folderPath + "/" + inputBagName;

      return g2o::loadFromBagFile(pathToBag, "/gridmap");
    }
};
```

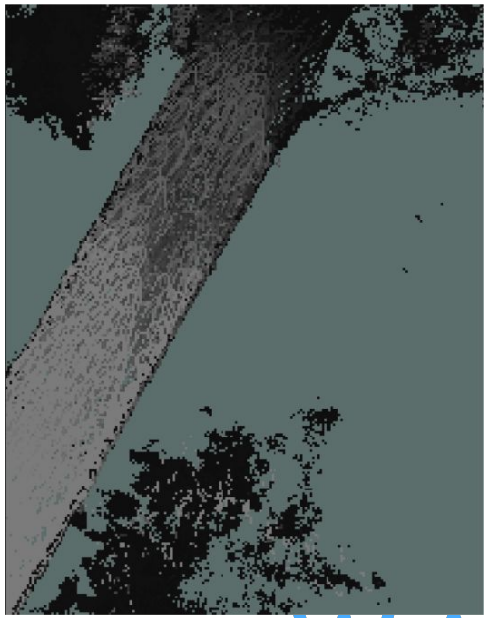
- Input
  - folder_path
    - parameters.yaml
  - bag_filename
- Output
  - folder_path
    - costmap (yaml and pgm)

Outputs with different cost per label

Cost: 0



Cost: 100



Next steps

- The conversion from **grid map** to **cost map** is done using the grid_map/grid_map_costmap2d package which uses absolute elevation to set the cost for the cells
- A better solution would be computing the normals and using them to know if the vehicle can reach that cell

WARP

Autonomous driving.
Everywhere.