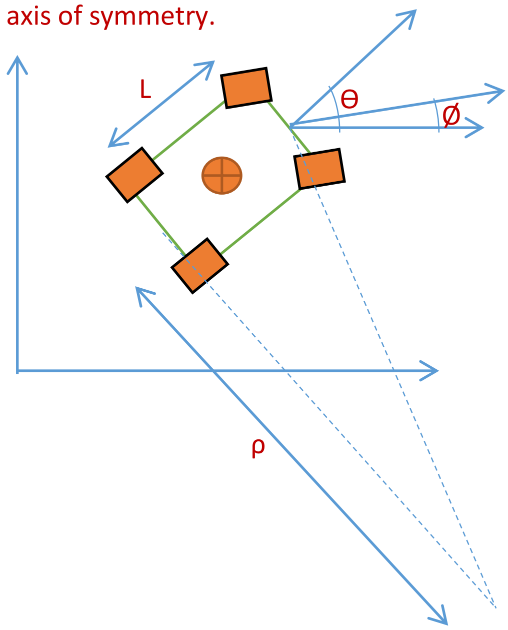
List of files attached with this answer sheet:

1	PWMOpenLoop.cpp
2	PWMClosedLoop.cpp
3	NavToGoal.cpp
4	myqbutton.h
5	myqbutton.cpp

Assessment: You have been assigned the task of developing a simple controller for a planar four-wheeled mobile robot that would enable it to autonomously navigate around its environment. In order to accomplish the forenamed objective, the following constituent sub-tasks have been assigned:

1. Derive the system's configuration kinematic model assuming that the robot's centre of mass coincides with its axis of symmetry.



Consider a 4 wheeled planar robot with Ackerman steering based geometry and COM at the geometric centre of the vehicle. For small Δt rhe robot moves approximately in the direction of its rear wheels, ie

$$\frac{\delta y}{\delta x} = tan\theta$$

$$\frac{\delta y}{\delta x} = \frac{\frac{\delta y}{\delta t}}{\frac{\delta x}{\delta t}} = \frac{\dot{y}}{\dot{x}}$$

Since $tan\theta = \frac{sin\theta}{cos\theta} = \frac{\dot{y}}{\dot{x}}$,

$$-\dot{x}sin\theta + \dot{y}cos\theta = 0$$

If ω is the distance travelled by the car with a turning radius of ρ, the dω = ρdϴ

Where $\rho = \frac{L}{tan\emptyset}$ ie, $d\theta = \frac{tan\emptyset}{L}$

Dividing both by dt,

We get $\dot{\theta} = \frac{\rho}{L} tan\emptyset$

Hence the differential equations of the motion in control input form is

$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ \theta \\ \emptyset \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} u_1 + \begin{bmatrix} cos\theta \\ sin\theta \\ \frac{tan\emptyset}{L} \\ 0 \end{bmatrix} u_2$$ where $u_1 = \dot{\emptyset}$ (steered wheel angle rate) and

$u_2 = V$ (forward velocity of the rear axle centre)

2. Formulate an expression for the wheel motor input commands, based on the configuration kinematic model that was previously computed, assuming that the robot is controlled in velocity mode.

   We have two control inputs for velocity

   $u_1 = \dot{\emptyset}$ (steered wheel angle rate) and
   $u_2 = V$ (forward velocity of the rear axle centre)

   Then the configuration transition equation will be

   $\dot{x} = u_2 \cos\theta$, $\dot{y} = u_2 \sin\theta$

   $\dot{\theta} = \frac{u_2}{L}\tan(u_1)$

   Where $\dot{x}$, $\dot{y}$ *and* $\dot{\theta}$ represent the x velocity, y velocity and the rate of change of orientation of the robot when controlled in velocity mode. The limitation is that $\emptyset_{max} < \frac{\pi}{2}$ which is usually the case for robots with ackerman steering.

3. Implement the velocity controller by means of a PWM function.

   PWM functions are generally micro controller dependent, however most of them involve a timer and a comparator that latches on to the timer for turning on for the particular duration of time.

   Sample code for open loop control >> See Attached: PWMOpenLoop.cpp

   Sample code for closed loop control >> See Attached: PWMClosedLoop.cpp

4. Provide a snippet of code enabling the use of a simple ROS-based mapping system.

   After setting the odometry nodes, the base controller node, and the PID nodes for driving, we need to configure the Navigation stack to perform SLAM .

   The gmapping node is the package to perform SLAM.

   The gmapping node inside this package mainly subscribes and publishes the following topics:

   Subscribed topics:

   • tf (tf/tfMessage): Robot transform that relates to Kinect, robot base and odometry

• scan (sensor_msgs/LaserScan): Laser scan data that is required to create the map

Published topics:

• map (nav_msgs/OccupancyGrid): Publishes the occupancy grid map data

• map_metadata (nav_msgs/MapMetaData): Basic information about the occupancy grid

The main gmapping launch file is given next.

It is placed in say mybot_bringup/launch/includes/gmapping_mybot.launch. This launch file launches the openni_launch file and the depth_to_laserscan node to convert the depth image to laser scan. After launching say Kinect nodes, it launches the gmapping node and the move_base configurations.

```
<launch>

<!-- Launches 3D sensor nodes -->

 <include file="$(find mybot_bringup)/launch/3dsensor.launch">

 <arg name="rgb_processing" value="false" />
<arg name="depth_registration" value="false" />
<arg name="depth_processing" value="false" />
 <arg name="scan_topic" value="/scan" />


</include>
<!-- Start gmapping nodes and its configurations -->
<include file="$(find mybot_bringup)/launch/includes/gmapping.launch.xml"/>
<!-- Start move_base node and its configuration -->
<include file="$(find mybot_bringup)/launch/includes/move_base.launch.xml"/>

</launch>
```

The next node we need to configure is move_base. Along with the move_base node, we need to configure the global and the local planners, and also the global and the local cost maps. We will first look at the launch file to load all these configuration files. The following launch file mybot_bringup/launch/includes/move_base.launch.xml will load all the parameters of move_base, planners, and costmaps:

```
<launch>

 <arg name="odom_topic" default="odom" />

<!-- Starting move_base node -->

 <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">

<!-- common parameters of global costmap -->

 <rosparam file="$(find mybot_bringup)/param/costmap_common_params.yaml" command="load" ns="global_costmap" />

<!-- common parameters of local costmap -->
```

```
<rosparam file="$(find mybot_bringup)/param/costmap_common_params.yaml" command="load" ns="local_costmap" />

<!-- local cost map parameters -->

<rosparam file="$(find mybot_bringup)/param/local_costmap_params.yaml" command="load" />

<!-- global cost map parameters -->

<rosparam file="$(find mybot_bringup)/param/global_costmap_params.yaml" command="load" />

<!-- dwa local planner parameters -->

<rosparam file="$(find mybot_bringup)/param/dwa_local_planner_params.yaml" command="load" />

<!-- move_base node parameters -->

<rosparam file="$(find mybot_bringup)/param/move_base_params.yaml" command="load" />

<remap from="cmd_vel" to="/cmd_vel_mux/input/navi"/>

<remap from="odom" to="$(arg odom_topic)"/>

</node>

</launch>
```

Some of the parameter files might look like this

```
costmap_common_params.yaml:

max_obstacle_height: 0.60
obstacle_range: 2.5
raytrace_range: 3.0

robot_radius: 0.45
inflation_radius: 0.50

#We can either choose map type as voxel which will give a 3D view of the world, or the other type, costmap which is a 2D view of the map. Here we are opting voxel.

map_type: voxel
#This is the z_origin of the map if it voxel
origin_z: 0.0

z_resolution: 0.2
z_voxels: 2
publish_voxel_map: false
observation_sources: scan

scan: {data_type: LaserScan, topic: scan, marking: true, clearing:true, min_obstacle_height: 0.0, max_obstacle_height: 3}
```

mybot_bringup/param/global_costmap_params.yaml might look like this

```yaml
global_costmap:

 global_frame: /map
 robot_base_frame: /base_footprint
 update_frequency: 1.0
 publish_frequency: 0.5
 static_map: true
 transform_tolerance: 0.5
```

mybot_bringup/param/local_costmap_params.yaml might look like this

```yaml
local_costmap:
 global_frame: odom
 robot_base_frame: /base_footprint
 update_frequency: 5.0
 publish_frequency: 2.0
 static_map: false
 rolling_window: true
 width: 4.0
 height: 4.0
 resolution: 0.05
 transform_tolerance: 0.5
```

Similarly in mybot_bringup/param/base_local_planner_params.yaml we set the configurations related to velocity, acceleration, and so on.

```yaml
TrajectoryPlannerROS:
max_vel_x: 0.3
 min_vel_x: 0.1
max_vel_theta: 1.0
 min_vel_theta: -1.0
 min_in_place_vel_theta: 0.6
acc_lim_x: 0.5
 acc_lim_theta: 1.0
yaw_goal_tolerance: 0.3
 xy_goal_tolerance: 0.15
sim_time: 3.0
 vx_samples: 6
 vtheta_samples: 20

# Trajectory Scoring Parameters
meter_scoring: true
 pdist_scale: 0.6
 gdist_scale: 0.8
 occdist_scale: 0.01
 heading_lookahead: 0.325
 dwa: true
```

```
#Oscillation prevention
oscillation_reset_dist: 0.05

# Differential-drive robot configuration : If the robot is holonomic configuration, set to true other vice set to false. Our robot is a non holonomic robot.
 holonomic_robot: false
 max_vel_y: 0.0
 min_vel_y: 0.0
 acc_lim_y: 0.0
 vy_samples: 1
```

The DWA planner is another local planner in ROS. Its configuration is almost the same as the base local planner we need, so we can copy it into chefbot_bringup/param/ dwa_local_planner_params.yaml.

Lastly, we configure the move_base node in move_base_node_params.yaml

```
#This parameter determine whether the cost map need to shutdown when move_base in inactive state
shutdown_costmaps: false
#The rate at which move base run the update loop and send the velocity commands
controller_frequency: 5.0
#Controller wait time for a valid command before a space-clearing operations
controller_patience: 3.0
#The rate at which the global planning loop is running, if it is 0, planner only plan when a new goal is received
planner_frequency: 1.0
#Planner wait time for finding a valid path befire the space-clearing operations
planner_patience: 5.0
#Time allowed for oscillation before starting robot recovery operations
oscillation_timeout: 10.0
#Distance that robot should move to be considered which not be oscillating. Moving above this distance will reset the oscillation_timeout
oscillation_distance: 0.2
# local planner - default is trajectory rollout
base_local_planner: "dwa_local_planner/DWAPlannerROS"
```

Now we can start running a gmapping for building the map.
Start the robot's tf nodes and base controller nodes: $ roslaunch mybot_bringup robot.launch

Start the gmapping nodes using the following command:
$ roslaunch mybot_bringup gmapping_mybot.launch

We should also have a keyboard_teleop started:
$ roslaunch mybot_bringup keyboard_teleop.launch

We can see the map building in RViz, which can be invoked using the following command:
$ roslaunch mybot_bringup view_navigation.launch

```
<launch>
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mybotbot_bringup)/rviz/navigation.rviz"/>
</launch>
```

After completing the mapping process, we can save the map using the following command: $ rosrun map_server map_saver -f /home/user/mymap

5. Demonstrate via ROS libraries and C++ code, how the outputs of the mapping system could be broadcasted and subsequently utilised to navigate the robot to desired locations in Cartesian/world coordinates.

   Suppose we have some locations in the map where we want mybot to go, then we can do it as follows

   Check out the file attached in the mail: NavToGoal.cpp

6. Append a button/checkbox to a custom ROS-based graphical user interface, whose toggling/checking activates the previously-designed controller (C++ code).

   Assume that we are already having a QT based plugin (rqt) and we need to add a button widget whose trigger launches the previous controller, we add the widget as follows and use the roslaunch through a system call to launch the node.

   Check out the files attached in the mail: myqbutton.h, myqbutton.cpp