Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

STERO

Sprawozdanie z projektu nr 2

Piotr Patek, Kacper Bielak

Warszawa, 2024

Spis treści

1. System badania odometrii robota TIAGo 2
1.1. Skrypt uruchomieniowy 2
1.2. Węzeł **rectangle_drawer** 3
1.2.1. Sterowanie - obrót i jazda do przodu 3
1.2.2. Główna pętla sterowania 4
1.2.3. Uaktualnianie położenia robota oraz obliczanie błędu 5
1.3. Testy systemu 5
2. Architektura robota-kelnera opisana z wykorzystaniem OPM 10

1. System badania odometrii robota TIAGo

W ramach projektu nr 2 należało stworzyć system do badania odometrii robota w symulowanym środowisku. W tym celu należało zaimplementować węzeł, który będzie sterował bazą robota w taki sposób, aby poruszała się po ścieżce w kształcie kwadratu. Dodatkowo system powinien przyjmować 3 argumenty: liczbę okrążeń, kierunek ruchu oraz długość boku rysowanego kwadratu. Po wykonaniu zadania system powinien generować raport zawierający chwilowe błędy średniokwadratowe oraz skumulowane błędy średniokwadratowe po każdym okrążeniu.

1.1. Skrypt uruchomieniowy

Do uruchomienia systemu przygotowano skrypt *rectangle_drawer.launch.py*, który oprócz 3 wymaganych parametrów przyjmuje parametr ścieżki do której zapisane mają być raporty wygenerowane w trakcie działania systemu. W przypadku nie podania ścieżki raport nie zostanie zapisany.

Skrypt przedstawiono na listingu poniżej:

```
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration


def generate_launch_description():
    ld = LaunchDescription()

    ld.add_action(DeclareLaunchArgument("side_length",
    description="Length of the square side drawn by TIAGo [m]",
                                        default_value='1.0', ))

    ld.add_action(DeclareLaunchArgument("loops",
    description="Number of loops that TIAGo will perform",
                                        default_value='1'))

    ld.add_action(DeclareLaunchArgument("clockwise",
    description="Direction in which the robot will be drawing the square",
    default_value="False", choices=["True", "False"]))

    ld.add_action(DeclareLaunchArgument("export_path",
    description="Path where summaries will be saved. "
    "If does not set then no summary will be saved",
    default_value=''))

    ld.add_action(
        Node(
```

```
            package='rectangle_drawer',
            executable='rectangle_drawer',
            parameters=[
                {"side_length": LaunchConfiguration("side_length"),
                 "loops": LaunchConfiguration("loops"),
                 "clockwise": LaunchConfiguration("clockwise"),
                 "export_path": LaunchConfiguration("export_path")
                 }
            ]
        )
    )

    return ld
```

Poniżej przedstawiono przykładowe uruchomienie systemu

```
ros2 launch rectangle_drawer rectangle_drawer.launch.py loops:=3 \
side_length=1.5 clockwise:=True export_path="/home"
```

1.2. Węzeł rectangle_drawer

Sygnał sterujący bazą mobilna robota generowany jest w specjalnie do tego stworzonym węźle **rectangle_drawer**. Subskrybuje on dwa tematy: */mobile_base_controller/odom* oraz */ground_truth_odom*. Podawana jest na nich odpowiednio odometria obliczona przez robota w środowisku symulowanym oraz odometria powstała na podstawie danych przekazywanych przez symulator Gazebo. Łatwo więc zauważyć, że dzięki jesteśmy w stanie sprawdzić jak bardzo nasz system lokalizacji za pomocą informacji wewnętrznych robota myli się względem położenia rzeczywistego i właśnie to zagadnienie było celem niniejszego projektu. Oprócz subskrypcji węzeł publikuje także sygnał sterujący dla bazy mobilnej na temat */cmd_vel*.

1.2.1. Sterowanie - obrót i jazda do przodu

Tak samo jak w przypadku problemu na zajęciach laboratoryjnych nr 4, możliwe było wydzielenie fragmentów kodu, które są powtarzalne i właśnie tak postąpiono zgodnie z zasadą DRY. Elementami tymi był manewr obrotu do zadanego kąta yaw definiowanego jako obrót wokół osi OZ układu bazowego robota oraz jazda do przodu o określoną odległość x.

Obrót do zadanego kąta yaw

W celu obrotu do zadanego kąta yaw zaimplementowano prosty regulator dwustanowy z histerezą. W pętli sterowania sprawdzana była aktualna orientacja robota, uaktualniana cały czas w callback'u do tematu */mobile_base_controller/odom*, i na jej podstawie wyliczany był uchyb, który normalizowany był do przedziału $[-180, 180]$. Jeżeli jego wartość bezwzględna była mniejsza od zadanej tolerancji to system uznawał, że wartość zadana została osiągnięta i przerywał pętlę sterowania, zwracając wykonanie do pętli głównej programu. W innym przypadku zadawana była prędkość kątowa w takim kierunku, aby zamiana kąta była jak mniejsza.

Kod regulatora przedstawiono na listingu poniżej;

```
...
    def _rotate_angle(self, deg):
        msg = Twist()

```

```
        while True:
            error = deg - self._angle
            if error > 180:
                error -= 360
            elif error < -180:
                error += 360

            if abs(error) < self._tolerance:
                self._publisher.publish(Twist())
                break

            msg.angular.z = self._rotation_speed if error > 0 else -self._rotation_speed
            self._publisher.publish(msg)
            time.sleep(0.01)
        self._publisher.publish(Twist())
...
```

Jazda do przodu o zadaną odległość x

W przypadku jazdy o zadaną odległość do przodu zadanie było jeszcze prostsze. Aktualne położenie robota tak samo jak orientacja uaktualniane są w odpowiednim callback'u i zapisywane są do zmiennych *self._x* oraz *self._y*.

Następnie w momencie wywołania funkcji *_drive_length* system zapisuje położenie robota na starcie wykonania ruchu i zadaje liniową prędkość dla bazy robota w osi OX. Prędkość ta utrzymywana jest tak długo jak spełniany jest warunek:

$$\sqrt{(x_s - x)^2 + (y_s - y)^2} < \text{meters}$$

1.2.2. Główna pętla sterowania

Cały program wykonania ruchu po ścieżce w kształcie kwadratu został zawarty w metodzie run, którą przedstawiono poniżej:

```
...
    def run(self):
        self._start_clock = self.get_clock().now().nanoseconds
        if self._clockwise:
            self._rotate_angle(90)
        else:
            self._rotate_angle(180)
        time.sleep(1)

        for k in range(self._loops):
            for i in range(1, 5):
                self._drive_length(self._side_length)

                if self._clockwise:
                    rotation_angle = 90 + 90 * i
                    if rotation_angle > 360:
                        rotation_angle -= 360
                    self._rotate_angle(rotation_angle)
                else:
```

```
20                      rotation_angle = 180 - 90 * i
21                      if rotation_angle < 0:
22                          rotation_angle += 360
23                      self._rotate_angle(rotation_angle)
24
25              self.get_logger().info(f"Loop {k+1} completed!")
26              self._calculate_loop_cumulative_error()
27
28          self.get_logger().info("Finished")
29          self.get_logger().info(self.generate_report())
30  ...
```

1.2.3. Uaktualnianie położenia robota oraz obliczanie błędu

Tak jak wspomniano pozycja robota była cały czas uaktualniana w programie w funkcji callback dołączonej do tematu */mobile_base_controller/odom*. Drugim zadaniem jakie było realizowane przez wspomnianą funkcję było obliczanie chwilowych błędów kwadratowych zarówno dla pozycji jak i orientacji robota. Chwilowy błąd kwadratowy orientacji zdefiniowany został jako:

$$(\text{yaw} - \text{yaw}_{\text{true}})^2$$

Natomiast błąd położenia obliczany był w następujący sposób:

$$(x - x_{\text{true}})^2 + (y - y_{\text{true}})^2$$

W tym momencie warto wspomnieć o tym czym są zmienne z tekstem *true* w indeksie dolnym. Pochodzą one z tematu */groud_truth_odom* i przypisywane są w odpowiedniej metodzie. Błędy kwadratowe położenia i orientacji obliczane były z częstotliwością 50 Hz, ponieważ była to częstotliwość z jaką kontroler bazy mobilnej publikuje odometrię.

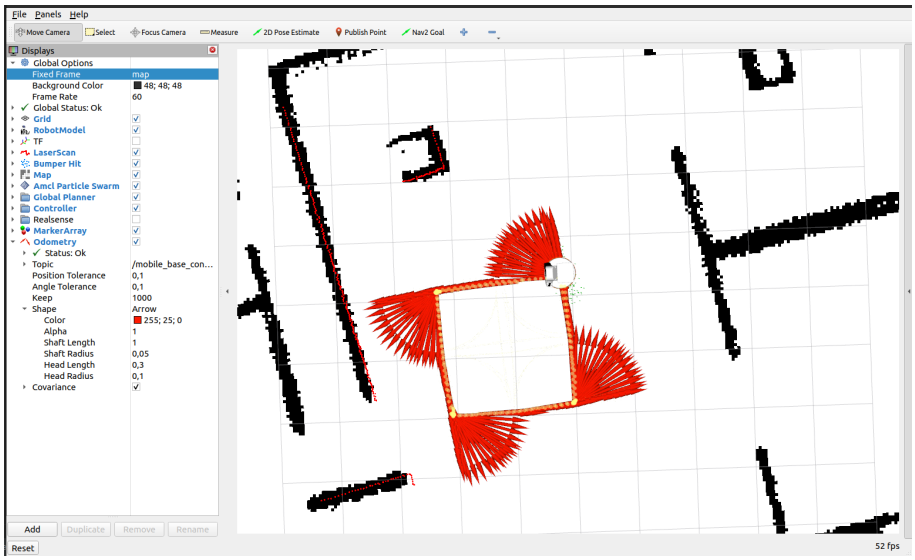
Chwilowy błąd średniokwadratowy oraz skumulowany błąd średniokwadratowy po okrążeniu

Zgodnie z poleceniem system generował chwilowe błędy średniokwadratowe, zarówno dla położenia jak i orientacji. Do tego celu stworzono specjalny timer, do którego dołączono metodę wywoływaną z częstotliwością 0,1 Hz. W niej uśredniany był błąd kwadratowy dla ostatnich 100 pomiarów, a następnie wynik ten zapisywany był do listy chwilowych błędów średniokwadratowych, które wyświetlane są po zakończeniu wykonania programu.

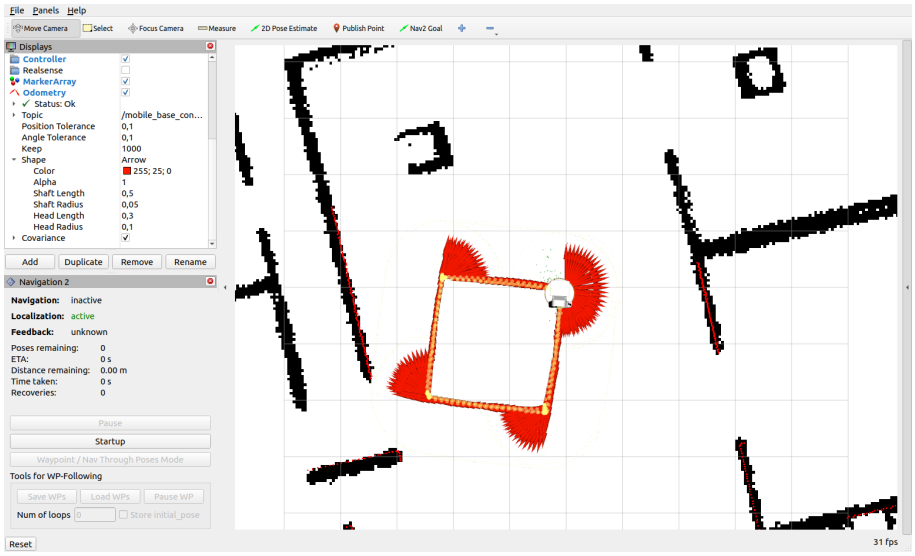
Obliczane błędy średniokwadratowe w trakcie jednego okrążenia były kumulowane tak i zapisywane do osobnej listy.

1.3. Testy systemu

Stworzony system został poddany testom w środowisku symulacyjnym. Najpierw narysowano kwadrat poruszając się w kierunku zgodnym z ruchem wskazówek zegara, a następnie przetestowano ruch w kierunku przeciwnym. Ostatecznie do wygenerowania dokładniejszych i bardziej reprezentatywnych danych dot. błędu uruchomiono system z parametrem okrążeń ustawionym na wartość 25.



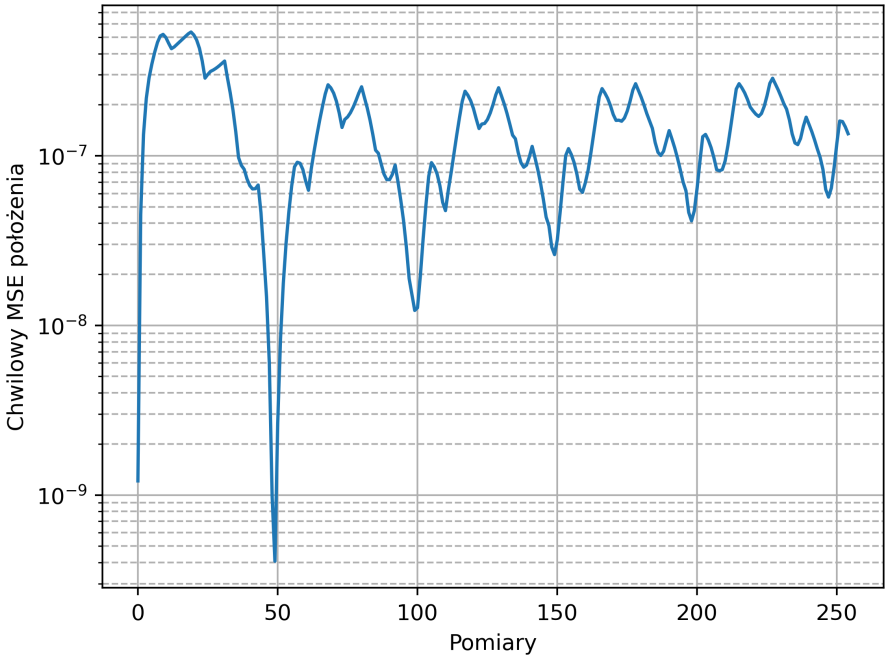
Rys. 1.1. Kwadrat narysowany przez robota TIAGo ruszając się zgodnie z ruchem wskazówek zegara



Rys. 1.2. Kwadrat narysowany przez robota TIAGo ruszając się przeciwnie do ruchu wskazówek zegara

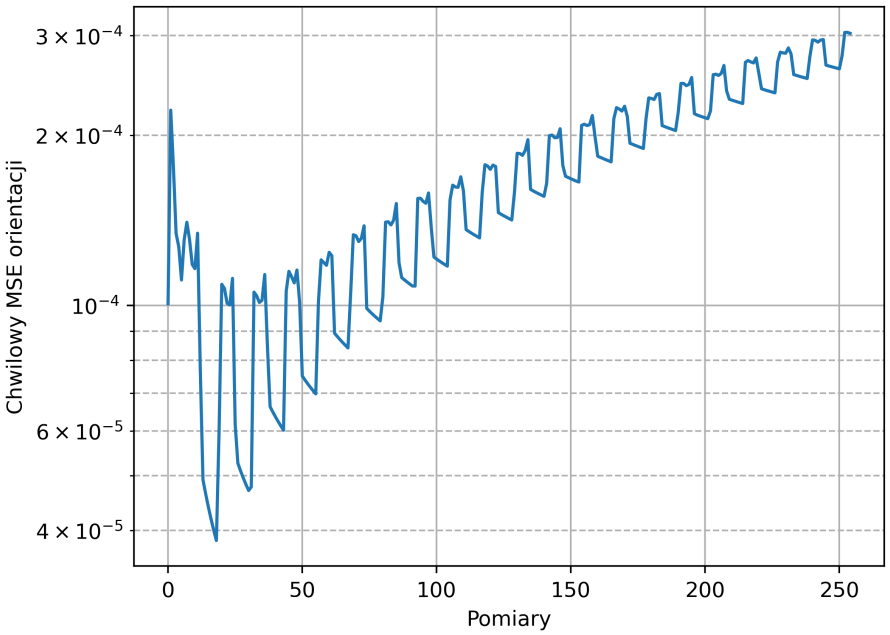
```
[rectangle_drawer-1] [INFO] [1734539261.946351386] [RectangleDrawer]:
[rectangle_drawer-1] ================================================
[rectangle_drawer-1] Temporary errors
[rectangle_drawer-1] ================================================
[rectangle_drawer-1] Second     Position        Orientation
[rectangle_drawer-1] ---------  --------------  ----------------
[rectangle_drawer-1] 28.00     0.00000594      2.03408904e-03
[rectangle_drawer-1] 48.00     0.00002372      4.01394532e-03
[rectangle_drawer-1] 68.00     0.00002650      6.82952151e-03
[rectangle_drawer-1] 88.00     0.00000839      1.10176593e-02
[rectangle_drawer-1] 108.00    0.00000762      1.56467327e-02
[rectangle_drawer-1] 128.00    0.00003520      2.20289930e-02
[rectangle_drawer-1] 148.00    0.00004676      2.84527833e-02
[rectangle_drawer-1] 168.00    0.00002139      3.69442110e-02
[rectangle_drawer-1] 188.00    0.00001696      4.53968071e-02
[rectangle_drawer-1] 208.00    0.00005777      5.60443120e-02
[rectangle_drawer-1] 228.00    0.00007734      6.66158770e-02
[rectangle_drawer-1] 248.00    0.00004141      7.95481508e-02
[rectangle_drawer-1]
[rectangle_drawer-1] ================================================
[rectangle_drawer-1] Cumulative errors
[rectangle_drawer-1] ================================================
[rectangle_drawer-1] Loop   Position        Orientation
[rectangle_drawer-1] -----  --------------  ----------------
[rectangle_drawer-1] 1      0.00006456      0.02389522
[rectangle_drawer-1] 2      0.00011096      0.10307272
[rectangle_drawer-1] 3      0.00019348      0.24760515
[rectangle_drawer-1]
[INFO] [rectangle_drawer-1]: process has finished cleanly [pid 14354]
vistek528@VisteKPC:~/STERO$
```

Rys. 1.3. Raport wygenerowany przez system dla 3 okrążeń



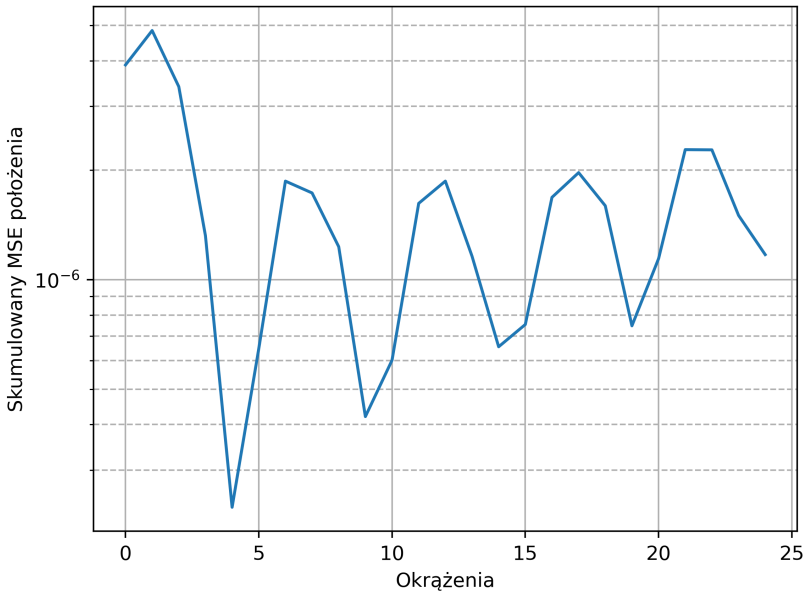
Rys. 1.4. Wykres chwilowych błędów średniokwadratowych położenia dla 25 okrążeń

Jak można zauważyć na rys. 1.4 wraz z czasem błąd średniokwadratowy położenia powoli wzrasta. Widoczne jest także, że wraz z rozpoczęciem ruchu robota błąd wzrasta, aby następnie lekko spaść w momencie zatrzymania robota na czas obrotu.



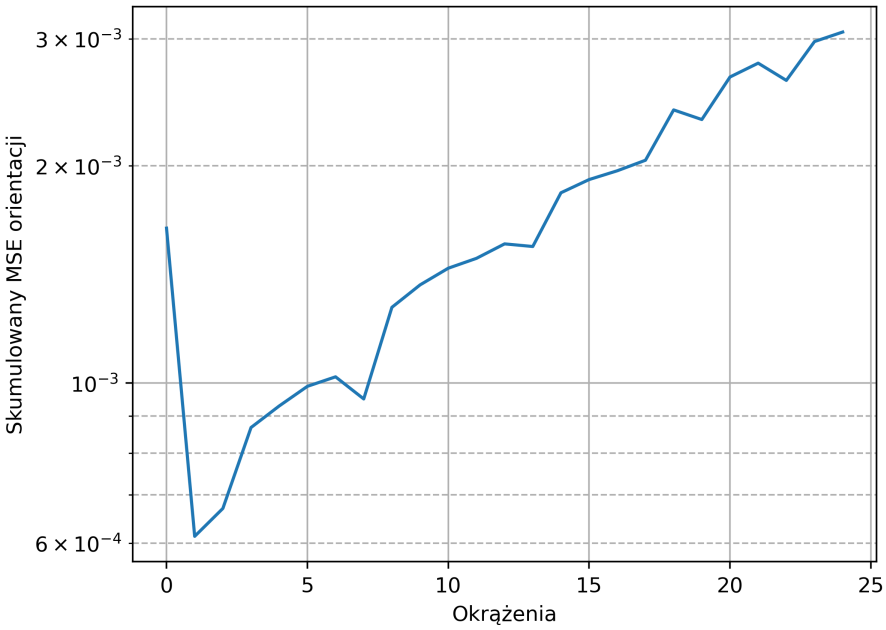
Rys. 1.5. Wykres chwilowych błędów średniokwadratowych orientacji dla 25 okrążeń

Wzrost błędu orientacji jest jeszcze bardziej zauważalny od błędu położenia, co widoczne jest na wykresie 1.5.



Rys. 1.6. Wykres skumulowanych błędów średniokwadratowych położenia dla 25 okrążeń

Skumulowany błąd średniokwadratowy położenia jeszcze wyraźniej pokazuje, że wraz z kolejnymi okrążeniami odometria generowana przez robota coraz bardziej różni się od rzeczywistego położenia robota w środowisku symulacyjnym.

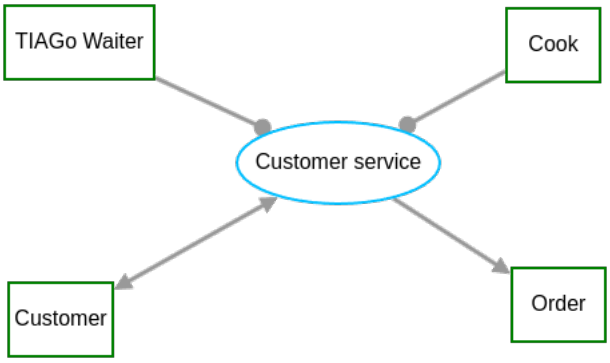


Rys. 1.7. Wykres skumulowanych błędów średniokwadratowych orientacji dla 25 okrążeń

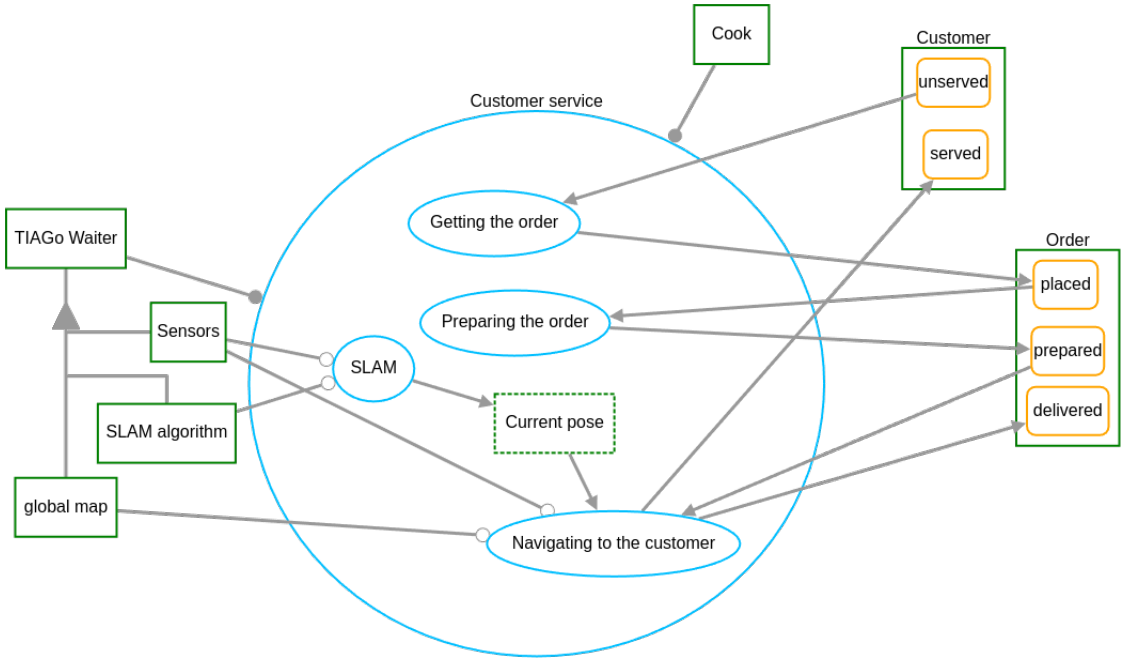
Podobnie wygląda sytuacja w przypadku skumulowanego błędu orientacji, co można zauważyć na rys. 1.7.

2. Architektura robota-kelnera opisana z wykorzystaniem OPM

Zgodnie z poleceniem w ramach projektu nr 2 stworzono diagramy OPD opisujące system robota-kelnera pracującego w restauracji. Na pierwszym diagramie przestawiono główny proces realizowany przez system, czyli obsługę klienta. W procesie tym biorą udział dwaj agenci - robot TIAGo w roli kelnera oraz kucharz. Dodatkowo, co logiczne, proces wpływa na klienta i generuje zamówienie.



Rys. 2.1. Diagram OPD reprezentujący system robota-kelnera w restauracji



Rys. 2.2. Szczegółowy diagram OPD reprezentujący system robota-kelnera w restauracji z elementami nawigacji

Na szczegółowym diagramie z głównego diagramu wydzielono podprocesy, a także zaznaczo-

no stany obiektów i część elementów składowych robota TIAGo wykorzystywanych w nawigacji. Najpierw robot przyjmuje zamówienie, przez co tworzony jest obiekt zamówienia z zainicjalizowanym pierwszym stanem - "złożone". Następnie zamówienie przygotowywane jest przez kucharza. Po tym etapie wykonywany jest proces nawigacji. Do niego wykorzystywana jest obecna pozycja robota, generowana na bieżąco przez proces SLAM'u. Do tego procesu wykorzystywane są odpowiednie algorytmy SLAM'u oraz czujniki. Na sam koniec, po dostarczeniu zamówienia, jego stan zmieniany jest na dostarczony, a stan klienta zmieniany jest na obsłużony.