



Testprotokoll

Testprotokoll der für die selbstgeschriebenen
Funktionen angefertigten Unit-Tests

Inhalt

1	Unit-Tests	1
1.1	Calculate_segments	1
1.1.1	Testfall 1	1
1.1.2	Testfall 2	2
1.1.3	Testfall 3	2
1.1.4	Ergebnis.....	3
1.2	Calculate_classification.....	3
1.2.1	Testfall 1	4
1.2.2	Testfall 2	5
1.2.3	Testfall 3	6
1.2.4	Ergebnis.....	6
1.3	Publish_marker.....	6
1.3.1	Testfall 1	6
1.3.2	Testfall 2	7
1.3.3	Testfall 3	7
1.3.4	Ergebnis.....	7

1 Unit-Tests

In dieser Arbeit selbstgeschriebene Quelltexte sollen anhand von drei Unit-Tests pro Funktion überprüft werden. Dafür werden die Funktionen *calculate_segments*, *calculate_classification* und *public_marker* überprüft. Für jede der Funktionen wurden sich somit drei jeweils sinnvolle Test-Fälle ausgedacht und überprüft. Zu beachten ist, dass die Unit-Tests aufgrund der selbstgestellten message-types im Workspace ausgeführt werden müssen. Die durchgeführten Unit-Test werden hier kurz erläutert.

1.1 Calculate_segments

Die Funktion *calculate_segments* filtert aus dem eingehenden *LaserScan* die einzelnen Segmente, welche dann im späteren Verlauf klassifiziert werden sollen. Ausgegeben werden die Daten als selbsterstellter message-type *segmentation_msg*.

```
segmentation_laserscan/segmentation_msg.msgs
uint8 amount_segments      #Anzahl der gefundenen Segmente
float32 angle_increment    #Winkelschritte zwischen den einzelnen Messpunkten [rad]
float32[] ranges           #Liste der gemessenen und verarbeiteten Abstände [m]
int32[] start_points       #Liste der Startpunkt der Segmente
int32[] stop_points        #Liste der Stopppunkte der Segmente
```

1.1.1 Testfall 1

Als erster Testfall werden in der *ranges* Liste zehn Messpunkte mit dem gleichen Abstand zum Sensor übergeben, zwischen denen jeweils ein Winkelschritt von 36° angenommen wird. Als Ausgabe wird erwartet, dass keine Segmente gefunden werden. Somit wird die in der *segmentation_msg* veröffentlichten *ranges* Liste auch gleich der eingegangenen Liste sein. Dadurch, dass keine Segmente gefunden wurden, werden auch die Listen *start_points* und *stop_points* leer bleiben.

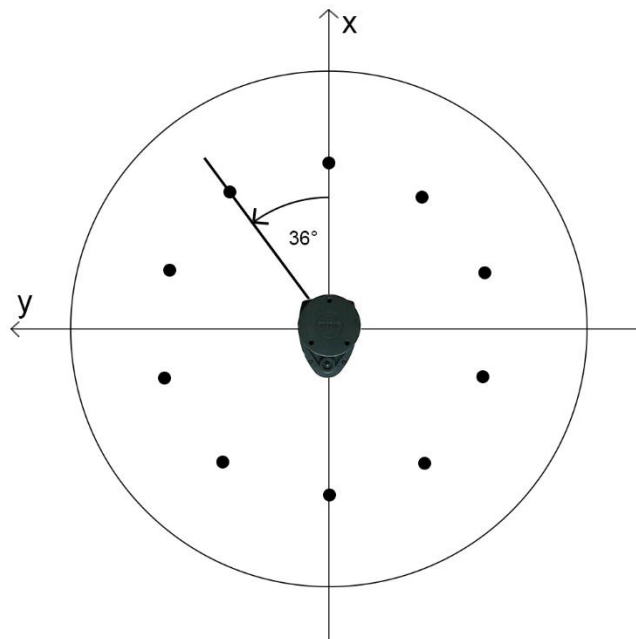


Abbildung 1: Testfall 1 calculate_segments

1.1.2 Testfall 2

Als zweiten Testfall wird angenommen, dass zwei Segmente gemessen werden. Die einzelnen Segmente sind dabei hintereinander platziert. Das erste Segment besteht aus zehn Messpunkten, welche alle in zwei Metern Entfernung gemessen wurden. Das zweite Segment besteht wiederum aus zehn Messpunkten, welche aber in einem Meter Entfernung gemessen wurden. Der Winkelschritt zwischen den einzelnen Punkten beträgt 1° .

Es wird erwartet, dass beide Segmente von der Funktion erkannt werden. Ebenfalls werden die Start- und Stopppunkte der einzelnen Segmente richtig übergeben. Die *ranges* Liste wird dabei nicht verändert.

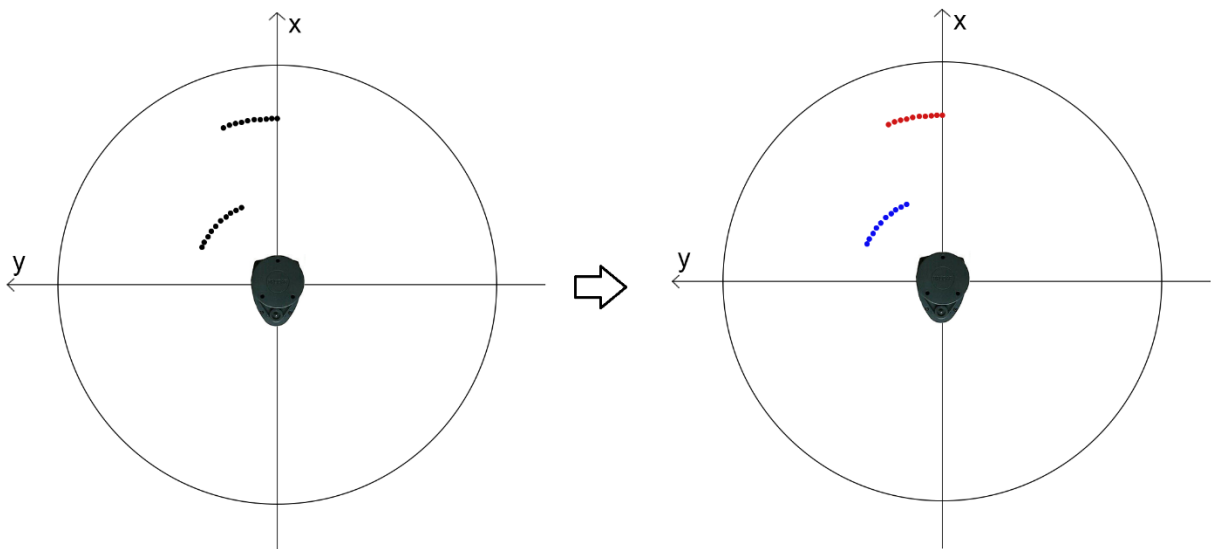


Abbildung 2: Testfall 2 calculate_segments

1.1.3 Testfall 3

Als dritter Testfall wird angenommen, dass ein Objekt gemessen wird, welches nicht durchgängig ist, wie zum Beispiel ein Gartenzaun. Die Messpunkte des Objekts befinden sich dabei in einem Abstand von zwei Meter. Die dahinterliegenden Messpunkte haben einen gemessenen Abstand von fünf Meter. Es wird außerdem ein Winkelschritt von 1° zwischen den Messpunkten angenommen.

Somit soll die dargestellte Punktwolke als erwartetes Ergebnis zu einem Segment zusammengefasst werden und die dazwischenliegenden Punkte neu berechnet werden. Es soll ein Segment erkannt werden. Ebenfalls werden die Start- und Stopppunkte des Segments richtig übergeben. Die *ranges* Liste wird dabei mit den neu berechneten Punkten ausgegeben.

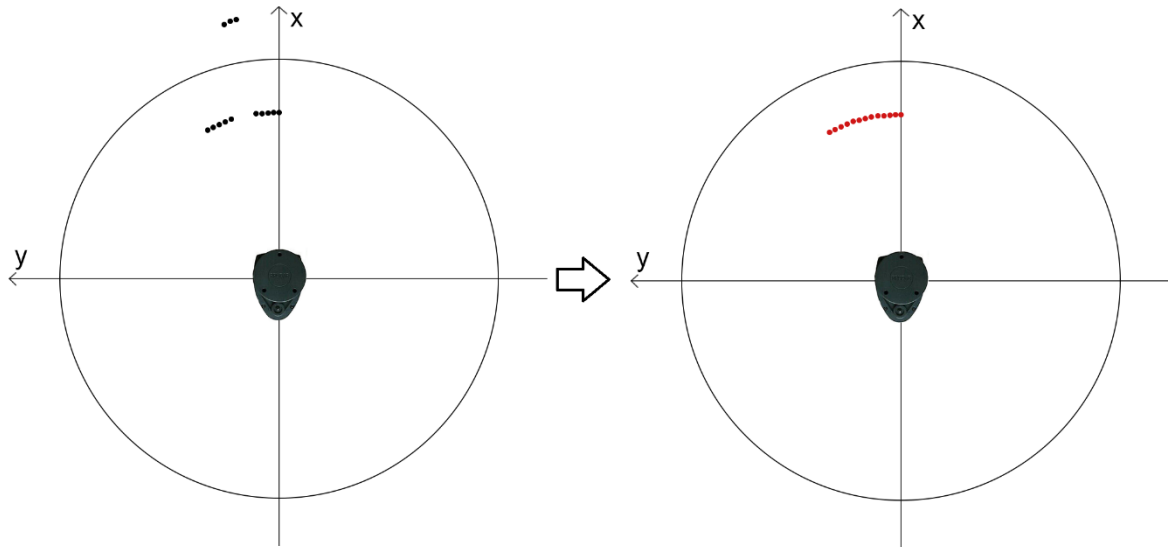


Abbildung 3: Testfall 3 calculate_segments

1.1.4 Ergebnis

Alle Testfälle konnten durch die Funktion *calculate_segments* wie erwartet berechnet werden.

```
tim@tim-VirtualBox:~/objectsegmentation_ws/src/segmentation_laserscan/unit_test$
python unit_test_segmentation.py
...
-----
Ran 3 tests in 0.002s
OK
```

Abbildung 4: Erfolgreiche Unit-Tests der Funktion *calculate_segments*

1.2 Calculate_classification

Die Funktion *calculate_classification* klassifiziert die vorher berechneten Segmente als rundes, rechteckig oder lineares Objekt. Anders geformte Segmente werden nicht ausgegeben. Ausgegeben werden die Koordinaten als selbsterstellter message-type *objects_msg*, welche zur Beschreibung der Position, Maße und Orientierung der einzelnen Objekte benötigt werden.

classification_segments/objects_msg.msgs

float32[] lines_start_x

#x-Koordinate der Linien-Startpunkte

float32[] lines_start_y

#y-Koordinate der Linien-Startpunkte

float32[] lines_stop_x

#x-Koordinate der Linien-Stopppunkte

float32[] lines_stop_y

#y-Koordinate der Linien-Stopppunkte

float32[] lines_closest_point_x

#x-Koordinate des Linien-Punktes mit dem geringsten Abstand zum Sensor

float32[] lines_closest_point_y

#y-Koordinate des Linien-Punktes mit dem geringsten Abstand zum Sensor

float32[] lines_middle_point_x

#x-Koordinate des Linien-Mittelpunktes

float32[] lines_middle_point_y

#y-Koordinate des Linien-Mittelpunktes

float32[] circles_center_x

#x-Koordinate der Kreismittelpunkte

float32[] circles_center_y

#y-Koordinate der Kreismittelpunkte

float32[] circles_radius

#Kreisradien

<code>float32[] rectangles_corner_start_x</code>	#x-Koordinate der Rechteck-Ecken, welche durch die Segment-Startpunkte entstehen
<code>float32[] rectangles_corner_start_y</code>	#y-Koordinate der Rechteck-Ecken, welche durch die Segment-Startpunkte entstehen
<code>float32[] rectangles_corner_max_distance_x</code>	#x-Koordinate der Rechteck-Ecken, welche durch den jeweiligen Segmentpunkt mit dem größten Abstand zum Start-Stop Vektor entstehen.
<code>float32[] rectangles_corner_max_distance_y</code>	#y-Koordinate der Rechteck-Ecken, welche durch den jeweiligen Segmentpunkt mit dem größten Abstand zum Start-Stop Vektor entstehen.
<code>float32[] rectangles_corner_stop_x</code>	#x-Koordinate der Rechteck-Ecken, welche durch die Segment-Stoppunkte entstehen
<code>float32[] rectangles_corner_stop_y</code>	#x-Koordinate der Rechteck-Ecken, welche durch die Segment-Stoppunkte entstehen
<code>float32[] rectangles_corner_max_distance_mirror_x</code>	#x-Koordinate der Rechteck-Ecken, welche durch die Spiegelung der „max_distance_points“ über den Start-Stop Vektor entstehen.
<code>float32[] rectangles_corner_max_distance_mirror_y</code>	#y-Koordinate der Rechteck-Ecken, welche durch die Spiegelung der „max_distance_points“ über den Start-Stop Vektor entstehen.
<code>float32[] rectangles_center_x</code>	#x-Koordinate der Rechteck Mittelpunkte
<code>float32[] rectangles_center_y</code>	#y-Koordinate der Rechteck Mittelpunkte

1.2.1 Testfall 1

Als erster Testfall wird ein Segment übergeben, dessen Messpunkte die Form eines Kreises beschreiben. Der Winkelschritt wird dabei mit 1° angenommen. Die auszugebene Position sowie der Radius wurden vorher mit dem Taschenrechner ausgerechnet. Aufgrund von Rundungsfehlern wird angenommen, dass die Differenz zwischen den selbstberechneten und von der Funktion ausgegebenen Werte nicht größer als 0,001 ist.

Somit wird erwartet, dass das Segment als Kreis erkannt wird und die Werte *circles_center_x*, *circles_center_y* und *circles_radius* richtig übergeben werden.

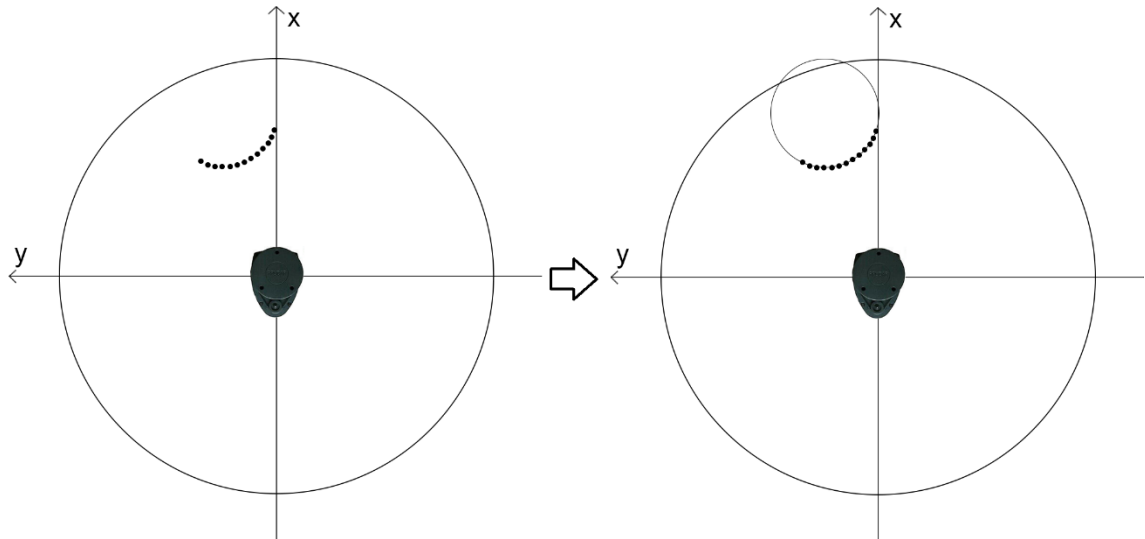


Abbildung 5: Testfall 1 calculate_classification

1.2.2 Testfall 2

Als zweiter Testfall wird ein Segment übergeben, dessen Messpunkte die Form eines Rechtecks beschreiben. Der Winkelschritt wird dabei mit 1° angenommen. Ebenfalls wurden die zu übergebenen Koordinaten vorher mit dem Taschenrechner ausgerechnet. Aufgrund von Rundungsfehlern wird angenommen, dass die Differenz zwischen den selbstberechneten und von der Funktion ausgegebenen Werte nicht größer als 0,001 ist.

Somit wird erwartet, dass das Segment als Rechteck klassifiziert wird und die Koordinaten der Rechteck-Ecken und des Rechteck-Mittelpunktes richtig übergeben werden.

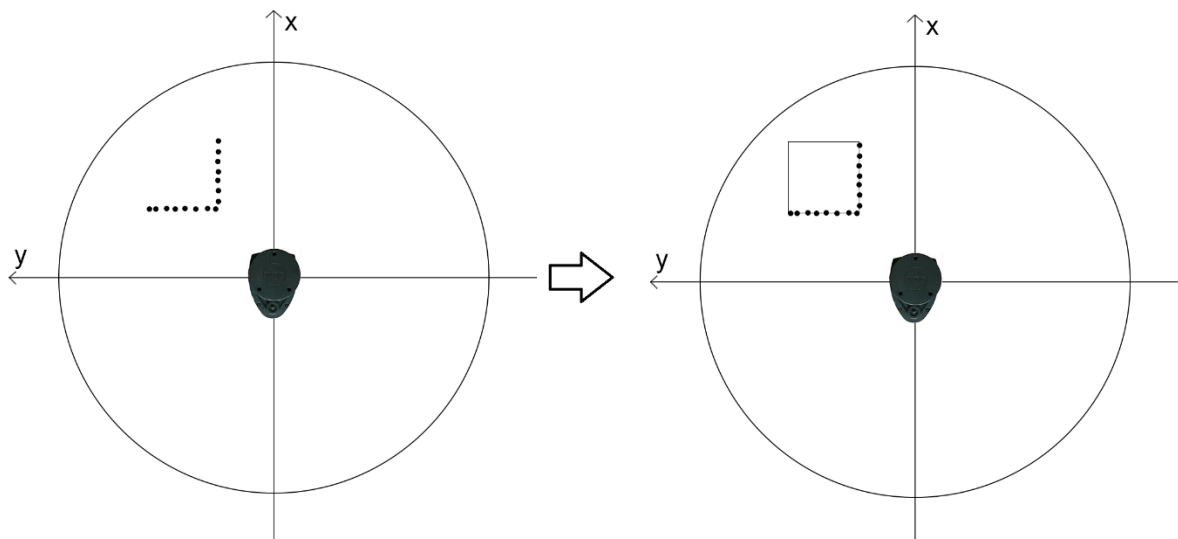


Abbildung 6: Testfall 2 calculate_classification

1.2.3 Testfall 3

Als dritter Testfall wird ein Segment übergeben, dessen Messpunkte die Form einer Linie beschreiben. Der Winkelschritt wird dabei mit 1° angenommen. Ebenfalls wurden die zu übergebenen Koordinaten vorher mit dem Taschenrechner ausgerechnet. Aufgrund von Rundungsfehlern wird angenommen, dass die Differenz zwischen den selbstberechneten und von der Funktion ausgegebenen Werte nicht größer als 0,001 ist.

Somit wird erwartet, dass das Segment als Linie klassifiziert wird und die Koordinaten der Linien Start- und Stopppunkte, sowie des Linienmittelpunktes und die des Linienpunktes mit der kürzesten Entfernung zum Sensor richtig übergeben werden.

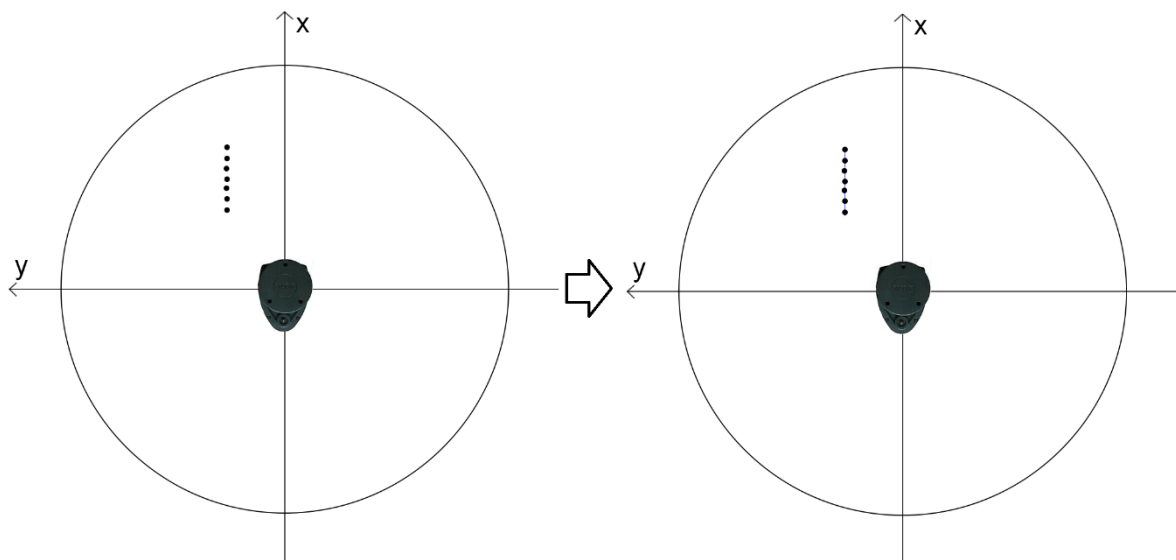


Abbildung 7: Testfall 3 calculate_classification

1.2.4 Ergebnis

Alle Testfälle konnten durch die Funktion *calculate_classification* wie erwartet berechnet werden.

```
tim@tim-VirtualBox:~/objectsegmentation_ws/src/classification_segments/unit_test
$ python unit_test_classification.py
...
-----
Ran 3 tests in 0.002s
OK
```

Abbildung 8: Erfolgreiche Unit-Tests der Funktion calculate_classification

1.3 Publish_marker

Die Funktion *publish_marker* veröffentlicht die vorher klassifizierten Objekte als ROS Datentyp *Marker*. Dabei berechnet sie aus den abonnierten Koordinaten Objektparameter wie Länge, Breite und Orientierung der verschiedenen Objekte.

1.3.1 Testfall 1

Als erster Testfall wurden die vorher ausgerechneten Koordinaten des Kreismittelpunktes und der Kreisradius übergeben.

Als Ergebnis wird erwartet, dass der Kreis als zylindrischer Marker ausgegeben wird.

1.3.2 Testfall 2

Als zweiter Testfall wurden die vorher ausgerechneten Koordinaten des Rechteckmittelpunktes und der Rechteck- Ecken übergeben. Die erwarteten Kantenlängen sowie der Orientierungswinkel wurden mit dem Taschenrechner ausgerechnet. Aufgrund von Rundungsfehlern wird angenommen, dass die Differenz zwischen den selbstberechneten und von der Funktion ausgegebenen Werte nicht größer als 0,001 ist.

Als Ergebnis wird erwartet, dass das Rechteck als quaderförmiger Marker ausgegeben wird.

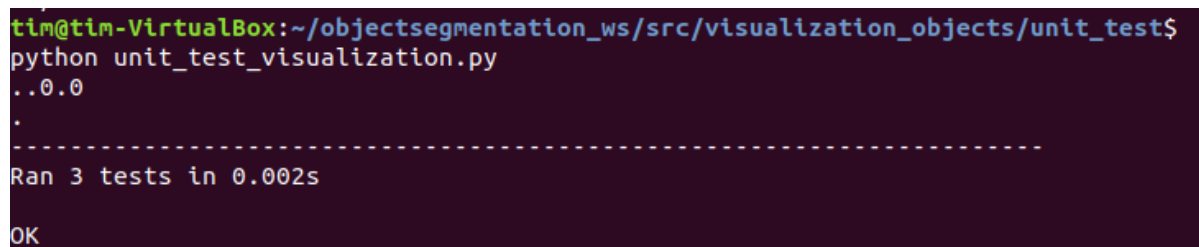
1.3.3 Testfall 3

Als dritter Testfall wurden die vorher ausgerechneten der Linien Start- und Stopppunkte, sowie des Linienmittelpunktes und die des Linienpunktes mit der kürzesten Entfernung zum Sensor übergeben. Die erwartete Linienlänge sowie der Orientierungswinkel wurden mit dem Taschenrechner ausgerechnet. Aufgrund von Rundungsfehlern wird angenommen, dass die Differenz zwischen den selbstberechneten und von der Funktion ausgegebenen Werte nicht größer als 0,001 ist.

Als Ergebnis wird erwartet, dass die Linie als sehr schmaler quaderförmiger Marker ausgegeben wird.

1.3.4 Ergebnis

Alle Testfälle konnten durch die Funktion *publish_marker* wie erwartet berechnet werden.



```
tim@tim-VirtualBox:~/objectsegmentation_ws/src/visualization_objects/unit_test$
python unit_test_visualization.py
..0.0
.
-----
Ran 3 tests in 0.002s

OK
```

Abbildung 9: Erfolgreiche Unit-Tests der Funktion *publish_marker*