

فاز اول: محاسبه $c, m, h, \text{smothed_h}$

در این بخش ابتدا باید مقادیر c و m را محاسبه کنیم. توابع مشخص شده در شکل های زیر این وظیفه را دارند. همچنین به دلیل اینکه ماکسیمم رنج دید سنسور لیدار 3.5 متر است مقادیر a و b به ترتیب 0.875 و 0.25 میباشد.

```
335 while not rospy.is_shutdown():
336     ## calculating polar obstacle density (c,m,h,h')
337     c = self.vision()
338     m = self.magnitude_calculator(c)
339     h = self.pod(m)
340     smothed_h = self.smothed_pod(h)
```

کد بالا ترتیب صدا شدن متد های مربوط به محاسبه c تا smothed_h را در لوپ اصلی کد نشان میدهد.

```
67 def vision(self):
68     laser_data = rospy.wait_for_message("/scan", LaserScan)
69     laser_data = laser_data.ranges[0:359]
70     return laser_data
71
```

```
72 ## m = (ci,j)^2 * (a - b*di,j)
73 def magnitude_calculator(self, c):
74     m = [None] * len(c)
75     for degree, dist in enumerate(c):
76         value = 0
77         if dist != inf:
78             value = self.a - (self.b * dist)
79         m[degree] = value
80     return m
81
```

دو تابع مشخص شده در شکل های بالا مقدار m را محاسبه میکنند. در صورتی که مقدار سنسور برای یک درجه inf بود برای آن مقدار 0 و در غیر این صورت طبق فرمول آمد در خط ۷۸ کد مقدار آن مشخص میشود.

```

82     ## polar obstacle density --- h_k
83     ## h_k = sigma(m)
84     ## k is number of sectors and is equal to degrees/number of degrees in each sector --> 360/5 = 72
85     def pod(self, m):
86         h = [0] * self.k
87         i = 0
88         for degree, value in enumerate(m):
89             if int(degree / self.alpha) != i:
90                 i += 1
91             h[i] += value
92         return h

```

تابع مشخص شده در شکل بالا مقدار polar obstacle density یا h را طبق فرمول زیر محاسبه میکند

$$h_k = \sum_{i,j} m_{i,j}$$

```

94     ## smoothed h
95     def smoothed_pod(self, h):
96         smoothed_h = [None] * self.k
97         for i in range(self.k):
98             if i == 0:
99                 smoothed_h[i] = (h[i - self.l] + 2 * h[i - self.l + 1] + 2 * h[i] + 2 * h[i + self.l - 1] + h[i + 1]) / (
100                     2 * self.l + 1)
101             elif i == 71:
102                 smoothed_h[i] = (h[i - self.l] + 2 * h[i - self.l + 1] + 2 * h[i] + 2 * h[i + 1]) / (2 * self.l + 1)
103             else:
104                 smoothed_h[i] = (h[i - self.l] + 2 * h[i - self.l + 1] + 2 * h[i] + 2 * h[i + self.l - 1] + h[
105                     i + self.l]) / (2 * self.l + 1)
106         return smoothed_h

```

در نهایت پس از محاسبه h یا pod مقادیر آن را طبق فرمول زیر نرمال میکنیم. تابع شکل بالا این کار را انجام میدهد.

$$h'_k = \frac{h_{k-l} + 2h_{k-l+1} + \dots + lh_k + \dots + 2h_{k+l-1} + h_{k+l}}{2l+1}$$

پس از محاسبات گفته شده در بالا به پلات کردن هیستوگرام های h و $smoothed_h$ در شرایط مختلف میپردازیم. از این کار برای بهتر مشخص کردن مقدار threshold برای پیدا کردن قعر و قله ها استفاده میکنیم.

توابع زیر برای پلات کردن هیستوگرام دایره ای و خطی استفاده شدند.

```

1089 def bar_plot(self, data):
1090     data = np.array(data)
1091     index2 = np.array([i for i in range(72)])
1092
1093     df = pd.DataFrame(list(zip(data, index2)), columns=['data', 'index'])
1094
1095     mask1 = data < self.threshold
1096     mask2 = data >= self.threshold
1097
1098     plt.bar(index2[mask1], data[mask1], color='green')
1099     plt.bar(index2[mask2], data[mask2], color='red')
1100     plt.show()

```

```

121 def bar_plot_circular(self, data):
122     data = np.array(data)
123     index = np.array([i for i in range(72)])
124
125     df = pd.DataFrame(
126         {
127             'index': index,
128             'data': data
129         })
130
131     plt.figure(figsize=(20, 10))
132     # plot polar axis
133     ax = plt.subplot(111, polar=True)
134     # remove grid
135     plt.axis('off')
136     upperLimit = max(data)
137     lowerLimit = 0
138     _max = df['data'].max()
139     slope = (_max - lowerLimit) / _max
140     heights = slope * df.data + lowerLimit
141     # Compute the width of each bar. In total we have  $2\pi = 360^\circ$ 
142     width = 2 * np.pi / len(df.index)
143     # Compute the angle each bar is centered on:
144     indexes = list(range(1, len(df.index) + 1))
145     angles = [element * width for element in indexes]
146
147     bars = ax.bar(
148         x=angles,
149         height=heights,

```

```

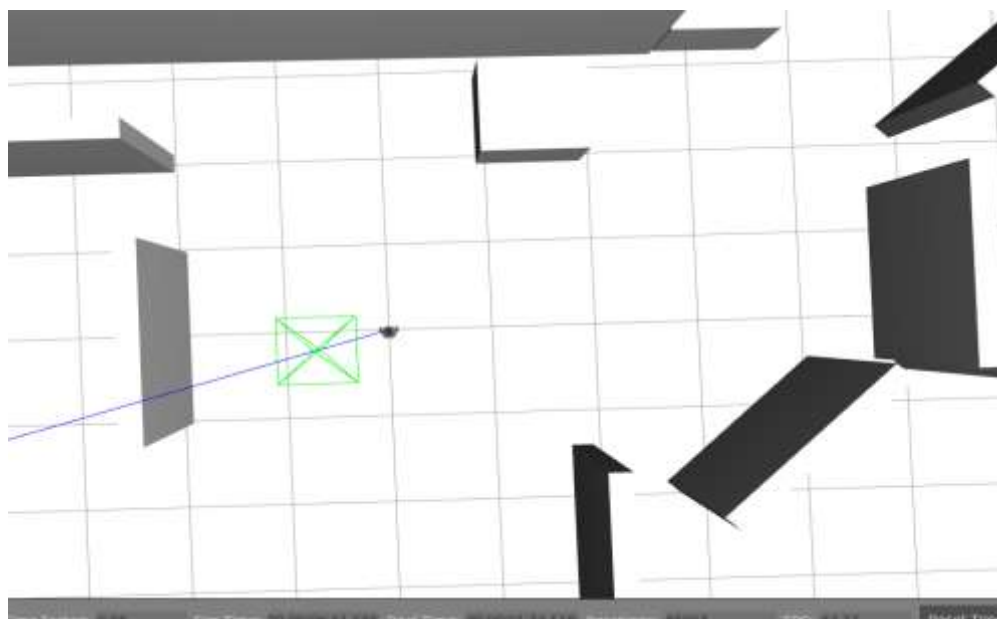
150         width=width,
151         bottom=lowerLimit,
152         linewidth=2,
153         edgecolor="white")
154     # Add labels
155     for bar, angle, height, label in zip(bars, angles, heights, df["index"]):
156
157         # Labels are rotated. Rotation must be specified in degrees :(
158         rotation = np.rad2deg(angle)
159
160         # Flip some labels upside down
161         alignment = ""
162         if angle >= np.pi / 2 and angle < 3 * np.pi / 2:
163             alignment = "right"
164             rotation = rotation + 180
165         else:
166             alignment = "left"
167
168         # Finally add the labels
169         ax.text(
170             x=angle,
171             y=lowerLimit + bar.get_height(),
172             s=label,
173             ha=alignment,
174             va="center",
175             rotation=rotation,
176             rotation_mode="anchor")
177     plt.show()

```

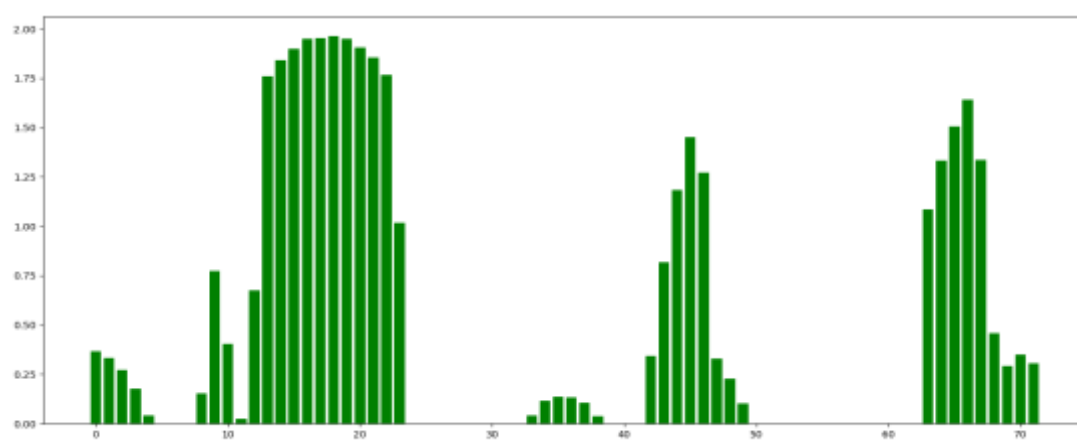
خروجی پلات های توابع بالا را در شکل های زیر میتوانید مشاهده نمایید.

۱. مقدار $\text{threshold} = 2$

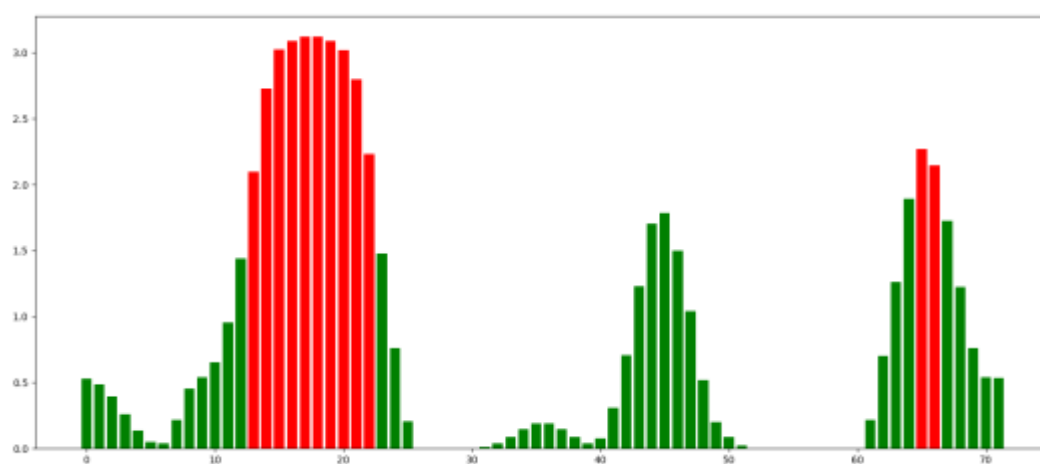
• ربات در نقطه $(0,0)$



موقعیت ربات در شبیه ساز

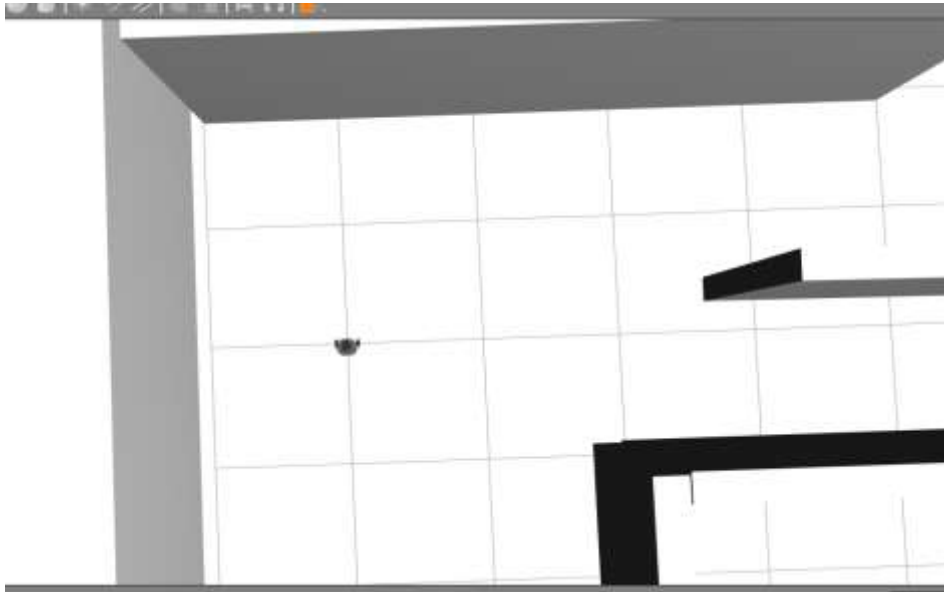


پلات h

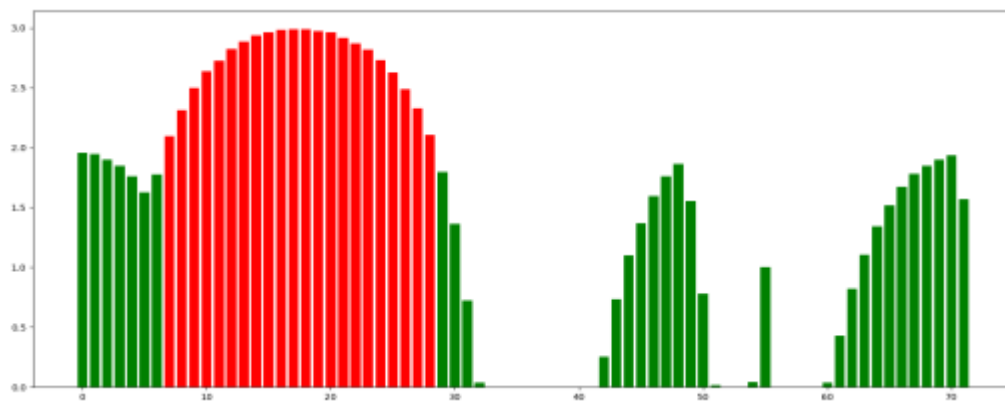


پلات $smothed_h$

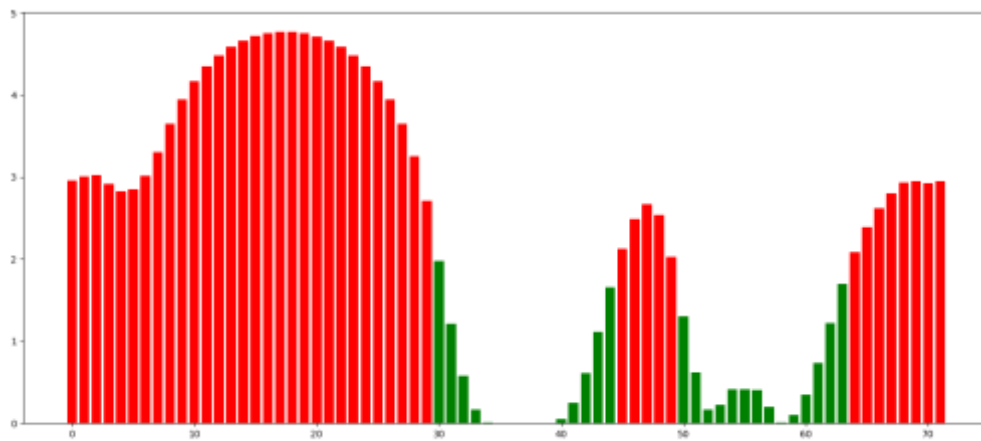
- ربات در نقطه $(-9,3)$



موقعیت ربات در شبیه ساز

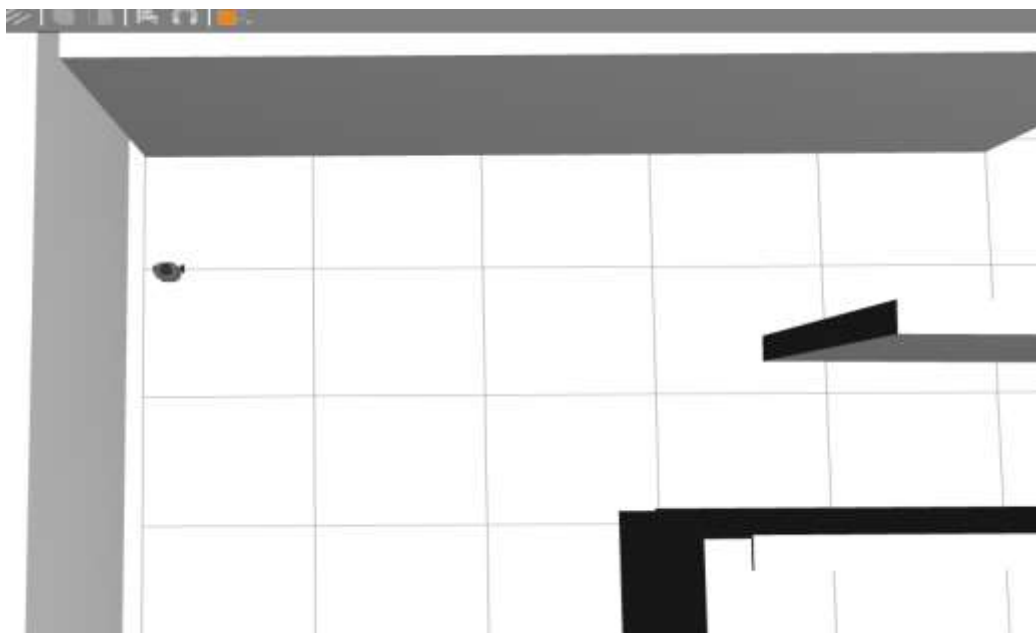


پلات h

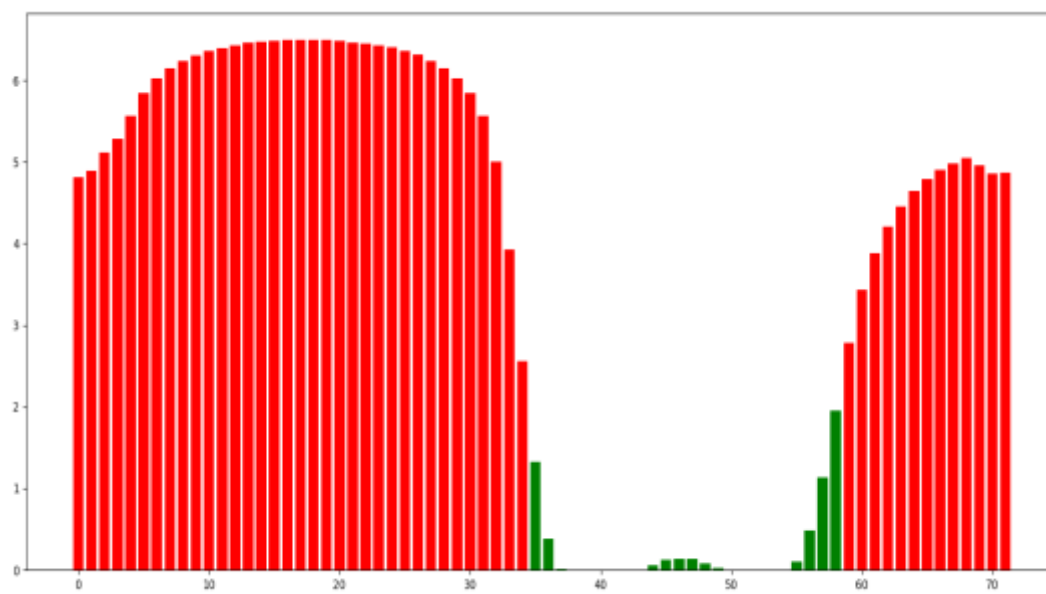


پلات h smoothed

- ربات در نقطه $(-9.85, 4)$



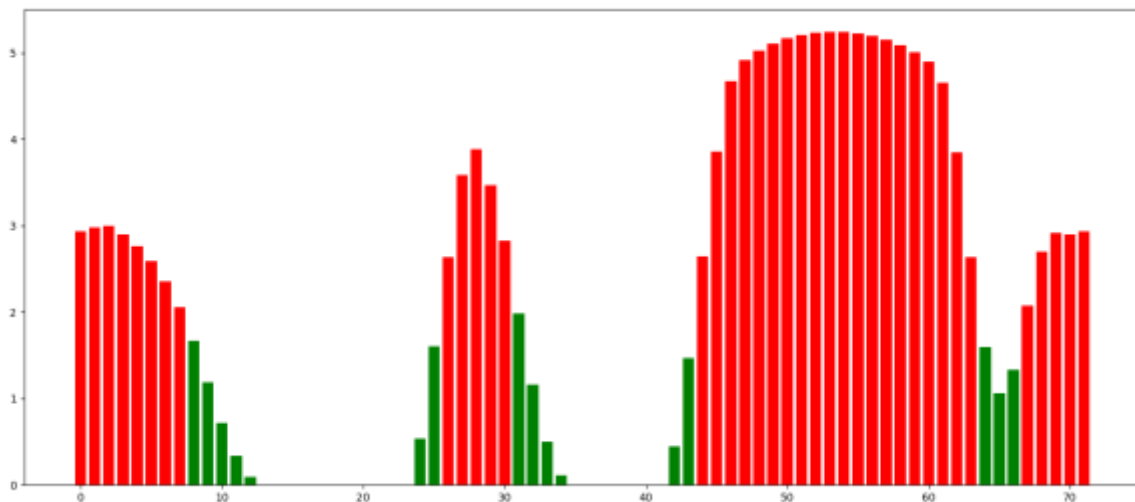
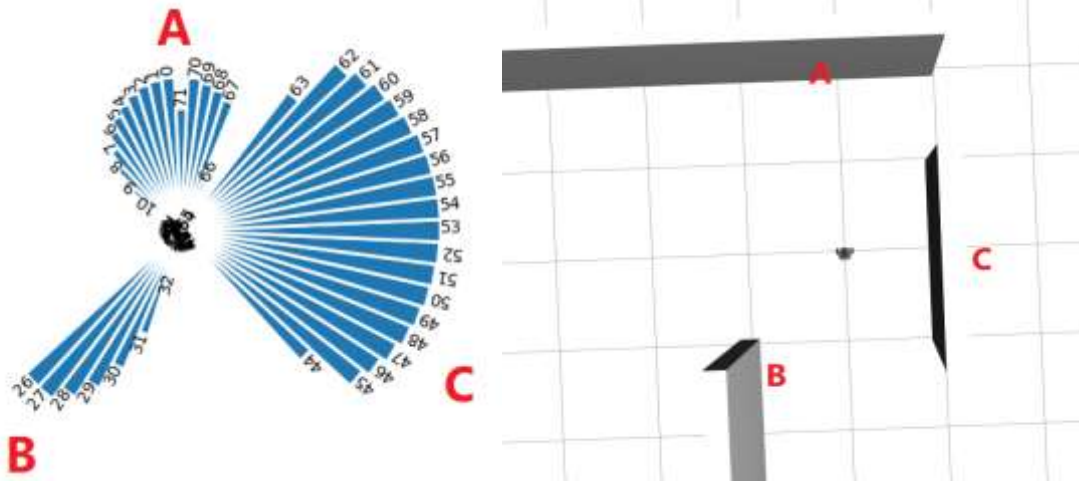
موقعیت ربات درون شبیه ساز



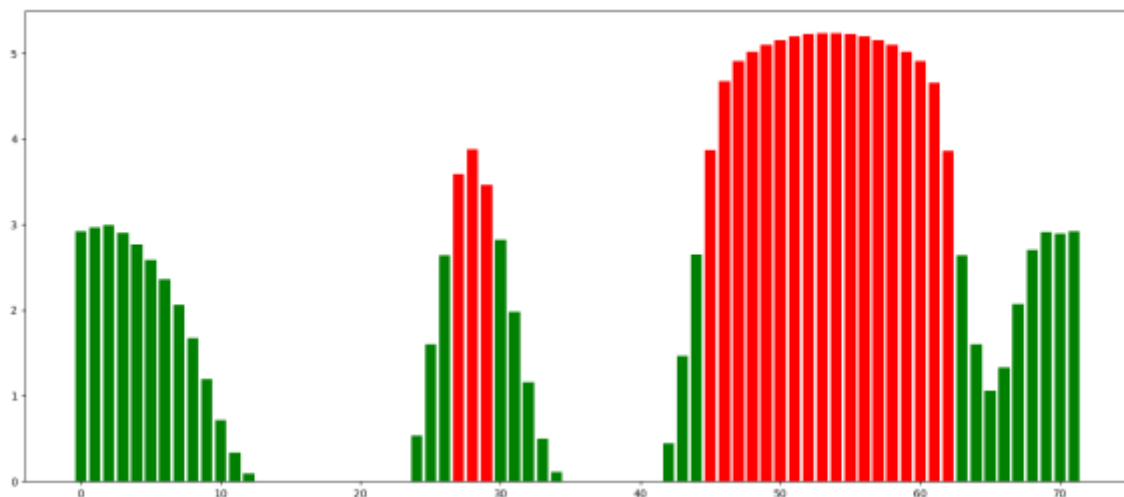
پلات smoothed pid

- ربات در نقطه $(-3,0)$

برای این نقطه پلات دایره ایی رسم شده در شکل زیر را مشاهده کنید. همانطور که مشخص است در اطراف ربات سه مانع A,B و C وجود دارد. پلات های دایره ایی و صاف از `smothed_h` آمده در تصاویر زیر نشان میدهد که توابعی که نوشته شده و در بالا توضیح داده شده است به درستی کار میکنند و میتوانند موانع را تشخیص دهند.



پس از پلات کردن مقادیر در موقعیت های متفاوت نقشه نتیجه گرفته شد که مقدار `threshold` برابر ۳ از باقی مقادیر مناسب تر عمل میکند. برای موقعیت بالا پلات `smothed h` را میتوانید در شکل زیر با `threshold = 3` مشاهده کنید.



فاز دوم: محاسبه زاویه حرکت یا θ

در این فاز از نتایج به دست آمده در فاز قبلی استفاده میکنیم و مقادیر قعر و قله را تشخیص میدهیم. سپس مقدار θ که بهترین زاویه برای حرکت کردن ربات است را به دست میآوریم.

برای به دست آوردن θ دو حالت وجود دارد. یا زاویه ایی که هدف با ربات دارد درون یکی از قعر ها که در پلات های بالا با رنگ سبز نشان داده شده است قرار گرفته که در این صورت θ برابر با اختلاف زاویه ربات با خود هدف میشود.

حالت دوم این است که زاویه ایی که هدف با ربات دارد درون یکی از قعر ها قرار ندارد. در این صورت باید نزدیک ترین قعر به هدف را پیدا کرده و سپس طبق فرمول $kn+kf / 2$ مقدار θ را محاسبه کنیم.

```

344     ## finding robot heading(in lidar sensor order) towards the target
345     angular_deffrences_with_target = self.target_angular_difference()
346     target_heading = self.heading_to_target(angular_deffrences_with_target)
347     target_sector = self.get_target_sector(target_heading)
348
349     ## finding theta using heading and smothed_h
350     valleys = self.valley_finder(smothed_h)
351     print(valleys)
352     theta_sector = self.find_theta(valleys, target_sector)
353     print("heading sector ", theta_sector)
354     theta = self.get_trigonometric_circle_theta(theta_sector)
355     print("heading theta ", theta)

```

شکل بالا ترتیب صدا شدن توابع مربوط به فاز دوم در لوپ اصلی کد را نشان میدهد. در سه تابع اول ابتدا سکتوری که هدف درون آن قرار گرفته شده است محاسبه میشود. سپس با استفاده از تابع valley_finder که در شکل زیر آمده است دره هارا با استفاده از نتایج فاز اول محاسبه میکنیم.

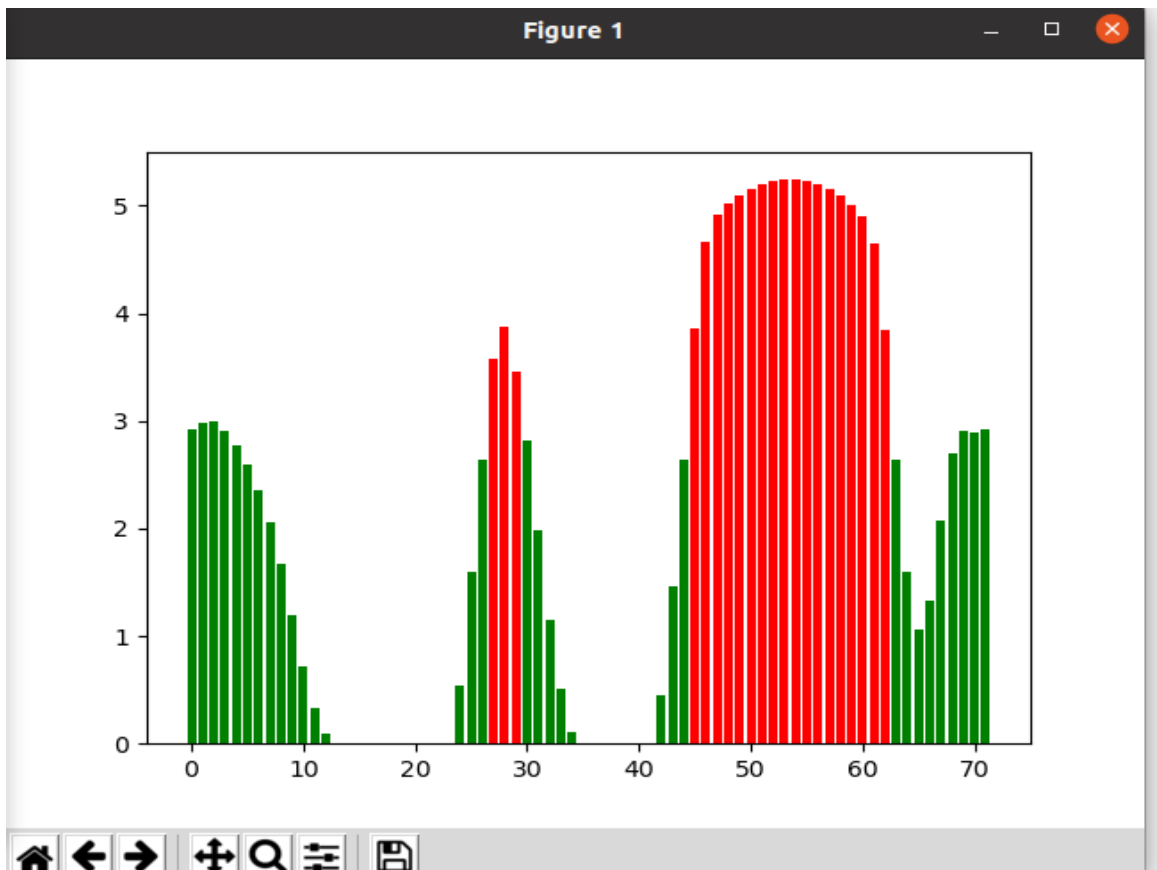
```

218     ## this method finds valleys according to VFH algorithm
219     ## returns a list of valleys with their sector numbers
220     def valley_finder(self, smothed_h):
221         rows, cols = (72, 2)
222         arr = [[0 for i in range(cols)] for j in range(rows)]
223         for i, v in enumerate(smothed_h):
224             arr[i][0] = i
225             arr[i][1] = v
226
227         valleys = []
228         flag = False
229         counter = 0
230         while True:
231             new_valley = []
232             if arr[counter][1] < self.threshold:
233                 new_valley.append(arr[counter])
234                 for j in range(counter + 1, len(arr)):
235                     if arr[j][1] < self.threshold:
236                         new_valley.append(arr[j])
237                     else:
238                         flag = True
239                         break
240             flag = True
241             if flag is True:
242                 counter = new_valley[-1][0] + 1
243                 valleys.append(new_valley)
244                 flag = False
245
246             else:
247                 counter += 1
248                 if counter == len(arr):
249                     break
250
251         valleys_sector_numbers = []
252         for valley in valleys:
253             new = []
254             for v in valley:
255                 new.append(v[0])
256             valleys_sector_numbers.append(new)
257
258         return valleys_sector_numbers
259
260     def circular_dist(self, lenght_of_list, idx_1, idx_2):
261         i = (idx_1 - idx_2) % lenght_of_list
262         j = (idx_2 - idx_1) % lenght_of_list
263         return min(i, j)

```

خروجی این تابع لیستی از دره ها با شماره سکتور هایشان است. نمونه خروجی در شکل زیر آمده است.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]
[63, 64, 65, 66, 67, 68, 69, 70, 71]
```



دو تابع بعدی که در خط های ۳۵۲ و ۳۵۴ صدا شده اند با استفاده از شماره سکتور هدف و لیست دره ها مقدار θ را بر حسب زاویه دایره مثلثاتی محاسبه میکنند.

فاز سوم: حرکت دادن ربات

در این فاز با استفاده از نتیجه فاز دوم یا همان θ ربات را به سمت هدف با استفاده از کد زیر حرکت میدهم. ربات ابتدا به سمت هدف پرخیده و سپس به سمت آن حرکت میکند.

```

307     def move_to_target(self, theta):
308         speed = Twist()
309         distance_error = self.euclidean_distance((self.x, self.y), (self.target.x, self.target.y))
310         if distance_error > 0.08:
311             if abs(theta - self.yaw) > 0.15:
312                 print("rotating")
313                 speed.linear.x = 0
314                 speed.angular.z = -0.25
315             else:
316                 print("moving to point")
317                 speed.linear.x = 0.2
318                 speed.angular.z = 0
319         else:
320             speed.linear.x = 0
321             speed.angular.z = 0
322
323         return speed
324
325     def run(self):
326         rospy.sleep(2)
327         twist = Twist()

```