به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

Advanced Robotics

پروژه ۲

مکان‌یابی ربات

نام و نام خانوادگی	مهیار ملکی فاطمه زهرا علی‌نژاد حسین آزاد ملکی
شماره دانشجویی	۸۱۰۱۰۰۴۷۶ ۸۱۰۱۰۰۴۲۰ ۸۱۰۱۰۰۲۸۵

فهرست

مقدمه

در این پروژه قصد داریم با استفاده از مدل خطای سنسوری و حرکتی ربات که در محیط Gazebo و جلسات آزمایشگاهی بدست آوردیم مکان یک ربات متحرک را با استفاده از الگوریتم فیلتر ذرات در یک محیط که نقشه آن را داریم بدست آوریم. در انتها ربات باید بتواند مکان خود را در نقشه اعلام کرده و همچنین اگر Kidnapping اتفاق بیوفتد و حین اجرای الگوریتم مکان ربات تغییر داده شود، ربات باید نسبت به آن مقاوم بوده و باز هم بتواند مکان خود را پیدا کند.

الگوریتم فیلتر ذرات

هدف الگوریتم Particle Filter Localization یا Monte Carlo Localization این است که ربات به این سوال که "کجا هستم؟" بتواند پاسخ دهد. فرض می‌شود که ربات نقشه محیط را دارد اما مکان خود را نمی‌داند. ربات باید بتواند مکان‌هایی را در نقشه پیدا کند که با چیزی که سنسورهایش دریافت می‌کنند مطابقت دارد. الگوریتم حدسیات زیادی از مکان ربات در کل نقشه در نظر می‌گیرد که به هر کدام particle گفته می‌شود. سپس آن‌چیزی که سنسورهای ربات دریافت می‌کنند را با چیزی که سنسورهای هر particle در آن مکان ممکن است دریافت کنند، مقایسه می‌کند. particle هایی که سنسورشان ورودی مشابه ربات دریافت می‌کند با احتمال بیشتری می‌توانند مکان رباتمان باشند. با حرکت ربات particleها نیز در محیط حرکت می‌کنند و هرچقدر که ربات بیشتر در محیط حرکت می‌کند، مکان دقیق آن با احتمال بیشتری یافت می‌شود. در واقع هر particle دارای یک وزن احتمالی است که نشان می‌دهد آن نمونه با چه احتمالی از تابع چگالی احتمال مکان ربات، نمونه برداری شده است. در نتیجه هر چه وزن particle هایی که در یک ناحیه قرار می‌گیرند بیشتر شود، نشان‌دهنده این است که احتمال قرارگیری ربات در این ناحیه بیشتر است.

- نکته: این الگوریتم بر اساس قائده Markov در نظر می‌گیرد که هر استیت فقط به استیت قبلی‌اش وابسته است، بنابراین محیطمان باید static باشد و در طول زمان تغییر نکند.

در این الگوریتم در ابتدا مجموعه‌ای از ذرات را به صورت تصادفی در نقشه انتخاب می‌کنیم و برای هرکدام وزن‌های یکسان در نظر می‌گیریم. سپس مراحل زیر را تا زمانی که ذرات به مکان ربات همگرا شوند تکرار می‌کنیم.

۱- ربات را به میزان مورد نظر حرکت می‌دهیم.

۲- با توجه به حرکت ربات، موقعیت و جهت ربات و ذرات را بروزرسانی می‌کنیم.

۳- مقدار اندازه‌گیری شده از سنسور لیزر ربات را با سنسور فرضی هر ذره مقایسه کرده و سپس به هر ذره با توجه به میزان مشابهت این دو مقدار یک وزن نسبت می‌دهیم.

۴- سپس با توجه به وزن ذرات Resampling انجام می‌دهیم:

- ٪۳۰ از بهترین ذرات را نگه می‌داریم
- ٪۵۵ از کل تعداد ذرات را با جایگزاری از بهترین ذرات بخش قبل انتخاب می‌کنیم، همچنین نویز حداکثر یک سانتی‌متری به آنها اعمال می‌کنیم
- ٪۱۵ باقی مانده را هم به صورت رندوم انتخاب می‌کنیم که در مرحله kidnapping موثر است

۵- تخمینمان از پوزیشن ربات را به روز رسانی می‌کنیم.

1. Algorithm **particle_filter**(S_{t-1}, u_t, z_t):
2. $S_t = \emptyset, \quad \eta = 0$
3. **For** $i = 1,\ldots,n$ — ***Generate new samples***
4. Sample index $j(i)$ from the discrete distribution given by w_{t-1}
5. Sample x_t^i from $p(x_t \mid x_{t-1}, u_t)$ using $x_{t-1}^{j(i)}$ and u_t
6. $w_t^i = p(z_t \mid x_t^i)$ — ***Compute importance weight***
7. $\eta = \eta + w_t^i$ — ***Update normalization factor***
8. $S_t = S_t \cup \{< x_t^i, w_t^i >\}$ — ***Add to new particle set***
9. **For** $i = 1,\ldots,n$
10. $w_t^i = w_t^i / \eta$ — ***Normalize weights***

شکل ۱- الگوریتم فیلتر ذرات

Kidnapping

در این قسمت می‌خواهیم مدتی پس از اجرای الگوریتم مکان ربات را به صورت دستی تغییر دهیم و الگوریتم باید نسبت به این تغییر robust باشد و به مکان قبلی بایاس نشود. برای حل این مشکل، ما در هر مرحله از Resampling ، ۱۵٪ از ذرات را به صورت رندوم انتخاب می‌کنیم که باعث می‌شود رباتمان به مکان قبلی بایاس نشود و درصورت تغییر در مکانش بتواند به مکان جدید خود همگرا شود.

توضیحات پیاده‌سازی

در این قسمت می‌خواهیم توضیحات مختصری در ارتباط با نحوه پیاده‌سازی پروژه و کلاس‌ها و توابع موجود بدهیم. برای اجرای پروژه باید فایل particle_filter.py اجرا شود.

کلاس Map

در ابتدای اجرای این کلاس از روی نقشه مختصات خود نقشه و مختصات چهار نقطه از چهار گوشه هر مانع در نقشه بدست می‌آید.

❖ **متغیرها:**

- متغیر self.global_map_position مختصات نقشه را در فضا نشان می‌دهد.
- متغیر self.rects مختصات چهارگوشه هر مانع را در فضا نشان می‌دهد.
- متغیر self.map_boundry مرزهای نقشه‌مان را در فضا مشخص می‌کند.
- متغیر self.all_map_lines تمام خطوط موجود در نقشه را مشخص می‌کند.
- متغیر self.polygan هر مانع را به صورت polygan نشان می‌دهد.

❖ **توابع:**

- تابع convert_point_to_line از روی چهار نقطه گوشه هر مانع، چهار خط مانع را می‌سازد.
- تابع add_offset مختصات هر گوشه از مانع را با مختصات کلی نقشه جمع می‌کند تا مختصات هر مانع در فضا بدست بیاید.

- تابع convert_to_poly از روی چهار نقطه گوشه هر مانع یک polygan برای هر مانع بدست می‌آورد.
- تابع map_boundry_func از روی مینیمم و ماکسیمم مختصات مرکز هر کدام از موانع حدود نقشه را بدست می‌آورد که تا کجا ادامه دارد.
- تابع plot_map نقشه را به همراه موانع موجود در آن رسم می‌کند.
- تابع find_intersection ورودی این تابع دو خط می‌باشد که می‌خواهیم ببینیم آیا این دو خط با هم تلاقی داردند یا خیر. اگر تلاقی دارند نقطه تلاقی‌شان بازگردانده می‌شود و اگر تلاقی ندارند false بازگردانده می‌شود.
- تابع check_is_collition یک نقطه دریافت می‌کند و با توجه به موانع موجود در نقشه بررسی می‌کند که آیا این نقطه با آنها برخورد دارد یا خیر.
- تابع out_of_range یک نقطه به عنوان ورودی می‌گیرد و بررسی می‌کند آیا این نقطه خارج از مرزهای نقشه قرار دارد یا خیر.
- تابع check_particle_path این تابع برای بررسی حرکت ذرات می‌باشد. ورودی این تابع نقطه شروع و پایان حرکت یک ذره است. سپس با استفاده از تابع find_intersection بررسی می‌شود که ایا خطی که ربات در این حرکت می‌پیماید با هیچ کدام از چهار خط همه موانع در نقشه تلاقی دارد یا خیر. در واقع بررسی می‌کند که این ذره در حرکت خود با هیچ کدام از موانع برخورد نداشته باشد.

کلاس Particle

❖ **متغیرها:**

- متغیر self.map یک instance از کلاس mapمان می‌باشد.
- متغیر self.particle_number تعداد ذراتمان را مشخص می‌کند.
- متغیر self.particles_pose پوزیشن هر ذره را مشخص می‌کند که شامل x و y و theta می‌باشد.
- متغیر self.particle_measurment اندازه سنسور هر ذره را مشخص می‌کند.
- متغیر self.weights وزن هر ذره را مشخص می‌کند.
- متغیر self.sensor_line نیز خط لیزر هر ذره را مشخص می‌کند.

❖ **توابع:**

- تابع randomize مختصات هر ذره را به صورت رندوم انتخاب می‌کند. x و y را به صورت رندوم یکنواخت در حدود مرزهای نقشه انتخاب می‌کند. تتا را از بین اعداد ۹۰، ۹۰-، ۱۸۰ و ۰ به صورت رندوم انتخاب می‌کند. اگر هر کدام از ذرات در collision بودند x و y شان را به صورت رندوم دوباره انتخاب می‌کند و آنقدر ادامه می‌دهد که دیگر آن نقطه در collision نباشد.
- در تابع calculate_weight می‌خواهیم با استفاده از مقداری که سنسور لیزر ربات بر می‌گرداند و همچنین اندازه‌های محاسبه شده برای سنسورهای ذرات، وزن ذرات را به روز رسانی کنیم. چون سنسور لیزر ربات دارای خطا می‌باشد بنابراین از مدل خطای این سنسور که در پروژه‌های قبلی بدست آوردیم استفاده می‌کنیم. ما اندازه‌گیری‌های سنسور لیزر ربات را در فواصل ۵، ۱۰، ۱۵، ۲۰، ۲۵، ۳۰، ۳۷ و ۴۰ انجام دادیم. بنابراین در ابتدا برای هر ذره محاسبه می‌کنیم که اندازه سنسور لیزر به کدام یک از این اعداد نزدیک‌تر است و سپس میانگین و انحراف معیار خطای همان فاصله را در نظر می‌گیریم. حال با استفاده از جمع مقدار سنسور هر ذره با میانگین خطای فاصله مورد نظر و انحراف معیار آن یک توزیع نرمان برای هر ذره بدست می‌آوریم. در نهایت مقدار سنسور اندازه‌گیری شده در ربات را در هر توزیع نرمال بدست آورده و این مقدار برابر وزن هر ذره می‌باشد.
- در تابع calculate_particles_sensors خط لیزر هر ذره را در ابتدا از مختصات آن ذره تا فاصله ۰.۴ متر جلوتر از آن در نظر می‌گیریم. حال با استفاده از تابع find_intersection بررسی می‌کنیم که آیا این خط با هیچ کدام از چهار خط موانعمان تلاقی دارد یا خیر. اگر تلاقی نداشته باشد این خط را در همان فاصله ۰.۴ در نظر می‌گیریم و measurement آن ذره را نیز ۰.۴ قرار می‌دهیم. اگر تلاقی داشته باشد measurement را فاصله آن ذره با نقطه تلاقی در نظر گرفته و خط لیزر ذره را نیز از مختصات آن ذره تا تلاقی در نظر می‌گیریم. در این اندازه‌گیری‌ها خط لیزر را در رایتای تتای آن ذره در نظر گرفته و محاسباتمان را با توجه به آن انجام می‌دهیم.
- تابع plot_particle ذرات را به همراه جهتشان در نقشه نمایش می‌دهد.
- هدف تابع move_angular پیاده‌سازی حرکت زاویه‌ای در ذرات است. در این قسمت از مدل خطای حرکت زاویه‌ای رباتمان در Gazebo استفاده می‌کنیک که در پروژه قبلی بدست آوردیم. در اینجا ما سه مقدار v و w و γ را داریم که به ترتیب معادل angular_linear، angular_angular و angular_gamma می‌باشند. حال با توجه به اینکه زاویه‌ای که ذرات می‌خواهند بچرخند ۹۰درجه یا ۹۰- درجه است، مقادیر میانگین و انحراف معیار سه متغیر بالا را بدست می‌آوریم. حال v و w را برابر یک مقدار رندوم از توزیع نرمال میانگین و انحراف معیارش قرار می‌دهیم. گاما را نیز برابر مقدار زاویه‌ای

که ربات باید بچرخد به علاوه یک مقدار رندوم از توزیع نرمال میانگین و انحراف معیار آن قرار می‌دهیم. حال با استفاده از فرمول زیر پوزیشن هر پارتیکل را تغییر می‌دهیم.

$$\hat{v} = v + \mathbf{sample}(\alpha_1|v| + \alpha_2|\omega|)$$
$$\hat{\omega} = \omega + \mathbf{sample}(\alpha_3|v| + \alpha_4|\omega|)$$
$$\hat{\gamma} = \mathbf{sample}(\alpha_5|v| + \alpha_6|\omega|)$$
$$x' = x - \frac{\hat{v}}{\hat{\omega}}\sin\theta + \frac{\hat{v}}{\hat{\omega}}\, sin(\theta + \hat{\omega}\Delta t)$$
$$y' = y + \frac{\hat{v}}{\hat{\omega}}\cos\theta - \frac{\hat{v}}{\hat{\omega}}\, cos(\theta + \hat{\omega}\Delta t)$$
$$\theta' = \theta + \hat{\omega}\Delta t + \hat{\gamma}\Delta t$$

- هدف تابع move_linear حرکت خطی ذرات می‌باشد. در اینجا از مدل خطای حرکت خطی رباتمان در Gazebo استفاده می‌کنیم. در اینجا ما فواصل ۵، ۱۰ و ۱۵ سانتی متر را داریم و با توجه به اینکه میزان حرکت خطی هر ذره به کدام یک از این فواصل نزدیک‌تر است میانگین و انحراف معیار خطایمان را بدست می‌آوریم. سپس یک نقطه میانگین از توزیع نرمال این میانگین و انحراف معیار بدست آورده و آن را به سرعت حرکتمان اضافه می‌کنیم. سپس موقعیت جدید هر ذره را با توجه به این سرعت بدست می‌آوریم. در نهایت چک می‌کنیم که این ذرات پس از حرکت در collision نباشند.
- در تابع resampling می‌خواهیم بخش resampling الگوریتم را پیاده‌سازی کنیم.
  - ۳۰٪ از بهترین ذرات را نگه می‌داریم
  - ۵۵٪ از کل تعداد ذرات را با جایگزاری از بهترین ذرات بخش قبل انتخاب می‌کنیم، همچنین نویز حداکثر یک سانتی‌متری به آنها اعمال می‌کنیم
  - ۱۵٪ باقی مانده را هم به صورت رندوم انتخاب می‌کنیم

  سپس تابع calculate_particles_sensors را برای محاسبه سنسور لیزر برای ذرات جدیدمان فراخوانی می‌کنیم.
- تابع center_point در بین بهترین ۳۰ درصد بهترین ذرات، ذره‌ای را که کمترین فاصله را با بقیه ذرات دارد باز می‌گرداند.

کلاس Robot

❖ **متغیرها:**

- متغیرهای self.x, self.y, self.theta موقعیت ربات و self.laser اندازه سنسور آن را نشان می‌دهند.
- متغیر self.angular_speed سرعت حرکت زاویه‌ای ربات را نشان می‌دهد.
- متغیر self.linear_speed سرعت حرکت خطی ربات را نشان می‌دهد.

❖ **توابع:**

- تابع laser_reader مقدار سنسور لیزر ربات را بر می‌گرداند.
- تابع odometry موقعیت ربات را که شامل x، y و theta می‌شود، در Gazebo باز می‌گرداند. این مقدار برای ترسیم محل ربات در نمودار و همچنین مقایسه موقعیت واقعی ربات و موقعیت بدست آمده توسط الگوریتم پس از اتمام اجرای الگوریتم می‌باشد.
- تابع rotation برای حرکت زاویه‌ای ربات می‌باشد.
- تابع translation برای حرکت خطی ربات می‌باشد.

کلاس ParticleFilter

❖ **متغیرها:**

- متغیر self.map یک instance از کلاس map می‌باشد
- متغیر self.particles یک instance از کلاس particle می‌باشد.

(در ابتدا self.particles.randomize را اجرا می‌کنیم تا به همه ذرات یک موقعیت رندوم نسبت دهیم. سپس self.particles.calculate_particles_sensors را اجرا می‌کنیم تا مقدار سنسور همه ذرات را محاسبه کنیم)

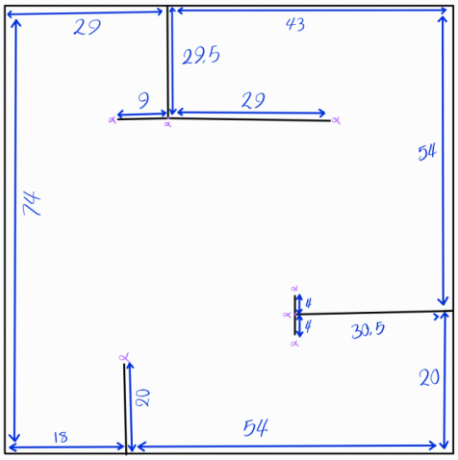
- متغیر self.particles_pose موقعیت هر ذره می‌باشد.
- متغیر self.number_of_particles تعداد ذرات را مشخص می‌کند.
- متغیر self.robot یک instance از کلاس robot می‌باشد.

❖ **توابع:**

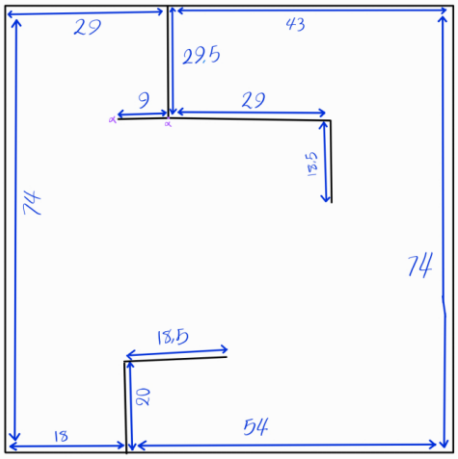
- تابع plot_current_state نقشه را به همراه ذرات و مکان واقعی ربات در هر مرحله نمایش می‌دهد.
- هدف تابع step اجرای یک تکرار از الگوریتم می‌باشد. در این تابع در ابتدا میزان حرکت و چرخش ربات را به عنوان ورودی از کاربر می‌گیریم. سپس با صدا زدن توابع self.robot.rotation و self.robot.translation رباتمان را حرکت می‌دهیم. در مرحله بعد با صدازدن توابع self.particles.move_angular و self.particles.move_linear ذراتمان را حرکت می‌دهیم. در مرحله بعد مقدار سنسور لیزر و وزن ذرات را با استفاده از توابع self.particles.calculate_particles_sensors و self.particles.calculate_weight بدست می‌آوریم. حال در هر گام میخواهیم ببینیم که آیا شرط همگراییمان برقرار شده است یا خیر، برای اینکه الگوریتممان را پایان دهیم. در ابتدا با استفاده از تابع self.particles.center_point ذره‌ای را که به همه نزدیک‌تر است پیدا می‌کنیم. سپس ذراتی را که در فاصله ۵ سانتی متری این ذره قرار دارند می‌یابیم. اگر تعداد این ذرات بیشتر یا مساوی ۸۰٪ کل ذرات باشد می‌توانیم الگوریتممان را خاتمه دهیم. اگر الگوریتم خاتمه پیدا نکند با استفاده از تابع self.particles.resampling مرحله resampling را اجرا می‌کنیم.
- هدف تابع particle_filter_algo اجرای همه تکرارهای الگوریتم است. تا زمانی که به انتهای الگوریتم نرسیدیم در ابتدا با استفاده از تابع self.plot_current_state وضعیت نقشه و ذرات و رباتمان را نمایش می‌دهیم. سپس تا زمانی که شرط خاتمه برقرار نشود تابع particle_filter.step را فراخوانی می‌کنیم.
- حال با اجرای فایل particle_filter.py در ابتدا آدرس نقشه و تعداد ذرات به عنوان ورودی از کاربر گرفته می‌شود. سپس یک instance از کلاس map ساخته می‌شود. در مرحله بعد یک instance از کلاس ParticleFilter ساخته می‌شود. در نهایت برای اجرای الگوریتم تابع particle_filter.particle_filter_algo صدا زده می‌شود.

تغییرات پیاده‌سازی برای ربات واقعی

۱. برای خواندن نقشه، نقاط هر گوشه از موانع به صورت دستی به دست آمده و درون متغیرهای نقشه جایگزاری شدند.



شکل ۲- نقشه تست ربات واقعی



شکل ۳- نقشه آزمون ربات واقعی

۲. مقادیر خطای سنسور و حرکت ربات با مقادیری که از مدلسازی سنسوری و حرکتی ربات در جلسات آزمایشگاهی بدست آمده بود جایگزین شدند.

۳. تاپیک‌های سنسوری و حرکتی ربات نیز جایگزین شدند. به عنوان مثال در محیط gazebo پیام‌های حرکتی از نوع twist بودند در حالی که در محیط واقعی پیام‌ها از نوع drive هستند و برای حرکت زاویه‌ای در محیط واقعی باید سرعت خطی چرخ‌ها را به سرعت زاویه‌ای تبدیل کرد.

۴. تبدیل واحدهای مورد نیاز نیز اعمال شدند، واحد مورد استفاده در محیط گزبو متر بود در حالی برای ربات واقعی میلی‌متر است.

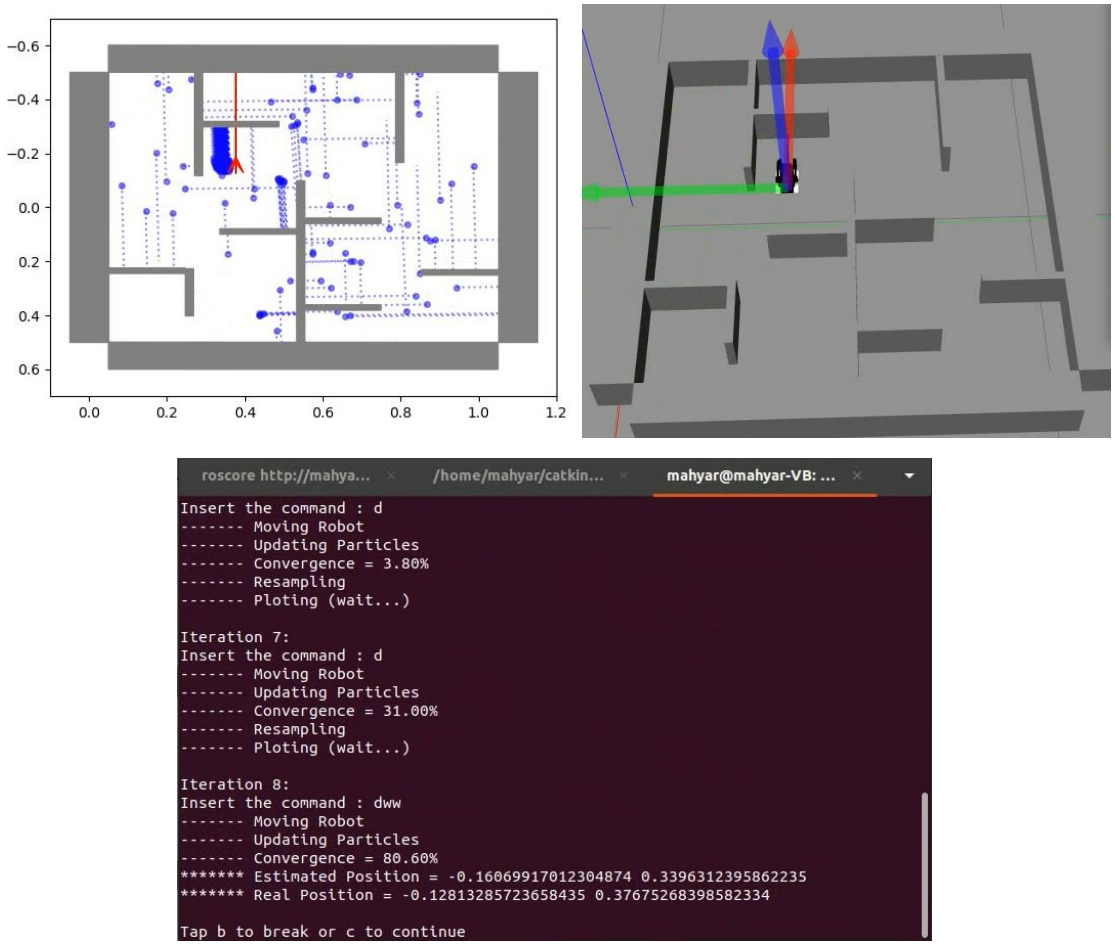
نتایج

محیط گزبو

همان طور که در تصاویر زیر مشخص است در اجرای ۱ از الگوریتم، particle ها پس از ۸ تکرار با شرایط خواسته‌شده همگرا شده‌اند و 80.06 درصد از آن‌ها در فاصله ۵ سانتی‌متری ربات قرار گرفته‌اند.

جدول ۱- نتایج دو بار اجرای الگوریتم در محیط گزبو

	تعداد ذرات	تعداد تکرار	درصد همگرایی	مختصات واقعی ربات	مختصات تخمین زده شده
اجرای ۱	500	8	80.06 %	-0.128, 0.376	-0.160, 0.339
اجرای ۲	500	14	81 %	-0.227, 0.644	-0.247, 0.645



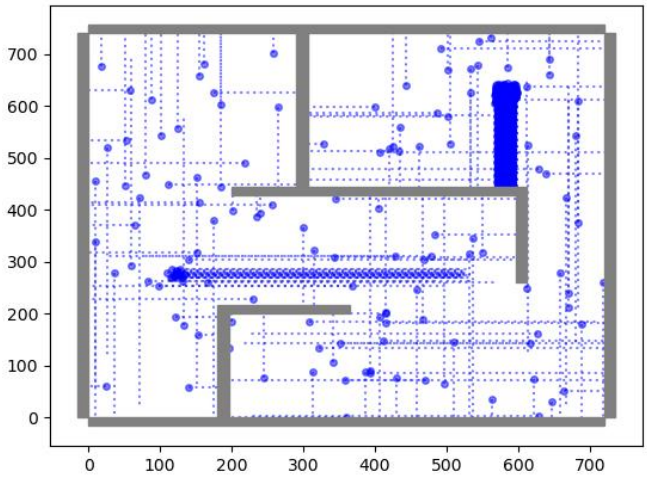
شکل ۴- تصاویر مربوط به اجرای ۱ در محیط گزبو

ربات واقعی

همان طور که در تصاویر زیر مشخص است در آزمون ربات واقعی، particle ها پس از ۲۳ تکرار با شرایط خواسته‌شده همگرا شده‌اند و 83.9 درصد از آن‌ها در فاصله ۵ سانتی‌متری ربات قرار گرفته‌اند. همچنین پس از انجام عمل kidnapping این اتفاق بعد از ۱۷ تکرار و با درصد 83.8 رخ داده‌است.

جدول ۲- نتایج عمل **localization** و **kidnapping** ربات واقعی

	تعداد ذرات	تعداد تکرار	درصد همگرایی	مختصات تخمین زده شده
Localization	1000	23	83.9 %	627.40, 683.39
Kidnapping	1000	17	83.8 %	581.23, 613.63



شکل ۵- نمودار همگرایی **Particle** ها در ربات واقعی پس از انجام **kidnapping**