

به نام خدا

# گزارش فاز نهایی پروژه نهایی

رباتیک پیشرفته

مهدی غیاثی  
۸۱۰۱۹۸۲۲۰

امیرمحمد رنجبر پازکی  
۸۱۰۱۹۹۳۴۰

دانشکده برق و کامپیوتر دانشگاه تهران

تابستان ۱۴۰۰

## • مقدمه:

در این فاز با استفاده از مدل حرکتی و سنسوری به دست آمده در فاز قبل، و همچنین فیلتر ذرات با حرکت دادن ربات به صورت تصادفی در فضا موقعیت ربات را به دست می‌آوریم.

## • مراحل:

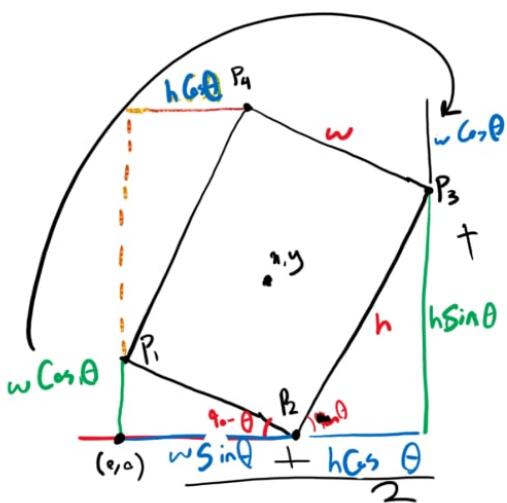
### + خواندن نقشه:

محیطها به صورت یک فایل **.world** می‌شود. این فایل‌های دارای فرمت **xml** هستند و برای خواندن آن‌ها از **ElementTree** در محیط پایتون استفاده شده است. این کلاس یک فایل **xml** را تجزیه می‌کند و سپس، می‌توان با استفاده از آن روی عناصر حرکت کرد.

تگ‌های **model** این فایل گرفته می‌شوند که شامل دو تگ **pose** و **link** هستند. تگ **pose** دارای موقعیت کلی آن جهان (مدل) است. در اینجا با خواندن آن به **X**، **Y** و **Z** و زوایای **roll**، **pitch** و **yaw** کلی نقشه پی می‌بریم که البته **X** و **Y** به کار می‌آیند چراکه نقشه‌های معمولاً زاویه ندارند و بر روی هوا نیز نیستند.  
Link‌ها خود دارای تعدادی زیرمجموعه هستند. هر تگ **pose** بیانگر موقعیت آن المان است. (مشابه موقعیت دنیا)

یک تگ **collision** نیز دارد که در آن یک تگ **geometry** وجود دارد که عرض و طول و ارتفاع آن قطعه را می‌دهد.

با خواندن این موارد می‌توان چهار گوشه هر مربع را محاسبه کرد که به صورت زیر به دست می‌آید.



$$C = \left( \frac{w \sin \theta + h \cos \theta}{2}, \frac{h \sin \theta + w \cos \theta}{2} \right)$$

$$P_1 = \left( x_c - \frac{w \sin \theta - h \cos \theta}{2}, y_c - \frac{h \sin \theta + w \cos \theta}{2} \right)$$

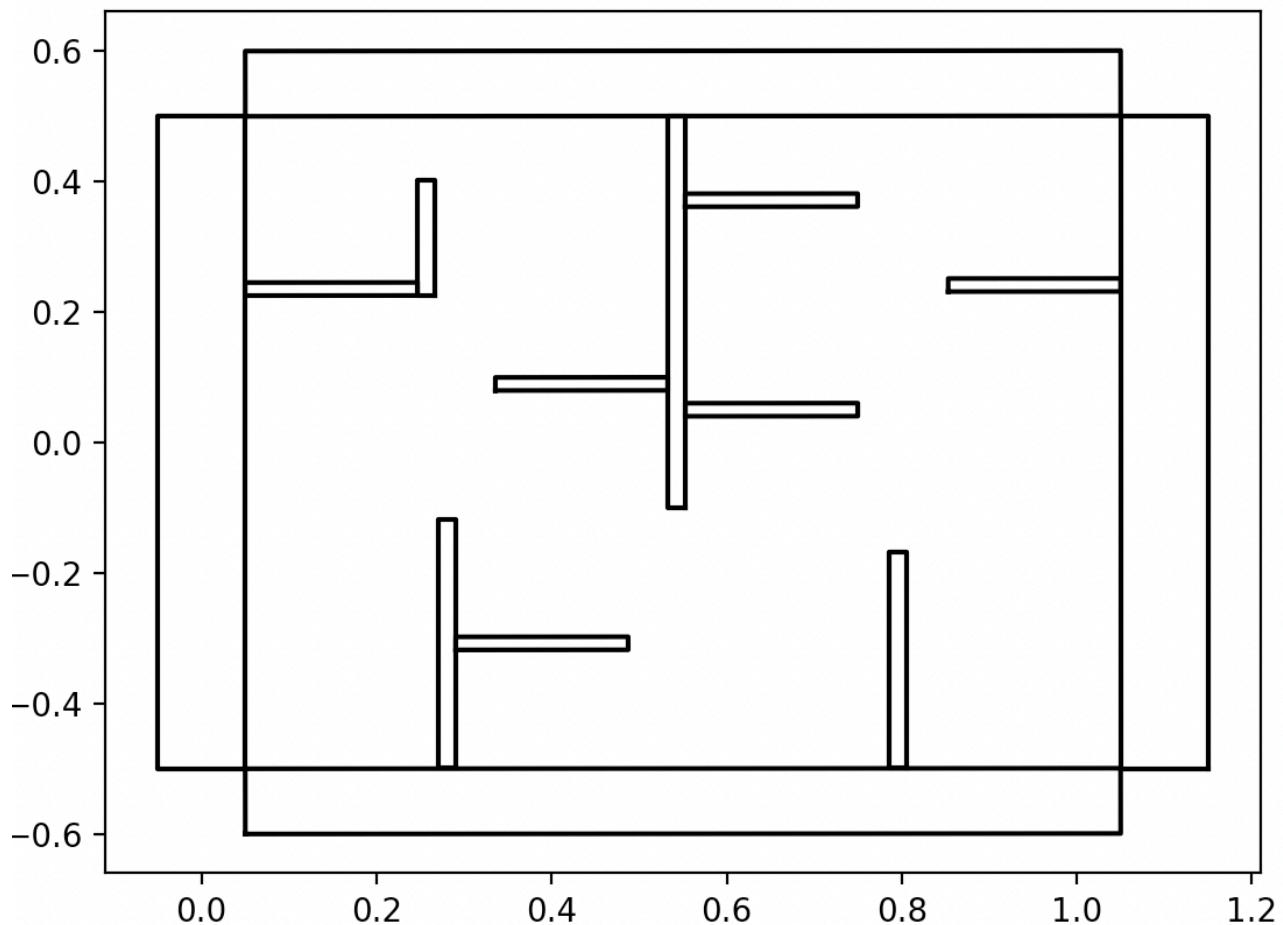
$$P_2 = \left( x_c + \frac{w \sin \theta - h \cos \theta}{2}, y_c - \frac{h \sin \theta - w \cos \theta}{2} \right)$$

$$P_3 = \left( x_c + \frac{w \sin \theta + h \cos \theta}{2}, y_c + \frac{h \sin \theta - w \cos \theta}{2} \right)$$

$$P_4 = \left( x_c - \frac{w \sin \theta + h \cos \theta}{2}, y_c + \frac{h \sin \theta + w \cos \theta}{2} \right)$$

حال با استفاده از این رابطه‌ها چهار نقطه هر مربع را محاسبه می‌کنید و در یک آرایه قرار می‌دهیم.  
همچنین، مراکز آن‌ها را نیز در لیست دیگری می‌ریزیم.

کلاس نقشه یک تابع دیگر دارد که با رسم این مستطیل‌ها نقشه را رسم می‌کند که در زیر نمونه‌ای از آن مشاهده می‌شود.



تعدادی تابع دیگر نیز نوشته شده‌اند که تمام خطوط و چندضلعی‌ها و همچنین مرز نقشه را برگرداند.  
همچنین چند تابع کمکی دیگر برای پیدا کردن تقاطع دو خط، معتبر بودن نقطه (در مرز بودن و در  
تقاطع نبودن) و همچنین تقاطع داشتن یا نداشتن نیز نوشته شده است.  
پس از این مرحله نقشه سه بعد تبدیل به یک نقشه دو بعدی قابل استفاده می‌شود.

کدهای این کلاس در فایل `map.py` موجود است. توابع کمکی نیز در فایل `map_utils.py` آمده‌اند.

### + حرکت ربات:

حرکت ربات مشابه آن‌چه در فاز اول (برای یافتن مدل سنسوری و حرکتی ربات) صورت گرفته بود انجام می‌گیرد.

به طور کلی، برای حرکت (translate) و چرخش (rotate) از دو کنترلر p استفاده می‌شود؛ در واقع یک error بر اساس وضعیت فعلی و وضعیت هدف به دست آمده، و بر این اساس سرعت حرکت/چرخش ربات تنظیم می‌شود.

همچنین یک حد بالا هم برای سرعت در نظر گرفته شده تا برای errorهای بزرگ، حرکت ربات بسیار ناگهانی نباشد (چرا که در این صورت، خطای حرکت زیاد می‌شود).

کنترلر p حرکت و چرخش ربات از رابطه‌های زیر پیروی می‌کنند:

$$error_{angle} = target\ angle - current\ angle$$

$$angular\ velocity^{(rad/s)} = 20 \times sign(error_{angle}) \times max(10, |error_{angle}|)$$

$$error_{linear} = target - current\ position$$

$$linear\ velocity^{(rad/s)} = 4 \times sign(error_{linear}) \times max(0.1, |error_{angle}|)$$

همچنین در مواقعي ممکن است ربات به دیواری گير کند (از آنجايي که سنسور ربات تنها يك خط در روبرو را مي‌بیند، ممکن است دیواری در گوشه را نبیند) و به اين ترتيب error<sub>linear</sub> هیچ‌گاه صفر نمی‌شود. برای جلوگیری از گیر کردن ربات در این وضعیت، يك تایم‌اوت ۳ ثانیه‌ای نیز برای کنترلرها در نظر گرفته شده است.

## + ذره‌ها، حرکت و نحوه‌ی وزن‌دهی به آن‌ها:

هر پارتيکل هم مانند خود ربات، سه مولفه‌ی x, y, theta را دارد. هنگامی که دستور چرخش به ربات داده می‌شود، theta<sub>i</sub> هر پارتيکل هم به همان مقدار می‌چرخد؛ و هنگامی که دستور حرکت به ربات داده می‌شود، تک تک پارتيکل‌ها هم در جهت theta<sub>i</sub> خود، به میزان distance حرکت می‌کند. (حرکات دورانی و translation خطاهایی به صورت یکدیگر دارند که از مدل حرکتی فاز یک قرار داده شده است).

هر پارتيکل در جهت theta<sub>i</sub> خود یک خط سنسور فرضی هم دارد (اگر ربات در نقطه‌ی پارتيکل باشد، احتمالاً چه داده‌ای از سنسور دریافت می‌کند). پس از انجام حرکت ربات، برای تک تک پارتيکل‌ها خط فرضی سنسور محاسبه شده و تقاطع احتمالی آن با خطوط مربوط به موانع نقشه به دست می‌آید.

بر این اساس، می‌توانیم احتمال درست بودن مکان و جهت این پارتيکل را بر اساس مشاهده‌ی فعلی ربات آپدیت کنیم. در واقع می‌خواهیم یک احتمال به صورت

$$P(\text{virtual sensor response} = a | \text{real sensor response} = b)$$

به دست آوریم که در آن  $a$  فاصله‌ی بین پارتیکل تا نقطه تقاطع، و  $b$  مشاهده‌ی واقعی ربات است.  
بر اساس این احتمال پارتیکل‌ها را وزن‌دهی می‌کنیم.  
وزن داده شده از رابطه‌ی زیر به دست می‌آید:

$$weight = pdf[real\ sensor\ response + 0.043] \ from \ N(virtual\ sensor\ response, 0.01049)$$

(دقت کنید که از فاز اول پاسخ سنسور به صورت  $N(-0.043, 0.1049)$  به دست آمده بود)  
در ادامه این وزن‌ها نرمالیزه شده و بر اساس این وزن‌های نرمالیزه شده، **resampling** انجام می‌گیرد که در ادامه به توضیح آن خواهیم پرداخت.  
+ نمونه‌گیری و تولید نمونه‌های جدید(دو روش):

برای نمونه‌گیری از دو روش **best select** و **roulette wheel** استفاده شده است.

در روش **roulette wheel** هشتاد درصد ذرات به میزان وزنشان وزن می‌گیرند و به صورت تصادفی با جایگذاری از این میان ذرات جدید به صورت وزن‌دار انتخاب می‌شوند. ۲۰ درصد ذرات به صورت کاملاً تصادفی تولید می‌شوند. دلیل این موضوع حل مشکل **kidnapping** است که در ادامه توضیح داده می‌شود.

در روش **best select** سعی شده تا از روشی ترکیبی استفاده شود. در این روش ۳۵ درصد بهترین ذرات نسل قبل نگه داشته می‌شوند. حال که ۵۰ درصد ذرات حول بهترین ذرات نسل قبل به صورت تصادفی تولید می‌شوند. ۱۵ درصد باقی‌مانده نیز به صورت تصادفی تولید می‌شوند تا مشکل **kidnapping** را حل کنند.

روش **best select** همگرایی سریع‌تر و دقیق‌تری دارد و به همین دلیل، این روش در نهایت انتخاب شد.

توابع این قسمت‌ها به صورت جدا در کد موجود است.

+ **رسم ذرات و ربات و مکان تخمینی:**

برای رسم موارد خواسته شده در کد، در فایل **visulization.py** توابع کمکی قرار داده شده‌اند. یکی از این توابع موقعیت یک ذره را به صورت فلشی که جهت آن را هم نشان دهد با رنگ خاصی رسم می‌کند. یک تابع دیگر نیز خط **sensor** ربات را تا نزدیک‌ترین دیوار رسم می‌کند تا بتوانیم کاملاً وضعیت را بدانیم.

برای رفع مشکل سرعت که در ادامه توضیح داده می شود، رسم که حدود یک ثانیه طول می کشید یکتابع جدا شده است و بر روی تردی جدا اجرا می شود.

در این تابع وضعیت ذرات به صورت فلش های قرمز رنگ، وضعیت ربات به صورت فلش آبی رنگ و تخمین موقعیت به صورت فلش سبز رنگ رسم می شود. همچنین خط لیزر ربات نیز به رنگ آبی رسم می شود.

#### + تخمین موقعیت ربات (دو روش):

برای تخمین موقعیت ربات از دو روش استفاده شده است. در یکی از روش ها میانگین ۳۰ درصد ذرات بالایی برای موقعیت نهایی گرفته شده است. البته برای زاویه mode حساب شده است چرا که نیاز است تا مقادیر خاصی به خود بگیرد.

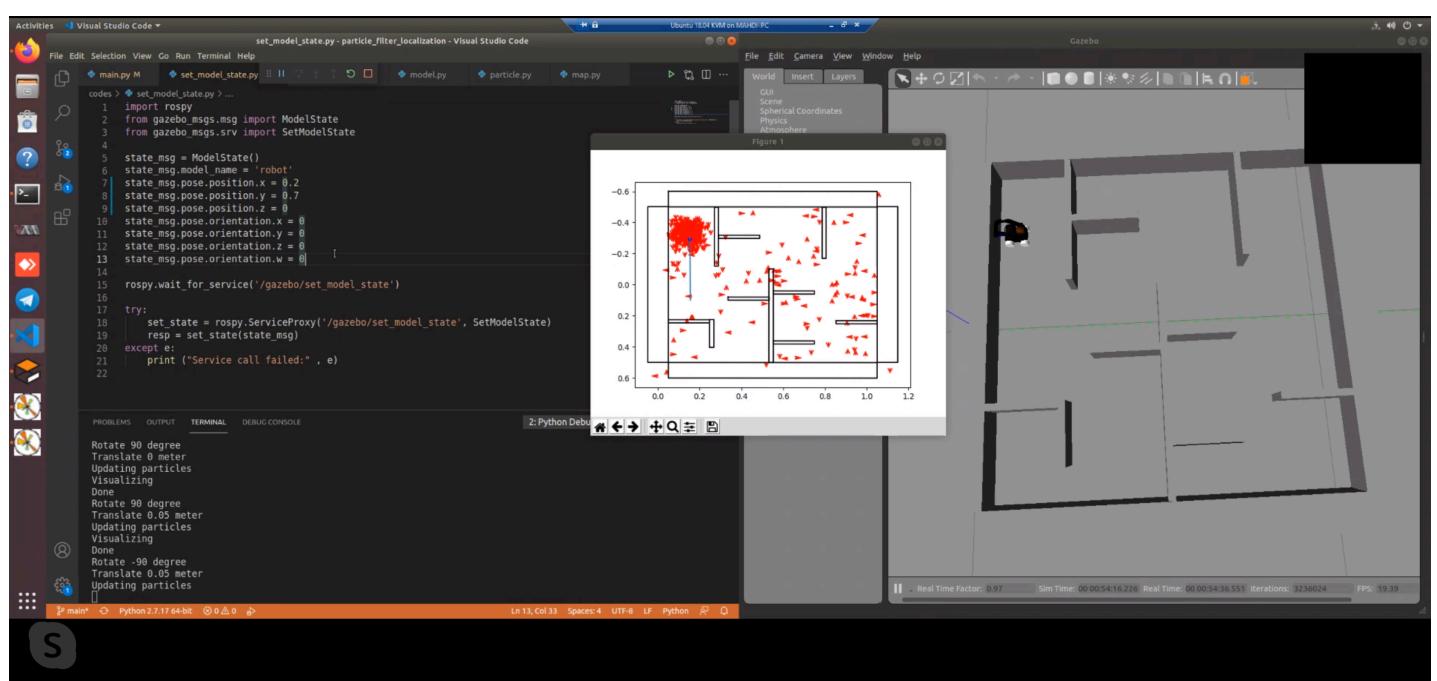
در روش دیگر ذرهای که از دیگر ذرات مجموع فاصله کمتری دارد به عنوان تخمین گزارش شده است. دلیل این موضوع آن است که این ذره به نوعی در مرکز تجمع ذرهای قرار دارد و می تواند نماینده خوبی برای مکان ربات باشد.

#### + پایان الگوریتم:

پایان الگوریتم زمانی است که بیش از ۶۰ درصد داده ها در فاصله کمتر از ۵ سانتی متری موقعیت واقعی ربات باشند. این به آن معناست که ذرات به خوبی در نقطه ربات جمع شده اند و می توانند موقعیت ربات را به خوبی تخمین بزنند.

#### + نمونه موفقیت آمیز اجرا:

در شکل زیر یک نمونه موفقیت آمیز اجرا مشاهده می شود که ذرات به خوبی در موقعیت ربات جمع شده اند و موقعیت ربات به خوبی تخمین زده شده است.



## ۰ چالش‌ها: + :Kidnapping

یکی از مشکلاتی که ممکن است برای ربات رخ دهد تعویض ناگهانی جای ربات است. یعنی در حین اجرا الگوریتم یا با داشتن حافظه‌ای از قبل، ربات در مکان دیگر شروع به کار کند. در این حالت اطلاعات قبلی اشتباه است و اگر ربات به آن‌ها بچسبد، نتیجه غلطی در پی دارد.

به همین دلیل در هر اجرا ۱۵ الی ۲۰ درصد ذرات جدید تولید می‌شوند تا در صورت تعویض جای ربات ذره‌ای در نزیکی آن باشد که بتواند وزنی بگیرد و ذرات دیگر را حول خود تولید کند و مکان قبلی را به نوعی پاک کند.

در اجرای موفق بخش قبل، ناگهان مکان ربات را به جایی دور از توده ذرات جابجا کردیم. همانطور که در تصاویر زیر مشاهده می‌کنید، گام به گام ذرات به مکان درست می‌روند و تخمین جدید به درستی زده می‌شود.

### + تفاوت فضاها و حرکات:

بعضی فضاها دارای تقارن‌هایی هستند که فیلتر ذرات را به اشتباه می‌اندازد و چند نقطه به عنوان مکان‌های محتمل انتخاب می‌شوند. در این فضاها نیاز است که بیشتر **explore** کنیم تا تفاوت‌ها را ببینیم.

در بعضی از نقشه‌ها نیاز است تا بیشتر از چرخش حرکت رو به جلو کنیم چراکه چرخش بیش از حد باعث حرکات تکراری می‌شود.

بالعکس در برخی دیگر از نقشه‌ها نیز است تا بیشتر بچرخیم تا بتوانیم به فضاها جدید بررسیم. به همین دلیل باید هایپرپارامترهای مدل با توجه به نوع نقشه **tune** شوند تا بتوانیم به پاسخ خوبی بررسیم.

### + نبود سنسورهای بیشتر و ساختار نامناسب ربات:

این ربات بسیار ساده می‌باشد و صرفاً یک سنسور **range finder** در جلوی خود دارد. با استفاده از این سنسور که به صورت خطی است ممکن است زوایای ربات به خوبی دیده نشوند و همچنین، حرکت پیچیده و تصمیم گیری و تفکر برای ربات امکان پذیر نباشد زیرا داده‌ی کمی دارد.

برای انتخاب مناسب حرکت عدم برخورد با مانع ممکن بود نیاز باشد تا ربات کمی در جا بچرخد و مسیر حرکت خود را بهتر ببیند چرا که سنسور ربات خطی است و ممکن مانع را در روبرت نبیند ولی چرخهای ربات به دیوار گیر کند و دیوار را بیاندازد یا خود به هوا برود و پس از بازگشت زاویه‌اش خراب شود.

مشکل گیر کردن ربات در دیوار نیز با استفاده از قرار دادن timeout برای آن حل شده است.



این ربات می‌توانست با داشتن یک lidar یا حداقل چند سنسور در کناره‌ها داده‌های بیشتری جمع کند و تصمیم‌های پیچیده‌تر و بهتری بگیرد. مسیرهای بهتری را بپیماید و کمتر به دیوارهای بخورد.

## + الگوریتم تصادفی و جایگزینی آن با الگوریتم‌های پیچیده‌تر مثل حل maze

### یا SLAM

در این پروژه برای پیاده‌سازی particle filter، حرکات به صورت تصادفی انجام می‌شوند. در هر مرحله ابتدا ربات یک زاویه چرخش از بین  $0^{\circ}$  و  $90^{\circ}$ - درجه انتخاب کرده و چرخش را انجام می‌دهد، سپس یک طول حرکت از بین  $5$ ،  $10$ ،  $15$ ،  $20$  و  $30$  سانتی‌متر را انتخاب کرده و حرکت را هم انجام می‌دهد.

(در انتخاب طول حرکت، داده‌ی ورودی از سنسور جلو هم لحاظ شده تا حتی‌الامکان از برخورد ربات با دیوار جلوگیری شود. مثلاً اگر سنسور جلو نشان می‌دهد که  $12$  سانتی‌متر تا دیوار فاصله داریم، حرکت  $15, 20$  و  $30$  سانتی‌متر انتخاب نمی‌شود)

این حرکت تصادفی سبب می‌شود که گاهی اوقات ربات مکان‌های تکراری را چند بار بازدید کند و داده‌های مفید کمتری در particle filter اعمال شود. این امر مکان‌یابی را کندتر می‌کند.

اگر بتوانیم روشی را به کار بگیریم که exploration بیشتری انجام دهد، شواهد بیشتری خواهیم دید و پارتیکل‌های نامناسب زودتر حذف می‌شوند، در نتیجه فرایند مکان‌یابی سریعتر به نتیجه می‌رسد.

برای مثال، می‌توان از یک روش حل maze برای حل مسئله استفاده کرد، به این صورت که ربات به یک دیوار بچسبد و در امتداد آن حرکت کند، تا نقاط مختلف نقشه را زودتر کشف کند.

البته به دلیل اینکه ربات تنها یک سنسور فاصله در جلوی خود دارد، چنین کاری کند خواهد بود، اما با این حال ممکن است در نقشه‌هایی که مکان‌یابی در آن‌ها سخت است (مثلاً نقشه‌های متقارن، در ادامه به طور خاص به آن می‌پردازیم) زودتر ما را به نتیجه برساند.

همچنین می‌توان با ترکیب SLAM با مکان‌یابی، در همین حین که SLAM در حال explore کردن نقشه برای ساخت نقشه است، مکان‌یابی را هم انجام داد. با این کار از explore کردن SLAM استفاده می‌شود تا شواهد بهتری توسط قسمت مکان‌یابی پیدا شده و همانطور که گفته شد، مکان‌یابی تسریع شود.

## + سرعت و رویکرد موازی سازی:

فرایند وزن‌دهی به پارتیکل‌ها و به طور خاص پیدا کردن virtual sensor response برای هر پارتیکل، عمل زمان‌بری است (به ازای هر پارتیکل، باید این خط با تمامی خطوط نقشه تقاطع داده شود). از آنجایی که تعداد پارتیکل‌های موجود در نقشه هم زیاد است (مثلاً  $500$  یا  $1000$  پارتیکل)، این عمل update کردن می‌تواند مقدار زیادی طول بکشد که این امر میزان stall شدن ربات بین تصمیم‌های حرکتی گرفته شده را زیاد می‌کند.

برای بهبود این مسئله، پیدا کردن وزن پارتيکل‌ها به صورت موازی صورت می‌گیرد تا با بهره‌گیری از پردازنده‌ی چند هسته‌ای، مدت زمان صرف شده برای محاسبه‌ی وزن‌ها کاهش یابد.

از آنجایی که محاسبه‌ی وزن هر پارتيکل از بقیه پارتيکل‌ها کاملاً مستقل است، به راحتی می‌توان این پردازش را به صورت موازی انجام داد.

برای ۵۰۰ پارتيکل روی یکی از نقشه‌های تستی، بدون موازی‌سازی، این فرایند روی کامپیوتر ما چیزی حدود ۸ ثانیه زمان می‌برد. با اضافه کردن پردازش موازی این زمان به ۲ ثانیه کاهش یافت. (ممکن است با انتقال این محاسبات به پردازنده گرافیکی، بتوان سرعت بسیار بیشتری هم به دست آورد، اما در این پروژه به موازی‌سازی روی سی‌پی‌یو اکتفا شده است.)

#### + تقارن نقشه:

نقشه‌هایی که فضاهای مشابهی در جاهای مختلفش وجود داشته باشد، مکان‌یابی را دشوارتر می‌کنند. چرا که مشاهدات ربات برای مثال با دو مکان مختلف نقشه همخوانی دارد و هیچکدام از این پارتيکل‌ها حذف نمی‌شوند.

این امر باید تا جایی ادامه پیدا کند که ربات شواهدی را در جایی از نقشه ببیند که فقط با یکی از این دو فضا همخوانی داشته باشد، تا در آنجا پارتيکل‌های فضای دیگر حذف شوند و مکان‌یابی خاتمه یابد.

در چنین نقشه‌هایی روش‌های پیشرفته‌تر برای مسیریابی (مانند آنچه در قسمت SLAM گفته شد) بهتر جواب می‌دهند.

یک حالت **extreme** این مسئله، نقشه‌ای است که کاملاً متقارن باشد. در چنین نقشه‌ای ربات هیچ‌گاه نمی‌تواند موقعیت خود را از بین چند موقعیت احتمالی به دست بیاورد، چرا که تمام شواهدی که ممکن است ربات به دست بیاورد با چند فضا قابل تطابق است.

در چنین محیط‌هایی، باید دست‌کم یک عامل که تقارن را بر هم بزند به نقشه اضافه شود، تا ربات با رفرنس قرار دادن آن عامل، بتواند موقعیت خود در نقشه را به دست بیاورد.  
به عنوان مثال، تصویر زیر یک نقشه‌ی متقارن را نمایش می‌دهد.

این نقشه از چهار فضا تشکیل شده است که هر چهار فضا دقیقاً مشابه یکدیگر هستند. ربات توانایی تفکیک این نقاط را ندارد. در صورتی که الگوریتم به اندازه‌ی کافی طولانی اجرا شود، به مرور بر حسب شانس ممکن است نمونه‌ها در برخی فضاهای بیشتر متمرکز شوند و در نهایت ربات یکی از این چهار فضا را خروجی دهد.

برای مثال در شکل زیر، پس از حدود یک دقیقه اجرای الگوریتم، پارتيکل‌های موجود در فضای پایین راست از بین رفته و پارتيکل‌های موجود در فضای پایین چپ هم کمتر شده است و تا مدتی دیگر حذف خواهد شد.

این حذف شدن تصادفی است. (بر حسب اتفاق چند نقطه بیشتر در یکی از ناحیه‌ها ایجاد شده و در resampling‌های بعدی نقاط دیگر را به سمت خود می‌کشد).

