Enhancing Trajectory Planning in Medical Robotics with Potential Fields: Leveraging Tensors for Accelerated Speed

Vincent Clifford
Technical University of Munich

Clinical Application Project Documentation - July 2024

Abstract

In the advancement towards digitizing operating rooms, a fundamental aspect involves the creation of a virtual representation of the operating space, enabling seamless navigation for surgical robots within it. In this clinical application project, I focused on navigating medical robots to the operating position. I will create a real-time virtual operating space from RBGD cameras in my bachelor thesis. Furthermore, I will address translating the trajectory planning of medical robots within this virtual environment.

1 Introduction

This project focuses on a medical robot's initial navigation stage – planning a safe and adaptable trajectory to the operating position. While the intricate movements of surgical instruments during the procedure are reserved for future development, trajectory planning in medical robotics presents unique challenges compared to navigation problems in other environments. Here, guaranteeing safety and adaptability within a dynamic environment is crucial.

In navigational problems in different settings, alternative trajectories might come at a higher cost but are still valid and reasonable routes. In the medical setting, this is not the case. Alternative trajectories of the robot might jeopardize the entire medical operation. Therefore, navigation in operating rooms must meet certain specific requirements. In the initial navigation stage, this requirement is the guarantee that the robot will avoid collisions with obstacles, especially the surgeon, at any cost. In addition, the robot's path should be smooth.

2 Potential Fields

Guaranteeing these requirements is not easy with traditional path-planning approaches. This project explores a novel approach specifically tailored for the delicate environment of the operating room – trajectory planning using potential fields. The idea of potential fields stems from physics, representing regions of influence exerted by forces. In the context of robot navigation, we can leverage this concept to create an artificial force field around the robot and obstacles within the workspace. Attractive and repulsive forces guide the robot's path toward the target location while ensuring it remains a safe distance from any potential collisions. With this approach, we aim to leverage the following benefits:

- Generation of smooth trajectories, following the laws of physics
- Real-time adaptability for multi-agent systems
- Short processing times

In our medical application, the potential field utilizes a single attractive force. This force originates from the robot's desired final position within the operating space, also referred to as the goal or target location. Conversely, all obstacles within the operating room generate repulsive forces. The strength of this repulsion force from each obstacle depends on two key parameters:

- *Distance of Influence*: This parameter defines the maximum distance at which the robot still experiences the repulsive force from an obstacle.
- *Attraction Strength*: This parameter determines the magnitude (amount) of repulsion exerted by the obstacle on the robot.

The work of Nasseri et al. [1] provided the foundation for exploring potential fields in trajectory planning. Wu et al. [2] propose concrete repulsive and attractive functions dependent only on the robot's position and the obstacle's position for calculating the attraction and repulsive force.

3 Modelling Forces

However, directly applying Wu et al.'s [2] proposal to the medical environment presented some challenges and unwanted

trajectories.

3.1 Repulsive Force

One challenge arose from the rate at which the repulsive force from obstacles decreased with distance. In Wu et al.'s [2] proposal, the rapid repulsive force falloff potentially caused the robot to "realize" an impending collision too late for safe course correction. This resulted in jerky movements as the robot attempted abrupt adjustments to avoid obstacles within the confined and delicate environment of the operating room.

Hence, a new repulsive force function had to be defined. We came up with a new similar function that mitigates this issue. In our trajectory planning algorithm, we use the following function to calculate the repulsive force for a single obstacle, with index i, at position $(x_{\text{obstacle}}, y_{\text{obstacle}})$:

$$F_{\text{singleRepulsive, i}}(x_{\text{robot}}, y_{\text{robot}}) = c' * e^{-c'' * \sqrt{(x_{\text{robot}} - x_{\text{obstacle}})^2 - (y_{\text{robot}} - y_{\text{obstacle}})^2}} \quad (1)$$

Parameters of an obstacle:

- $c' > 0$: Attraction strength of obstacle. The bigger this constant, the bigger the magnitude of the repulsive force.
- $c'' > 0$: Distance of influence of obstacle. The higher this constant is, the faster the drop-off of the repulsive force.

A key distinction between our proposed function and the one defined by Wu et al. [2] lies in how it handles the repulsive force based on the distance to the obstacle. Unlike Wu et al.'s [2] approach, our function eliminates the need for case-based calculations depending on the robot-obstacle distance. This simplification unlocks the power of mathematical objects called tensors, enabling parallel computation of the potential field value. In order to achieve real-time planning in the dynamic environment of the operating room, we prioritize parallelization of the distance calculation algorithm. This optimization is crucial for meeting the performance requirements of this application. We will discuss the further performance gains offered by utilizing tensors for computation in Section 6. For now, we first establish the core functionality of the algorithm before delving into real-time optimization strategies.

Wu et al. [2] make a case distinction under the following condition and return the repulsive force value of 0 iff.

$$\sqrt{(x_{\text{robot}} - x_{\text{obstacle}})^2 - (y_{\text{robot}} - y_{\text{obstacle}})^2} > \textit{distance of influence} \quad (2)$$

Recall, *distance of influence* is a parameter of each individual obstacle. By design, my proposed function, constructed without the need for case distinctions and relying on an exponential function, never returns a repulsive force value of precisely 0. However, in scenarios where Wu et al.'s [2] function would yield 0, our offers a value infinitesimally close to 0. This small value is neglectable in path planning and enables parallel computation of the repulsive force value. Note that this property only holds for rather large *distances of influence*. This is sufficient because most obstacles in the operating room don't have small distances of influences.

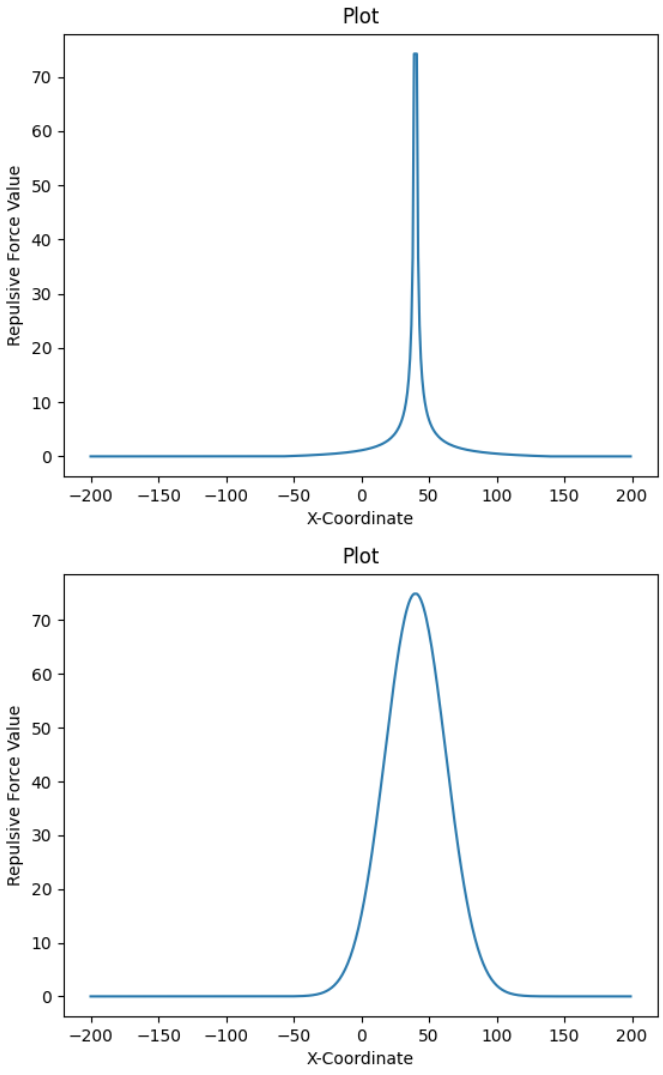


Figure 1: Comparison between Wu et al.'s and our function.

Unlike the inverse-square law observed in physics and used by Wu et al. [2], which leads to undesirable jerkiness in trajectory planning, our proposed repulsive function leverages an exponential distribution. This choice avoids the abrupt dropoff of a function following the inverse-square law while maintaining a rather "inverse-squared-like shape" compared to a linear function. As a result, the exponential function balances responsiveness to nearby obstacles with computational efficiency for path planning.

This can be seen best by an example in Figure 1. Consider the one-dimensional environment depicted in Figure 1, where the only obstacle is at $x = 40$. This figure compares two repulsive value functions used for robot navigation. The upper image shows the function proposed by Wu et al. [2] Here, the repulsive force value is highest directly at the obstacle ($x = 40$) and rapidly decreases as the agent moves away. In contrast, the lower image depicts our proposed function. While it also assigns the maximum force at the coordinates of the obstacle, the repulsive effect extends over a greater distance. This smoother falloff helps trajectory planning algorithms generate smoother, less jerky paths to navigate around the obstacle.

Within the operating room environment, numerous obstacles are present. We can sum each *i-th* object's individual repulsive force, $F_{\text{SingleRepulsive, i}}$, to efficiently compute the entire repulsive force, $F_{\text{TotalRepulsive}}$, at a specific location. Assuming we have n objects, the total repulsive force can be expressed mathematically as:

$$F_{\text{TotalRepulsive}}(x_{\text{robot}}, y_{\text{robot}}) = \sum_{i=1}^{n} F_{\text{SingleRepulsive, i}}(x_{\text{robot}}, y_{\text{robot}}) \quad (3)$$

3.2 Attraction Force

Wu et al. [2] proposes a unique solution to model the attraction force. They essentially flip the attraction force and model it using a repulsive function.

This can be explained best by an analogy between a magnet and the robot: Imagine a magnet at the target and one on the robot with opposite poles. In this setting, the magnet would pull the robot towards the goal. This is how an attraction force is typically thought of. Now, let's draw a line between the robot and the target. Imagine placing a new magnet directly on this line, a bit behind the robot. Fictively, the magnet is "chasing/pushing" the robot to the goal. The imaginary magnet has the same pole orientation as the one on the robot. Due to the same poles of the robot and the robot behind the magnet, the robot is pushed to the goal, resulting in the same movement of the robot.

Wu et al. [2] model the attraction force as a force that pushes the robot toward the goal, as described above. Figure 2 shows an example of this, a one-dimensional environment with only a target at $x = -50$.

Our project uses the core concept of the attraction force function proposed by Wu et al. [2]. However, we've introduced a simplification to enhance efficiency. In the original function, a distinction was made in the calculations based on the robot's distance to the goal. While this can provide more nuanced results, we found that the impact on the overall trajectory path was minimal. By removing this case distinction, we can streamline the calculation process. This allows

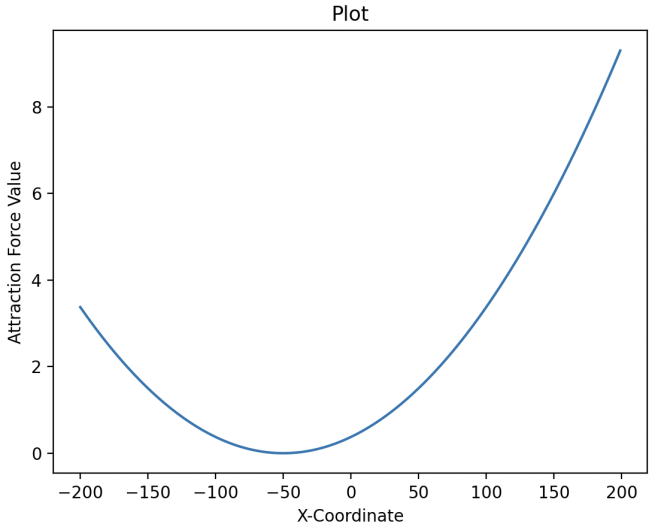


Figure 2: Attraction force value plot for a 1D environment with the target located at $x = -50$.

us to compute the attraction force value for any location much faster, making our implementation more efficient without significantly compromising the outcome. In this project, the following function was used to calculate the attraction force of a robot given a certain position. c' being the attraction strength of the target.

$$F_{\text{Attraction}}(x_{\text{robot}}, y_{\text{robot}}) = c' * ((x_{robot} - x_{obstacle})^2 + (y_{robot} - y_{obstacle})^2) \quad (4)$$

Given that the attractive force is effectively negated and hereby transformed into a repulsive force, the total force acting on the robot at a specific location can be obtained by simply summing the repulsive and attractive force contributions. Mathematically, this is expressed as:

$$F_{\text{Total}}(x_{\text{robot}}, y_{\text{robot}}) = F_{Attraction}(x_{robot}, y_{robot}) + F_{TotalRepulsive}(x_{robot}, y_{robot}) \quad (5)$$

This summation is visualized in Figure 3, which depicts the total force acting on the robot as the combined effect of the previously defined attractive force (Figure 1) and repulsive force (Figure 2).

4 Distance Calculations

In path planning simulations, calculating distances between the robot and surrounding elements is crucial for both attractive and repulsive force functions. While we typically model the robot as a circle for simplicity, real-world operating rooms

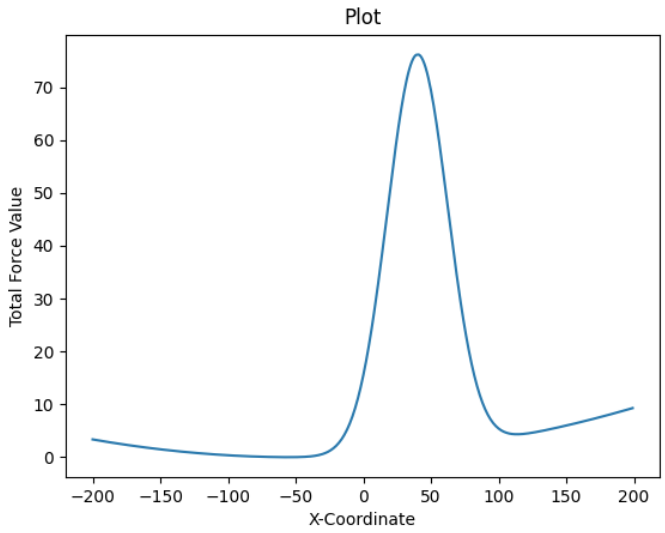


Figure 3: Total force value plot for a 1D environment with an obstacle at $x = 40$ and the target at $x = -50$.

contain obstacles with various shapes. We model obstacles as circles or polygons to account for this complexity. Determining the distance between the robot and a circular obstacle is straightforward. We can calculate it by finding the Euclidean distance between the centers of both circles and then subtracting the sum of their radii:

$$\tilde{d}(x_r, y_r, r_r, x_c, y_c, r_c) = \sqrt{(x_r - x_c)^2 + (y_r - y_c)^2} - r_r - r_c \tag{6}$$

$$d(x_r, y_r, r_r, x_c, y_c, r_c) = \begin{cases} 0, & \text{if } \tilde{d}(x_r, y_r, r_r, x_c, y_c, r_c) \leq 0 \\ \tilde{d}(x_r, y_r, r_r, x_c, y_c, r_c), & \text{else} \end{cases} \tag{7}$$

Unlike circles, calculating the distance between the robot and a polygonal obstacle presents a greater challenge. Here, we need to identify the closest edge of the polygon to the robot's position. Our approach involved exploring various methods and iterations to determine the most efficient way to calculate this minimum distance.

As a first approach, we aimed to expedite the initial concept validation by leveraging existing *Python* libraries for point-to-polygon distance calculations. Since our trajectory planning utilized Python, this approach offered readily available functionality and reduced development time on this aspect. This strategy proved valuable for a rapid proof-of-concept, allowing us to demonstrate the effectiveness of potential fields in the operating room environment within a few weeks.

However, this approach presented a significant performance bottleneck. Computing the repulsive force for 17 polygons

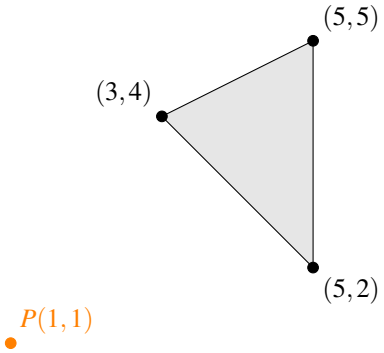


Figure 4: Polygon with vertices $(3,4)$, $(5,5)$, $(5,2)$, and an orange point at $(1,1)$.

(each with 3 edges) within an 800×640 environment, which included computing the distance between the robot and each polygon, resulted in computation times exceeding 1 minute. This latency is unacceptable for real-time path-planning applications. Driven by the need for real-time performance, we developed a custom distance calculation algorithm. This in-house solution addressed the limitations of existing libraries and enabled parallelization, a critical factor for achieving real-time responsiveness.

We will illustrate our methods for computing distances using a concrete example. This example is visually represented in Figure 4. In this scenario, we aim to determine the distance between a robot positioned at $(1,1)$ and a polygon comprising three edges, defined by the vertices $(3,4)$, $(5,5)$, and $(5,2)$. All our proposed techniques involve computing the distance to each edge of the polygon and subsequently selecting the minimum distance. How the distance to each edge is computed varies by our approaches.

Our initial approach involved the following steps: For each pair of neighboring vertices A and B, such as $(3,4)$ and $(5,2)$, we perform the following steps:

1. We determine the slope of the line connecting the two vertices. For this particular pair, the slope is calculated as $t = \frac{\Delta y}{\Delta x} = \frac{2-4}{5-3} = -1$.

2. With the slope in hand, we proceed to compute the slope of the line orthogonal to the edge's line. Remember: Two lines, l and l', are orthogonal if and only if their slopes satisfy the following equation: $slope_l \times slope_{l'} = -1 \Leftrightarrow slope_{l'} = \frac{-1}{slope_l}$. Therefore, to determine the slope of the line orthogonal to the edge's line, we calculate $\text{slope}_{\text{orthogonal}} = \frac{-1}{\text{slope}_{\text{edge}}}$. In our example, this yields $\text{slope}_{\text{orthogonal}} = \frac{-1}{-1} = 1$.

3. Next, we derive the equation of the orthogonal line passing through the robot's location. Substituting the robot's coordinates and the slope of the orthogonal line into the general equation for a line, $y = mx + t$, we find

$1 = 1 \times 1 + t \Leftrightarrow t = 0$. Thus, the equation for the orthogonal line is: $y = 1x$.

4. Similarly, we calculate the equation for the edge bounded by the vertices A and B by inserting the calculated slope from step 1 and the coordinates of A and B. For example, inserting the vertex A, will yield: $y = mx + t \Leftrightarrow 4 = -1 \times 3 + t \Leftrightarrow t = 7$. Thus, the equation for the edge is: $y = -2x + 7$.

5. We then find the intersection point between the orthogonal line and the edge's line: $1x + = -1x + 7 \Leftrightarrow 2x = 7 \Leftrightarrow x = 3.5$. The intersection point is $P(3.5, 3.5)$.

6. Finally, we determine the distance of the robot R to the intersection point P using the Euclidean distance formula: $d = \sqrt{(3.5-1)^2 + (3.5-1)^2} \approx 3.54$.

However, we abandoned this approach to calculate distances to polygons rather quickly for two reasons:

- If a polygon had a vertical edge, our approach would fail to calculate the distance correctly. This comes from step 1 of the previously described algorithm. The slope of the edge's line would not be defined because $\Delta x = 0$, and we would therefore divide by 0. To overcome this issue, we must make a case distinction based on a polygon's x-coordinates of neighboring vertices. This is undesirable because it introduces case distinctions and complexity. Parallelizing this algorithm would, therefore, be less efficient due to pipeline stalls or wrong pipeline predictions.
- The algorithm calculates the distance to an edge by creating an orthogonal line from a point to the edge's line. However, this orthogonal line might intersect with the edge's line beyond the actual edge segment defined by the two vertices (endpoints) of the line. If this happens, the calculated distance might not reflect the true distance between the point and the edge itself. There is no indication in our algorithm that informs us whether this is the case.

Building upon the insights from the previous analysis, the second iteration of the algorithm incorporates vector calculations for a more robust and efficient approach. This revised method forms the foundation of our current trajectory planning implementation. To illustrate the workings of this vector-based approach, we revisit the example scenario depicted in Figure 4. Furthermore, Figure 5 provides additional visualizations to enhance understanding. The revised algorithm computes the distance to a single edge bounded by the vertices A and B and a point P as follows. We will demonstrate this with the example points $A(3,4), B(5,2)$ and $P(1,1)$ again.

1. Calculate the vector $\vec{AB}$. For this particular pair $\vec{AB} = (2,-2)^T$.

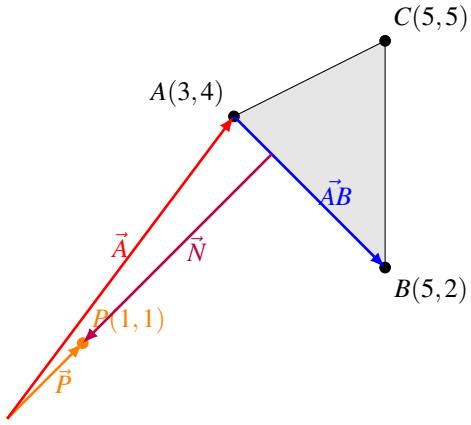


Figure 5: Polygon with vertices $(3,4)$, $(5,5)$, $(5,2)$, and an orange point at $(1,1)$, with the vector $\vec{P}$ from $(0,0)$ to $P(1,1)$.

2. Determine the normal (orthogonal) vector to $\vec{AB}$. In two-dimensional space, this can be achieved by swapping the components of the vector and changing the sign of one entry. Therefore, $\vec{N} = (2,2)^T$.

3. Solve the equation $\vec{A} + i \cdot \vec{AB} = \vec{P} + j \cdot \vec{N}$ for i and j. This equation represents finding the coefficients i and j such that the point $\vec{A} + i \cdot \vec{AB}$ lies on the line defined by $\vec{P}$ and is orthogonal to the direction given by $\vec{N}$. In other words, we are determining how many times we need to add the vector $\vec{AB}$ to $\vec{A}$ to reach a point on the line defined by $\vec{P}$ and orthogonal to the direction given by $\vec{N}$.

   The given equation can be formulated as a matrix-vector problem $Ax = b$, enabling solution using standard matrix solvers:

$$\begin{bmatrix} a_x \\ a_y \end{bmatrix} + i \begin{bmatrix} ab_x \\ ab_y \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \end{bmatrix} + j \begin{bmatrix} n_x \\ n_y \end{bmatrix} \tag{8}$$

$$\Leftrightarrow i \begin{bmatrix} ab_x \\ ab_y \end{bmatrix} - j \begin{bmatrix} n_x \\ n_y \end{bmatrix} = \begin{bmatrix} p_x - a_x \\ p_y - a_x \end{bmatrix} \tag{9}$$

$$\Leftrightarrow \begin{bmatrix} ab_x & -n_x \\ ab_y & -n_y \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} p_x - a_x \\ p_y - a_x \end{bmatrix} \tag{10}$$

   For our example we solve:

$$\begin{bmatrix} 2 & -2 \\ -2 & -2 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1-3 \\ 1-4 \end{bmatrix} \quad \Leftrightarrow \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 0.25 \\ 1.25 \end{bmatrix} \tag{11}$$

4. If $i \in [0,1]$, then the point P lies within the 2D space, such that there exists an orthogonal vector $\vec{N}$ to $\vec{AB}$ that originates from a point q on line segment $\overline{AB}$ (defined by vertices A and B) that intersects point P. This point q can be either an endpoint (A or B) or a point located

on the interior of the line segment. Then the distance of the point to the edge is $|j\vec{N}| = \sqrt{(j \cdot n_x)^2 + (j \cdot n_y)^2} = |j| \cdot \sqrt{(n_x)^2 + (n_y)^2}$.

Because $i = 0.25 \in [0,1]$ the distance in our example can be computed as $|j\vec{N}| = \sqrt{(1.25 \cdot 2)^2 + (1.5 \cdot 2)^2} = |1.25| * \sqrt{(2)^2 + (2)^2} \approx 3.54$, the same value as determined by our first method. This is as expected.

Alternatively, if P cannot be reached by an orthogonal vector originating from a point on the line segment defined by the vertices A and B, the closest point on the line segment to P is either A or B. Therefore, the distance can be computed using the Euclidean distance formula: $min\{\sqrt{(p_x - a_x)^2 + (p_y - a_y)^2}, \sqrt{(p_x - b_x)^2 + (p_y - b_y)^2}\}$.

The closest point to a polygon is determined by calculating the distance between the point and each edge of the polygon using the algorithm previously described. The minimum distance among these calculated distances corresponds to the point closest to the original point.

Our current distance calculation algorithm hasn't yet addressed a critical edge case: where the point of interest lies inside the polygon. In this case, the algorithm would incorrectly identify the closest edge as one within the polygon. This is an undesirable outcome.

We implemented an additional check after calculating the distance to the closest polygon edge to address this limitation. This check verifies if the point of interest could reside inside the polygon. A common approach for such a point-in-polygon test involves creating a bounding rectangle around the polygon.

The bounding rectangle serves as a preliminary check to exclude points obviously outside the polygon. We construct this rectangle by finding the maximum and minimum x and y coordinates among all the polygon's vertices. These extrema, which define the opposite corners of the rectangle, may not necessarily belong to the same vertex. If any of the following conditions are met for the point of interest (x_{point}, y_{point}):

- $x_{point} < x_{min}$
- $x_{point} > x_{max}$
- $y_{point} < y_{min}$
- $y_{point} > y_{max}$

then the point lies outside the bounding rectangle and can be definitively excluded from residing within the polygon. This significantly reduces unnecessary computations for distance checks. To optimize performance, we pre-compute the bounding box for each polygon at creation time. This bounding box efficiently encapsulates the polygon's extent. The minimum and maximum x and y coordinates are then stored as attributes of the polygon object itself, enabling efficient retrieval for future calculations. A visualization of this principle can be seen in Figure 6.

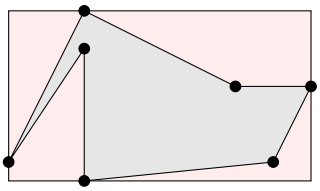


Figure 6: Example of bounding box for sanity check.

If the initial bounding rectangle check fails to exclude the point, a more refined test is necessary to definitively determine whether it lies inside or outside the polygon. This is achieved using a "ray-casting algorithm." This approach involves shooting a ray (imaginary line segment) from a fixed point outside the polygon towards the point of interest. The algorithm then counts the number of intersections between the ray and all the polygon's edges. An odd number of intersections indicates that the point lies inside the polygon, while an even number (including zero) signifies it's outside.

The computational cost of ray casting can vary depending on the chosen origin point for the ray (the point from which it shoots). To optimize performance, we employ a strategic approach that minimizes the number of necessary intersection calculations between the ray and the polygon's edges.

We adopted a strategic choice for the ray's origin point to minimize the computational cost of intersection calculations. Given the polygon's edges and the point of interest, we consistently shoot rays from a fixed location at $x_{start} = x_{min} - 1$ and $y_{start} = y_{point}$. Here, x_{min} is the minimum x-coordinate obtained from the previously computed bounding box, ensuring the ray originates outside the polygon. We subtract 1 from x_{min} for extra certainty. The y-coordinate of the ray's origin is set equal to the y-coordinate of the point in question. This configuration creates a horizontal ray. Notably, determining intersections between a horizontal ray and an arbitrary edge of the polygon is computationally simpler compared to rays with at angle. This optimization significantly reduces the complexity of the ray-casting algorithm.

Determining whether a horizontal ray with the start point $S(x_{start}, y_{start})$ and an edge, bounded by the vertices $A(x_a, y_a)$ and $B(x_b, y_b)$ intersect, can be computed with simple if-statements. The ray intersects with the edge under the following conditions:

- $y_{start} < max\{y_a, y_b\}$
- $y_{start} > min\{y_a, y_b\}$
- $x_{start} < x_a + \frac{y_{start} - y_a}{y_b - y_a} * (x_b - y_a)$

Therefore, to check whether a point resides inside a polygon, we check the above three conditions for all edges. If an odd number of edges fulfill all three previously mentioned conditions, the point lies within the polygon, and the distance to the polygon should be set to zero. Otherwise, it lies outside,

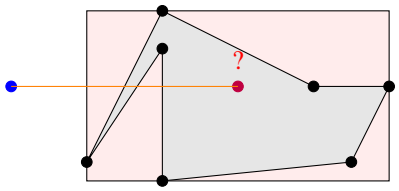


Figure 7: Visualization of ray-casting algorithm. The red point is the point in question. The ray is shot from the blue point.

and the previously computed distance remains valid. Figure 7 visualizes this ray-casting technique.

Now, we can compute the distance to polygon and circle obstacles. Therefore, we can determine each obstacle's individual repulsive force value and thus calculate the total repulsive force for a specific location (x, y). We will return to the topic of calculating distances once we parallelize our entire algorithm. For this, the distance computation has to be altered slightly. For example, we can't get the i value by solving a matrix-vector problem. This does not scale well. We'll return to this topic in Section 6.

5 Learning to Steer: A Primer on Path Planning

So far, we have addressed constructing the repulsive and attraction force value functions and how to calculate distances to objects of different shapes. We have set the groundwork for actual path planning using potential fields.

The general idea of the planning algorithm is as follows: For each point/pixel in our environment, we calculate the potential field value function. Recall that this value is the sum of all the individual repulsive forces and the attraction force acted upon the agent in that location.

Then, we construct a map with these values, assigning each point in 2D space a potential field value. This value represents the amount of repulsion acted upon the agent in this location. One can imagine the potential field value as a measurement of how urgently the robot wants to exit this state. We propose a path-planning algorithm that minimizes the accumulated repulsive force over the entire trajectory path. An example environment with 38 obstacles and the environment's corresponding map, each modeled as a circle, can be seen in Figure 8.

One small adjustment to the potential field value must still be made before performing a search on the map. To ensure that the robot does not collide with any obstacle, we set the potential field value in the location of the obstacle to infinity.

Following the potential field calculation, we employ a pathfinding algorithm on the generated map. The robot's current location serves as the starting point, while the operating table's coordinates define the goal.

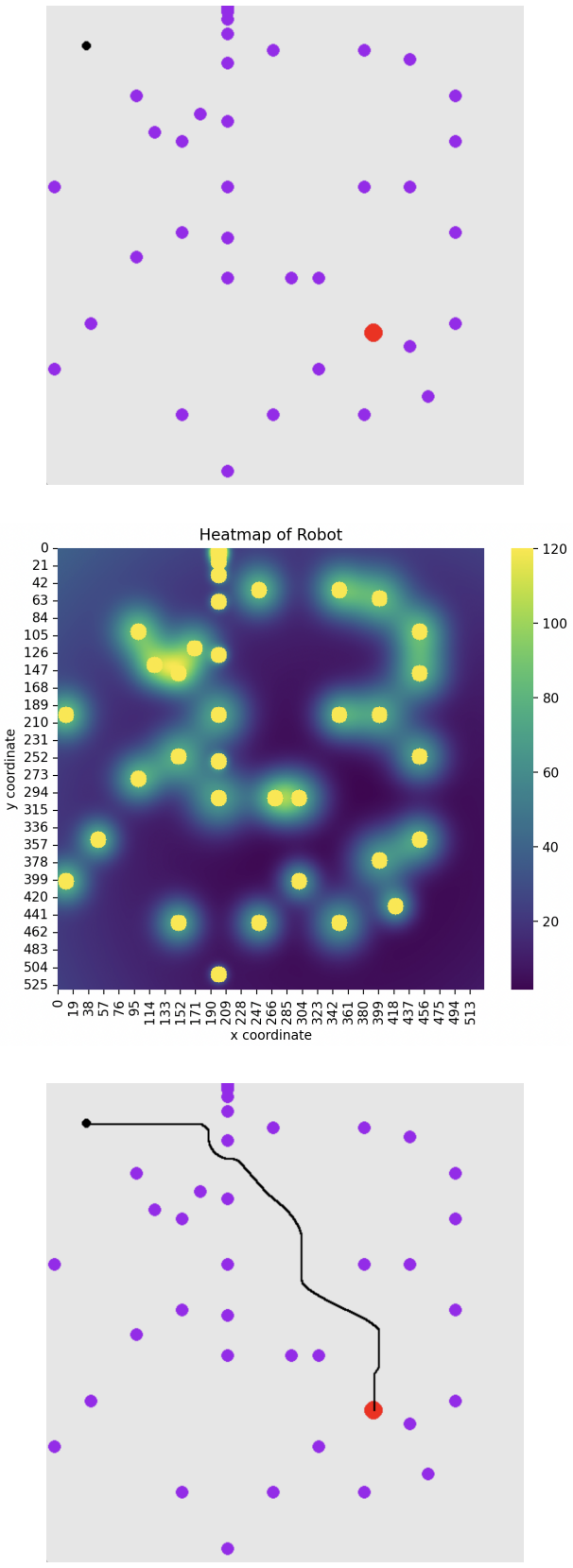


Figure 8: Example environment 1 with 38 obstacles (purple), the corresponding heat map and the agent's trajectory with our proposed algorithm.

The potential field value at a location determines the cost associated with moving the robot to that location. Our algorithm only allows the robot to move north, south, west, and east. Combinations of direction in a single move, e.g., northwest, are prohibited. Then, we perform an altered version of Dijkstra's algorithm on this map. Altered in the sense that if the algorithm expands a node of its frontier with a potential field value of infinity, we abort the path planning. A potential field value of infinity would imply that the path to the location would include a location with infinite repulsive force. Since we want to guarantee that the robot doesn't collide in these situations, we abort the trajectory planning. In this environment, no path to the location exists which does not collide with an obstacle.

6 Shifting Gears: Optimizing the Algorithm for Real-time Performance

This path-planning strategy works well in theory. Implementing the algorithm in practice is another story. The main bottleneck is the computational time needed to compute the repulsive force for each obstacle at each location.

Our initial implementation prioritized rapid prototyping by sequentially calculating the potential field value for each location. This straightforward approach facilitated quick development and effectively validated the core functionality of the algorithm. However, this method suffers from high computational cost. For instance, processing environments like the one shown in Figure 8 or one with only 13 triangles both required approximately 50 seconds to find a trajectory path. Real-time performance is crucial for practical medical applications, as frequent trajectory adjustments are necessary in dynamic environments such as in the operating room. Therefore, a more computationally efficient approach had to be thought of to achieve real-world viability.

While manually multithreading the repulsive force calculations emerged as a potential solution to the computational bottleneck, we ultimately opted against this approach. Multithreading introduces thread overhead, impacting performance. Additionally, manual implementation carries the inherent risks associated with concurrency - race conditions can be notoriously difficult to debug, and ensuring software quality becomes significantly more complex. Furthermore, manually written multithreading may not fully utilize the underlying hardware architecture, potentially leaving performance on the table.

Since *Python* is our choice of programming language, multithreading is especially tricky. Due to the *Global Interpreter Lock*, only one thread can execute bytecode at a time. This means that while you can create multiple threads, they won't truly run in parallel for CPU-bound tasks. The calculation of the potential field value is such a CPU-bounded task.

Our current implementation for repulsive force computation leverages the capabilities of the *PyTorch* framework. While traditionally used for deep learning, *PyTorch* excels at efficient tensor operations. In deep learning, tensors represent the core building blocks of neural networks. *PyTorch* facilitates the weight adjustments ("learning") through operations on these tensors. Crucially, *PyTorch* offers automatic multithreading, making it well-suited for computationally intensive tasks like ours. This even extends to utilizing external CUDA-compliant GPUs for further performance gains.

While the traditional application of tensors lies in deep learning, we've effectively repurposed them to calculate the potential field value efficiently. By leveraging *PyTorch's* tensor-based computations, the entire trajectory path calculation in Figure 8 achieves an average runtime of just 0.75 seconds. This represents a remarkable 65x speedup compared to the sequential algorithm, highlighting the significant efficiency gains achieved with *PyTorch*.

In the next subsections, we will dive into the tensor math for computing the potential field value in less than a second.

The general algorithmic approach with tensors is that tensors calculate the repulsive field values and the attraction field value for all locations in our environment simultaneously. However, our environment contains obstacles modeled by both circles and polygons, which require different algorithms for distance calculations to determine the repulsive field value. Therefore, we need to split the computation for the repulsive field values. We calculate the repulsive field values for circles and polygons separately, then add the repulsive field values together.

6.1 Tensor-Based Repulsive Force Calculation for Circular Obstacles

To establish the mathematical foundation, we define the following variables:

- Environment Bounds: $width$ and $height$ represent the dimensions of our environment. Referring to Figure 8, these values are $width = 800$ and $height = 640$, respectively.
- Obstacles: A list named $obstacles$ stores all circular objects within the environment. Each element, denoted by $obstacles_i.vektor$, holds the coordinates for the *i-th* circular object. The number of circular obstacles is stored in the variable $amount\_circles$ and $amount\_polygons$.

We employ tensors to efficiently compute the sum of repulsive forces exerted by all circular obstacles at each location within the environment. Tensors are essentially multidimensional arrays that offer significant advantages for numerical computations.

The computational steps for calculating the entire repulsive force for all circle obstacles at all locations in our environment at once with tensors are the following:

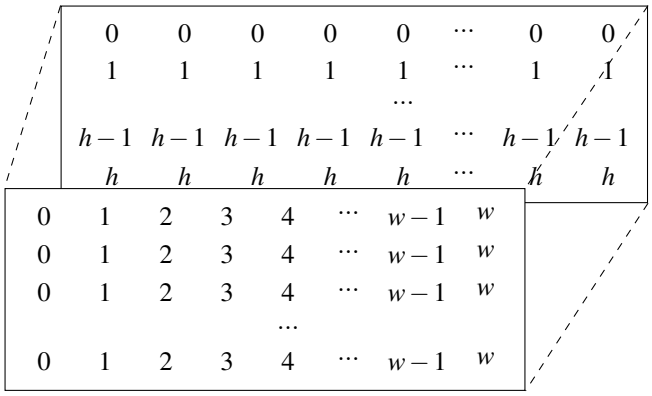


Figure 9: Visualization of the "base" tensor. The size of the tensor is $(height+1) \times (width+1) \times 2$. w and h are the *width* and *height* of our environment, respectively.

1. Create a "base" tensor T with $T \in \mathbb{R}^{(height+1) \times (width+1) \times 2}$. The entries of T are: $\forall i \in [height+1], j \in [width+1] : T_{i,j,0} = i,\ T_{i,j,1} = j$. A visualization of this Tensor can be seen in Figure 9.

2. Define an "obstacle position tensor" $O_p \in \mathbb{R}^{amount\_obstacles \times 2}$ that stores the x and y-coordinate of the i-th obstacle in the *i-th* row of the tensor O_p. Mathematically speaking: $\forall i \in [amount\_circles] : O_{p_i} = obstacles_i.vector$. (Assuming obstacles are zero-indexed).

3. To compute the potential field value, we need to create two additional 1D tensors: "obstacles distance of influence" (O_d) and "obstacle attraction value" (O_a). These tensors, both of size $\mathbb{R}^{amount\_obstacles}$, store the distance of influence and attraction value of the *i-th* obstacle at the *i-th* position, respectively: $\forall i \in [amount\_circles] : O_{d_i} = obstacles_i.distance\_of\_influence,\ O_{a_i} = obstacles_i.attraction$.

4. Now that we have our tensors defined, *PyTorch's* powerful broadcasting capabilities come into play. Broadcasting allows us to perform mathematical operations on tensors of different sizes, simplifying calculations.

   To use *PyTorch's* broadcasting feature, we still have to add dimensions to the previously defined tensors so that the tensor dimensions align according to the broadcasting rules and the math works out. For the tensor T, we add a fictive second dimension (using 0-indexed counting). The size of the tensor now is $[width+1,\ height+1,\ 1,\ 2]$. For the tensor O_p, we add two fictive "zeroth" dimensions, the resulting tensor size being $[1,\ 1,\ amount\_obstacles\ 2]$.

5. After adjusting the dimensions of T and O_p we simply subtract O_p from T: T - O_p. This results in a tensor of size $[width+1,\ height+1,\ amount\_obstacles,\ 2]$. But what does this tensor represent?

   This tensor encodes the difference vectors between each point (x,y) and every obstacle in the list. Each entry in the third dimension represents the vector from point (x,y) to the corresponding obstacle (*i-th* obstacle for the *i-th* entry in the second dimension). The first element within this third-dimension-entry signifies the x-component of the difference vector, while the second element represents the y-component.

6. To determine the distance between each point (x,y) and all obstacles, we employ a two-step process leveraging the pre-computed difference vectors in tensor T.

   First, we calculate the Euclidean distance for each vector in the third dimension of T. This involves squaring each element within a vector (representing the difference between the point and an obstacle) and summing those squares. Essentially, we compute the dot product of each difference vector with itself. This operation collapses the fourth dimension, resulting in a 3D tensor.

   In the second step, we apply the square root element-wise to the resulting 3D tensor. This transformation yields the final distances between each point and all obstacles.

   To access the distance between a specific point (x,y) and the *i-th* circular obstacle, one can perform a lookup at position $T_{y,x,i}$. This retrieves the corresponding distance value from the tensor.

7. We then leverage the pre-processed tensors to calculate the repulsive force exerted by each obstacle on all points (x,y). This involves the following steps:

   - Weighted Influence: We multiply the collapsed distance tensor T with the "obstacle distance of influence" tensor with the help of broadcasting. This emphasizes the influence of closer obstacles on the repulsive force.
   - Exponential Scaling: We raise the resulting element-wise product to the power of e (Euler's number). This introduces a non-linear relationship where closer obstacles contribute more significantly to the force, as described by the 1st function introduced in this paper.
   - Attraction Integration: Finally, we multiply the result with the "obstacle attraction value" tensor. This incorporates the obstacle's inherent attraction strength.

   Now, each value in the Tensor $T_{y,x,i}$ represents the single repulsive force in the location (x,y) generated by the *i-th* obstacle in the obstacle list.

8. We sum over all the single repulsive forces in the second dimension of the tensor T. This will collapse another dimension of our tensor. Hence, T will be 2D after summing in the second dimension. Through this, we added all the single repulsive forces of the obstacles together in order to obtain the total repulsive force for all circular obstacles in a given position (x,y).

9. One final step involves setting the repulsive field values for obstacle coordinates (x,y) and the surrounding obstacle's coordinates to positive infinity. This ensures the path avoids these areas completely, guaranteeing no collisions. This calculation is the only point where the algorithm directly evaluates each location sequentially. To achieve this, we iterate through all obstacles in a list and set the field values in their vicinity to infinity.

6.2 Tensor-Based Repulsive Force Calculation for Polygon Obstacles

Currently, only repulsive forces for circular obstacles have been calculated (as described in Section 6.1). We now focus on extending this to polygonal obstacles.

Similar to the circular case, this algorithm for computing repulsive forces with tensors will also result in a tensor containing the corresponding repulsive field values. Ultimately, these repulsive field tensors for both circular and polygonal obstacles will be added to create a single tensor representing the total repulsive field.

Let's delve into the specifics of computing repulsive force values for polygonal obstacles using tensors. Figuring out the specific algorithm to calculate the repulsive field value for the polygon was the most difficult part of this clinical application project. Many hours went into coming up with a parallelizable algorithm.

We based the first algorithm approach on the sequential algorithm described in Section 4. The core idea is that we only need to compute the distance between each edge of the obstacle and take a minimum of these values. This must be done for all obstacles and all positions in our environment.

We started off by using *PyTorch* to only calculate the distance between a point and all edges of an obstacle in parallel. Evaluating each point in our environment will still be done sequentially.

To do this, we needed to extend the math used in Section 4. Assuming a polygon is bounded by n vertices, we must solve equation (10) n times to get the distance to all edges. Performing this operation in *PyTorch* is not as straightforward as one might assume.

The sequential algorithm mentioned in Section 4 effectively solves a rank-2 system n times. However, since *PyTorch* does not have the capability to solve n 2-rank matrix-vector problems in parallel, we rewrite the n sub-problems to one single matrix-vector problem that can be executed in parallel. This can be done by rewriting the entire problem with simple linear algebra. The solution vector of the rewritten problem will have $2 \cdot n$ entries, where each entry at position $2k$ stores the i value (Recall that the i value and its meaning was addressed in Section 4) of the *k-th* obstacle's edge. At the position $2k+1$ in the new solution vector, the j value of the *k-th* obstacle's edge is stored.

The matrix A of newly formulated problem $Ax = b$ is a 2×2 block diagonal matrix. Each block A_k stores the x and y coordinates of the *k-th* edge vector and the coordinates of the normal vector of the edge vector. This is the same as in equation 10. Vector b is simply a list containing all the individual b vectors from the original n rank-2 matrix-vector problems, where the *k-th* and *(k-th + 1)* entry in the list corresponds to the b vector of the *k-th* subproblem. This is visualized in equation 12.

With this reformulated problem, in order to compute the distance between a point (x,y) and a polygonal obstacle, we formulate the matrix-vector equation as defined in Equation 12 and solve it.

We quickly recognized limitations with this initial approach. Calculating distances to obstacle edges with values of i outside the range $[0,1]$ presented additional complexities and overhead. However, the most significant issue was the scalability of this matrix-vector approach. This approach required allocating a large amount of unnecessary memory.

To illustrate, consider a scenario with a typical environment size of 800×640 pixels and 10 obstacles, each with 10 edges. In this case, we would need to solve roughly 50 million equations in parallel. Fortunately, this isn't a CPU-intensive task due to the matrix structure involved (a collection of 2×2 blocks). Instead, the problem becomes entirely memory-bound. The resulting matrix would be massive, with dimensions of 50 million by 50 million, requiring storage for 25 quadrillion ($25 \cdot 10^{14}$) values. This memory footprint is simply impossible.

Fortunately, not all hope is lost. A mere fraction $\frac{2}{50.000.000} = 0,00000004 = 0,000004\%$ of the entries in Matrix A are non-zero. *PyTorch* does not automatically recognize this sparsity, even when specifying the matrix as a block matrix using the *torch.block_diag()* function. Therefore, it becomes necessary to manually develop a new mathematical formulation that effectively utilizes the problem's characteristics. This new approach should specifically avoid storing the zero values.

We propose an approach based on 1D vector mathematics that addresses the previously described problem by relying solely on element-wise operations: addition, subtraction, multiplication, and division. To explain this approach, we look back at Equation 10. Here, we solved the matrix-vector problem with Gauss-Elimination while explicitly storing all zero values in Matrix A. We propose to solve the problem in an alternative way. From Equation 10 we can derive two equations:

$$\begin{pmatrix} ab_{1_x} & -n_{1_x} & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ ab_{1_y} & -n_{1_y} & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & ab_{2_x} & -n_{2_x} & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & ab_{2_y} & -n_{2_y} & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & ab_{3_x} & -n_{3_x} & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & ab_{3_y} & -n_{3_y} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \ldots & ab_{m_x} & -n_{m_x} \\ 0 & 0 & 0 & 0 & 0 & 0 & \ldots & ab_{m_y} & -n_{m_y} \end{pmatrix} \begin{pmatrix} i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \\ \vdots \\ i_m \\ j_m \end{pmatrix} = \begin{pmatrix} p_{1_x} - a_{1_x} \\ p_{1_y} - a_{1_y} \\ p_{2_x} - a_{2_x} \\ p_{2_y} - a_{2_y} \\ p_{3_x} - a_{3_x} \\ p_{3_y} - a_{3_y} \\ \vdots \\ p_{m_x} - a_{m_x} \\ p_{m_y} - a_{m_y} \end{pmatrix} \tag{12}$$

$$\begin{aligned} &\text{I}: \quad a_x + i \cdot ab_x = p_x + j \cdot n_x \\ &\text{II}: \quad a_y + i \cdot ab_y = p_y + j \cdot n_y \end{aligned} \tag{13}$$

Equation I can be solved for j and then be inserted into II:

$$\begin{aligned} &\text{I}: \quad a_x + i \cdot ab_x = p_x + j \cdot n_x \\ &\Leftrightarrow \quad a_x + i \cdot ab_x - p_x = j \cdot n_x \\ &\Leftrightarrow \quad j = \frac{a_x + i \cdot ab_x - p_x}{n_x} \end{aligned} \tag{14}$$

Inserting 14 *into* II:

$$\begin{aligned} &a_y + i \cdot ab_y = p_y + j \cdot n_y \\ \Leftrightarrow\ &a_y + i \cdot ab_y = p_y + \frac{a_x + i \cdot ab_x - p_x}{n_x} \cdot n_y \\ \Leftrightarrow\ &n_x \cdot (a_y + i \cdot ab_y) = p_y \cdot n_x + (a_x + i \cdot ab_x - p_x) \cdot n_y \\ \Leftrightarrow\ &n_x \cdot a_y + i \cdot ab_y \cdot n_x = p_y \cdot n_x + a_x \cdot ab_y + i \cdot ba_x \cdot n_y - p_x \cdot n_y \\ \Leftrightarrow\ &i \cdot ab_y \cdot n_x - i \cdot ab_x \cdot n_y = p_y \cdot n_x + a_x \cdot n_y - n_x \cdot a_y - p_x \cdot n_y \\ \Leftrightarrow\ &i = \frac{p_y \cdot n_x + a_x \cdot n_y - n_x \cdot a_y - p_x \cdot n_y}{ab_y \cdot n_x - ab_x \cdot n_y} \\ \Leftrightarrow\ &i = \frac{n_x \cdot (p_y - a_y) + n_y \cdot (a_x - p_x)}{ab_y \cdot n_y - ab_x \cdot n_y} \end{aligned} \tag{15}$$

With this reformulation, only 9 operations on single scalar values must be performed to determine the i value of a position (x,y) and an arbitrary edge of an obstacle. This is done without the overhead of storing zero values. To calculate the distance of a position to an obstacle, we require the j value. This j value can be computed similarly with only scalar operations:

$$\begin{aligned} &\text{I}: \quad a_x + i \cdot ab_x = p_x + j \cdot n_x \\ &\Leftrightarrow \quad i \cdot ab_x = p_x + j \cdot n_x - a_x \\ &\Leftrightarrow \quad i = \frac{p_x + j \cdot n_x - a_x}{ab_x} \end{aligned} \tag{16}$$

Inserting 16 *into* II:

$$\begin{aligned} &a_y + i \cdot ab_y = p_y + j \cdot n_y \\ \Leftrightarrow\ &a_y + \frac{p_x + j \cdot n_x - a_x}{ab_x} \cdot ab_y = p_y + j \cdot n_y \\ \Leftrightarrow\ &a_y \cdot ab_x + (p_x + j \cdot n_x - a_x) \cdot ab_y = (p_y + j \cdot n_y) \cdot ab_x \\ \Leftrightarrow\ &a_y \cdot ab_x + p_x \cdot ab_y + j \cdot n_x \cdot ab_y - a_x \cdot ab_y = p_y \cdot ab_x + j \cdot n_y \cdot ab_x \\ \Leftrightarrow\ &a_y \cdot ab_x + p_x \cdot ab_y - a_x \cdot ab_y - p_y \cdot ab_x = j \cdot n_y \cdot ab_x - j \cdot n_x \cdot ab_y \\ \Leftrightarrow\ &j = \frac{a_y \cdot ab_x + p_x \cdot ab_y - a_x \cdot ab_y - p_y \cdot ab_x}{n_y \cdot ab_x - n_x \cdot ab_y} \\ \Leftrightarrow\ &j = \frac{ab_x \cdot (a_y - p_y) + ab_y \cdot (p_x - p_y)}{n_y \cdot ab_x - n_x \cdot ab_y} \end{aligned} \tag{17}$$

Using equations 15 and 17, we have formulated all mathematical expressions to calculate the total repulsive force for all polygonal obstacles at every location in parallel using tensors. The computational steps are as follows:

1. Again, like for circles, create a "base" tensor. However, there is one small difference to the "base" tensor of circles. The individual tensors in the second dimension / each 2D sheet will be stored separately. We call the tensors the X and Y tensor, with $P_x, P_y \in \mathbb{R}^{(height+1) \times (width+1)}$. The entries of P_x and P_y are: $\forall i \in [height+1], j \in [width+1] : P_{x_{i,j}} = i,\ P_{y_{i,j}} = i$.

2. Let the variable *amount_vertices* denote the total amount of vertices in the entire environment. In other words, it's the sum of the polygon's vertices list length.

   Create an "A x-coordinate tensor" A_x and a "A y-coordinate" tensor A_y, with A_x, $A_y \in \mathbb{R}^{amount\_vertices}$. These store the x and y coordinates of all the vertices of all polygons. The "A x-coordinate tensor" comprises the x-coordinates of all the vertices from obstacle 1, followed by the x-coordinates of all the vertices from obstacle 2, and so forth. The "A y-coordinate tensor" contains the same only for the y-coordinates of the obstacles.

3. Define an "AB x-coordinate tensor", an "AB y-coordinate tensor" a "N x-coordinate tensor" and a "N

y-coordinate tensor", labeled AB_x, AB_y, N_x and N_y, all $\in \mathbb{R}^{amount\_vertices}$ respectively. These store the x and y-coordinates of the vectors $\vec{AB}$ and $\vec{N}$, in the same ordering as the tensors A_x and A_y do.

4. To be able to use *PyTorch*$'s$ broadcasting capabilities, we still have to adjust the tensor sizes of the tensors P_x and P_y. We add a new 3D dimension by unsqueezing in the second dimension (Remember: We start to count at 0). We repeat this tensor *amount_vertices* times and stack them together. Therefore, $P_x, P_y \in \mathbb{R}^{(height+1)\times(width+1)\times amount\_vertices}$.

5. Calculation of a new "i-value tensor" and "j-value tensor," I and J, with both $I, J \in \mathbb{R}^{amount\_vertices}$. All of the operations - addition, subtraction, multiplication, and division - are performed cell-wise on the entries of the tensor. These same operations, previously described for single scalar values in Equations 15 and 17, are now applied to a tensor:

   - $I = \frac{N_x \cdot (P_y - A_y) + Y \cdot (A_y - X)}{AB_y \cdot N_y - AB_X \cdot N_Y}$
   - $J = \frac{AB_x \cdot (A_Y - P_y) + AB_y \cdot (P_x - P_y)}{N_y * AB_x - N_x * AB_y}$

6. Computation of a new "n distance" tensor $N_{distance}$. This vector will store the distances of the orthogonal vectors $\vec{N_i}$, whose coordinates are stored in N_{x_i} and N_{y_i}. To calculate $N_{distance}$, we simply take the norm $\vec{N_i}$. Hence:

   - $N_{distance} = \sqrt{(N_x)^2 + (N_y)^2}$

7. Using these tensors, we can now compute the distance to each edge by multiplying the j-value of an edge with its orthogonal vector, $\vec{N}$. The mathematical details for this computation were explained in Section 4. Consequently, we obtain a new tensor D_{edge}, where $D_{edge} = J \cdot N_{distance}$.

8. Now, a tensor that stores the minimum distance of the point to the vertices that bound the edge must be computed.

   For this, we instantiate the same base tensor while computing the distances to circles. Recall this tensor is called T and is of size $[(height+1) \times (width+1) \times 2]$, with entries: $\forall i \in [height+1], j \in [width+1] : T_{i,j,0} = i,\ T_{i,j,1} = j$.

   Additionally, we create a "coordinate tensor for (polygon) obstacles," $C \in \mathbb{R}^{amount\_vertices \times 4}$. It may seem strange that this tensor has a width of 4. However, this is intentional to use the broadcasting capabilities of *PyTorch*. A row in this tensor will store the coordinates of bounding vertices of an edge in our environment. Therefore, $2 * 2 = 4$ entries are required. The order of the entries in this tensor for the bounding vertices must be the same as defined in Step 2.

9. To let the math work out in *PyTorch*, the entries in T still need to be repeated *amount_vertices* times in a newly created 2*nd* dimension. Remember, again, we index our dimensions starting at 0.

10. Compute a tensor "distance to minimum vertex," D_{vertex}. This tensor is computed as follows: $D_{vertex} = (T - C)^2$. We resize that tensor immediately to the size of $[(height+1) \times (width+1) \times amount\_vertices \times 2 \times 2]$. Why do we do this, and what does this tensor now represent?

    An entry $[y, x, i, a, b]$ in this tensor represents the difference of x/y coordinates (depending on whether $b = 0$ or $b = 1$) of the first or second bounding vertex (depending on $a = 0$ or $a = 1$) of the *i-th* edge in our environment to the point (x, y).

11. The only remaining steps to calculate the minimum distance to the bounding vertex of an edge to a point (x, y) are:

    - Sum the tensor D_{vertex} in the fourth dimension. This will collapse the tensor to the size: $[(height+1) \times (width+1) \times amount\_vertices \times 2]$.
    - Take the square root of each element in the collapsed tensor D_{vertex} individually.

      With this operation, we have calculated the Euclidean distance to each vertex in our environment from every point x, y. The distances to the vertices of the *i-th* edge in the environment from point (x, y) are stored in $D_{vertex_{y,x,i,0}}$ and $D_{vertex_{y,x,i,1}}$.
    - To get the distance to the closest vertex of an edge, we simply have to take the minimum in the 3rd dimension. This will collapse the tensor D_{vertex} even further, resulting in a size of $[(height+1) \times (width+1) \times amount\_vertices]$.

12. So far, we have constructed three crucial tensors, which we will use for further work. Before I formalize the next steps, I would like to summarize these components and give an intuition for the next steps:

    - I (i-value tensor): $I_{y,x,z}$ stores the *i-value* of the *z-th* edge in our environment to the position (x, y).
    - D_{edge} (distance to edge tensor): $D_{edge_{y,x,z}}$ stores the distance of the position (x, y) to the *z-th* edge in our environment, regardless of its i-value.
    - D_{vertex} (distance to vertex tensor): $D_{vertex_{y,x,z}}$ stores the distance of the position (x, y) to the closest vertex that bounds the *z-th* edge in our environment.

    All of these tensors are of size of $[(height+1) \times (width+1) \times amount\_vertices]$.

Remember, the goal of all these operations is to calculate a tensor of size $[(height+1) \times (width+1) \times amount\_polygons]$. So far, we have all the components: For every polygon obstacle edge, we have stored the *i-value* in I, $j \cdot \vec{N}$ in D_{edge} and $min\{\sqrt{(p_x-a_x)^2+(p_y-a_y)^2}, \sqrt{(p_x-b_x)^2+(p_y-b_y)^2}\}$ in D_{vertex}.

One now must take the distance values from the tensor D_{edge} or D_{vertex}, depending on the *i-value*.

We create a new tensor D, which represents this filtering. This tensor has the same size as the I, D_{edge} and D_{vertex} tensor. The entries in this tensor are as follows: $D_{y,x,z} = D_{edge_{y,x,z}} \quad iff. \quad 0 \leq I_{y,x,z} \leq 1$. Else the entry is $D_{y,x,z} = D_{vertex_{y,x,z}}$.

13. Now, only the minimum distances of these distances for a polygon must be taken. We want to do this in parallel for every polygon. Here, one must be cautious. Theoretically, we would like a tensor that stores the distances to each edge for every polygon in the 3rd dimension. However, this is not feasible because the amount of edges a polygon has varies from polygon to polygon. Hence, the amount of entries in the third dimension would vary. This is not allowed by *PyTorch*.

    We add "fake/fictive" of $+\infty$ values in the tensor D to address this issue, which will not alter the effect of taking the minimum over the distance values.

    We introduce *max_edges*, a variable that stores the largest number of edges encountered in any polygon within our environment.

    For each polygon, defined by x number of edges, $max\_edges - x$ fictive values of $+\infty$ are added to the tensor D at the according position in the depth (2nd) dimension.

14. The tensor is now reshaped to the size: $[(height+1) \times (width+1) \times amount\_polygons \times max\_edges]$.

15. Afterward, we take the minimum in the 3rd dimension. With this operation, D is collapsed to the size $[(height+1) \times (width+1) \times amount\_polygons]$, which is the final desired distance tensor.

16. Following steps 7 and 8 from section 6.1, we process the final distance tensor: we apply the repulsive field value function and perform summation along the second dimension. This results in a repulsive field value tensor encompassing all polygons within the environment.

17. Lastly, we still have to set all points residing inside a polygon and the coordinates in the vicinity of the obstacle to positive infinity. This is done iteratively with the previously described ray-casting algorithm.

6.3 Tensor-Based Attraction Force Calculation

After having calculated the repulsive field value tensor for both circles and polygons, the attractive field value tensor still has to be computed. This is fairly straightforward compared to the computation of the repulsive field value tensor for polygons:

1. Again, create a "base" tensor T with $T \in \mathbb{R}^{(height+1) \times (width+1) \times 2}$. The entries of T are: $\forall i \in [height+1], j \in [width+1] : T_{i,j,0} = i,\ T_{i,j,1} = j$.

2. Create a "target tensor" $G \in \mathbb{R}^{1 \times 2}$ which stores the x and y coordinate of the goal.

3. Calculate the "distance goal tensor" D_g by computing $D_g = (T-G)^2$.

4. Sum D_g in the 2nd dimension, collapsing D_g to a tensor of size: $[(height+1) \times (width+1)]$.

5. Lastly, multiply the tensor with a coefficient c representing the goals' inherent attraction strength.

After the computation of the repulsive field value tensor for polygons, the repulsive field value tensor for circles, and the attraction force value tensor, these three tensors are added together to yield the final potential field value tensor. We then perform a slightly altered Dijkstra's algorithm on this tensor to find the path that minimizes the total potential field value.

7 Examples

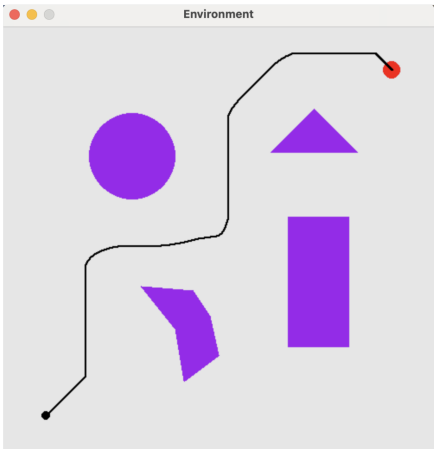
The results of the entire algorithm can be seen in Figure 10. In the left column, you can see the path the robot takes with the previously described algorithm. On the right, the corresponding heatmaps are depicted. The brighter the color of the heatmap is, the stronger the repulsive force is.

I chose these three environments because Wu et al. [2] uses these environment as examples as well. By recreating the exact environment and letting our algorithm find a trajectory, we can compare the results. This is left for future work.

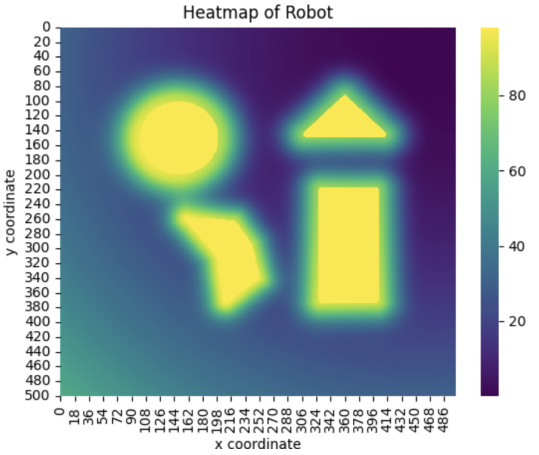
8 Future Work

Since I've only been working on this project since mid-March 2024, there is still a lot of work to do. Future work could include developing an alternative search strategy to Dijkstra's algorithm on the potential field value tensor in order to find an even smoother trajectory.

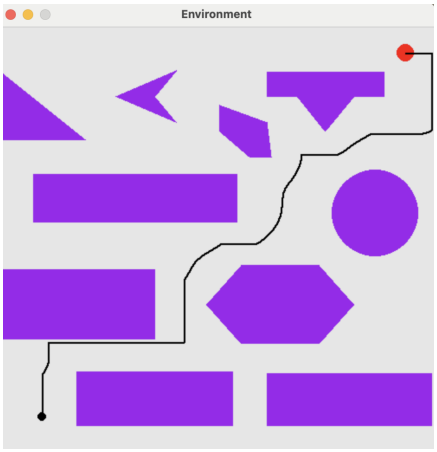
Additionally, there are still many optimizations that can improve the performance of the algorithm. To accelerate the algorithm, we can explore leveraging dedicated GPUs for large tensor operations. Additionally, implementing efficient algorithmic optimizations can yield significant speedups.



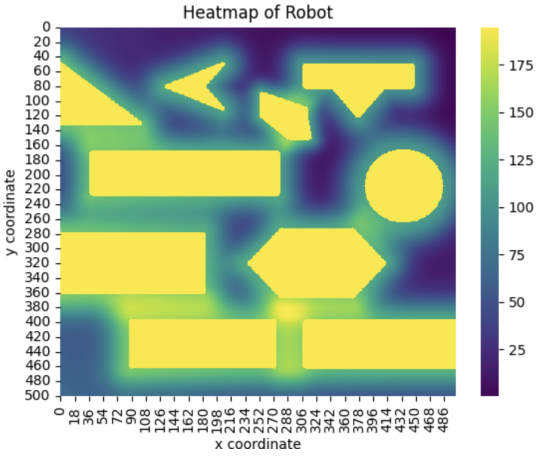
(a) Example environment 2



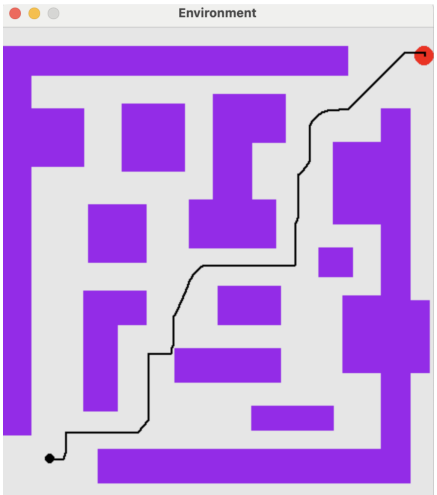
(b) Heatmap environment 2



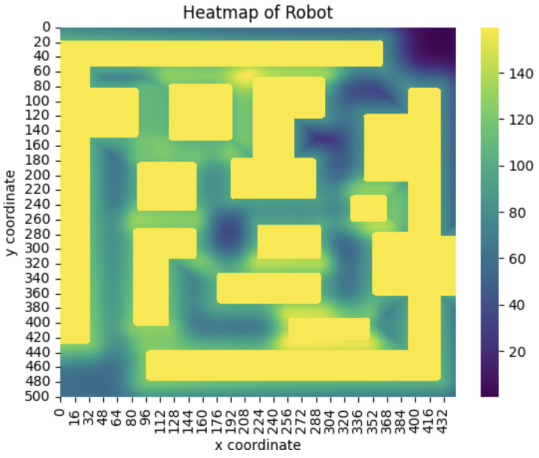
(c) Example environment 3



(d) Heatmap environment 3



(e) Example environment 4



(f) Heatmap environment 4

Figure 10: Three example environments and the corresponding trajectories produced by our algorithm.

So far, we assume that we receive the coordinates of the obstacles and the goal of the operating room. The entire program that retrieves these coordinates from RGBD camera pictures has been neglected so far. This is also a crucial aspect of real-world deployment, as it involves computer vision techniques for object detection and pose estimation in potentially cluttered environments. This is future work for my Bachelor's thesis and will be addressed starting today, after having finished my CAP.

Acknowledgments

I want to express my appreciation to Adjunct Professor Ali Nasseri at TUM for giving me the opportunity to work in his lab. Additionally, I'd like to thank my supervisor, Angelo Henriques, and co-supervisor, Korab Hoxha, for supporting me, as well as all other colleagues who made my time at MAPS extraordinary.

References

[1] Ali Nasseri and Masoud Asadpur. Control of flocking behaviour using informed agents. 2011.

[2] Baoping Jiang Zhengtian Wua, Jinyu Dai and Hamid Reza Karimi. Robot path planning based on artificial potential field with deterministic annealing. 2022.