JACOBS
UNIVERSITY

# Comparison between A* and Dijkstra Algorithm with different Kernel Sizes, Obstacle Weights and Heuristics using Occupancy Grid in an Autonomous Vehicle

by

**Abrish Sabri**

Bachelor Thesis in Computer Science

Prof. Dr. Francesco Maurelli, Dr. Szymon Krupiski & Arturo Gomez Chavez
Bachelor Thesis Supervisors

Date of Submission: August 7, 2019

Jacobs University — Focus Area Mobility

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Abrish Sabri Bremen,Germany, 7 August 2019

## Abstract

The aim of this research paper is to find the shortest and smartest path to relocate the robot in a simulation by comparing the two algorithms: Dijkstra and A*; by comparing different heuristics, changing the size of kernel with an alternation in values of obstacle weights. The smartest path is also observed by overall processing time taken as well as the space and time complexity utilized. The kernel sizes are obtained by downsampling the images and the data through the method of convolution via filter2D. The kernel sizes applied are 3, 5 and 7. In some cases, the heuristics are applied with tie-breaking mechanism. This strategy was initially appealed to tackle the memory fault that occured in the implementation. The path computation is followed through by converting the 1D indexes into 2D through resolution. This path in resolution is written to a YAML file and fed it to ROS to detect the path development thoroughly. Between Dijkstra and A*, A* calculates the optimum path. The shortest path is given by A* with an euclidean heuristic by the kernel size of 3. A confused path is computed when the obstacle weights are modified. Thus, in conclusion it is of crucial importance to choose the optimal weights. Although the time complexity is the same for both algorithms, Dijkstra has better space complexity than A* however, it does not find the optimal solution. A* also calculates the best path much faster than Dijkstra. In few instances, a different result is given as the kernel size increases. In closing, euclidean computes the shortest, safest and the most efficient path 50% of the time.

# Contents

# 1 Introduction

Spatial model of a robot environment is a map and the process to build a map is called mapping. Occupancy grid mapping refers to a group of computer algorithms in probabilistic robotic autonomy for mobile / versatile robots which addresses the complication of generating maps from noisy and uncertain sensor measurement data with the assumption that the robot pose (position and orientation) is known. Data about the environment can be gathered from sensors progressively or be loaded from prior knowledge. Laser range finders, bump sensors, cameras and depth sensors are commonly used to discover obstacles in the robots environment.

Occupancy grid map is an array with occupancy variables. The term occupancy is characterized as a random variable. A random variable is a function that maps the sample space to the real numbers. Each element of an occupancy grid is represented with a corresponding occupancy variable which is an evenly spaced field of binary random variables each representing the presence of an obstacle at that location in the environment. Occupancy grid mapping requires bayesian filtering algorithm to maintain an occupancy grid map. Bayesian filtering applies recursive update to the map. A robot can never be certain about the world so the probabilistic idea of the occupancy rather than occupancy itself is utilized.

Method that is using occupancy grid divides area into cells (e.g. map pixels) and assign them as occupied or free. One of the grid cells is marked as robot position and another as a destination. All the grid cells are independent of each other. The occupancy probability of an occupied grid cell is: $p(m_i) = 1$, if the grid cell is not occupied: $p(m_i) = 0$, and if there is no given knowledge of the grid cell: $p(m_i) = 0.5$. The state of each grid cell is assumed to be static.

Finding the trajectory is based on finding the shortest line that do not cross any of the occupied cells. This is a difficult issue in developing autonomous frameworks in light of the fact that limiting the danger of impacts removes the productivity of most navigation strategies. This problem is liable to vulnerability and incomplete data with respect to the condition of the vehicle, the obstacles, and the responses of the vehicle to inputs. Robust methodologies for safe and efficient navigation require replanning to compensate for uncertainty and changes in the environment. The success of such methodologies is dependent on the nature of detection, planning and control, and on the transient interactions between those tasks [1]. This paper involves comparing two algorithms with additional assistance of kernel sizes, different heuristics and obstacle weights and tested in a simulation that incorporates these sensors to find the most efficient trajectory plan. It focuses on finding a continuous path that can be followed from an initial configuration (or state) to the final configuration. A safe path is one that prevent collisions with obstacles, and an efficient path is one which minimises cost.

For my thesis, I compare Dijkstra and A* with different kernel sizes, obstacle weights and heuristics in the simulation. Dijkstra is a classic Greedy algorithm due to the process of making locally optimal decision such as chosing the closest node and restarting the process; the problem is not broken down into subproblems . A* algorithm is a classic breadth-first-search (BFS) algorithm for finding the shortest path between two points due to its optimisation capability. It processes nodes in increasing order of their distance from the source, which are also called root nodes. The shortest path between two nodes is a path with the shortest length (i.e. least number of edges), also called link-distance.

- Let $G = (u, v)$ be a weighted undirected graph, with weight function $w : E \mapsto R$ mapping edges to real-valued weight. If $e(u, v)$, then we write $w(u, v)$ for $w(e)$.
- The length of a path $p =< v_0, v_1, v_2, v_3 ... v_k >$ would be the total of the weights of its constituent edges as in:

$$length(p) = \sum_{1}^{k} w(v_{i-1}, v_i)$$

- The distance from $u$ to $v$, denoted by $\delta(u, v)$ is the length of the minimum path if there is a path from $u$ to $v$ and $\infty$ is otherwise.

The general idea of Dijkstra algorithm is to report nodes in increasing order of their distances from the root node while constructing the shortest path tree edge by edge; at each step adding one new edge, corresponding to the construction of the shortest path to the current new node. This is accomplished in the following steps:

- Maintain an estimate $d[v]$ of the $\delta(s, v)$ of the shortest path for each node $v$.
- Always $d[v] \geq \delta(s, v)$ and $d[v]$ equals the length of a known path ($d[v] = \infty$ if we have no path so far).
- Initially, $d[s] = 0$ and all other $d[v]$ values are set $\infty$. The algorithm will process the nodes one by one in some order. The processed node's estimate will be validated as being the real shortest distance; i.e. $d[v] = \delta(s, v)$.

The term processing a node u means finding new paths and updating $d[v]$ for all $v \epsilon\ adj[u]$ if necessary. The process by which an estimate is updated is called relaxation. When all nodes have been processed, $d[v] = \delta(s, v)$ for all $v$ [2]. The path computed using the classic Dijkstra algorithm is the shortest; however, it may not be the most feasible.

The A* algorithm follows a technique that makes use of well informed search procedure. It computes the shortest path between two nodes in a graph. While Dijkstra algorithm indiscriminately picks the next node accessible, the A* algorithm uses heuristic that estimates the distance from any node to the target to choose the best node leading to the target. This estimate acts as an archetype for the algorithm and speeds up the calculation.

During the A* algorithm, each node has one of the following three states:

- The node is not known and thus has not been processed therefore there is no path from the root to the current node.
- The node is in the priority queue. Some route prompting current node is known, yet there might be a shorter way.
- The node is completely processed. The shortest path from the starting node to the current one is known.

The algorithm first adds the starting node to the empty priority queue. Each node in the priority queue has a f-value. This value is the sum of the distance from the root node to the current node and the estimate of its distance to the targeted node. The node with the smallest f-value drives the priority queue and will be processed next.The algorithm now takes the node with the minimum f-value from the priority queue until the queue is empty or a path to the targeted node has been found. In the event that the node taken from the queue is the targeted node, at that point the algorithm has found the shortest path

and the program terminates. If the priority queue becomes empty, then no path from the start to the target is possible and the algorithm terminates. Subsequent to processing a node from the priority queue, its neighbors are examined. The algorithm recognizes three cases:

- The neighbor has already been processed at that point the algorithm does nothing
- The neighbor is already in the priority queue, If the current path is an alternate shortcut, update its f-value.
- The neighbor is not in the priority queue. Process the f-value of the node and add it to the priority queue.

Each time when updating the cost of some node, the algorithm saves the edge that was used for the update as the predecessor of the node. Towards the end of the algorithm, the shortest path to each node can be constructed by going backwards using the predecessor edges until the starting node is reached. In the event that a node cannot be reached from the starting node, the cost of that node remains infinite. [3].

Dijkstra algorithm is ensured to locate the shortest path given the input graph. A* algorithm is ensured to find the shortest path if the heuristic is never larger than the true distance. As the heuristic decreases, A* transforms into Dijkstra algorithm. As the heuristic increases, A* transforms into Greedy Best-first Search.

The major disadvantage of Dijkstra algorithm is the fact that it runs a blind search there by consuming a lot of time and in addition a waste of necessary resources and as a result it generally makes it slower than A*. Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path. Dijkstra algorithm has an order of $n^2$ so it is efficient enough to use for relatively large problems.

## 2 Statement and Motivation of Research

In recent years, a whole lot of conclusive research results have been concluded using Dijkstra and A* algorithms to be the most fitting method for path planning in autonomous vehicles. [2] proposes a robotic path planning using a multilayer dictionary which provides more comprehensive data structure for Dijkstra algorithm in an indoor environment application where GPS coordinates and compass orientation are unreliable. Another group of researchers in [4] analyzed node-based optimal algorithms based on 3D path planning. Node based optimal algorithms deal with nodes and arcs weight information. Their task is to find the optimal path through calculating the cost by traversing through the nodes. Essentially, two main examples of node-based optimal algorithms are Dijkstra and A*. Their survey concluded that these type of algorithms cannot be further optimize the result beyond the decomposition of the environment. The results of these kind of algorithms rely much on the preconstructed graph and can be combined with other methods to achieve global optimal. [5] developed an interpolation based method for ideal cost-to-go function based on Dijkstra and A*. It produced an effective method for estimating feedback of a plan and determined the shortest path for motion over a simplicial complex of an arbitrary dimension. The paper also demonstrated that the computational cost is significantly reduced by implementing an A* like heuristic.

There also has been some extensive research comparing the two algorithms A* and Dijkstra in terms of their computational cost, efficiency and simplicity. After considerable amount of analysis of the two algorithms [6] Dijkstra is significantly similar to A*, except there is no heuristic; H is always 0 because of which the algorithm expands out equally in all directions. In other cases, A* scans the area only in the direction of the destination. Thus, Dijkstra ends up exploring a sizeable area before the target is found making it slower than the A*. However, both of the algorithms have their own importance; Dijkstra is mostly used when the destination of the target is unknown while A* is mainly applied when both the source and the destination is known. As an illustration, there is a delivery service unit that needs to pick up an accessory from a branded store. It may know where several stores of that brand are but it wants to go to the closest one. Here, Dijkstra is better than A* because we do not know which one is closest. The only alternative is to repeatedly use A* to find the distance to each one and then choose that path. There are probably countless similar situations where the location whereabouts might be known but not know where it might be or which one might be closest. Therefore, A* is better when we both know the initial point and final point. A* is both thorough (finds a path if one exists) and optimal (always finds the shortest path) if the admissible heuristic function is used.

In the field of path planning for motion in virtual environments and artificial intelligence such as games, there have been some group of researchers focusing on navigation to find the optimized path [[7],[8],[9]]. The authors in [8] compared various path finding algorithms in unmanned aerial vehicles for detecting targets and keeping them in its sensor range in various environments and their performance compared to establish and monitor a path for communication. K. Khantanapoka and K.Chinnasarn [9] compared the path finding algorithms in intelligent environments in order to find the contrast in their time and space complexity. Sathyaraj et al. [7] developed a path planning strategy which let the randomly deployed autonomous robots in the environment move forward till an obstacle is met. For each agent (robot), an estimation of the relative location of the obstacle nearby according to the measurements of infrared sensors were deduced which let the common

agent diverge before collision. Since common agents are used to imitate real people, this moving strategy satisfies the real situation that people walk around. For pursuer, it needs to navigate to specific position when evader is located therefore the researchers applied Dijkstra for path planning.

On the contrary, instead of focusing only on finding the optimized path using the Dijkstra and A*, the proposed method in this thesis uses the sensors implemented in the simulation. This thesis is based on a simulation in the loop (SIL) procedure that incorporates real observations into the simulation in an effortless manner by synchronization of simulated conditions with real-world data. The simulation helps in analyzing and optimizing critical components like robot localization considering the components behavior in deep sea robotic operations and shows the benefit of the presented simulation in the loop approach in the context of DexROV research project. The SIL framework synchronizes simulated and real-world data by incorporating environmental and spatial feedback collected from field-trials which provides an augmented virtual environment reflecting environmental/spatial conditions from real-world missions to test, benchmark and compare behaviors of system modules, preserves the benefits of continuous system integration to perform such benchmarks using real or simulated components or a combination of both and allows to perform tests on distributed deployment, interfaces/pipelines, data regressions/degradation, and fault recovery/safety [A]. This simulation was created by the Robotics research group of Prof. Andreas Birk. It is provided by Jacobs University Bremen under the supervision of Dr. Francesco Maureli, Dr. Szymon Krupinski and Arturo Gomez Chavez.

The proposed method takes advantage of the prior knowledge that the sensors provide such as camera view, position, resolution, height and width of the grid. Since the simulation illustrates an underwater vehicles environment with a vessel nearby, the physics of the vehicle is disabled in order to get a more precise measurement of the mapping for simplicity reasons. Subsequently, making this measurement off-center than the real world calculations. For simplicity, it is assumed that the prior knowledge of the simulation/ map services is believable. The occupancy grid map is based on real-time information and it updates with the detection based on camera image. The contribution focuses on planning the global path for autonomous navigation, however, in this thesis, the focus will be in checking which grid cell is occupied or free in the occupancy grid and the time it takes to find the optimal path from starting position to the goal.

# 3 Representation of the Data

As mentioned above in this paper, the two algorithms implemented are Dijkstra and A* for finding the optimized path for the robot used in the simulation. A generative implementation of these two algorithms could have been constructed by explicitly coding basic rules in some logic or formal grammar. However, due to the utility of the simulation extended expertise was required. It is of crucial importance how data set is represented and what information is retrieved from the data set. Subsections 3.1, 3.2 and 3.3 describe the representation of simulation performed, data collected, and data utilized, respectively. Subsection 3.4 describes the procedures of transforming that data into an image.

## 3.1 Representation of Simulation performed & ROS

Since the simulation is part of the DexRov research project, evidently it is run on the ROS. ROS (Robot Operating System) provides tools and libraries to help software developers simplify the task of creating complex and robust robot behavior across a variety of robot applications. ROS greatly aids in displaying the valuable data needed to perform the task required such as width and height between the travelled distance, resolution, the initial position of the robot and the goal it needs to reach. By running the command rostopic echo /projected_map the data is presented in the form of pose (position and orientation) [1]. The simulation is initiated by launching a vehicle, vessel and octomap launch file; followed by starting Rviz (ROS visualization), a 3D visualizer for displaying sensor data and state information from ROS. It displays live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more. Rviz is an essential part of this section. The robot moves through a joystick and the software displays the live representation of the trajectory mapped by the robot's movement [2].

```
#INITIAL POSITION IN METERS (START)
pose:
  position:
    x: 10.6115179532
    y: -12.484507967
    z: 2.94719725414
  orientation:
    x: -0.00106181641047
    y: 0.00037097801211
    z: -0.943184935889
    w: -0.332266326362
---
```

```
#FINAL POSITION IN METERS (GOAL)
pose:
  position:
    x: 1.21908376738
    y: 4.85488249354
    z: 2.88283420207
  orientation:
    x: -0.000842838538483
    y: -0.000929650401635
    z: -0.67355026858
    w: 0.739140352754
---
```

Figure 1: Data set displayed for the source and destination of the robot through Rviz

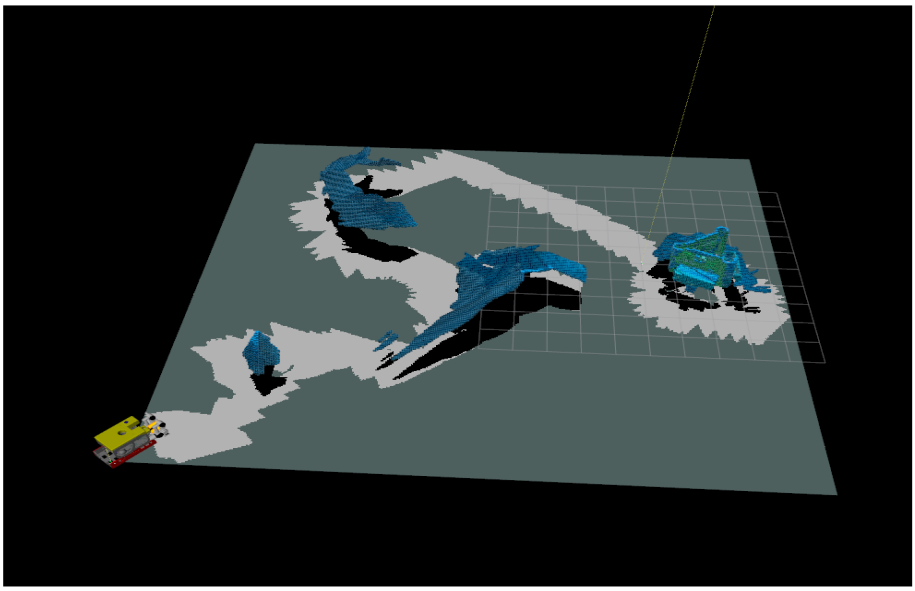


Figure 2: Projected map rendered via Rviz

Gazebo is used to simulate the robot's route. This software produces the actual simulation of the underwater environment where the robot, vessel is shown by a third body. This part of the software involves disabling the physics of the environment. The software acts as an eye while Rviz collects all the data and utilizes this information to produce an occupancy grid and an octomap [3].

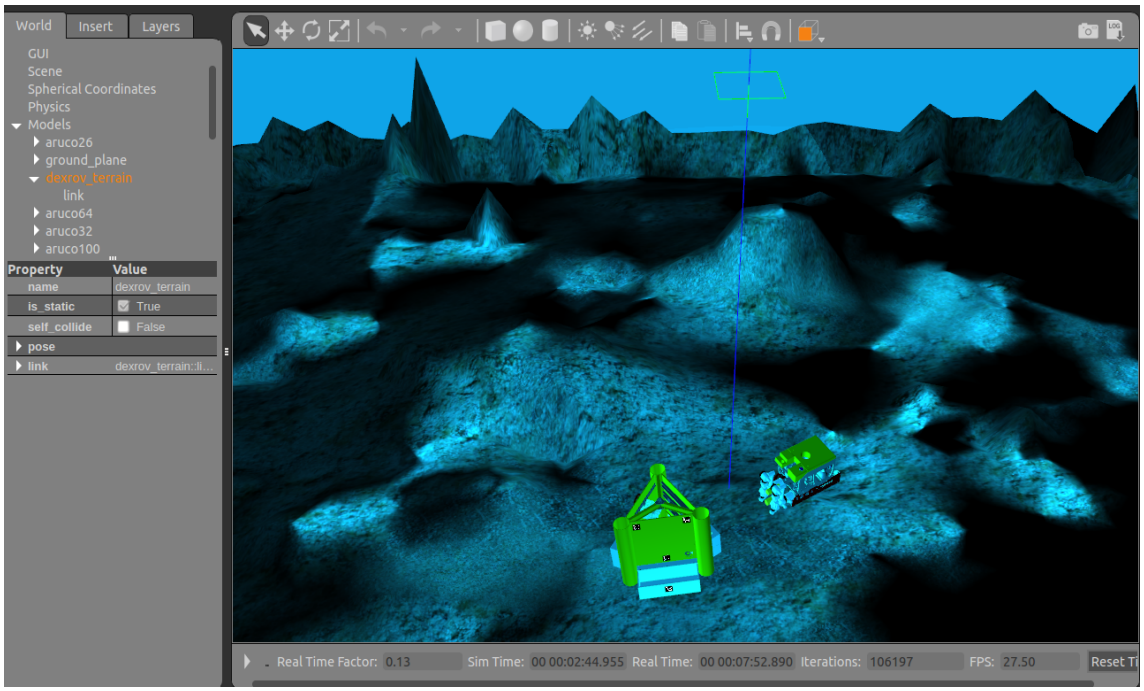



Figure 3: Simulation of the environment via Gazebo

## 3.2 Representation of Data

The data is displayed in left tab of Rviz. It exhibits vital things about the ROS Topics that have been enabled and are currently running. For this simulation, there are at least 150 ROS topics active varying from stereo, depth, and underwater camera / left and right , each having their numerous parameters such as image_raw, state, rotation, etc. Topics containing information about rov, gazebo, octomap, projected map are also active. The data collected for this research exists of stereo camera, ROV model, projected_map, pose, grid, and octomap. Rviz also generates a text file that displays the map as a 1D array. The text file is indeed applied in the usage of Dijkstra and A*. The data obtained in the text file is displayed as an array of 100, 0 and -1. 100 implies that the cell in the map is occupied/blocked, 0 implies that the cell is free/empty while -1 implies that the cell is unexplored. Based on this information the data is further probed on.

## 3.3 Representation of Data Set/Text file

The data set is displayed in a 1D array. The data is converted into a 2D array by using the given values of the width,height and the resolution. This data is then converted to be represented as a graph to be used in the path planning algorithms by traversing through each cell one by one and checking the cell's neighbours. Logically, each cell would have 8 neighbours at most. The graph is formulated by comparing the values between the neighbour cell and the current cell. If the value of the cell in the map is 100, 0 , -1, in the graph it would have a value of 100,1,50, respectively. After creating the graph, the safest shortest path from source to the destination is found through the two path planning algorithms. The path (display of various cells) is extracted from the cells in the map and multiplied by the resolution. These values are stored as X and Y values into a YAML file and used in the launch file.

## 3.4 Representation of Images

In a spatial transformation each point $(x, y)$ of an input image coordinates is mapped to a point $(u, v)$ in a new coordinate system. After algorithms read data files and compute the path, the path is iterated through and each coordinate value is multiplied by the given resolution in order to convert the coordinates into meters. Before converting the coordinates, a rotation operator performs a geometric transform which maps the coordinates $(x, y)$ in an input image onto a position of $(u, v)$ in an output image by rotating it through a specified angle about an origin. Rotation is mostly used to improve the visual appreance of an image.

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} xcos\theta & -ysin\theta \\ xsin\theta & ycos\theta \end{bmatrix}$$

$\theta$ = 90

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

$$x' = -y$$
$$y = x$$

The coordinates in meters are multipled by the resolution of the defined kernel size to scale the image coordinates properly. This method is called scaling.

x = (vecTemp[i].second)*-1
x = x*resolution*KERNEL_SIZE
y = (vecTemp[i].first)
y = y*resolution*KERNEL_SIZE

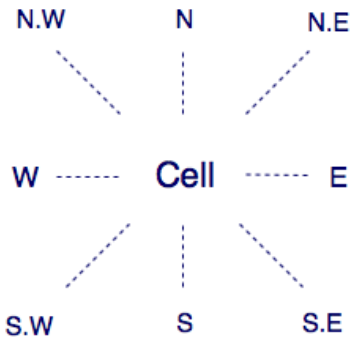
VecTemp: contains the path

The x and y values are stored in another vector that stores the path in resolution in a pair form.

# 4 Method of Implementation

## 4.1 Implementation of Dijkstra:

For Dijkstra, the implementation idea was taken by [6]. Before computing the Dijkstra, the data read by the text file is mapped and then converted into a graph of nodes based on each neighbour of each cell in the map. The data of graph is stored by using vector of vectors. The row of the graph is constructed by multiplying the row of the cell in the map to the max width of the map and adding the column number. The column of the graph is based on the current cell's neighbours position in the map. If the north cell above the current cell exists the value of the column will be computed by subtracting 1 from the row, multiplied by the width and since it is in the same column, the column value will remain the same. Ideally, if we consider north east neighbour of the current cell, the row will be subtracted by 1 and the column will be increased by 1. Each cell has 8 neighbours at most and they are computed as such:




$$Cell - CurrentCell(x, y)$$
$$N - North(x - 1, y)$$
$$S - South(x + 1, y)$$
$$E - East(x, y + 1)$$
$$W - West(x, y - 1)$$
$$N.E - NorthEast(x - 1, y + 1)$$
$$N.W - NorthWest(x - 1, y - 1)$$
$$S.E - SouthEast(x + 1, y + 1)$$
$$S.W - SouthWest(x + 1, y - 1)$$

Considering $i$ as row and $j$ as column for the graph; to compute $i$ and $j$ the formulas followed are:

$i$ = (row of the current cell*width of the map)+column of the current cell

For north neighbour:

$j$ = (row of the neighbour cell * width of the map)+column of the neighbour cell

Written in the program as:

$$i = (row * W) + col$$
$$j = (rneigh * W) + cneigh$$

Note: The formula of $i$ remains the same , $j$ changes according to the neighbour. Evidently, the values of rneigh and cneigh are found out by the position of the neighbour.

Since the real data set consists of $366 * 362$ by $362 * 366$ value set, to test the concept and the working of the algorithm, the test data set (map) taken included $3 * 3$ value set.

| | | |
|---|---|---|
| 0 | -1 | 0 |
| 100 | 0 | -1 |
| -1 | 0 | 100 |

As previously mentioned, $0$,$-1$ and $100$ depicted unique values in the graph. The graph computed is:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 50 | 0 | 100 | 1 | 0 | 0 | 0 | 0 |
| 50 | 0 | 50 | 100 | 50 | 50 | 0 | 0 | 0 |
| 0 | 50 | 0 | 0 | 1 | 50 | 0 | 0 | 0 |
| 100 | 100 | 0 | 0 | 100 | 0 | 100 | 100 | 0 |
| 1 | 50 | 1 | 100 | 0 | 50 | 50 | 1 | 100 |
| 0 | 50 | 50 | 0 | 50 | 0 | 0 | 50 | 100 |
| 0 | 0 | 0 | 100 | 50 | 0 | 0 | 50 | 0 |
| 0 | 0 | 0 | 100 | 1 | 50 | 50 | 0 | 100 |
| 0 | 0 | 0 | 0 | 100 | 100 | 0 | 100 | 0 |

While the graph is filled by converting the map into 2D by using the logic of rows and columns, the graph itself is read by an entirely different logic. The first row represents each cell methodically and the values in the column represents each cells relation to their neighbour cell.

For example: Reviewing the first column: the first cell of the map relation with all other cells; the first value is $0$ because it has no connection with itself of course, the second value is $50$ because the next neighbour is $-1$ and the third value is $0$ because there is no connection between the first cell and the third cell in the map and so on.

$$
\begin{array}{lllll}
0 & \rightarrow & \text{-1} & = & 50 \\
0 & \rightarrow & 0 & = & 1 \\
0 & \rightarrow & 100 & = & 100 \\
0 & \rightarrow & 0 & = & 1 \\
0 & \rightarrow & \text{-1} & = & 0 \\
0 & \rightarrow & \text{-1} & = & 0 \\
0 & \rightarrow & 0 & = & 0 \\
0 & \rightarrow & 100 & = & 0
\end{array}
$$

Using this logic, the graph will always have the size of $H * W$. $H$ and $W$ being the height and width of the map respectively. In this case, if the map is $3$ by $3$, the graph will have the size of $9$ by $9$. The graph will also always be a left diagonal matrix. Dijkstra is applied on the graph and the shortest path from the source to the destination is computed. A separate function for Dijkstra is written in which a set is created to keep track of shortest path tree set meaning nodes for which minimum distance is calculated from the source and confirmed. An array $dist[i]$ having the size of $H * W$ is created to hold the shortest distance from the source to the node for each node. Initially, the shortest path tree set is empty and values of distances for all the nodes are maximum number of the integer (INT_MAX) or INFINITE. For the source node, the distance value is allotted to be 0 so that the source acts as the starting point. The program processes a loop to find the shortest path for all the nodes. It firsts select a node $l$ that does not exist in the set but has a minimum distance from the source and adds it to the set. It iterates through all the adjoining nodes and detects if the sum of $l's$ distance from the source and weight of the next adjoining edge is less than the distance value of iterated node, then it updates the value of that node.

```
 for (int i = 0; i < H*W - 1; i++)
{
   int l = mDist(dist, spt);
   spt[l] = true;
   for (int v = 0; v < H*W; v++)
   {
      if (!spt[v] && graph[l][v] &&
         dist[l] != INT_MAX && dist[l]+ graph[l][v] < dist[v])
      {
         dist[v] = dist[l] + graph[l][v];
      }
   }
}
```

Listing 1: Code implemented to find the shortest path for all vertices

Since the path is in 2D and it represents x and y coordinates, the path is stored in a pair of doubles of vectors of vector. The path is extracted from the map instead of the graph because map depicts the real number of cells. After extracting the path, the x and y values for the simulation are computed by dividing and finding the modulus of the cells position number respectively and multiplied by the resolution value to be portrayed

appropriately for the robots orientation.

```
void Path2D(vector<vector<int> >&v1,vector<vector<pair<double,double> > >&v)
{
   double x,y;
   for(int i=0; i<v1.size();i++)
   {
      vector<pair<double,double> > tmp;
      for(int j=0; j<v1[i].size();j++)
      {
         x = (v1[i][j])%W;
         x = x*-1;
         x = x*resolution*KERNEL_SIZE;
         y = (v1[i][j])/W;
         y = y*resolution*KERNEL_SIZE;
         x = x+ORIGIN_MAP;
         y = y-ORIGIN_MAP;
         tmp.push_back(make_pair(x,y));
      }
   v.push_back(tmp);
   }
}
```

Listing 2: Code implemented to print the path in resolution form.

The first vector of vector of integers $v_1$ is the vector that stores the path in cell(map) values. It has to be an integer because the modulus operator, % (remainder operator) is a binary operator meaning it only takes two operands at a time. It also only manages integer types such as int, short, and long, etc. Thus, another vector of vector of pairs of doubles have to be initialized to store the path in resolution format because the resolution has a value of $0.0500000007451$ which is a double.

The libraries used to get the complete functionality of the code are the following:

#include <stdio.h>
#include <limits.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <iomanip> #include "yaml-cpp/yaml.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

stdio.h header in this research paper is used for the variable type: FILE. An object type to store information for a file stream. fstream and sstream are classes to read and operate on files. Through the combination of these various libraries functions the data from the file is read and stored in the vector of vector of integers which is called map in this paper. iostream uses the objects cout, cin, etc for sending data to and from the standard streams

input and output. These objects are also a part of namespace std. Including the vector header file helps construct the vector and provides with vector functions which makes adding and deleting elements from the vector easier. Vectors are sequence containers that changes size dynamically. Logically they are the same as arrays that can be resized. Internally, vectors also uses dynamically allocated array to store their elements. Vectors are very efficient at accessing its elements and relatively efficient at adding or removing elements from its end comparing to other dynamic sequence containers. It is important to state that operations that involve inserting or removing elements at positions other than the end, vectors usually perform worse than the other dynamic sequence containers. The header <limits.h> provides various properties such as macros defined in the header, and value limit for several variable types. INT_MAX is used several times during implementing the algorithms. This means an integer can store up to a maximum value of $2147483647$. The iomanip header file is used for the setprecision() function. It is used to affect the state of iostream objects. Yaml-cpp header file has a YAML parser and emitter in C++ for the purpose to make and write yaml files. The last two opencv header files are used for image processing and HighGUI module in opencv is responsible for quick and simple GUIs.

Some of the globally defined variables and macros are the following:

```
using namespace std;
using namespace cv;
#define KERNEL_SIZE 3
#define ORIGIN_MAP 15
#define resolution 0.0500000007451;
int H ,W;
vector<vector<int> > map;
vector<vector<int> > graph;
vector<vector<int> >v1;
vector<vector<pair<double,double> > >v2;
```

These vectors were chosen to be defined globally because the vectors map, graph, $v_1$ and $v_2$ and the integers $H$ and $W$ are accessed by multiple functions. The advantages of global variable are that they can be accessed from all the functions defined in the program and the variables are only needed to be declared only once. Global variables general perception is that they are unsafe and provide a high risk to a program's computability. If too many variables are declared as global they will remain in the memory till program is executed and terminated. The data stored as a global variable is also vulnerable to other functions that can easily modify it. However, in this case, map, graph, $v_1$ and $v_2$ are not changed anywhere in the function, the value is the same and with each function the values of these variables are updated which are utilized somewhere else. Similarly, $H$ and $W$ are initialized in the main function but used in a lot of functions that are needed in the computation of the map and the graph. The main problem claimed against global variables are that they are changeable. One can allot the variables as a watchpoint while debugging. It would be impossible to follow this procedure with a local variable as watchpoints are canceled when a variable goes out of scope but with a global variable, the variable can be queried at any time. Lastly, resolution, KERNEL_SIZE and ORIGIN_MAP (starting position of the robot) are manoeuvred as a macro. One significant difference between a macro and a global variable is that macro is not stored in the memory, in this

case, resolution is just a substitute for the value $0.0500000007451$ which is what we need to multiply the values in the code above to get the final result. In this research, three different kernel sizes are used; 3,5 and 7. The chosen kernel size is defined initially, similarly origin map states the initial coordinates of the robot to match the simulation. Macros are also preprocessor directives, their values cannot be changed like variables. Therefore, this method is memory efficient for the defined macros but no so much for the other variables as the variables such as map, graph, etc. need to have memory allocated in order for the data to be accessed, modified and computed.

## 4.2 Implementation of A*

The implementation of A* imitates the same idea as the Dijkstra. The main difference lies in some additional functions. As previously mentioned, A* search algorithm selects a cell based on its f-value which is a variable equivalent to the sum of two other variables; g and h. With each iteration, it selects the cell that holds the minimum f value and process that cell. g is defined as the cost to move from the initial point to the stated cell on the graph, backing the path generated from the starting cell to the given cell. h is defined as the estimated cost to move from the given cell on the graph to the destination cell. This variable is also called heuristic. The heuristic is a simple estimate of the distance between each cell and the destination cell, therefore it is especially important that the computation of H is simple and easy as the value will be determined at least once for each cell before reaching the destination cell. The implementation of this H value varies depending on the properties of the graph being processed. Using a good heuristic is important in determing the performance of A*.

- At one extreme, if h is 0, then only g plays a role, and A* turns into Dijkstra which is guaranteed to find a shortest path.
- If h is always lower than (or equal to) the cost of moving from current node to the goal, then A* is guaranteed to find a shortest path. The lower the h, the more A* expands, making it slower.
- If h is exactly equal to the cost of moving from current node to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can not make this happen in all cases, you can make it exact in some special cases. It is nice to know that given perfect information, A* will behave perfectly.
- If h is sometimes greater than the cost of moving from current node to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if h is very high relative to g, then only h plays a role, and A* turns into Greedy Best-First-Search.

Heuristic is found through various methods of calculations, three common heuristic functions are mentioned below.

1. **Manhattan Distance**
2. **Diagonal Distance**
3. **Euclidean Distance**

These are the approximation methods to compute the value of H, there are methods to compute the exact value of H, however, they are not efficiently utilized because they are generally very time consuming.

The Manhattan distance is computed by summing the absolute values of differences in destination cells x and y coordinates and the current cells x and y coordinates respectively i.e.,

$$h(n) = |n.x - Z.x| + |n.y - Z.x|$$

*n = current cell*
*Z = destination cell*

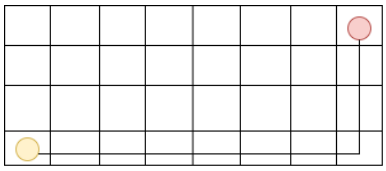


Figure 4: The Manhattan Distance Heuristics.

Assume orange point as the starting node/cell and red point as the destination cell

This is the most simple heuristic method and is ideal for graphs/grids/maps that allow movements in four directions (up,down,left,right)

There are two types of diagonal distance. One being the uniform cost of diagonal planning where the cost of diagonal planning is equivalent to the cost of non-diagonal. It is computed by finding the maximum of absolute values of differences in the destination cells x and y coordinates and the current cells x and y coordinates respectively i.e.,

$$h(n) = c.max(|n.x - Z.x| + |n.y - Z.x|)$$

*n = current cell*
*Z = destination cell*
*c = cost of movement*

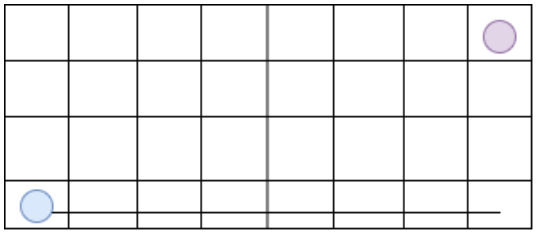


Figure 5: The Uniform Diagonal Distance Heuristics.

Assume blue point as the starting node/cell and purple point as the destination cell

The other diagonal distance occurs when the cost of diagonal planning varies from the cost of non-diagonal.

$$h(n) = c\_d\_min + c\_n(d\_max - d\_min)$$

$$d\_max = max(|n.x - Z.x|, |n.y - Z.y|)$$

$$d\_min = min(|n.x - Z.x|, |n.y - Z.y|)$$

*n = current cell*
*Z = destination cell*
*c = cost of movement*
*c_n = cost of non-diagonal movement*
*c_d = cost of diagonal movement*
*c_d = c_n x $\sqrt{2}$ = c_n x 1.414*

The non-uniform diagonal distance cost is simply computed by summing the minimum cost value of diagonal movement and the differences in maximum and minimum value of cost of non-diagonal movement in the destination cells x and y coordinates and the current cells x and y coordinates respectively. Both of the diagonal distances is used for graphs/grids/maps that allow movement in eight directions.

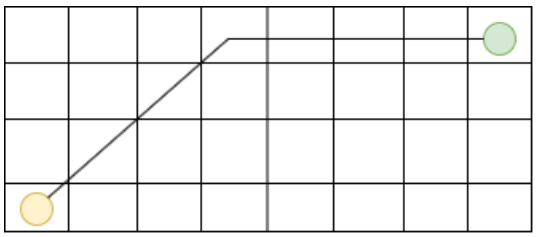


Figure 6: The Diagonal Distance Heuristics.

Assume yellow point as the starting node/cell and green point as the destination cell

The euclidean distance is computed by finding the distance between the current cell and the destination cell using the distance formula. Sum of x and y coordinates in the differences of destination cells x and y coordinates and the current cells x and y coordinates respectively. This distance is used for graphs/grids/maps that allow movement at any angle.

$$h(n) = \sqrt{(n.x - Z.x)^2 + (n.y - Z.y)^2}$$

*n = current cell*
*Z = destination cell*

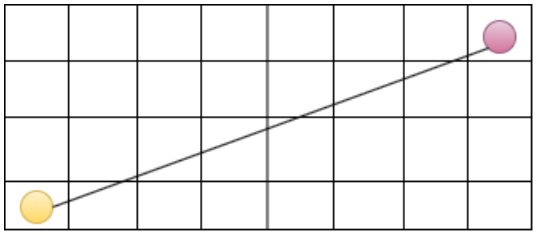


Figure 7: The Euclidean Distance Heuristics.

Assume orange point as the starting node/cell and purple point as the destination cell

The data was tested on different heuristic functions through the test data set. However, this time along with a 3x3 value set, the functions were tested on different data set to have a better estimate such as: 3x6, 4x6, 10x10, 10x16, 16x6.

The results of the path can be found at B

5 out of 6 trials demonstrate similar results for manhattan and diagonal heuristics while 1 out of 6 trails show similarity between euclidean and the others. Euclidean distance is shorter than manhattan or diagonal distance and gives the shortest paths but on hindsight A* takes longer to run. Looking closely, it is evident that euclidean turns at any angle and tries to find the shortest path as compare to manhattan and diagonal which only checks their adjacent cells (nodes) and moves closer to the destination point accordingly.

```
double EuclideanHVal(int row, int col, Pair destination)
{
   //W = row, H = col.
   return ((double)sqrt((row - destination.first) * (row - destination.first) +
       (col - destination.second) * (col - destination.second)));
}
```

Listing 3: Code implemented to find the Euclidean Heuristic function

Note: Usage of Pair makes it easier to extract the first and second elements rather than using a loop or an array. Explained further down below.

Besides the additional heuristic function in the implementation of A*, two more functions were added, one to determine if the source or the destination is blocked and the other to check if the destination has been found. The added libraries to complete the functionality of A* are the following:

#include <cfloat>
#include <cmath>
#include <set>
#include <stack>
#include<string>
#include <utility>

cfloat header defines the characteristics of floating types with their minimal or maximal values FLT_MAX is used several times during implementing the algorithms. This means a float can store up to a maximum value of 1e+37. It is used in storing the values of f,g and h. String header file contains the operations on string such as getline(). The set and stack headers define the set and stack container classes. Utility is a header file which contains params/objects of two different types of values. It is used in make_pair function

The altered global defined variables and macros are the following:

#define KERNEL_SIZE 3
#define ORIGIN_MAP 5
#define OH 366 //original height
#define OW 362 //original width
#define OSx 320 //original source x
#define OSy 40 //original source y
#define ODx 320 //original destination x
#define ODy 300 //original destination y
//resolution values
double res3 = 0.15;
double res5 = 0.25;
double res7 = 0.35;
typedef pair<int, int> $Pair$;
typedef pair<int,int> pathTemp;
typedef pair<double,double> $pathStore$;
typedef pair<int, pair<int, int> > $pPair$;
vector<pathStore> vecPath;

In our implementation, typedef pair $<int, int> Pair$ is initialized twice, one with data type of stack and the other with vector. To make things easier, stack$< Pair > Path$ is utilized for storing the row and column of the path in 1D. vector$< Pair > vecTemp$ stores the same data in 2D, since the data required is in resolution which is a double value, we initialize typedef pair$< double, double > pathStore$ once; equivalent to vector$< pathStore > vecPath$. $vecPath$ stores the path in 2D with the resolution. vector$< pathStore >$ is also used as a function that returns the trace of the path from source to the destination. Defining typedef pair$< int, int > Pair$ and typedef pair$< int, pair < int, int >> pPair$ is globally appropriate for our implementation because $Pair$ itself is used five times in different functions as $Pair\ dest$ and $Pair\ src$. $pPair$ is used for openlist. Most importantly, the major difference between Dijkstra and astar implementation is the added usage of $Pair$ that is because in A* applications the destination and the source is usually known. Keeping this in mind, the source and the destination cells are given in 2D in the main function as x and y coordinates as one Pair which makes it simpler to extract the first and

second components when needed. $std :: stack$ is only initialized once, because stack is not flexible enough to provide several operations as compare to $std :: vector$. In case of $std :: stack$, operations are only performed in a calculated manner, where the elements are only needed to $push()$ above the last element or $pop()$ the last element. While $std :: vector$ has several accessibility and modification operations, elements can be inserted in between or be erased in between. Lastly, vector class underlying data structure is a dynamic array, it represents the list of objects under a resizable array. Vector is also the best choice in retrieval cases.

For open list we use a set data structure of C++ STL ($pPair$) which is implemented as Red-Black Tree. and for closed list we used a boolean hash table for best performance. Also to lessen the time taken to calculate g, dynamic programming is used.

Red-Black Tree is a self balancing binary search tree where every node (cell in our case) pursues specific rules such as the root of tree is always black, a node is either red or black, a red node cannot have a red parent or red child and all leaves are black. Most of this tree operations take O(h) time where h is the height of the tree. If after every insertion and deletion of a node in open list it is guaranteed that the height of the tree remains $O(log\ n)$ then for all other operations such as e.g., search, max, min, insert, delete.. etc it is also $O(log\ n)$. The height of the tree is always $O(log\ n)$ when n is the number of nodes (cells).

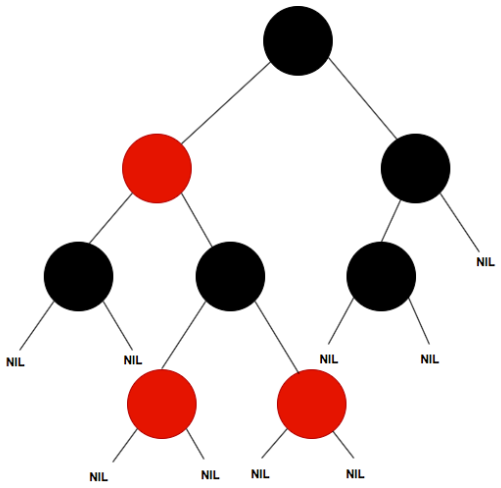



Figure 8: Displays an example of a red-black tree

A struct is also defined to carry some essential parameters

```
struct cell
{
// Width and Height index of its parent
// Note that 0 <= i <= W-1 & 0 <= j <= H-1
int parent_i, parent_j;
// f = g + h
// g = movement cost
```

```
// h = heuristic
double f, g, h;
};
```

This struct is declared as a 2D array to hold the details of a cell in a function that traces the path from the source to the destination cell and in a function that finds the shortest path between the source and the destination cell

$$cell\ cellInfo[W][H];$$

```
vector<pathStore> trace(cell **cellInfo, Pair destination)
{
   printf("\nThe Path is ");
   int RW = destination.first;
   int RH = destination.second;
   double x,y;
   int m,n;
   stack<Pair> Path;

   while (!(cellInfo[RW][RH].parent_i == RW && cellInfo[RW][RH].parent_j ==
       RH))
   {
      Path.push(make_pair(RW, RH));
      int temp_RW = cellInfo[RW][RH].parent_i;
      int temp_RH = cellInfo[RW][RH].parent_j;
      RW = temp_RW;
      RH = temp_RH;
   }

   vector<pathTemp> vecTemp;

   Path.push(make_pair(RW, RH));
   while (!Path.empty())
   {
      pair<int, int> p = Path.top();

      Path.pop();
      vecTemp.push_back(make_pair(p.first,p.second));
      cout << "-> (" << p.first << "," << p.second << ") ";
   }

   cout << endl;

   for(int i=0; i<vecTemp.size(); i++)
   {

      x = (vecTemp[i].second)*-1;
      x = x*resolution*KERNEL_SIZE;
      y = (vecTemp[i].first)*resolution*KERNEL_SIZE;
      x = x+ORIGIN_MAP;
      y = y-ORIGIN_MAP;
```

```
        vecPath.push_back(make_pair(x,y));
    }
    cout << "\nPath in resolution" << endl;
    for(int i =0; i<vecPath.size();i++)
    {
        cout << fixed << setprecision(20)<< vecPath[i].first << " " <<fixed <<
            setprecision(20)<< vecPath[i].second << " " << endl;
    }
    cout << endl;



    Mat imgmat = imread("../images/map_image.jpg");
    Vec3b color;
    color[0] = 0;
    color[1] = 255;
    color[2] = 0;

    for(int i =0; i< vecTemp.size();i++)
    {
        m = (vecTemp[i].second)*KERNEL_SIZE;
        n = (vecTemp[i].first)*KERNEL_SIZE;
        imgmat.at<Vec3b>(Point(m,n)) = color;
    }

    imshow("window",imgmat);
    imwrite("../images/manhattan.jpg",imgmat);
    waitKey(0);

    return vecPath;
}
```

Listing 4: Code implemented to trace the path from the source to the destination cell. Converts the path from 1D into 2D using resolution [10]

The first part of this code contains changing the coordinates of the path into resolution, multiplying those values by the given kernel_size [5.2] and applying mathematical operations relative to the robots origin position to x and y respectively. Before multiplying the x value with the resolution, x and y values are rotated by a $90^{\circ}$ angle in order to display the tracked path from the map to an image. The openCV operations are discussed in detail in Evaluation section of this paper.

A* is implemented by initializing two lists; open list and closed list. Open list is a pair of pair of int while closed list uses a boolean hash table. A* function takes three parameters; the graph, source cell coordinates and the destination cell coordinates. First it checks if the source and the destination coordinates are valid, and whether the source and the destination is accessible and not blocked. The closed list is initialized as a boolean 2D array and initialized to false; stating it is empty.

$$bool\ closedList[H][W];$$
$$memset(closedList, false, sizeof(closedList));$$

The struct cell is declared as:

$$cellInfo[i][j].f = FLT\_MAX;$$
$$cellInfo[i][j].g = FLT\_MAX;$$
$$cellInfo[i][j].h = FLT\_MAX;$$
$$cellInfo[i][j].parent\_i = -1;$$
$$cellInfo[i][j].parent\_j = -1;$$

And then the parameters are initialized as:

$$i = source.first;$$
$$j = source.second;$$
$$cellInfo[i][j].f = 0.0;$$
$$cellInfo[i][j].g = 0.0;$$
$$cellInfo[i][j].h = 0.0;$$
$$cellInfo[i][j].parent\_i = i;$$
$$cellInfo[i][j].parent\_j = j;$$

Open list is created by using typedef pair $< int, pair < int, int >> pPair$ structure that stores the information of f,i and j; where $f = g + h$ and $i$ and $j$ are the row and column index of the cell.

$$set < pPair > openList;$$

Initially open list holds the source cell and its f is given a value of zero. Open list iterates through all the cells inside and finds the cell with the least f. It pops that cell off the open list and generates that cells 8 neighbours and sets the parents of that cell equal to that cell. For each neighbour it checks whether the neighbour is the destination:

neighbour.g = cell.g + distance between the neighbour and cell
neighbour.h = distance from destination cell to the neighbour cell
neighbour.f = neighbour.g + neighbour.h

A* also checks If a cell is in the same position as its neighbour in the open list and has a lower f than the neighbour then it skips that neighbour. If a cell is in the same position as its neighbour in the closed list and holds a lower f value than the neighbour or if it is blocked, the neighbour is skipped, the cell is added to the open list and the parameters of the cell are updated and f,g and h are stored.

```
// To store the 'g', 'h' and 'f' of the 8 successors
double gNew, hNew, fNew;
```

```
int rneigh, cneigh, gi, gj;
gi = (i*W)+j;
// North neighbour
if (isValid(i - 1, j,row_max,col_max) == true)
{
   if (isDestination(i - 1, j, destination) == true)
   {
      cellInfo[i - 1][j].parent_i = i;
      cellInfo[i - 1][j].parent_j = j;
      cout << "The destination cell is found" << endl;
      vecPath = trace(cellInfo, destination);
      ifFound = true;
      return;
   }
      else if (closedList[i - 1][j] == false &&
      isUnBlocked(graph, i - 1, j) == true)
   {
      rneigh = i-1;
      cneigh = j;
      gj = (rneigh*W)+cneigh;
      gNew = cellDetails[i][j].g + graph[gi][gj];
      hNew = EuclideanHVal(i - 1, j, destination);
      fNew = gNew + hNew;
      if (cellInfo[i - 1][j].f == FLT_MAX ||
      cellInfo[i - 1][j].f > fNew)
      {
         openList.insert(make_pair(fNew,make_pair(i - 1, j)));
         cellInfo[i - 1][j].parent_i = i;
         cellInfo[i - 1][j].parent_j = j;
         cellInfo[i - 1][j].f = fNew;
         cellInfo[i - 1][j].g = gNew;
         cellInfo[i - 1][j].h = hNew;
      }
   }
}
```

The iteration does not stop until all the elements inside open list have been processed. After the loop, the open list is empty and the function concludes whether the destination cell has been found or not. The struct is deallocated and the function ends.

# 5 Evaluation of the Investigation

## 5.1 Tie Breaking Mechanism

Despite the improvements in search algorithms for cost optimal path planning, the exponential growth of the extent of the search frontier in A* is unavoidable. Tie-breaking significantly affects the performance of search algorithms when there are zero cost operators that incite large level areas in the search space. A* needs to apply tie-breaking techniques in order to choose which node to extend when multiple nodes have a similar evaluation score.

The A* implementation is integrated with a tie-breaking mechanism after the simple implementation fails to find the path in our real graph i.e., 366x362. The algorithm terminates with a Killed by 9 error message code, a SIGKILL. This error is caused by OutOfMemory. In order to resolve this complication, tie-breaking mechanism was implemented in the algorithm. One approach was to find a way to break ties by preferring paths that were along the straight line from the source cell to the destination cell.

```
double TieBreakHValue(int W, int H, Pair source, Pair destination)
{
   double dx1;
   double dy1;
   double dx2;
   double dy2;
   double cross;
   double heuristic = 0;
   if(map1[row][col] == 3) {
      heuristic = 1000;
      return heuristic;
   }
   dx1 = row - destination.first;
   dy1 = col - destination.second;
   dx2 = src.first - destination.first;
   dy2 = src.second - destination.second;
   cross = dx1*dy2 - dx2*dy1;
   heuristic += cross*0.001;
   return heuristic;
}
```

Listing 5: Tie Breaking Mechanism [11]

This chunk of code calculates the vector cross product between the source and the destination vector and the current cell to destination vector. When these vectors do not queue up, the cross product is larger. The code gives more value to a path that lies along the straight line from the source cell to the destination cell. However, this tie-breaking technique fails drastically as there are several obstacles in our predefined graph.

Tested with our previous 3x3 map:
Source: (0,0) , Destination: (2,0)

0, -1, 0,
100, 0,-1,
-1, 0,100

The path is (0,0) $\rightarrow$ (1,0) $\rightarrow$ (2,0)

Evidently, this is incorrect as (1,0) is an obstacle and the robot can not go through this cell. To fix this problem an if statement was added in case the cell has a value of 100, update the value of heuristic to 1000.

However, tie-breaking mechanism did not solve the OutOfMemory error. This problem occurs in Dijkstra as well. With the original map data set requiring a vector of 132492 (366x362) by 132492 to be allocated, the process allocates too much memory putting pressures on the OS. The OS kills such processes for the sake of system stability. When the process is killed, its action is logged in at /var/log/messages. Each process has a limited free storage associated with it for dynamic memory allocation. When the application devours all the memory from load, at that point the process crashes with OutOfMemory error. After 1 minute and 5 seconds, the process is terminated after CMPRS reaches 59G+. CMPRS shows the measure of bytes of compressed data in memory that belongs to the process.

## 5.2 Convolution with filter2D

Subsequently, filter2D method was utilized through downscaling the real map by three times. It is an opencv function that uses a method known as convolution by using a kernel. Convolution is a procedure that takes place between all the parts of images and the kernel. The kernel is an operator such as a fixed array of integers consisting of an anchor point located at the center of the array.

Essentially this filter calculates the value of convolution by placing the kernel anchor on top of the intent pixel. The kernel coefficients are multiplied by the corresponding image pixel values and summed together, the result is placed at the anchor and the process is repeated for the rest of the pixels.

$$dst(x,y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} kernel(i,j) * src(x+i-anchor_i, y+j-anchor_j)$$

This equation manifests all the existing procedures described above in filter2D. [12]

**Function:** void **filter2D**(InputArray **src**, OutputArray **dst**, int **ddepth**, InputArray **kernel**, Point **anchor**=Point(-1,-1), double **delta**=0, int **borderType**=BORDER_DEFAULT )

The Parameter denotes:

***src:*** *input image*
***dst:*** *output image*
***ddepth:*** *depth of the destination image*
*When ddepth = -1 the output image has the same depth as the source*

***kernel:*** *convolution kernel*
***anchor:*** *indicates the relative position of the determined pixel.*
*Default value (-1,-1) indicates the kernel center*
***delta:*** *value added to filtered pixels. Default value = 0*
***borderType:*** *pixel extrapolation method*

Owing to the fact that convolution method functions by multiplying the kernel coefficients, the values of the original map are changed from 100 to 3 such that extreme bias is avoided, in any other way if there is even one 100 the result would be so high that the whole block will be perceived as an obstacle.

These values are stored as a vector and later converted into Mat float. In order to extract the reduced map, the filter2D is applied on Mat float to an integer 8UC1. The reason being that an image cannot allocate floating values. It only keeps integers 0 to 255. If the image is of floating type then any value greater than 1.0 is shown as a white pixel and value less than 0.0 is shown as a black pixel. While converting, the matrix stores values from the source matrix and then rounds them to the nearest possible value of the destination data type. If the value is out of range, it picks up either minimum or maximum values. In this example, all the values which are negatives will become 0 in the destination matrix as the destination type is CV_8U1 and the minimum possible value is 0. All the floating point values will be floored. No automatic mapping is done. Therefore, to convert the float matrix to int in the scale from 0 to 255, float values are rescaled so they fit in the destination matrix scale. The minimum and maximum values are found through the minMaxLoc opencv function and applied in scaling.

To properly scale the values the following function is applied:

```
void displaymat(Mat& matname, Mat &dst)
{
   double minVal;
   double maxVal;
   Point minLoc;
   Point maxLoc;
   minMaxLoc(matname,&minVal,&maxVal,&minLoc,&maxLoc);
   cout << "min: " << minVal << endl;
   cout << "max: " << maxVal << endl;
   if (minVal!=maxVal){
      matname.convertTo(dst,CV_8U,255.0/(maxVal-minVal),-255.0*minVal/(maxVal-minVal));
   }
}
```

```
int main ( int argc, char** argv )
{
   ifstream inFile;
   ofstream outFile;
   string strFileName = "num2.txt";
   string strOutFile = "downsample7.txt";
   int r = 366, c = 362;
   float V[r*c];
```

```
readFile(strFileName, V, inFile);


//Copying vector to Mat
Mat dst1;

Mat M(r,c,CV_32FC1,V);
printvals(M);

// Declare variables
Mat intMat;
Mat dst;
Mat kernel;
Point anchor;
double delta;
int ddepth;
int kernel_size;
double minVal;
double maxVal;
Point minLoc;
Point maxLoc;
Mat downsample(dr,dc,CV_32FC1);
Mat intdownsample;
int newrow = 0, newcol = 0;

// Initialize arguments for the filter
anchor = Point( -1, -1 );
delta = 0;
ddepth = -1;

// Update kernel size for a normalized box filter
kernel_size = 3;
kernel = Mat::ones( kernel_size, kernel_size, CV_32FC1 )/
    (float)(kernel_size*kernel_size);
// Apply filter
cv::filter2D(M, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT );

// Min and Max value of Mat

displaymat(M,intMat);
imshow("Filter Map",intMat);
imwrite("Filter_Map.jpg",intMat);

int dr = ceil(float(M.rows)/kernel_size);
int dc = ceil(float(M.cols)/kernel_size);

cout << dr << " " << dc << endl;
//downsampling - kernel operation
for(int i = kernel_size/2; i<dst.rows; i+=kernel_size)
{
   newcol=0;
   for(int j = kernel_size/2; j<dst.cols; j+=kernel_size)
   {
      if(dst.at<float>(i,j) > 0)
      {
         downsample.at<float>(newrow,newcol) = 3;
```

```
            }
            else if(dst.at<float>(i,j) < 0)
            {
               downsample.at<float>(newrow,newcol) = -1;
            }
            else
            {
               downsample.at<float>(newrow,newcol) = 0;
            }
            newcol++;
         }
            cout << endl;
            newrow++;
         }

   displaymat(downsample,intdownsample);
   printvals(downsample);

   imshow("downscaled3",intdownsample);
   imwrite("downscaled3.jpg",intdownsample);
   writeFile(strOutFile,downsample,outFile);
   waitKey(0);

   return 0;

}
```

Listing 6: C++ Program to implement filter2D for reducing the map

This function not only reduces the original map by the given kernel_size but prints out the coordinates of the map in reduced map and the images. The A* then use the reduced data as shown in [4.2] and computes the path on it. This code is also used for different kernel_sizes such as 3, 5 and 7 and tested on A* with different heuristics.

The path printed by A* after applying convolution is reduced to 122 x 121 which is 3 times less of the original map. Before computing the path, the x and y coordinates are multiplied with the applied kernel_size. The source is 10,10 and the destination is 120,120.

## 5.3 Path Outline

Path outline for coordinates:
source = 30,30 ; destination = 360,360

### 5.3.1 Shortest Path in A*

The path computed can be found at C. With A*, the total number of cells explored were 108 in kernel size 3, 36 in kernel size 5 ad 25 in kernel size 7.

### 5.3.2 Shortest Path in Dijkstra

Path is computed in terms of nodes. In kernel size 3, the total cells explored are 231. In kernel size 5, the cells explored are 65 and in size 7, 53 cells are explored.

Dijkstra's path is shown in nodes to show that Dijkstra explores more map than A* (C). Since Dijkstra has no heuristic, the path travels from one node to another and does not pick another node in any direction but one that is connected to the current node. The path computed in resolution is stored in a YAML file and configured on the simulator for the robot to move accordingly. The computation in Dijkstra depends on the distance of previous nodes before the current node therefore Dijkstra also prints all the paths from the source uptill the last node before the destination. Appropriately, only the path between the source and the destination (last path) is printed and mentioned in this thesis. The distance is also calculated and printed for the same reason.

The results prove Dijkstra explores more area than A*. The nodes explored in different kernel sizes are more than twice the nodes or cells explored in A*.

Note: This conclusion was drawn by comparing the results of 10 different coordinate sets with various kernel sizes, Dijkstra almost always explores double the amount of nodes explored in A*

## 5.4 YAML Configuration

After these coordinates are converted into resolution; they are stored in a YAML file in order for the simulation to run. The robot moves accordingly and computes the same path displayed in the images below 10b.

```
YAML::Emitter yaml_out;
yaml_out << YAML::BeginMap;
yaml_out << YAML::Key << "waypoints";
yaml_out << YAML::Value << YAML::BeginSeq ;
for(int i =0; i<vecPath.size();i++)
{
   yaml_out << YAML::BeginMap;
   yaml_out << YAML::Key <<"position";
   yaml_out << YAML::Value << YAML::BeginMap;
   yaml_out << YAML::Key << "x";
   yaml_out << YAML::Value << vecPath[i].first;
   yaml_out << YAML::Key << "y";
   yaml_out << YAML::Value << vecPath[i].second;
   yaml_out << YAML::Key << "z";
   yaml_out << YAML::Value << "4.5";
   yaml_out << YAML::EndMap;
   yaml_out << YAML::Key << "orientation";
   yaml_out << YAML::Value << YAML::BeginMap;
   yaml_out << YAML::Key << "x";
   yaml_out << YAML::Value << "0.0444210774910485";
   yaml_out << YAML::Key << "y";
   yaml_out << YAML::Value << "-0.03997364552703113";
   yaml_out << YAML::Key << "z";
```

```
   yaml_out << YAML::Value << "0.7459565426241741";
   yaml_out << YAML::Key << "w";
   yaml_out << YAML::Value << "0.66330815768691";
   yaml_out << YAML::EndMap;
   yaml_out << YAML::EndMap;
}
yaml_out << YAML::EndSeq;
yaml_out << YAML::EndMap;
cout << "Here's the output YAML:\n" << yaml_out.c_str();

cout << endl;

ofstream inFile;
inFile.open("../config/yamlastardata.yaml");
inFile << yaml_out.c_str();

inFile.close();
```

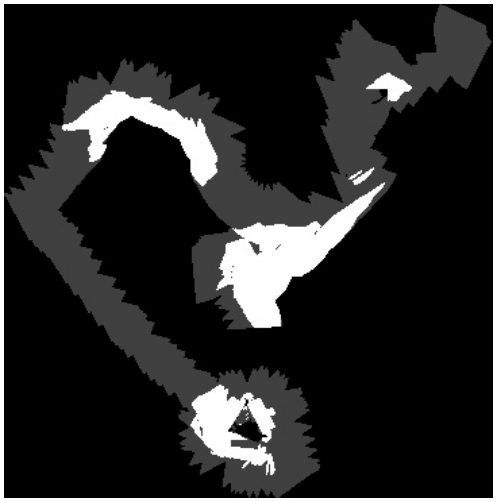
Listing 7: Code to store the coordinates in a YAML file

```yaml
waypoints:
  - position:
      x: 13.499999977647001
      y: -13.499999977647001
      z: 4.5
    orientation:
      x: 0.0444210774910485
      y: -0.03997364552703113
      z: 0.7459565426241741
      w: 0.66330815768691
  - position:
      x: 13.3499999754117
      y: -13.499999977647001
      z: 4.5
    orientation:
      x: 0.0444210774910485
      y: -0.03997364552703113
      z: 0.7459565426241741
      w: 0.66330815768691
  - position:
      x: 13.1999999731764
      y: -13.499999977647001
      z: 4.5
    orientation:
      x: 0.0444210774910485
      y: -0.03997364552703113
      z: 0.7459565426241741
      w: 0.66330815768691
  - position:
      x: 13.0499999709411
      y: -13.499999977647001
      z: 4.5
```

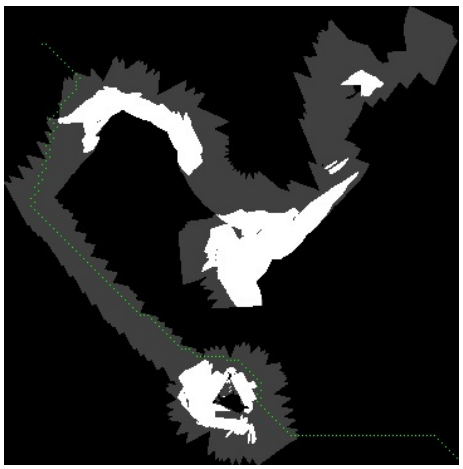
Figure 9: Output of YAML file: File name: yamlastardata.yaml

## 5.5 Path displayed as seen on simulator

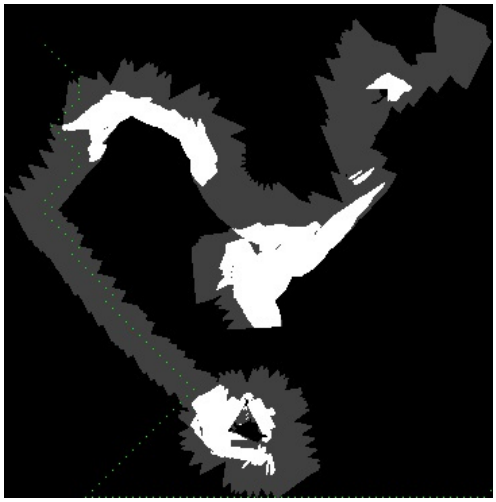
The following images display the result of the path computed:
Original coordinates: source = 30,30 ; destination = 360,360



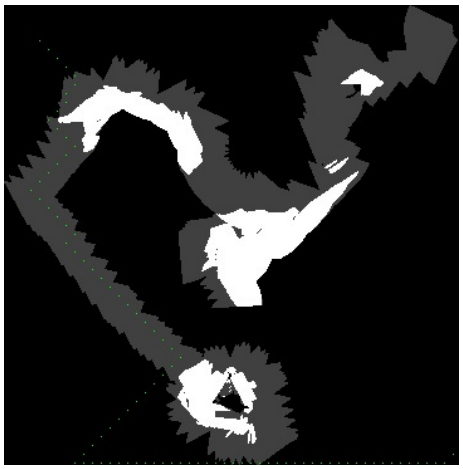
(a) Original map



(b) Path computed using Euclidean Heuristic on A* with Kernel Size 3



(c) Path computed using Euclidean Heuristic on A* with Kernel Size 5



(d) Path computed using Euclidean Heuristic on A* with Kernel Size 7

Figure 10: Path computed on original map with downscaled data

## 5.6 Downsampled Images

These images are produced by 5.2. 12 displays the same path as in 10. 10 uses the original filtered map with dimensions 366 by 362 to display the result of varied kernel size data. Kernel Size 3 has a dimension of 122 by 121, kernel size 5 has a dimension of 74 by 73 and kernel size 7 has a dimension of 53 by 52. The dimension variation is the cause of the difference between 10b, 10c and 12. The path in the latter is more clear and concised as compare to the path computed in the orignal map because of less resolution. The difference in path formation is also visible from kernel size 5, A* distinguishes the white block as a bigger obstacle than it actually is and changes its route.

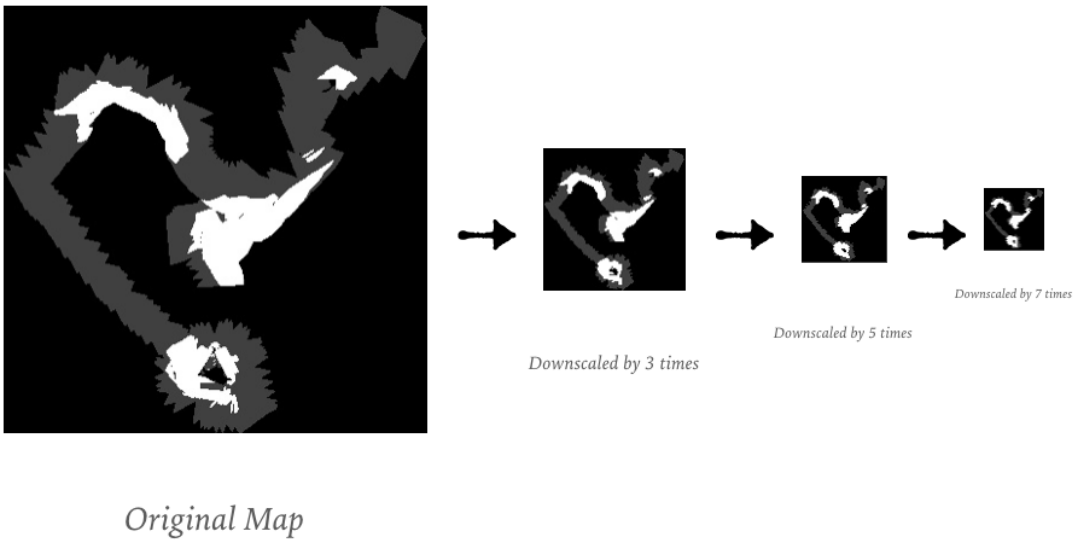



Figure 11: Set of downscaled images of the map

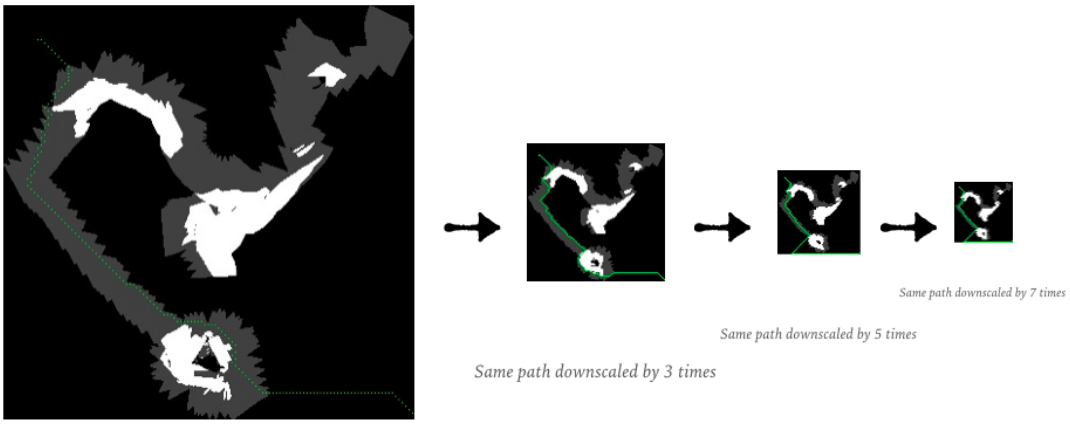



Figure 12: Set of downscaled images with the path computed

## 5.7 Time & Space Complexity

The Dijkstra algorithm is implemented as a depth first search algorithm with adjacent arrays being the main data structure so the time complexity will be O($V^2$). However, the way this algorithm was written to conduct this research was to have a two-dimensional graph of $H * W$ by $H * W$ and to access each node, the number of nodes are given, then the time complexity will be proportional to $O(H * W \times H * W)^2$

The length of longest path = $x$. For each node, it's siblings are stored so when it's children have been visited and the parent node is explored, the next sibling to visit should be known. For $x$ nodes throughout the path, extra $s$ nodes are stored for each of the $x$ nodes. So the most memory it can take up is the longest possible path. Thus, the space complexity of Dijkstra function is $O(x * s)$ when it's searching for the shortest path. However, in order to store the nodes in an array, it takes $O(V)$ and to store the shortest distance for every node it takes another $O(V)$ which equals to $O(V)$ since 2 is constant; equivalent to $O(H * W)$.

A* function itself has the time complexity of $O(E)$ where $E$ is the number of edges in the graph as the graph is connected by a series of edges from the source cell to the destination cell. The total number of real edges results in $O(H * W \times H * W)$. Altogether it has the same worse time as the Dijkstra $O(H * W \times H * W)^2$.

A* uses more memory but it is the absolute algorithm meaning if the node is in the lowest depth possible, it will give the optimal solution. The space complexity of A* is $O(H * W)$.

| Algorithm | Kernel Size | Heuristic | Time |
|---|---|---|---|
| Dijkstra | 3 | - | 5.8 s |
| Dijkstra | 5 | - | 4.2 s |
| Dijkstra | 7 | - | 2.9 s |
| A* | 3 | euclidean | 1 s |
| A* | 3 | manhattan | 93ms |
| A* | 3 | diagonal | 83ms |
| A* | 5 | euclidean | 30 ms |
| A* | 5 | manhattan | 26 ms |
| A* | 5 | diagonal | 26 ms |
| A* | 7 | euclidean | 26 ms |
| A* | 7 | manhattan | 20 ms |
| A* | 7 | diagonal | 19 ms |

Table 1: Total time taken to compute paths with different heuristics and kernel sizes

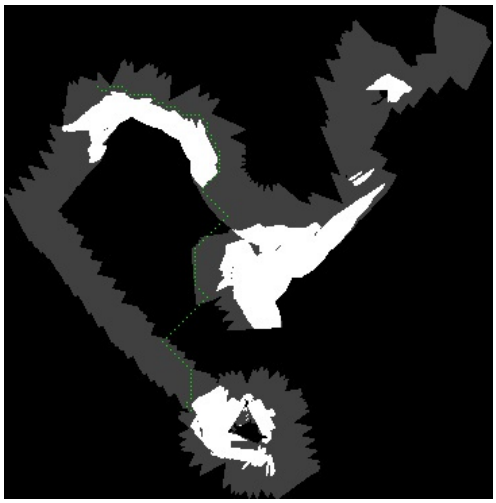
# 6 Comparision with Heuristics & Obstacle Weights

## 6.1 Path Computation and Comparison with different Heuristics

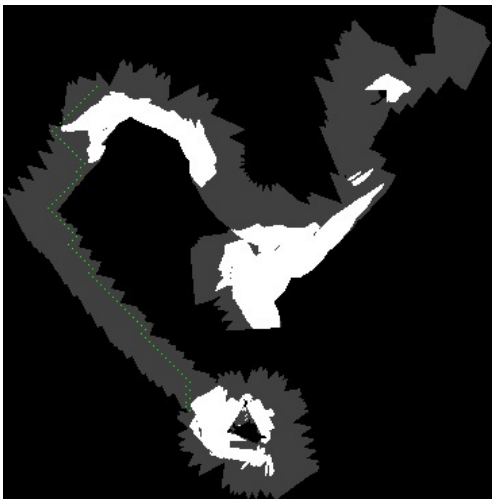
A* is computed with three different heuristics; Manhattan, Diagonal and Euclidean. Moreover, these three heuristics are compared with 10 different coordinate sets.

1. Source: 30,30 Destination: 360,360
2. Source: 60,70 Destination: 60,340
3. Source: 60,70 Destination: 140,300
4. Source: 60,90 Destination: 180,210
5. Source: 60,60 Destination: 180,315
6. Source: 300,70 Destination: 300,210
7. Source: 300,70 Destination: 180,315
8. Source: 180,180 Destination: 315,315
9. Source: 40,320 Destination: 340,60
10. Source: 40,320 Destination: 300,320

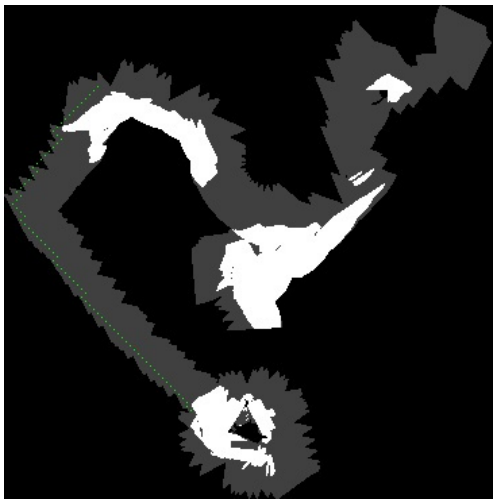
The paths are computed in detail in C with the three heuristics mentioned. The images below display the different paths with three heuristics. The difference in path is visible. As the kernel size increases, the path with different heuristics might compute a different path but might have the same length or number of cells explored. In some cases, two heuristics give the shortest path such as in coordinates 3,5,6,7,8 and 9. While with kernel size 7 all three heuristics have the same length in coordinates 2,3,5 and 6. After comparing the results, Euclidean computes the shortest path 86% of the time , Manhattan comes second with 63% and Diagonal with 53%. When acknowledged singly, Euclidean gives the shortest path 50% of the time while Manhattan and Diagonal gives the shortest path 33% and 16% of the time respectively.



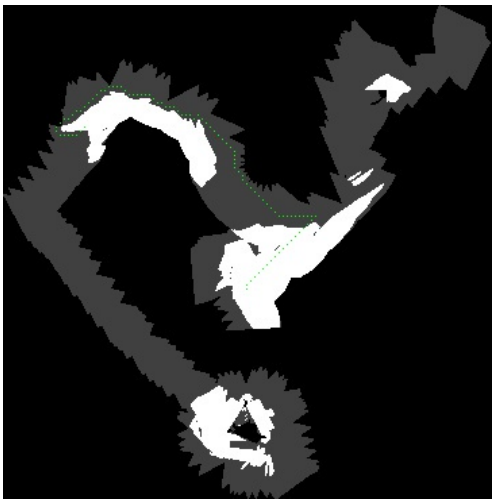
(a) Diagonal Heuristic with coordinates of 3



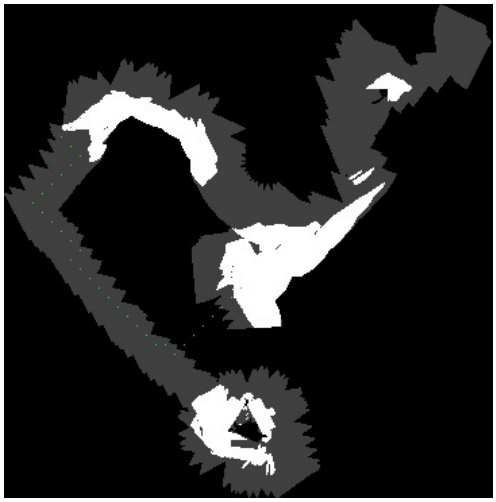
(b) Manhattan Heuristic with coordinates of 3



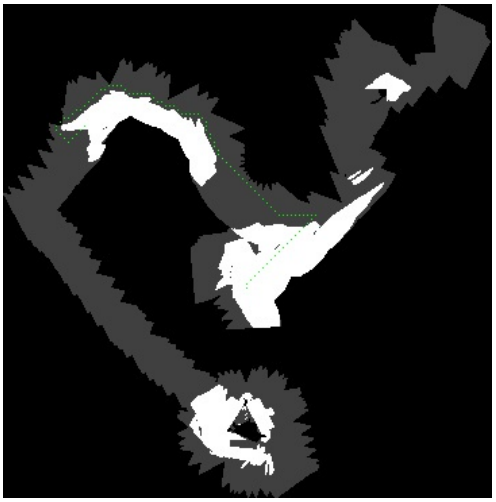
(c) Euclidean Heuristic with coordinates of 3



(d) Diagonal Heuristic with coordinates of 4

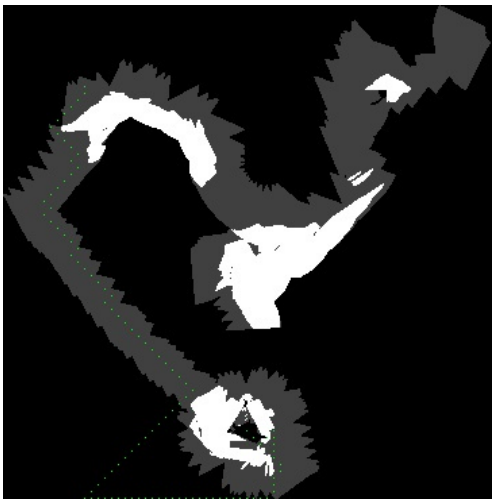


(e) Manhattan Heuristic with coordinates of 4



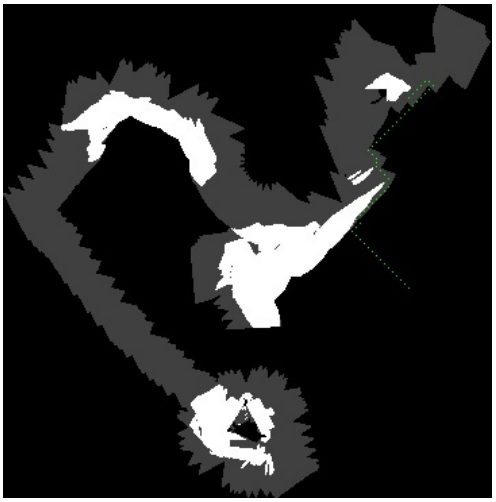
(f) Euclidean Heuristic with coordinates of 4

(a) Diagonal Heuristic with coordinates of 5

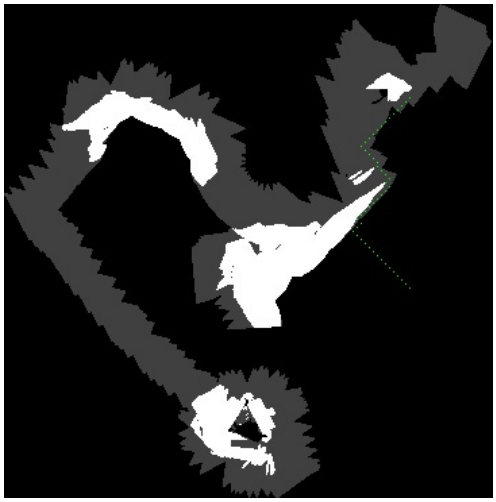


(b) Manhattan Heuristic with coordinates of 5

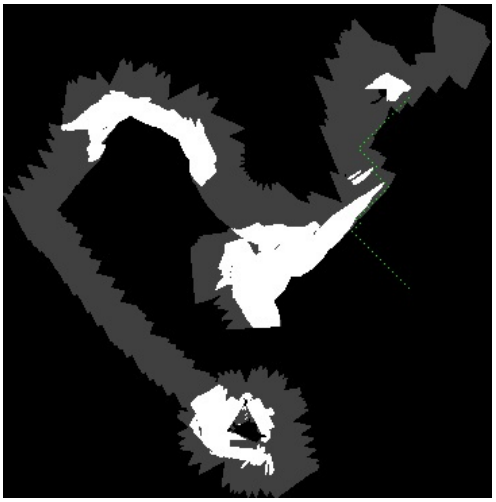
(c) Euclidean Heuristic with coordinates of 5



(d) Diagonal Heuristic with coordinates of 6



(e) Manhattan Heuristic with coordinates of 6



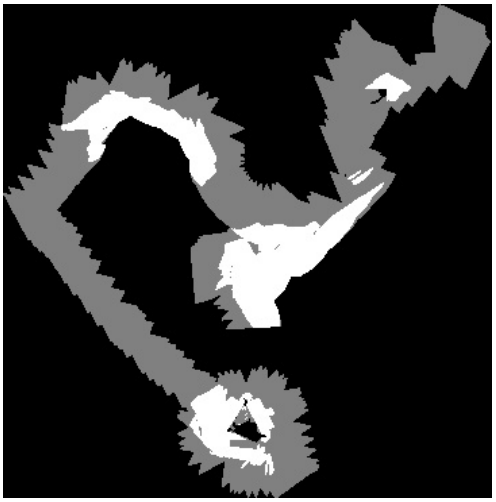
(f) Euclidean Heuristic with coordinates of 6

Figure 14: Paths with different heuristics

## 6.2 Path Computation and Comparison with different Obstacle Sizes

In order to emphasize on the obstacle, free and unknown cell values. The values were switched from 100, -1 and 0 to 100 , 2 and 1 and to 60,50 and 40. 100 and 60 are for obstacles. -1 and 50 for unseen cells and 1 and 40 for free cells.

(a) Filtered Map of 100, 2 & 1



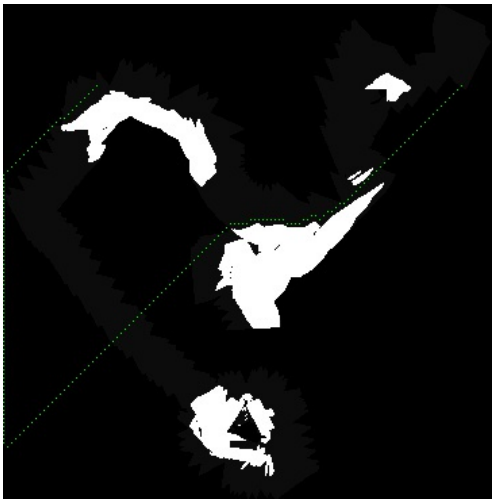
(b) Filtered Map of 60,50 & 40

As evident by the images, the free cells in 15a is barely visible, the grey area is closer to black since the values are close. In 15b the free cells have a lighter grey color closer to white.

By changing the values of the depicted values of the graph, the path changes completely. When the value from 50 to 2 is altered, there is almost no recognization of free cells because weighted value of 2 is much closer to 1 where 1 depicts the value of a free cell. Thus the same path in 10b is very different in 16a by using the same heuristic. The path size is therefore also larger, it explores 226 vertices compared to 164 in the original path computed. The path difference is also visible between 16c and 14f. In both images, the path does not follow the free and known route and rather hits the boundaries first, since there is no recognization of those cells and follows the boundaries until it comes closer to the boundary that is closest to the destination point. This similarity is presented in the rest of the images as well.

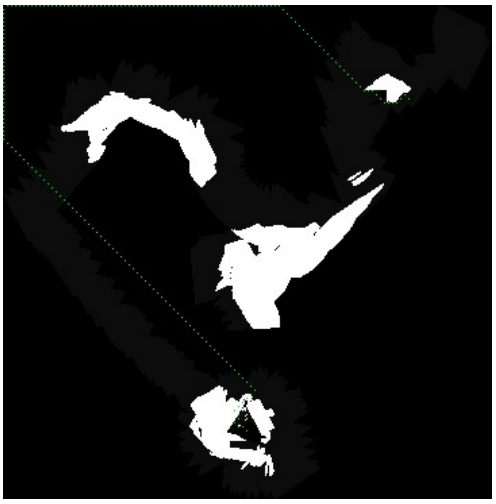
When the values are changed to 60,50 & 40. The path somewhat follows the known free cells but because of values being so close, in some cases, it passes through the obstacles while in other cases the unknown cells are mistaken with free cells. The path in 17a and 17c is still similar to the original path computed. 17b and 17e completely disregards the free and unknown cell notion and computes the shortest path by making a straight line and passes through the obstacles respectively.

(a) Coordinates with no. 1

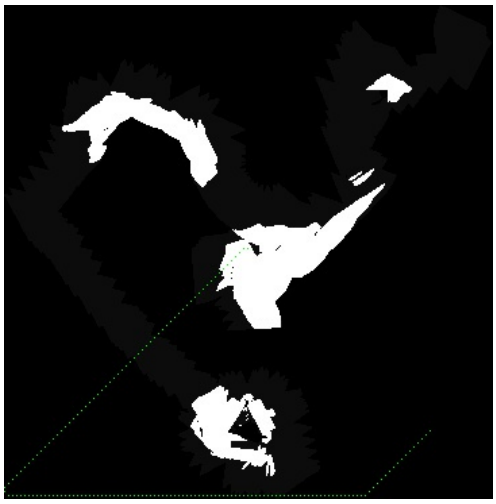


(b) Coordinates with no. 2

(c) Coordinates with no. 6

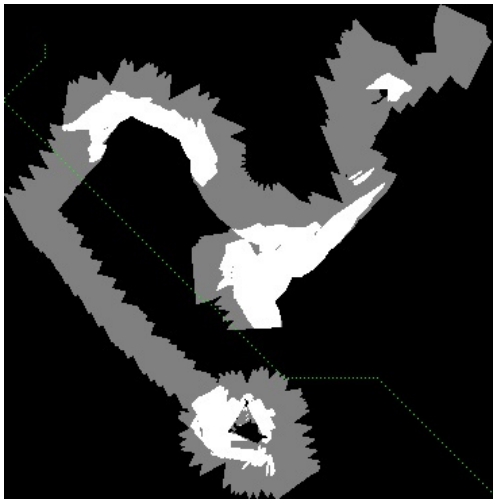


(d) Coordinates with no. 7

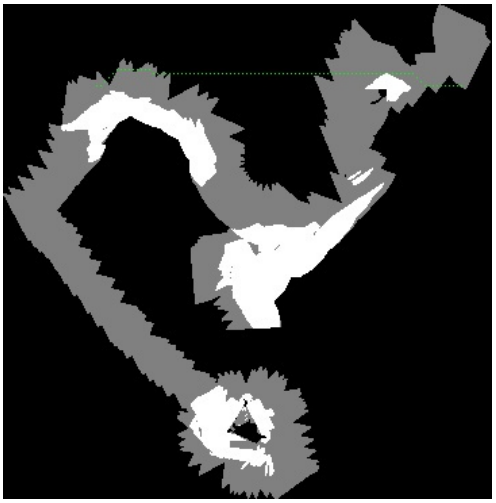


(e) Coordinates with no. 8

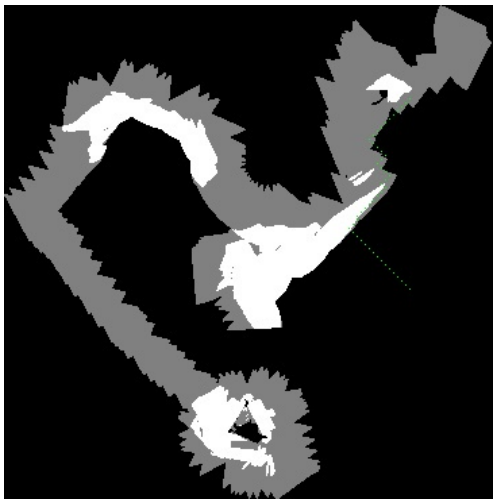
Figure 16: Paths computed on image 15a



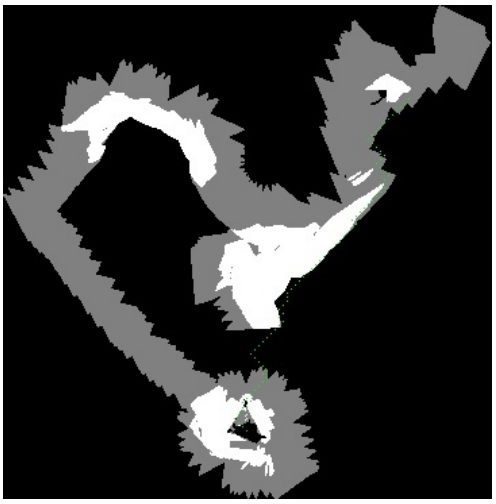
(a) Coordinates with no. 1



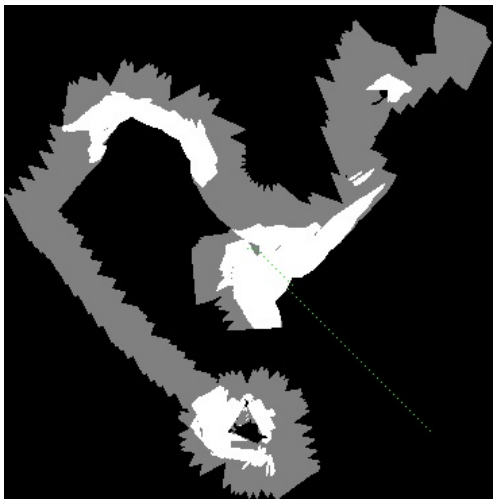
(b) Coordinates with no. 2



(c) Coordinates with no. 6



(d) Coordinates with no. 7



(e) Coordinates with no. 8

Figure 17: Paths computed on image 15b

# 7 Compilation & Simulation Start-up Process

## 7.1 Gazebo Simulator

To run the simulation on Gazebo the following steps are taken:

1. **roslaunch dexrov_meta startup_simulartion.launch ;** starts up the simulation
2. **roslaunch dexrov_meta startup_vehicle_sim.launch ;** vehicle is imported in the simulation
3. **roslaunch dexrov_meta startup_vessel_sim.launch ;** vessel is imported in the simulation
4. **roslaunch dexrov_meta rviz.launch ;** for launching rviz and visualization purpose to keep track of all the topics and nodes
5. **roslaunch dexrov_meta perception_octomap.launch ;** for launching the octomap
6. **rosrun joy joy_node_dev:=/dev/input/js1 ;** in case the joystick does not work
7. **roslaunch abrish navigate_path.launch ;** This command makes the robot move through the YAML file. YAML file is stowed in navigate_path.launch file <arg name="config_filename" default="$(find abrish)/config/yamlastardata.yaml"$/>

## 7.2 Compilation of Algorithms

To run the algorithms the following steps are followed:

- g++ filter2D.cpp -o filter2D ‘pkg-config –cflags –libs opencv‘
- g++ -std=c++11 astar.cpp -o astar -lyaml-cpp ‘pkg-config –cflags –libs opencv‘
- g++ -std=c++11 dij.cpp -o dij -lyaml-cpp ‘pkg-config –cflags –libs opencv‘

# 8 Drawbacks & Potential Improvements

Before unfolding an algorithm the right data structure has to be adopted for the optimization of the algorithm. For the open list, the best choice is a list that has both a fast insert and extract minimum operation. It is more important for the insert operation to have better performance of the insert operation than the extract minimum operation since the algorithm on average does not extract all the cells (nodes) inserted into the open list. These functionalities exist in the priority queues such as a binary heap and a Fibonacci heap. Binary heap gives a time complexity of $\Theta$(log n) for both insert and extract minimum operations. The fibonacci heap further optimizes the insert operation to a linear time $\Theta$(1). Another less recurrent operation that occurs is decrease key, when the g cost of a cell (node) in the open list needs an update. The fibonacci heap beats the rest of the data structure in this regard with a linear decrease key time complexity $\Theta$(1).

Both algorithms can be implemented with an adjacency list which will improve the time complexity by $O(N + E)$ where $N$ is the number of nodes/vertices and $E$ is the number of edges.

Although A* produces the shortest optimal path but it is not always consistent as it relies heavily on the heuristics

# 9 Conclusion

Compared to A*, Dijkstra has a much better space complexity however it may not find the optimal solution. Dijkstra does not compute the most optimal shortest path but stops when it finds a minimal path from one node to another. Although the time complexity is the same for both algorithms, A* calculates the shortest path much faster than Dijkstra. A* computes the optimal path with kernel size 3 with the euclidean heuristic function. This research paper also deduce the importance of values assigned to various objects to indicate their characteristics. The values should be optimum otherwise a confused path will be evaluated. In addition, the kernel sizes are crucial, less resolution and data on the image and the graph, returns an altered path as less resolution disturbs the significance of mixed objects defined.

# 10 Further Research

In this paper, only one type of map image from the simulation is explored. For further research, the algorithms should be tested on variety of map images. The map representation can make a huge difference in the performance and path quality as well, therefore the algorithms can be tested on various representation of maps such as polygonal maps or navigation meshes. This research can be further expanded by working with the robots closely and adding the usage of sensors conveniently. It would be interesting to see the results of the coordinates that are known or near obstacles. Furthermore, A* can be tested with exact heuristics by precomputing the length of the shortest path between every pair of nodes.

# 11 Acknowledgements

I want to express my sincerest gratitude to those who have helped me. I am grateful to my Professor and supervisor Szymon Krupinski for his advice, assistance and suggesstions and Prof. Francesco Maurelli for supporting this research. I would like to thank my bachelor supervisor, Arturo Gomez Chavez without his insight, supervision, guidance and patience, I would not have tackled and address the problems, I came across during this thesis.
I also want to thank Malte Granderath for helping me with implementation problems and thank you Maia Tomadze, Matius Sulung and Frederik Florenz for the study sessions.
In addition, I would like to thank my friends Murt and Zazzle for helping me survive the stress and not letting me give up.

# References


[1] Andrew B Walker et al. Hard real-time motion planning for autonomous vehicles. *Walker, A.—" Hard Real-Time Motion Planning for Autonomous Vehicles PhD thesis, Swinburne University*, 2011.

[2] Syed Abdullah Fadzli, Sani Iyal Abdulkadir, Mokhairi Makhtar, and Azrul Amri Jamal. Robotic indoor path planning using dijkstra's algorithm with multi-layer dictionaries. In *2015 2nd International Conference on Information Science and Security (ICISS)*, pages 1–4. IEEE, 2015.

[3] Lisa Velden. An informed search for the shortest path.

[4] Liang Yang, Juntong Qi, Dalei Song, Jizhong Xiao, and volume=2016 pages=5 year=2016 publisher=Hindawi Publishing Corp. Han, Jian journal=Journal of Control Science and Engineering. Survey of robot 3d path planning algorithms.

[5] Dmitry S Yershov and Steven M LaValle. Simplicial dijkstra and a* algorithms for optimal feedback planning. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3862–3867. IEEE, 2011.

[6] Abhishek Goyal, Prateek Mogha, Rishabh Luthra, and Neeti Sangwan. Path finding: A* or dijkstras. *International Journal in IT and Engineering*, 2(01), 2014.

[7] Mengzhe Zhang. *Path planning for autonomous vehicles*. PhD thesis, 2014.

[8] B Moses Sathyaraj, Lakhmi C Jain, Anthony Finn, and S Drake. Multiple uavs path planning algorithms: a comparative study. *Fuzzy Optimization and Decision Making*, 7(3):257, 2008.

[9] K. Khantanapoka and K. Chinnasarn. Pathfinding of 2d 3d game real-time strategy with depth direction a algorithm for multi-layer. In *2009 Eighth International Symposium on Natural Language Processing*, pages 184–188, Oct 2009.

[10] Andris Jansons. A star (a*) path finding c, May 2018.

[11] Amit Patel. Heuristics from amit's thoughts on path finding, Feb 2019.

[12] Making your own linear filters, 2017.

# Robust Continuous System Integration for Critical Deep-Sea Robot Operations Using Knowledge-Enabled Simulation in the Loop

Christian A. Mueller*, Tobias Doernbach*, Arturo Gomez Chavez*, Daniel Köhntopp*, Andreas Birk

***Abstract*— Deep-sea robot operations demand a high level of safety, efficiency and reliability. As a consequence, measures within the development stage have to be implemented to extensively evaluate and benchmark system components ranging from data acquisition, perception and localization to control. We present an approach based on high-fidelity simulation that embeds spatial and environmental conditions from recorded real-world data. This *simulation in the loop* (SIL) methodology allows for mitigating the discrepancy between simulation and real-world conditions, e.g. regarding sensor noise. As a result, this work provides a platform to thoroughly investigate and benchmark behaviors of system components concurrently under real and simulated conditions. The conducted evaluation shows the benefit of the proposed work in tasks related to perception and self-localization under changing spatial and environmental conditions.**

## I. Introduction

The rapid progress of Unmanned Underwater Vehicle (UUV) capabilities in recent years has increased their use in inspection and mapping activities as they offer higher data transmission rates through acoustics, more accurate navigation and denser environment 3D models with energy-efficient sensors. UUV systems are also increasingly applied to areas which are inaccessible and hazardous to humans. For example, the UK Health & Safety Executive's (HSE) 2015–2016 Offshore Safety Statistics [1] reports 53 major injuries and more than 400 dangerous occurrences, most of them performing maintenance and construction activities. However, the development and continuous evaluation of such underwater robotic systems typically requires the organization of a crew (intendant, operator, navigator) and an adequately equipped vessel to deploy, operate, and retrieve the robot offshore. This rapidly increases the effort and cost of each development cycle. Specialized testing and deployment strategies are required because effort and costs in case of failure are higher by several orders of magnitude than in ground robotics, for example, when a UUV malfunctions in deep sea and cannot be retrieved anymore. Consequently, efficient strategies have to be incorporated to validate effectiveness, robustness and reliability of the developed capabilities. As to alleviate these costs and efforts, we propose a methodology that uses a simulator for underwater robotic activities and integrates parts of the development stack with real-world data recorded from field trials.

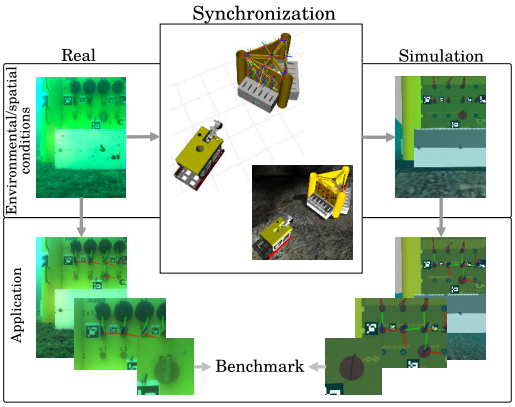



Fig. 1. Illustration of the proposed *simulation in the loop* (SIL) methodology showing a perception task application (valve and lever pose estimation, see Section V-B) within the DexROV project [2].
**Supplementary video:** http://robotics.jacobs-university.de/videos/sil-2018

In the context of the EU-funded research project *DexROV* (Effective Dexterous ROV Operations in Presence of Communication Latencies [2]), in our previous work [3] we proposed a versatile integration and validation architecture that allows for pre-deployment testing using simulated and real system components besides each other in a seamless way. That work focused on the *continuous system integration and deployment* of a fully integrated system that may contain simulated components due to their developmental stage.

In the work presented here, however, our goal is a *simulation in the loop* (SIL) architecture (see Fig. 1) which allows for extensive *system benchmarking*. Hence, our particular focus is set on closing the discrepancy between simulated and real-world data. As a result, our proposed framework

- *synchronizes* simulated and real-world data by incorporating environmental and spatial feedback collected from field-trials which
- *provides* an augmented virtual environment reflecting environmental/spatial conditions from real-world missions to test, benchmark and compare behaviors of system modules,
- *preserves* the benefits of continuous system integration

*These authors contributed equally to this work and share first authorship. The authors are with the Robotics Group, Computer Science & Electrical Engineering, Jacobs University Bremen, Germany, {chr.mueller,t.fromm,a.gomezchavez,d.koehntopp, a.birk}@jacobs-university.de.

The research leading to the presented results has received funding from the European Union's Horizon 2020 Framework Programme (H2020-EU.3.2.) within the project (ref.: 635491) "Effective dexterous ROV operations in presence of communication latencies (DexROV)". The authors would like to thank the DexROV team within the Space and Innovations Division of COMEX S.A., Marseille, France, for their support in collecting the experimental data described in this paper.

to perform such benchmarks using real or simulated components or a combination of both, and allows to

- *perform* tests on distributed deployment, interfaces/pipelines, data regression/degradation, and fault recovery/safety as described in [3].

## II. Related Work

DexROV [2] features a full-fledged Unmanned Underwater Vehicle (UUV) system, deployed from a vessel in the Mediterranean to perform perception as well as dexterous manipulation. In our application, an artificial testing panel has been set up to allow for interventions including, but not limited to, visual inspection, docking, and manipulation of valves, handles and other movable parts. Such robotic underwater operations require accurate pose estimation of task-related objects like levers and valves to reduce the risk of costly failures. Hence, robust detection of spatio-temporal reference points is of particular interest during these tasks.

Due to the noisy nature of underwater scenarios and the precision required for manipulation tasks, we exploit a-priori knowledge about the environment. Using known landmarks allows the system to operate with low-quality data which commonly appears in deep-sea sensing, e.g. acoustic sensors affected by salinity and temperature, and camera images distorted by light backscatter. Such landmarks have been used for navigation and docking of underwater vehicles [4], [5] since underwater no global positioning information is available. Additionally, artificial structure-based perception has been frequently used both in ground [6] and underwater robotics [7], [8]. For these reasons, we equipped the described testing panel with visual markers [9] to exploit it as a reference structure for enhancing localization and object perception capabilities.

The major goal of the proposed SIL methodology is robust projection of real conditions to simulation, including task-related objects. Inferences from simulated interaction of robots with the environment is a well-established approach [10], [11], [12]. Additionally, several frameworks exist which allow for the reproduction of experiments from a knowledge base incorporating experimental data and information deduced thereof [13], [14]. However, synchronizing the state of the simulation with real-world data for reasoning and benchmarking has still not been covered extensively in the literature.

Although numerous sophisticated simulators exist for ground and aerial robots, there is a limited number for underwater applications [15], [16] due to the difficulty of modeling hydrodynamic forces and environment light. Closest to the DexROV scenario is the UUV Simulator package [17] which serves similar use cases like intervention tasks using a UUV manipulator. This package, as well as our approach, builds upon the established Gazebo [18] simulator. Further on, to guarantee authentic projection of real conditions to simulation, we exploit visual markers attached to the testing panel as reference points, described in Section IV. The work in [19] and our experimental evaluation in Section VI-B validate the use of visual markers subject to underwater image distortions.

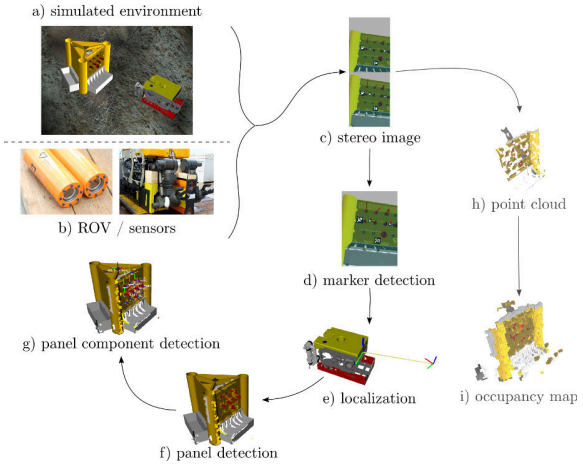



Fig. 2. System Architecture Overview – subfigures h) and i) are not regarded in the context of this paper, see [3] instead.

In the remainder of this paper, we describe how to use artificial man-made structures to set up the simulation environment for perception and manipulation tasks and interlock it with continuous system integration. In Section VI, we show the performance improvements achieved through experiments in object pose estimation and robot localization as real data is iteratively used to refine the simulated scenario.

## III. Simulation for Continuous System Integration

In a real-world robotic system, a large number of system integration hours have to be invested before being able to deploy it into a realistic scenario. Regarding the example of DexROV, the research project features an fully-fledged UUV for open sea usage and, amongst others, equipped with a stereo perception system and a manipulator. Within this project, several real-world field trials have been scheduled and partly performed already. However, the different workgroups do not have access to the hardware prior to the annual trials despite the need to thoroughly test their developed hardware and software components.

For this reason, high priority was assigned to establish a simulation base before implementing any task-specific software modules in order to allow developers to work independently and in parallel. The architecture of such a simulation framework depends on the application scenario, though the underlying principles and procedures of continuous system integration as described in detail in our previous work [3] translate to any desired area of robot operation.

Fig. 2(a) shows an example of the DexROV simulation environment including a man-made testing panel which features different valves and levers, specifically generated to test the manipulation capabilities of the robot. The main target of using simulation in a continuous system integration fashion is to validate a plethora of different components in simulation and to infer from their behavior onto real-world missions. Hence, the simulation framework for the respective application scenario, i.e. a deep-sea setting like

DexROV, needs to provide its capabilities in a component-wise structure where simulated can be effortlessly replaced with real elements. As for inter-component communication, since such a variable system encompasses many interacting modules, all components are integrated into a middleware like ROS. This allows operability through remote and unstable network connections as described in [2] and required in harsh deployment environments.

Using the principle of continuous system integration, a complex system architecture can be established, maintained, and exploited like the application example in Fig. 2. All the components pictured therein have been developed, enhanced, and evaluated following this schema. In Section V, we describe in detail the business logic of some components as well as how the simulation framework has been exploited to improve their utility and usability. However, first of all, the next section explains how real-world data is incorporated into the simulation environment for fast component tuning that, in turn, yields high-accuracy results in real-world tests.

## IV. Increasing Continuous System Integration Robustness Using Simulation in the Loop

A developmental procedure that incorporates a simulation of the application scenario is a powerful tool to decrease the overall project costs by accelerating the planning and execution of field trials. It also increases the system reliability by benchmarking its behavior in extreme border conditions and without exposing valuable equipment to danger.

The concept of *simulation in the loop* (SIL) goes one step further. Instead of conducting working cycles in a sequential development-evaluation process with data generated from simulation and later from real-world field trials, SIL aims to combine both steps. The recorded real-world data is projected into the simulation environment using similar conditions such as the robot configuration in space, perceived sensor data and environmental constraints.

To conflate the *spatial conditions* present in real-world data with simulation, spatio-temporal reference points are used during the field trials. Crucial prerequisite of these reference points is their accurate detection and pose estimation in 3D space. In the DexROV scenario, the testing panel is exploited as such a reference point since the pose estimation of the panel is a major project objective, further described in Section V-A. To guarantee robust pose estimation, the panel is augmented with visual markers, specifically ArUco markers [9] which provide high pose accuracy. Given the augmented panel model, the observation of markers in the recorded real-world data allows to take the panel as a *visual landmark* and infer the robot pose with respect to it. This inference allows to project the relative spatial circumstances between panel and robot into simulation (Fig. 3b). Likewise, states of panel components (e.g. valves, switches or wheels) from real observations can also be accordingly projected to panel components in simulation (Fig. 3e).

Furthermore, real observed *environment conditions*, like camera image noise, haze or illumination, can be estimated to reflect similar conditions in simulation. For this, we rely on Gazebo's built-in sensor noise models, scene fog and lighting options as well as the UUV Simulator camera plugin [17]. Consequently, *spatial* and *environment conditions* perceived from real observations are continuously reflected in simulation in a cylic manner, as shown in Fig. 3. This *processing loop* of real observations acquisition and their projection into simulation is shown in Alg. 1.

**Algorithm 1** Simulation in the loop (SIL)

**Input:** real-world sensor data $\mathcal{R}(\mathcal{E})$, task $\mathcal{T}$
  initialize knowledge base (see Section V)
  infer environment conditions $\mathcal{E}$ from sensor data $\mathcal{R}(\mathcal{E})$
  initialize simulation environment, spawn testing panel model
  **for all** real-world samples $r(t) \in \mathcal{R}$ at time steps $t$ **do**
    detect visual markers in $r(t)$ (see Fig. 3a)
    estimate panel pose in odometry frame ${}^{O}_{P}\mathbf{T}$ from marker poses (Section V-A)
    infer robot pose in odometry frame ${}^{O}_{R}\mathbf{T}$ (see Fig. 3b)
    set robot pose in simulation according to ${}^{O}_{R}\mathbf{T}$ (see Fig. 3c)
    generate simulated sensor data $s(t) \in \mathcal{S}(\mathcal{E})$
    create benchmarking sensor data pair $b(t) \leftarrow \langle r(t), s(t) \rangle$ (Figs. 3a and 3d)
    calculate measure $m(\mathcal{T}, t)$ using $b(t)$ under $\mathcal{R}(\mathcal{E})/\mathcal{S}(\mathcal{E})$ domain (Section V)
    $\mathcal{B} \leftarrow \mathcal{B} \cup b(t)$, $M \leftarrow M \cup m(\mathcal{T}, t)$
  **end for**
**Output:** synchronized benchmarking data sequence $\mathcal{B} \leftarrow \mathcal{R}(\mathcal{E}) \cap \mathcal{S}(\mathcal{E})$, sequence of measures $M(\mathcal{T})$ for task $\mathcal{T}$

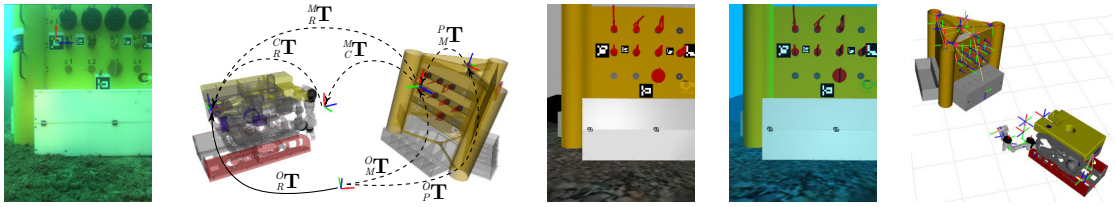
Through this loop process several specialized optimization and benchmarking tasks $\mathcal{T}$ can be performed iteratively based on simulated $\mathcal{S}$ and real-world data $\mathcal{R}$, such as object recognition, manipulation or 3D modeling. This yields a corresponding sequence $M(\mathcal{T})$ of individual measures $m(\mathcal{T}) \in M$ which have to be defined depending on the task $\mathcal{T}$ prior to running the simulation loop. Several examples for such measures are described and used in our experimental evaluation (Section VI).

Additionally, environmental conditions $\mathcal{E}$ in simulation can be adapted to compare the methods performance (e.g. robot localization, panel pose estimation) under various configurations with respect to their performance under real field trial conditions. For numeric optimization, a respective task-dependent measure $m(\mathcal{T})$ can be utilized. This capability of the proposed approach is particularly valuable in continuous system development under challenging and dynamic conditions, such as in deep-sea projects like DexROV.

## V. Application-Relevant Benchmarking Tasks

Since deep-sea missions are cost-intensive and bear a risk to life and equipment, prior knowledge about the mission decreases risk of failures and increases safety. Particularly in visual inspection or manipulation tasks of man-made structures, the incorporation of prior knowledge can be exploited to increase efficiency and effectiveness of conducted missions. Therefore, a *knowledge base* is built which contains properties of task-related objects. Along with offline information, like CAD models and kinematic descriptions of the robot and testing panel, the knowledge base also contains online state information gathered over the execution course of the task, e.g. the current robot and object poses.

Using this prior knowledge and online state information, a multitude of different validation and optimization tasks can be carried out with the presented setup. In this section, we describe several application scenarios where the SIL



(a) Real-world camera image (b) Robot–panel space transformations (c) Simulated camera image (d) Adapted simulated camera image (e) Panel with projected kinematic model

Fig. 3. (*Spatial condition*) Synchronization of observed robot pose in real-world data (a, b) with simulation environment (c). (*Environmental condition*) Simulated images are shown in (c) and (d) accordingly. (e) Panel component kinematic state projected in simulation.

methodology provides significant benefits for validation and benchmarking. All these benchmarking tasks $\mathcal{T}_P$, $\mathcal{T}_H$ and $\mathcal{T}_L$ are described in detail while their respective results can be found in our experimental evaluation.

### A. Panel Pose Estimation ($\mathcal{T}_P$)

Our panel pose estimation approach described below is the basis for projecting the panel model and its kinematic properties into the simulation as illustrated in Fig. 3e. The estimation of accurate panel poses is crucial for reliable manipulation of valves and handles. Our approach incorporates offline knowledge such as the panel CAD model and visual markers placed at predefined locations. Based on this augmentation of the panel with markers, the panel pose in odometry frame ${}^{O}_{P}\mathbf{T}$ can be reliably estimated using the detected marker pose w.r.t. the camera frame ${}^{C}_{M}\mathbf{T}$, the camera pose on the robot frame ${}^{R}_{C}\mathbf{T}$, the panel pose in marker frame ${}^{M}_{P}\mathbf{T}$, and the current robot pose in odometry frame ${}^{O}_{R}\mathbf{T}$, see Fig. 3b:

$$ {}^{O}_{P}\mathbf{T} = {}^{O}_{R}\mathbf{T}\,{}^{R}_{C}\mathbf{T}\,{}^{C}_{M}\mathbf{T}\,{}^{M}_{P}\mathbf{T} \tag{1} $$

Consequently, $n$ marker observations lead to $n$ panel pose estimates ${}^{O}_{P}\mathbf{T}$ that eventually allow to compute the pose mean which includes mean position and orientation, determined by *spherical linear interpolation (Slerp)* [20].

### B. Panel Handle Pose Estimation ($\mathcal{T}_H$)

Once the panel pose has been estimated, the panel kinematic model can be exploited to approximate the orientations of each component (handles, wheels, valves, etc.). Accurate orientations are necessary to guarantee reliable manipulation of targets as required by further mission tasks.

Using the described knowledge base, a *region of interest* (ROI) in form of a 3D oriented bounding box is extracted according to the component model dimensions. Then an image-based approach estimates the component orientation using the extracted ROI projected into the image space. Fig. 4a shows the input image from one monocular view of the real stereo camera, and Fig. 4b shows the computed ROI of the handle labelled C3. For precise localization of the handles, the *superellipse-guided active contours segmentation* algorithm [21] is applied to the image patches representing each handle ROI. This algorithm is a particularly good fit

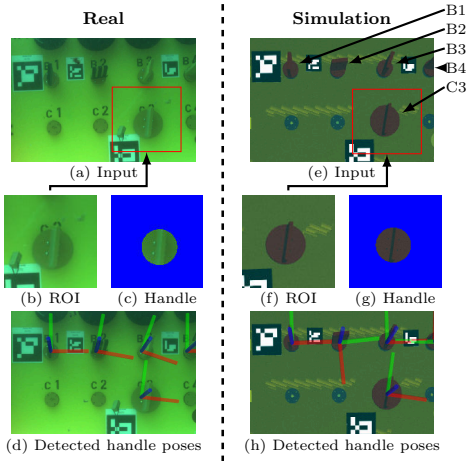



Fig. 4. Stages of the handle pose estimation algorithm. It can be tested for identical viewpoints on real ((a)–(d)) and simulation data (e)–(h)).

to our use case because the handles are round or ellipsoid-shaped when viewed from a frontal perspective.

Fig. 4c shows the result of using Fig. 4b as input for the superellipse-guided active contours segmentation. Subsequently, the handle state given by the lever position can be inferred from the most prominent straight edge in the lever's image. A Canny edge detector is used to detect edge points, followed by a Hough transformation for lines. Fig. 4d shows the estimated handle poses overlaid over the panel. Based on these estimated orientations, the overall state of the panel is accordingly updated (Fig. 3e). Then, thanks to the synchronized simulation, the algorithm is tested with an identical camera viewpoint in simulation for comparison, as shown in Fig. 4e–f; note that, handle pose estimates retrieved from real (Fig. 4d) and simulated (Fig. 4h) data may deviate due to different signal-to-noise in the respective data.

To further enhance the robustness of the algorithm when used in the envisioned scenario, a moving average of the detected lever orientations is employed to mitigate the effects of incorrect estimations on single frames. Moreover, both images from the stereo camera are used separately to estimate the handle pose, which gives us two samples from different perspectives at each time instance.

### C. Robot Localization ($\mathcal{T}_L$)

Accurate self-localization of vehicles is a challenging task, especially in the deep-sea domain, due to noisy sensor

readings typically based on acoustic devices like Ultra-Short Baseline (USBL) systems, single-beam or multi-beam sonars, Doppler Velocity Log (DVL) or relative readings provided by Inertial Navigation Systems (INS). Consequently, localization methods rely on multiple modalities to increase reliability [22], [23]. A typical and well-established approach to deal with sensor fusion is the Extended Kalman filter (EKF) [24] which allows to incorporate these modalities while considering their individual uncertainty.

However, as discussed in the previous section, reliable dexterous manipulation is a requirement in the DexROV scenario. To ensure robust control of the manipulator arm, accurate robot pose estimates are needed. Hence, we exploit the panel as a visual landmark again due to its static pose on the seafloor and its visual augmentation with multiple markers. Once the panel pose has been estimated, the robot pose can be inferred and used as an additional EKF input modality. In the following we describe our EKF-based localization system incorporating sensor readings and visual landmarks.

*1) Sensor Readings*

The robot setup provides a bank of sensors including INS, DVL, and USBL which together allow to gather readings of the current robot state regarding translation, orientation, linear/angular velocities and accelerations.

*2) Visual Landmark-Based Localization*

Fig. 3a shows a sample pose estimate of a visual marker, note that the panel is partially observed, used to infer the panel pose though the space transformations shown in Fig. 3b. Now the panel is taken as a fixed landmark and the robot pose ${}^{O}_{R}\mathbf{T}$ can be estimated as follows:

$$ {}^{O}_{R}\mathbf{T} = {}^{O}_{P}\mathbf{T}\ {}^{P}_{M}\mathbf{T}\ {}^{M}_{C}\mathbf{T}\ {}^{C}_{R}\mathbf{T} \tag{2} $$

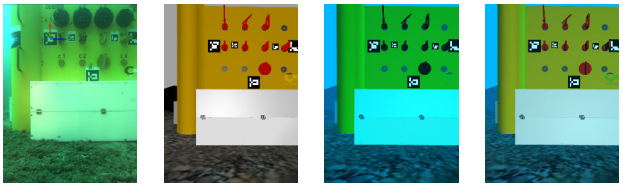
where ${}^{O}_{P}\mathbf{T}$ is the panel pose in odometry frame, ${}^{P}_{M}\mathbf{T}$ is one marker pose in panel frame, ${}^{M}_{C}\mathbf{T}$ is the camera pose w.r.t. the marker and ${}^{C}_{R}\mathbf{T}$ is the robot fixed pose w.r.t. the camera. Further on, the means of robot position ${}^{O}_{R}\bar{\mathbf{p}}$ and orientation ${}^{O}_{R}\bar{\mathbf{q}}$ w.r.t. the odometry frame are estimated from multiple marker detections using *Slerp*. In addition, a covariance matrix ${}^{O}_{R}\mathbf{C}$ for the robot pose is computed:

$$ {}^{O}_{R}\mathbf{C} = \mathrm{diag}(\sigma^2_{\mathbf{p}_x}, \sigma^2_{\mathbf{p}_y}, \sigma^2_{\mathbf{p}_z}, \sigma^2_{\mathbf{q}_\phi}, \sigma^2_{\mathbf{q}_\theta}, \sigma^2_{\mathbf{q}_\psi}). \tag{3} $$

The full robot pose estimate ${}^{O}_{R}\mathbf{T} = \langle {}^{O}_{R}\bar{\mathbf{p}}, {}^{O}_{R}\bar{\mathbf{q}} \rangle$ along with the respective covariance matrix ${}^{O}_{R}\mathbf{C}$ is then taken as an input for the localization filter in the final setup. Alternatively, it can be used as a ground truth value to optimize each of its components, i.e. sensor biases and associated covariances.

*3) Extended Kalman Filter*

In this work, we apply an *Extended Kalman Filter* (EKF) [24] to estimate the robot pose over time considering a state space consisting of position $x, y, z$, orientation $\phi, \theta, \psi$, translational $\dot{x}, \dot{y}, \dot{z}$, and angular velocities $\dot{\phi}, \dot{\theta}, \dot{\psi}$ as well as translational accelerations $\ddot{x}, \ddot{y}, \ddot{z}$. We only incorporate direct sensor measurements to the EKF, no integrated or differentiated values. INSs produce angular and linear accelerations, a DVL provides position outputs in form of altitude readings and linear velocities, and the mentioned landmarks are incorporated as pose readings. To increase the localization filter robustness, obvious outliers from sensor readings are rejected heuristically, and the pose inputs inferred from visual markers are tuned based on our experimental results.



(a) $m(\mathcal{T}_E) = 1.0$ (b) $m(\mathcal{T}_E) = 0.32$ (c) $m(\mathcal{T}_E) = 0.48$ (d) $m(\mathcal{T}_E) = 0.65$

Fig. 5. Simulated image adaptation to real environment conditions. Images correspond to (a) real-world, (b) default and (c–d) light adapted simulation.

## VI. Experimental Evaluation

In the following experiments we exploit the SIL concept with real-world data recorded during field trials in Marseille, France, in July 2017. The goal is to demonstrate the effectiveness of the proposed method to benchmark critical system components and several mission tasks $\mathcal{T}_i$ in deep-sea operations. This data contains sequences where the robot was used to verify the integrity of the testing panel and expected handle positions at up to 100 meters below sea level.

In the first experiment we perform an *environment condition adaptation* task ($\mathcal{T}_E$) to find the best possible simulation setup with respect to environment conditions. Next, we evaluate the performance of three benchmarking tasks: *panel pose estimation* ($\mathcal{T}_P$), *panel handle pose estimation* ($\mathcal{T}_H$) and *robot localization* ($\mathcal{T}_L$).

### A. Environment Condition Adaptation ($\mathcal{T}_E$)

In this task the simulated environment conditions $\mathcal{E}$ are tuned to reflect real environment conditions with high fidelity. We focus on adapting the light behavior to replicate the underwater camera distortions, i.e. light and color attenuation. The simulated stereo camera applies an exponential attenuation on the pixel intensity as described in [17]:

$$ i^*_c = i_c e^{-z a_c} + (1 - e^{-z a_c}) b_c \quad \forall c \in \{R, G, B\} \tag{4} $$

where $i_c$ and $b_c$ correspond to the pixel and background intensity value for color channel $c$, and $a_c$ is a color-dependent attenuation factor. The attenuation depends on the distance $z$ to the object projected on the camera pixel, which is extracted directly from the simulator depth-camera plugin.

We define the measure $m(\mathcal{T}_E)$ equivalent to the Feature Similarity Index (FSIM) [25] image quality measure between synchronized real and simulated images. 50 images recorded from the field trial were used to heuristically determine the most adequate light parameters for Equation 4; these images include different distances to the panel and perspectives. Fig. 5 shows some adapted image instances and their respective $m(\mathcal{T}_E) \in [0, 1]$. Fig. 5d shows the optimized adapted image under environment conditions $\mathcal{E}^*$, which are used in the next experiments to find the *expected system performance* in simulation.

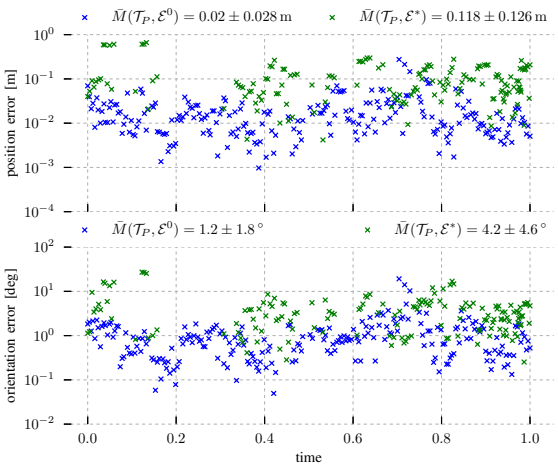



Fig. 6. Panel detection error $M(\mathcal{T}_P)$ for noise-free $\mathcal{E}^0$ and underwater $\mathcal{E}^*$ environment conditions

### B. Panel Pose Estimation ($\mathcal{T}_P$)

This first benchmarking task $\mathcal{T}_P$ evaluates the accuracy of the panel pose estimation, as it is the starting point for other tasks like handle pose estimation. This consequently validates the robustness of the used visual markers.

In simulation the robot navigates as in Fig. 8c; the trajectory was computed by extracting the robot poses given by the detected markers on real data and using them as waypoints in simulation. In this way, the same visual perspectives as in the field trial are obtained which represent a common routine trajectory given by the robot operators. Thus, we can determine the expected error as the difference between the ground-truth panel pose in simulation ${}^{O}_{P}\mathbf{T}_S$ and the panel pose determined from marker detection ${}^{O}_{P}\mathbf{T}_M$:

$$\begin{aligned} m(\mathcal{T}_P, \mathcal{E}) =& d\left({}^{O}_{P}\mathbf{T}_S,\, {}^{O}_{P}\mathbf{T}_M\right) \\ =& \langle d\left({}^{O}_{P}\bar{\mathbf{p}}_S,\, {}^{O}_{P}\bar{\mathbf{p}}_M\right), d\left({}^{O}_{P}\bar{\mathbf{q}}_S,\, {}^{O}_{P}\bar{\mathbf{q}}_M\right)\rangle \end{aligned} \quad (5)$$

where $d\left({}^{O}_{P}\bar{\mathbf{p}}_S,\, {}^{O}_{P}\bar{\mathbf{p}}_M\right)$ is the Euclidean distance between positions and $d\left({}^{O}_{P}\bar{\mathbf{q}}_S,\, {}^{O}_{P}\bar{\mathbf{q}}_M\right)$ is the minimal geodesic distance between orientations [26] under conditions $\mathcal{E}$.

Fig. 6 shows the mean $\bar{M}(\mathcal{T}_P, \mathcal{E})$ and standard deviation $\sigma(M(\mathcal{T}_P, \mathcal{E}))$ for all panel observations under noise free $\mathcal{E}^0$ and underwater-like conditions $\mathcal{E}^*$. As expected, our method has high accuracy under noise-free environment, and underwater image distortions decrease the accuracy and number of detections. With $\mathcal{E}^*$ conditions, we can expect a translation and orientation error of $0.118\,\mathrm{m}$ and $4.2\,^\circ$ respectively. For visual survey and navigation tasks this error is minimal. It can be overcome by image registration methods and the variance $\sigma^2(M(\mathcal{T}_P, \mathcal{E}^*))$ can be used to fine-tune the robot pose covariance matrix ${}^{O}_{R}\mathbf{C}$ (Section V-C) to improve localization, as it is an estimation of pose precision. For manipulation tasks, applying a moving average over the outputs produces good results as explained in Section V-B. Tasks $\mathcal{T}_H$ and $\mathcal{T}_L$ show this in the next experiments.

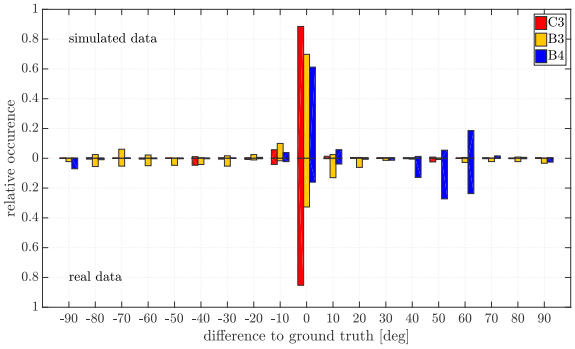



Fig. 7. Normalized histograms of handles orientation error $M(\mathcal{T}_H)$ for simulated and real data

### C. Panel Handle Pose Estimation ($\mathcal{T}_H$)

The previously described handle pose estimation was tested with the SIL framework, synchronizing the real and simulated camera point of view. Fig. 7 shows the normalized histograms of the error $M(\mathcal{T}_H)$ between the estimated handle orientation and ground truth from simulated and real data; note that, ground truth handle orientations are retrieved from visual inspection. In the course of the development, B1 and B2 have been modified to enhance the estimation accuracy by coloring the respective lever black in order to investigate contrast benefits for edge-based image algorithms as the one used (see Figs. 4a and 4e). This modification causes a discrepancy between simulated and real data, therefore B1 and B2 are disregarded in the results shown in Fig. 7. In general, the results in Fig. 7 show that the handle orientation estimates are predominantly within a low -10° to 10° error range. Depending on viewpoint, handle type and orientation, outliers can be observed for B4 in the range of 60° to 70°.

### D. Robot Localization ($\mathcal{T}_L$)

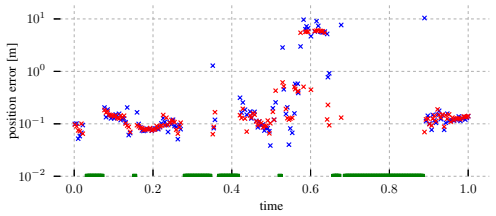
The final task $\mathcal{T}_L$ to benchmark with our SIL methodology is the localization method described in Section V-C. First we validate the use of visual landmarks in the localization filter through simulation under the found $\mathcal{E}^*$ conditions, then with real data we compare the task performance with and without the use of visual landmarks, and finally we tune the EKF parameters based on the results from tasks $\mathcal{T}_{L1}$ and $\mathcal{T}_P$.

*1) $\mathcal{T}_{L1}$ – localization in simulation*

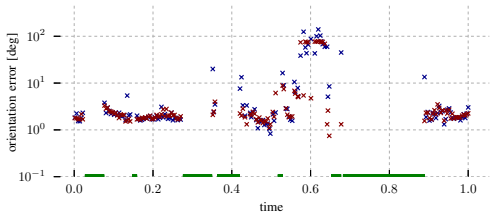
In the case of real-world underwater localization, no accurate ground-truth sensing is available. For this reason, the performance of the proposed localization filter that integrates visual landmarks into the EKF has to be tested in simulation first; and deployed afterwards in the field.

In this task $\mathcal{T}_{L1}$ the simulated robot again follows the trajectory shown in Fig. 8c. During this movement, the ground-truth robot pose in simulation ${}^{O}_{R}\mathbf{T}_S$ is recorded alongside the robot pose determined through the detected marker pose ${}^{O}_{R}\mathbf{T}_M$ and the localization filter ${}^{O}_{R}\mathbf{T}_F$. Note that the EKF receives only the visual landmark-based pose estimates to prove that it converges to ground truth.

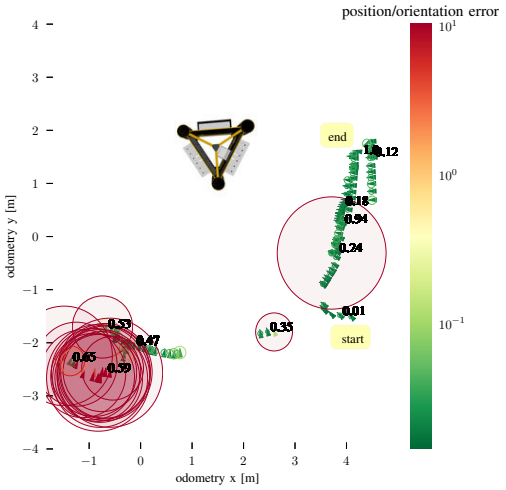
The pose estimate error of ${}^{O}_{R}\mathbf{T}_M$ and ${}^{O}_{R}\mathbf{T}_F$ with respect to simulation ground truth are shown in Fig. 8, denoted as $m_{S,M}(\mathcal{T}_{L1}) = d\left({}^{O}_{R}\mathbf{T}_S, {}^{O}_{R}\mathbf{T}_M\right)$ and $m_{S,F}(\mathcal{T}_{L1}) =$




(a) Position errors $d\left({}^{O}_{R}\bar{\mathbf{p}}_S,\ {}^{O}_{R}\bar{\mathbf{p}}_M\right)$ and $d\left({}^{O}_{R}\bar{\mathbf{p}}_S,\ {}^{O}_{R}\bar{\mathbf{p}}_F\right)$. No marker detected for sampling times marked green




(b) Orientation errors $d\left({}^{O}_{R}\bar{\mathbf{q}}_S,\ {}^{O}_{R}\bar{\mathbf{q}}_M\right)$ and $d\left({}^{O}_{R}\bar{\mathbf{q}}_S,\ {}^{O}_{R}\bar{\mathbf{q}}_F\right)$. No marker detected for sampling times marked green




(c) Robot poses (triangles) with orientation error $d\left({}^{O}_{R}\bar{\mathbf{q}}_S,\ {}^{O}_{R}\bar{\mathbf{q}}_F\right)$ (triangle color) and position error $d\left({}^{O}_{R}\bar{\mathbf{p}}_S,\ {}^{O}_{R}\bar{\mathbf{p}}_F\right)$ (circle color, log-scaled circle radius). Some time marks are shown next to some poses for reference

Fig. 8. $\mathcal{T}_{L1}$ results: position and orientation errors between ground-truth robot pose and marker-based / localization filter-based robot pose estimates, while the robot moves around the panel on a trajectory recorded in field trials

$d\left({}^{O}_{R}\mathbf{T}_S,\ {}^{O}_{R}\mathbf{T}_F\right)$ as in Equation 5. The trajectory in Fig. 8c and the detailed error breakdown in Fig. 8a–b show that whenever no marker has been detected for a while, the EKF error increases significantly on the next reading, but then quickly re-converges towards ground truth. On parts of the trajectory where markers are visible constantly, the localization error decreases satisfactorily below $0.3\,\mathrm{m}/3\,^\circ$ e.g. between time marks $0.1$ and $0.25$.

*2) $\mathcal{T}_{L2}$ – real-world localization using only navigation sensor data*

In order to get a baseline to compare the performance of the localization filter when integrating visual landmarks, in this subtask only DVL and INS measurements are used as inputs. This is also done because our focus is on tuning the use of visual markers in the EKF because navigation sensors are not integrated in the simulation at this development stage.

To evaluate this and the next tasks, see Table I-II, we use the robot pose estimate given by the marker ${}^{O}_{R}\mathbf{T}_M$ as reference ground truth, and compute the measure $m_{M,F}(\mathcal{T}_{Li}) = d\left({}^{O}_{R}\mathbf{T}_M,\ {}^{O}_{R}\mathbf{T}_F\right)$ plus the *lag-one autocorrelation* $m_A(\mathcal{T}_{Li}) = \sum_t {}^{O}_{R}\mathbf{T}_F(t)\,{}^{O}_{R}\mathbf{T}_F(t-1)$ on the EKF-predicted poses. $m_A(\mathcal{T}_{Li})$ is a measure of trajectory smoothness, important to prevent the robot from performing sudden jumps that can interfere with manipulation tasks.

*3) $\mathcal{T}_{L3}, \mathcal{T}_{L4}, \mathcal{T}_{L5}$ – real-world localization with visual markers*

In these tasks, we show the localization results using all sensor data recorded in field trials along with visual landmark-based pose estimates. A description of the respective tasks is given in Table I. The corresponding results are shown in Table II and Fig. 9.

As expected, the EKF instance with only navigation sensors as an input ($\mathcal{T}_{L2}$ – blue in Fig. 9) bears the largest error. Integrating the visual markers ($\mathcal{T}_{L3}$ – orange) significantly reduces the error and increases the number of jerky movements while navigating; this results in the worst $m_A(\mathcal{T}_{Li})$. Finally, we show that, based on previous tasks $\mathcal{T}_P$ and $\mathcal{T}_{L1}$ developed through our SIL methodology, the localization performance can be optimized adjusting the pose estimates covariances ($\mathcal{T}_{L4}$ – green) and rejecting outliers ($\mathcal{T}_{L5}$ – red). $\mathcal{T}_{L4}$ and $\mathcal{T}_{L5}$ yield similar accuracies, but the latter achieves the smoothest navigation trajectory. Certainly, as more sensors are integrated in simulation, performance can be further enhanced through simulation in the loop.

TABLE I. Description of localization tasks $\mathcal{T}_{Li}$

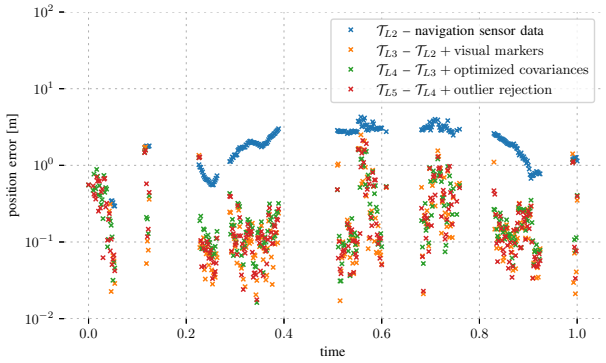
| Task | Description |
|---|---|
| $\mathcal{T}_{L2}$ | EKF with real-world data and only navigation sensors |
| $\mathcal{T}_{L3}$ | EKF with real-world data, using navigation sensors and visual markers (default parameters) |
| $\mathcal{T}_{L4}$ | $\mathcal{T}_{L3}$, plus covariance ${}^{O}_{R}\mathbf{C}$ of the robot pose estimates from marker detections adjustment with results from task $\mathcal{T}_P$, i.e. using $(0.126\,\mathrm{m})^2$ and $(4.6\,^\circ)^2$ (see Fig. 6) as diagonal values for single marker detections |
| $\mathcal{T}_{L5}$ | $\mathcal{T}_{L4}$, plus rejection of pose estimates whose distance $d\left({}^{O}_{R}\mathbf{T}_M,\ {}^{O}_{R}\mathbf{T}_F\right)$ to the current prediction are greater than $1\,\mathrm{m}$ and $12\,^\circ$; according to $\mathcal{T}_{L1}$ and Fig. 8a–8b |

TABLE II. Tasks $\mathcal{T}_{Li}$ measure results

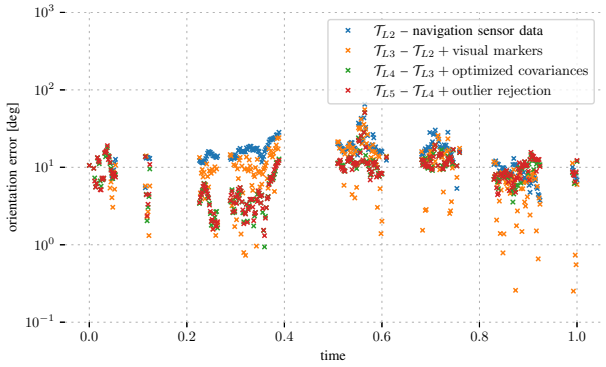
| | $\mathcal{T}_{L2}$ | $\mathcal{T}_{L3}$ | $\mathcal{T}_{L4}$ | $\mathcal{T}_{L5}$ |
|---|---|---|---|---|
| $\bar{m}_{M,F}(\mathcal{T}_{Li}\langle\bar{\mathbf{p}}\rangle)$[m] | 2.11±0.94 | 0.26±0.39 | 0.29±0.32 | 0.28±0.35 |
| $\bar{m}_{M,F}(\mathcal{T}_{Li}\langle\bar{\mathbf{q}}\rangle)$[deg] | 15.59±7.33 | 10.24±7.57 | 8.82±5.17 | 8.86±5.19 |
| $m_A(\mathcal{T}_{Li})$ | 0.95 | 0.72 | 0.91 | 0.94 |

## VII. Conclusion

Deep-sea robotic operations are cost intensive, and demand robustness and high reliability under harsh conditions. Measures have to be taken to guarantee the safety of the




(a) Position error $d\left({}^{O}_{R}\bar{\mathbf{p}}_M, {}^{O}_{R}\bar{\mathbf{p}}_F\right)$




(b) Orientation error $d\left({}^{O}_{R}\bar{\mathbf{q}}_M, {}^{O}_{R}\bar{\mathbf{q}}_F\right)$

Fig. 9. Localization benchmark tasks $\mathcal{T}_{Li}$ results

crew and the equipment. This includes not only robust pose estimation and localization algorithms, but also a versatile development framework including a realistic testbed.

In this work we presented a simulation in the loop (SIL) procedure that incorporates real observations into the simulation in a seamless manner by synchronization of simulated conditions with real-world data. Consequently, the components development progress can be instantaneously verified and benchmarked under simulated and real conditions. In our experimental evaluation we showed the benefit of the presented SIL approach on the DexROV research project. We were able to analyze and optimize critical components like robot localization considering the components' behavior under various environmental and spatial conditions.

## References


[1] UK Health & Safety Executive (HSE), "Offshore Safety Statistics Bulletin," http://www.hse.gov.uk/offshore/statistics/hsr2016.pdf, 2016, accessed: 2018-02-01.

[2] J. Gancet, P. Weiss, G. Antonelli, M. Pfingsthorn, S. Calinon, A. Turetta, C. Walen, D. Urbina, S. Govindaraj, P. Letier, X. Martinez, J. Salini, B. Chemisky, G. Indiveri, G. Casalino, P. di Lillo, E. Simetti, D. de Palma, A. Birk, T. Fromm, C. Mueller, A. Tanwani, I. Havoutis, A. Caffaz, and L. Guilpain, "Dexterous Undersea Interventions with Far Distance Onshore Supervision: the DexROV Project," in *IFAC Conference on Control Applications in Marine Systems*, 2016.

[3] T. Fromm, C. A. Mueller, M. Pfingsthorn, A. Birk, and P. Di Lillo, "Efficient Continuous System Integration and Validation for Deep-Sea Robotics Applications," in *Oceans (Aberdeen)*, 2017.

[4] M. Hildebrandt, L. Christensen, and F. Kirchner, "Combining Cameras, Magnetometers and Machine-Learning into a Close-Range Localization System for Docking and Homing," in *Oceans (Anchorage)*, 2017.

[5] A. Murarka, G. Kuhlmann, S. Gulati, M. Sridharan, C. Flesher, and W. C. Stone, "Vision-based Frozen Surface Egress: a Docking Algorithm for the Endurance AUV," in *Unmanned Untethered Submersible Technology*, 2009.

[6] R. Salas-Moreno, R. Newcombe, H. Strasdat, P. Kelly, and A. Davison, "SLAM++: Simultaneous Localisation and Mapping at the Level of Objects," in *Conference on Computer Vision and Pattern Recognition*, June 2013, pp. 1352–1359.

[7] D. Ribas, P. Ridao, J. D. Tardós, and J. Neira, "Underwater SLAM in man-made structured environments," *Journal of Field Robotics*, vol. 25, no. 11-12, pp. 898–921, 2008.

[8] B. Englot and F. Hover, "Inspection Planning for Sensor Coverage of 3D Marine Structures," in *International Conference on Intelligent Robots and Systems*, 2010.

[9] S. Garrido-Jurado, R. Munoz-Salinas, F. Madrid-Cuevas, and M. Marin-Jimenez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[10] T. Fromm and A. Birk, "Physics-Based Damage-Aware Manipulation Strategy Planning Using Scene Dynamics Anticipation," in *International Conference on Intelligent Robots and Systems*, 2016.

[11] P. Battaglia, J. Hamrick, and J. Tenenbaum, "Simulation as an engine of physical scene understanding." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 110, no. 45, pp. 18 327–32, 2013.

[12] L. Kunze and M. Beetz, "Envisioning the qualitative effects of robot manipulation actions using simulation-based projections," *Artificial Intelligence*, vol. 247, pp. 352–380, 2015.

[13] M. Beetz, M. Tenorth, and J. Winkler, "Open-EASE – A Knowledge Processing Service for Robots and Robotics/AI Researchers," in *International Conference on Robotics and Automation*, 2015.

[14] F. Lier, M. Hanheide, L. Natale, S. Schulz, J. Weisz, S. Wachsmuth, and S. Wrede, "Towards automated system and experiment reproduction in robotics," in *International Conference on Intelligent Robots and Systems*, 2016.

[15] M. Prats, J. Pérez, J. Fernández, and P. Sanz, "An Open Source Tool for Simulation and Supervision of Underwater Intervention Missions," in *International Conference on Intelligent Robots and Systems*, 2012.

[16] O. Kermorgant, "A Dynamic Simulator for Underwater Vehicle-Manipulators," in *Simulation, Modeling, and Programming for Autonomous Robots*, 2014.

[17] M. Manhães, S. Scherer, M. Voss, L. Douat, and T. Rauschenbach, "UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation," in *Oceans (Monterey)*, 2016.

[18] N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *International Conference on Intelligent Robots and Systems*, 2004.

[19] D. dos Santos Cesar, C. Gaudig, M. Fritsche, M. dos Reis, and F. Kirchner, "An Evaluation of Artificial Fiducial Markers in Underwater Environments," in *Oceans (Genova)*, 2015.

[20] K. Shoemake, "Animating rotation with quaternion curves," in *Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85, 1985, pp. 245–254.

[21] D. Koehntopp, B. Lehmann, D. Kraus, and A. Birk, "Segmentation and Classification Using Active Contours Based Superellipse Fitting on Side Scan Sonar Images for Marine Demining," in *International Conference on Robotics and Automation*, 2015.

[22] L. Chen, S. Wang, K. McDonald-Maier, and H. Hu, "Towards autonomous localization and mapping of AUVs: a survey," *International Journal of Intelligent Unmanned Systems*, vol. 1, no. 2, pp. 97–120, 2013.

[23] D. Li, D. Ji, J. Liu, and Y. Lin, "A Multi-Model EKF Integrated Navigation Algorithm for Deep Water AUV," *International Journal of Advanced Robotic Systems*, vol. 13, no. 1, p. 3, 2016.

[24] T. Moore and D. Stouch, "A Generalized Extended Kalman Filter Implementation for the Robot Operating System," in *International Conference on Intelligent Autonomous Systems*, 2014.

[25] L. Zhang, L. Zhang, X. Mou, and D. Zhang, "FSIM: A Feature Similarity Index for Image Quality Assessment," *Transactions on Image Processing*, vol. 20, no. 8, pp. 2378–2386, Aug 2011.

[26] D. Huynh, "Metrics for 3D Rotations: Comparison and Analysis," *Journal of Mathematical Imaging and Vision*, vol. 35, no. 2, pp. 155–164, 2009.

# B Test Data Set

**For 3x3; source: (0,1) destination (3,2)**
**Manhattan:**
(0,1) → (1,2) → (2,3) → (3,2)
**Diagonal:**
(0,1) → (1,1) → (2,1) → (3,2)
**Euclidean:**
(0,1) → (1,1) → (2,2) → (3,2)

**For 3x6; source: (0,5) destination (3,2)**
**Manhattan:**
(0,5) → (1,4) → (2,3) → (3,2)
**Diagonal:**
(0,5) → (1,4) → (2,3) → (3,2)
**Euclidean:**
(0,5) → (1,4) → (2,3) → (3,2)

**For 4x6; source: (0,5) destination (3,0)**
**Manhattan:**
(0,5) → (0,4) → (0,3) → (1,2) → (2,1) → (3,0)
**Diagonal:**
(0,5) → (0,4) → (0,3) → (1,2) → (2,1) → (3,0)
**Euclidean:**
(0,5) → (1,4) → (2,3) → (2,2) → (3,1) → (3,0)

**For 10x10; source: (0,9) destination (2,0)**
**Manhattan:**
(0,9) → (0,8) → (0,7) → (0,6) → (0,5) → (0,4) → (0,3) → (0,2) → (1,1) → (2,0)
**Diagonal:**
(0,9) → (0,8) → (0,7) → (0,6) → (0,5) → (0,4) → (0,3) → (0,2) → (1,1) → (2,0)
**Euclidean:**
(0,9) → (0,8) → (0,7) →(0,6) → (0,5) → (0,4) → (1,3) → (1,2) → (2,1) → (2,0)

**For 10x16; source: (0,15) destination (9,0)**
**Manhattan:**
(0,15) → (0,14) → (0,13) → (0,12) → (0,11) → (0,10) → (0,9) → (1,8) → (2,7) → (3,6) → (4,5) → (5,4) → (6,3) → (7,2) → (8,1) → (9,0)
**Diagonal:**
(0,15) → (0,14) → (0,13) → (0,12) → (0,11) → (0,10) → (0,9) → (1,8) → (2,7) → (3,6) → (4,5) → (5,4) → (6,3) → (7,2) → (8,1) → (9,0)
**Euclidean:**
(0,15) → (1,14) → (2,13) → (3,12) → (4,11) → (4,10) → (5,9) → (5,8) → (6,7) → (6,6) → (7,5) → (7,4) → (8,3) → (8,2) → (9,1) → (9,0)

**For 16x6; source: (0,15) destination (5,0)**
**Manhattan:**
(0,15) → (0,14) → (0,13) → (0,12) → (0,11) → (0,10) → (0,9) → (0,8) → (0,7) → (0,6) → (0,5) → (1,4) → (2,3) → (3,2) → (4,1) → (5,0)
**Diagonal**
(0,15) → (0,14) → (0,13) → (0,12) → (0,11) → (0,10) → (0,9) → (0,8) → (0,7) → (0,6) → (0,5) → (1,4) → (2,3) → (3,2) → (4,1) → (5,0)
**Euclidean:**
(0,15) → (0,14) → (0,13) → (0,12) → (0,11) → (0,10) → (1,9) → (1,8) → (2,7) → (2,6) → (3,5) → (3,4) → (4,3) → (4,2) → (5,1) → (5,0)

# C Paths with different Heuristics

### SOURCE 30,30 DESTINATION 360,360

### Kernel Size 3

**Dijkstra**
Path computed in vertices:
10→122→244→366→488→610→732→854→976→1098→1220→1342→1464→1586→1708→
1830→1952→2074→2196→2318→2440→2562→2684→2806→2928→3050→3172→3294→
3416→3538→3660→3782→3904→4026→4148→4149→4150→4272→4273→4274→4275→
4276→4398→4518→4640→4762→4884→5006→5128→5248→5368→5488→5608→5728→
5848→5968→5967→5966→5844→5843→5842→5841→5840→5839→5838→5958→6080→
6202→6324→6446→6568→6690→6812→6934→7056→7178→7300→7422→7544→7666→
7788→7910→8032→8154→8276→8398→8399→8400→8522→8523→8524→8646→8647→
8648→8649→8650→8772→8773→8774→8775→8897→8898→8899→8900→8901→9023→
9024→9025→9026→9148→9149→9029→9030→9031→9153→9154→9155→9156→9278→
9279→9280→9281→9403→9404→9405→9406→9407→9529→9649→9769→9889→10009→
10129→10249→10369→10489→10609→10729→10849→10969→11089→11209→11329→
11449→11569→11689→11688→11687→11686→11685→11684→11562→11561→11560→
11559→11558→11557→11556→11434→11433→11432→11431→11430→11429→11549→
11548→11668→11790→11912→12034→12156→12278→12400→12522→12644→12766→
12888→13010→13132→13254→13376→13498→13620→13742→13864→13986→14108→
14230→14352→14474→14596→14718→14719→14720→14721→14722→14723→14724→
14725→14726→14727→14728→14729→14730→14731→14732→14733→14734→14735→
14736→14737→14738→14739→14740→14741→14742→14743→14744→14745→14746→
14747→14748→14749→14750→14751→14752→14753→14754→14755→14756→14757→
14758→14759→14760→14761 = 231

**A***

**Euclidean:**
(10,10) →(10,11) →(11,12) →(12,13) →(13,14) →(14,15) →(15,16) →(16,17) →
(17,18) →(18,19) →(19,20) →(20,19) →(21,19) →(22,19) →(23,18) →(24,17) →
(25,16) →(26,15) →(27,15) →(28,14) →(29,13) →(30,12) →(31,13) →(32,14) →
(33,13) →(34,12) →(35,13) →(36,12) →(37,11) →(38,12) →(39,11) →(40,12) →
(41,11) →(42,10) →(43,11) →(44,10) →(45,9) →(46,10) →(47,9) →(48,8) → (49,9) →
(50,8) →(51,7) →(52,8) →(53,7) →(54,8) →(55,9) →(56,10) → (57,11) →(58,12) →
(59,13) →(60,14) →(61,15) →(62,16) →(63,17) →(64,18) → (65,19) →(66,20) →
(67,21) →(68,22) →(69,23) →(70,24) →(71,25) →(72,26) → (73,27) →(74,28) →
(75,29) →(76,30) →(77,31) →(78,32) →(79,33) →(80,34) → (81,35) →(82,36) →
(82,37) →(82,38) →(83,39) →(84,40) →(85,41) →(86,42) → (87,43) →(88,44) →
(89,45) →(90,46) →(91,47) →(91,48) →(91,49) →(92,50) → (93,51) →(93,52) →
(93,53) →(93,54) →(93,55) →(93,56) →(93,57) →(93,58) → (94,59) →(94,60) →
(94,61) →(94,62) →(95,63) →(96,64) →(97,65) →(98,66) → (99,67)→(100,68)→
(101,67)→(102,68)→(103,67)→(104,68)→(105,67)→(106,68)→ (107,69)→
(108,70) →(109,71)→(110,72) →(111,73) →(112,74) →(113,75) →(114,76) →

(114,77) →(114,78) →(114,79) →(114,80) →(114,81) →(114,82) →(114,83) →
(114,84) →(114,85) →(114,86) →(114,87) →(114,88) →(114,89) →(114,90) →
(114,91) →(114,92) →(114,93) →(114,94) →(114,95) →(114,96) →(114,97) →
(114,98) →(114,99) →(114,100) →(114,101) →(114,102) →(114,103) →(114,104)→
(114,105)→(114,106)→(114,107)→(114,108) →(114,109)→(114,110)→(114,111) →
(114,112)→(114,113)→(114,114)→(115,115) →(116,116) →(117,117) →(118,118) →
(119,119) →(120,120) = 164

**Diagonal:**

(10,10) →(10,11) →(11,12) →(12,13) →(13,14) →(14,15) →(15,16) →(16,17) →
(17,18) →(18,19) →(19,20) →(20,21) →(21,22) →(22,23) →(21,24) →(21,25) →
(20,26) →(20,27) →(20,28) →(20,29) →(21,30) →(22,31) →(22,32) →(22,33) →
(22,34) →(22,35) →(22,36) →(23,37) →(24,38) →(24,39) →(25,40) →(25,41) →
(25,42) →(26,43) →(27,44) →(27,45) →(27,46) →(28,47) →(27,48) →(28,49) →
(29,50) →(30,51) →(31,52) →(32,53) →(33,54) →(34,55) →(35,56) →(36,57) →
(37,57) →(38,57) →(39,57) →(40,57) →(41,58) →(42,59) →(43,59) →(44,60) →
(45,61) →(46,62) →(47,63) →(48,62) →(49,61) →(50,60) →(51,59) →(52,58) →
(53,57) →(54,56) →(55,55) →(56,56) →(57,55) →(58,55) →(58,54) →(59,53) →
(60,52) →(61,53) →(62,52) →(63,51) →(64,52) →(65,51) →(66,52) →(67,53) →
(68,52) →(69,51) →(70,50) →(71,51) →(72,50) →(73,49) →(74,48) →(75,47) →
(76,46) →(77,45) →(78,44) →(79,43) →(80,42) →(81,41) →(82,40) →(83,39) →
(84,40) →(85,41) →(86,42) →(87,43) →(88,44) →(89,45) →(90,46) →(91,47) →
(92,48) →(93,49) →(94,50) →(95,49) →(95,48) →(96,47) →(97,46) →(98,45) →
(99,44) →(100,43) →(101,44) →(102,45) →(103,45) →(104,44) →(105,43) →
(106,44) →(107,45) →(108,46) →(109,47) →(110,48) →(111,49) →(112,50) →
(112,51) →(112,52) →(112,53) →(112,54) →(113,55) →(114,56) →(115,57) →
(116,58) →(117,59) →(117,60) →(117,61) →(117,62) →(117,63) →(117,64) →
(117,65) →(117,66) →(117,67) →(117,68) →(117,69) →(117,70) →(117,71) →
(117,72) →(116,73) →(115,74) →(115,75) →(114,76) →(114,77) →(114,78) →
(114,79) →(114,80) →(114,81) →(114,82) →(114,83) →(114,84) →(114,85) →
(114,86) →(114,87) →(114,88) →(114,89) →(114,90) →(114,91) →(114,92) →
(114,93) →(114,94) →(114,95) →(114,96) →(114,97) →(114,98) →(114,99) →
(114,100) →(114,101)→(114,102)→(114,103)→(114,104)→(114,105)→(114,106) →
(114,107) →(114,108) →(114,109)→(114,110)→(114,111)→(114,112)→(114,113)→
(114,114) →(115,115) →(116,116) →(117,117) →(118,118) →(119,119) →(120,120) =
196

**Manhattan:**

(10,10) →(10,11) →(11,12) →(12,13) →(13,14) →(14,15) →(15,16) →(16,17) →
(17,18) →(18,19) →(19,20) →(20,21) →(21,20) →(22,20) →(23,20) →(24,19) →
(25,18) →(26,17) →(27,16) →(28,15) →(29,14) →(30,13) →(31,12) →(32,13) →
(33,14) →(34,15) →(35,16) →(36,17) →(37,18) →(38,19) →(39,20) →(40,19) →
(41,18) →(42,19) →(43,18) →(44,17) →(45,16) →(46,15) →(47,14) →(48,13) →
(49,12) →(50,11) →(51,12) →(52,13) →(53,14) →(54,15) →(55,16) →(56,17) →
(57,18) →(58,17) →(59,16) →(60,17) →(61,18) →(62,19) →(63,20) →(64,21) →
(65,22) →(66,21) →(67,22) →(68,23) →(69,24) →(70,25) →(71,26) →(72,27) →
(73,28) →(74,29) →(75,30) →(76,31) →(77,32) →(78,33) →(79,34) →(80,35) →
(81,34) →(82,35) →(83,36) →(84,37) →(85,38) →(86,39) →(87,40) →(88,41) →
(89,42) →(90,43) →(91,44) →(92,45) →(93,46) →(93,47) →(93,48) →(93,49) →
(93,50) →(93,51) →(93,52) →(93,53) →(93,54) →(94,55) →(95,56) →(95,57) →
(95,58) →(95,59) →(95,60) →(95,61) →(96,62) →(96,63) →(97,64) →(98,65) →

(98,66) →(99,67) →(100,68) →(101,69) →(102,70) →(103,69) →(104,70) →
(105,71) →(106,72) →(107,73) →(108,72) →(109,73) →(110,74) →(111,73) →
(112,74) →(113,75) →(114,76) →(114,77) →(114,78) →(114,79) →(114,80) →
(114,81) →(114,82) →(114,83) →(114,84) →(114,85) →(114,86) →(114,87) →
(114,88) →(114,89) →(114,90) →(114,91) →(114,92) →(114,93) →(114,94) →
(114,95) →(114,96) →(114,97) →(114,98) →(114,99) →(114,100)→(114,101)→
(114,102)→(114,103)→(114,104)→(114,105) →(114,106) →(114,107) →(114,108)→
(114,109)→(114,110)→(114,111)→(114,112)→(114,113) →(114,114) →(115,115) →
(116,116) →(117,117) →(118,118) →(119,119) →(120,120) = 165

**Kernel Size 5:**

**Dijkstra**
0→74→148→222→296→370→444→518→592→666→740→814→888→962→890→
818→892→893→894→895→896→897→971→1045→1046→1120→1121→1122→
1196→197→1271→1345→1419→1493→1567→1640→1713→1786→1860→1933→
2007→2081→2155→2229→2303→2304→2378→2379→2307→2308→2309→2310→
2238→2166→2094→2095→2023→2024→1952→1880→1807→1733→1732→1731→1803
= 65

**A***

**Euclidean:**
(6,6) →(7,7) →(8,8) →(9,9) →(10,10) →(11,11) →(12,11) →(13,11) → (14,11) →
(15,10) →(16,9) →(17,8) →(18,7) →(19,8) →(20,9) →(21,10) → (22,11) →(23,10) →
(24,11) →(25,10) →(26,9) →(27,8) →(28,7) →(29,6) → (30,7) →(31,6) →(32,7) →
(33,8) →(34,9) →(35,10) →(36,11) →(37,10) → (38,11) →(39,12) →(40,13) →(41,14) →
(42,15) →(43,16) →(44,15) →(45,16) → (46,17) →(47,18) →(48,19) →(49,20) →
(50,21) →(51,22) →(52,23) →(53,24) → (54,25) →(55,26) →(56,27) →(57,28) →
(58,27) →(59,26) →(60,25) →(61,24) → (62,23) →(63,22) →(64,21) →(65,20) →
(66,19) →(67,18) →(68,17) →(69,16) → (70,15) →(71,14) →(72,13) →(73,12) →
(73,13) →(73,14) →(73,15) →(73,16) → (73,17) →(73,18) →(73,19) →(73,20) →
(73,21) →(73,22) →(73,23) →(73,24) → (73,25) →(73,26) →(73,27) →(73,28) →
(73,29) →(73,30) →(73,31) →(73,32) → (73,33) →(73,34) →(73,35) →(73,36) →
(73,37) →(73,38) →(73,39) →(73,40) → (73,41) →(73,42) →(73,43) →(73,44) →
(73,45) →(73,46) →(73,47) →(73,48) → (73,49) →(73,50) →(73,51) →(73,52) →
(73,53) →(73,54) →(73,55) →(73,56) → (73,57) →(73,58) →(73,59) →(73,60) →
(73,61) →(73,62) →(73,63) →(73,64) → (73,65) →(73,66) →(73,67) →(73,68) →
(73,69) →(73,70) →(73,71) →(72,72) = 128
**Diagonal:**
(6,6) →(7,7) →(8,8) →(9,9) →(10,10) →(11,11) →(12,12) →(13,13) → (14,12) →
(14,11) →(15,10) →(16,9) →(17,8) →(18,7) →(19,8) →(20,9) → (21,10) →(22,11) →
(23,10) →(24,11) →(25,10) →(26,9) →(27,8) →(28,7) → (29,6) →(30,7) →(31,6) →
(32,7) →(33,8) →(34,9) →(35,10) →(36,11) → (37,10) →(38,11) →(39,12) →
(40,13) →(41,14) →(42,15) →(43,16) →(44,15) → (45,16) →(46,17) →(47,18) →
(48,19) →(49,20) →(50,21) →(51,22) →(52,23) → (53,24) →(54,25) →(55,26) →
(56,27) →(57,28) →(58,27) →(59,26) →(60,25) → (61,24) →(62,23) →(63,22) →
(64,21) →(65,20) →(66,19) →(67,18) →(68,17) → (69,16) →(70,15) →(71,14) →

(72,13) →(73,12) →(73,13) →(73,14) →(73,15) → (73,16) →(73,17) →(73,18) →
(73,19) →(73,20) →(73,21) →(73,22) →(73,23) → (73,24) →(73,25) →(73,26) →
(73,27) →(73,28) →(73,29) →(73,30) →(73,31) → (73,32) →(73,33) →(73,34) →
(73,35) →(73,36) →(73,37) →(73,38) →(73,39) → (73,40) →(73,41) →(73,42) →
(73,43) →(73,44) →(73,45) →(73,46) →(73,47) → (73,48) →(73,49) →(73,50) →
(73,51) →(73,52) →(73,53) →(73,54) →(73,55) → (73,56) →(73,57) →(73,58) →
(73,59) →(73,60) →(73,61) →(73,62) →(73,63) → (73,64) →(73,65) →(73,66) →
(73,67) →(73,68) →(73,69) →(73,70) →(73,71) → (72,72) = 129

**Manhattan:**

(6,6) →(7,7) →(8,8) →(9,9) →(10,10) →(11,11) →(12,11) →(13,11) → (14,11) →
(15,10) →(16,9) →(17,8) →(18,7) →(19,8) →(20,9) →(21,10) → (22,11) →(23,10) →
(24,11) →(25,10) →(26,9) →(27,8) →(28,7) →(29,6) → (30,7) →(31,6) →(32,7) →
(33,8) →(34,9) →(35,10) →(36,11) →(37,10) → (38,11) →(39,12) →(40,13) →(41,14) →
(42,15) →(43,16) →(44,15) →(45,16) → (46,17) →(47,18) →(48,19) →(49,20) →
(50,21) →(51,22) →(52,23) →(53,24) → (54,25) →(55,26) →(56,27) →(57,28) →
(58,27) →(59,26) →(60,25) →(61,24) → (62,23) →(63,22) →(64,21) →(65,20) →
(66,19) →(67,18) →(68,17) →(69,16) → (70,15) →(71,14) →(72,13) →(73,12) →
(73,13) →(73,14) →(73,15) →(73,16) → (73,17) →(73,18) →(73,19) →(73,20) →
(73,21) →(73,22) →(73,23) →(73,24) → (73,25) →(73,26) →(73,27) →(73,28) →
(73,29) →(73,30) →(73,31) →(73,32) → (73,33) →(73,34) →(73,35) →(73,36) →
(73,37) →(73,38) →(73,39) →(73,40) → (73,41) →(73,42) →(73,43) →(73,44) →
(73,45) →(73,46) →(73,47) →(73,48) → (73,49) →(73,50) →(73,51) →(73,52) →
(73,53) →(73,54) →(73,55) →(73,56) → (73,57) →(73,58) →(73,59) →(73,60) →
(73,61) →(73,62) →(73,63) →(73,64) → (73,65) →(73,66) →(73,67) →(73,68) →
(73,69) →(73,70) →(73,71) →(72,72) = 128

**Kernel size 7**

**Dijkstra:**

0→53→106→159→212→265→318→371→424→477→426→479→428→429→430→431→
484→537→590→591→592→593→646→699→751→804→856→908→960→1013→1066→
1119→1172→1173→1174→1175→1176→1177→1126→1075→1024→972→920→869→
817→766→714→663→610→557→506→454→402 = 53

**A***

**Euclidean:**

(4,4) →(5,5) →(6,6) →(7,7) →(8,8) →(9,8) →(10,7) →(11,7) → (12,6) →(13,5) →
(14,6) →(15,7) →(16,8) →(17,7) →(18,6) →(19,5) → (20,4) →(21,3) →(22,4) →
(23,5) →(24,6) →(25,7) →(26,8) →(27,7) → (28,8) →(29,9) →(30,10) →(31,11) →
(32,12) →(33,13) →(34,14) →(35,15) → (36,16) →(37,17) →(38,18) →(39,19) →
(40,20) →(41,19) →(42,18) →(43,17) → (44,16) →(45,15) →(46,14) →(47,13) →
(48,12) →(49,11) →(50,10) →(51,9) → (52,8) →(52,9) →(52,10) →(52,11) →
(52,12) →(52,13) →(52,14) →(52,15) → (52,16) →(52,17) →(52,18) →(52,19) →
(52,20) →(52,21) →(52,22) →(52,23) → (52,24) →(52,25) →(52,26) →(52,27) →
(52,28) →(52,29) →(52,30) →(52,31) → (52,32) →(52,33) →(52,34) →(52,35) →
(52,36) →(52,37) →(52,38) →(52,39) → (52,40) →(52,41) →(52,42) →(52,43) →

(52,44) →(52,45) →(52,46) →(52,47) → (52,48) →(52,49) →(52,50) →(51,51) = 92

**Diagonal:**
(4,4) →(5,5) →(6,6) →(7,7) →(8,8) →(9,9) →(10,8) →(10,7) → (11,6) →(12,5) →
(13,4) →(14,3) →(15,2) →(16,1) →(17,0) →(18,1) → (19,0) →(20,1) →(21,2) →
(22,3) →(23,4) →(24,5) →(25,6) →(26,7) → (27,8) →(28,7) →(29,8) →(30,9) →
(31,10) →(32,11) →(33,12) →(34,13) → (35,14) →(36,15) →(37,16) →(38,17) →
(39,18) →(40,19) →(41,18) →(42,17) → (43,16) →(44,15) →(45,14) →(46,13) →
(47,12) →(48,11) →(49,10) →(50,9) → (51,8) →(52,7) →(52,8) →(52,9) →(52,10) →
(52,11) →(52,12) →(52,13) → (52,14) →(52,15) →(52,16) →(52,17) →(52,18) →
(52,19) →(52,20) →(52,21) → (52,22) →(52,23) →(52,24) →(52,25) →(52,26) →
(52,27) →(52,28) →(52,29) → (52,30) →(52,31) →(52,32) →(52,33) →(52,34) →
(52,35) →(52,36) →(52,37) → (52,38) →(52,39) →(52,40) →(52,41) →(52,42) →
(52,43) →(52,44) →(52,45) → (52,46) →(52,47) →(52,48) →(52,49) →(52,50) →
(51,51) = 94

**Manhattan:**
(4,4) →(5,5) →(6,6) →(7,7) →(8,8) →(9,8) →(10,7) →(11,6) → (12,5) →(13,5) →
(14,6) →(15,7) →(16,8) →(17,7) →(18,6) →(19,5) → (20,4) →(21,3) →(22,4) →
(23,5) →(24,6) →(25,7) →(26,8) →(27,7) → (28,8) →(29,9) →(30,10) →(31,11) →
(32,12) →(33,13) →(34,14) →(35,15) → (36,16) →(37,17) →(38,18) →(39,19) →
(40,20) →(41,19) →(42,18) →(43,17) → (44,16) →(45,15) →(46,14) →(47,13) →
(48,12) →(49,11) →(50,10) →(51,9) → (52,8) →(52,9) →(52,10) →(52,11) →(52,12) →
(52,13) →(52,14) →(52,15) → (52,16) →(52,17) →(52,18) →(52,19) →(52,20) →
(52,21) →(52,22) →(52,23) → (52,24) →(52,25) →(52,26) →(52,27) →(52,28) →
(52,29) →(52,30) →(52,31) → (52,32) →(52,33) →(52,34) →(52,35) →(52,36) →
(52,37) →(52,38) →(52,39) → (52,40) →(52,41) →(52,42) →(52,43) →(52,44) →
(52,45) →(52,46) →(52,47) → (52,48) →(52,49) →(52,50) →(51,51) = 92

**SOURCE 180,180 DESTINATION 315,315**

**Kernel Size 3**

**Euclidean:**
(60,60) →(60,59) →(61,58) →(62,57) →(63,56) →(64,55) →(65,54) →(66,53) →
(67,52) →(68,51) →(69,50) →(70,49) →(71,48) →(72,47) →(73,46) →(74,45) →
(75,44) →(76,43) →(77,42) →(78,41) →(79,40) →(80,39) →(81,38) →(82,37) →
(82,38) →(83,39) →(84,40) →(85,41) →(86,42) →(87,43) →(88,44) →(89,45) →
(90,46) →(91,47) →(91,48) →(91,49) →(92,50) →(93,51) →(93,52) →(93,53) →
(93,54) →(93,55) →(93,56) →(93,57) →(93,58) →(94,59) →(94,60) →(94,61) →
(94,62) →(95,63) →(96,64) →(97,65) →(98,66) →(99,67) →(100,68) →(101,67) →
(102,68) →(103,69) →(104,70) →(105,71) →(106,72) →(107,73) →(107,74) →
(107,75) →(107,76) →(107,77) →(107,78) →(107,79) →(107,80) →(107,81) →
(107,82) →(107,83) →(107,84) →(107,85) →(107,86) →(107,87) →(106,88) →
(106,89) →(106,90) →(106,91) →(106,92) →(106,93) →(106,94) →(106,95) →
(106,96) →(106,97) →(106,98) →(106,99) →(106,100) →(106,101) →(106,102) →
(106,103) →(106,104) →(105,105) = 94

**Diagonal:**
(60,60) →(60,59) →(61,58) →(62,57) →(63,56) →(64,55) →(65,54) →(66,53) →
(67,52) →(68,51) →(69,50) →(70,51) →(71,52) →(72,51) →(73,50) →(74,49) →
(75,48) →(76,47) →(77,46) →(78,45) →(79,44) →(80,43) →(81,42) →(82,41) →
(83,40) →(84,39) →(85,40) →(86,41) →(87,42) →(88,43) →(89,44) →(90,45) →
(91,46) →(92,47) →(93,48) →(93,49) →(93,50) →(93,51) →(93,52) →(93,53) →
(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →(95,60) →(95,61) →
(96,62) →(96,63) →(97,64) →(98,65) →(98,66) →(99,67) →(100,68) →(101,69) →
(102,70) →(103,69) →(104,70) →(105,71) →(106,72) →(107,73) →(107,74) →
(107,75) →(107,76) →(107,77) →(106,78) →(106,79) →(106,80) →(106,81) →
(106,82) →(106,83) →(106,84) →(106,85) →(106,86) →(106,87) →(106,88) →
(106,89) →(106,90) →(106,91) →(106,92) →(106,93) →(106,94) →(106,95) →
(106,96) →(106,97) →(106,98) →(106,99) →(106,100) →(106,101) →(106,102) →
(106,103) →(106,104) →(105,105) = 94
**Manhattan:**
(60,60) →(60,59) →(61,58) →(62,57) →(63,56) →(64,55) →(65,54) →(66,53) →
(67,52) →(68,51) →(69,50) →(70,51) →(71,52) →(71,53) →(72,54) →(73,55) →
(74,56) →(75,57) →(76,58) →(77,57) →(78,58) →(79,59) →(80,58) →(81,57) →
(82,56) →(83,55) →(84,54) →(85,53) →(86,52) →(87,51) →(88,50) →(89,49) →
(90,48) →(91,47) →(92,48) →(93,49) →(93,50) →(93,51) →(93,52) →(93,53) →
(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →(95,60) →(95,61) →
(96,62) →(96,63) →(97,64) →(98,65) →(98,66) →(99,67) →(100,68) →(101,69) →
(102,70) →(103,69) →(104,70) →(105,71) →(106,72) →(107,73) →(107,74) →
(107,75) →(107,76) →(107,77) →(106,78) →(106,79) →(106,80) →(106,81) →
(106,82) →(106,83) →(106,84) →(106,85) →(106,86) →(106,87) →(106,88) →
(106,89) →(106,90) →(106,91) →(106,92) →(106,93) →(106,94) →(106,95) →
(106,96) →(106,97) →(106,98) →(106,99) →(106,100) →(106,101) →(106,102) →
(106,103) →(106,104) →(105,105) = 94

**Kernel Size 5**

**Euclidean:**
(36,36) →(37,36) →(38,35) →(39,34) →(40,33) →(41,32) →(42,31) →(43,30) →
(44,29) →(45,28) →(46,27) →(47,26) →(48,25) →(49,24) →(50,23) →(51,24) →
(52,25) →(53,26) →(54,27) →(55,28) →(55,29) →(56,30) →(57,29) →(58,28) →
(59,27) →(60,26) →(61,25) →(62,24) →(63,23) →(64,22) →(65,21) →(66,20) →
(67,19) →(68,18) →(69,17) →(70,16) →(71,15) →(72,14) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →
(73,39) →(73,40) →(73,41) →(73,42) →(73,43) →(73,44) →(73,45) →(73,46) →
(73,47) →(73,48) →(73,49) →(73,50) →(73,51) →(73,52) →(73,53) →(73,54) →
(73,55) →(73,56) →(73,57) →(73,58) →(73,59) →(73,60) →(73,61) →(73,62) →
(73,63) →(73,64) →(73,65) →(73,66) →(73,67) →(73,68) →(73,69) →(73,70) →
(73,71) →(72,72) →(71,72) →(70,72) →(69,72) →(68,72) →(67,72) →(66,72) →
(65,72) →(64,72) →(63,72) →(62,72) →(61,72) →(60,72) →(59,72) →(58,72) →
(57,72) →(56,72) →(57,71) →(58,70) →(59,69) →(60,68) →(61,67) →(62,66) →
(63,65) →(64,64) →(63,63) = 123

**Diagonal:**
(36,36) →(36,35) →(35,34) →(34,33) →(34,32) →(34,31) →(35,30) →(36,31) →
(37,30) →(38,31) →(39,30) →(40,31) →(41,30) →(42,29) →(43,28) →(44,27) →
(45,26) →(46,25) →(47,24) →(48,23) →(49,22) →(50,21) →(51,22) →(52,23) →
(53,24) →(54,25) →(55,26) →(56,27) →(57,28) →(58,27) →(59,26) →(60,25) →
(61,24) →(62,23) →(63,22) →(64,21) →(65,20) →(66,19) →(67,18) →(68,17) →
(69,16) →(70,15) →(71,14) →(72,13) →(73,12) →(73,13) →(73,14) →(73,15) →
(73,16) →(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →(73,23) →
(73,24) →(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →(73,31) →
(73,32) →(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →(73,39) →
(73,40) →(73,41) →(73,42) →(73,43) →(73,44) →(73,45) →(73,46) →(73,47) →
(73,48) →(73,49) →(73,50) →(73,51) →(73,52) →(73,53) →(73,54) →(73,55) →
(73,56) →(73,57) →(73,58) →(73,59) →(73,60) →(73,61) →(73,62) →(73,63) →
(73,64) →(73,65) →(73,66) →(73,67) →(73,68) →(73,69) →(73,70) →(73,71) →
(72,72) →(71,72) →(70,72) →(69,72) →(68,72) →(67,72) →(66,72) →(65,72) →
(64,72) →(63,72) →(62,72) →(61,72) →(60,72) →(59,72) →(58,72) →(57,72) →
(58,71) →(59,70) →(60,69) →(61,68) →(62,67) →(63,66) →(64,65) →(64,64) →
(63,63) = 129
**Manhattan:**
(36,36) →(36,35) →(35,34) →(34,33) →(34,32) →(34,31) →(35,30) →(36,31) →
(37,30) →(38,31) →(39,30) →(40,31) →(41,30) →(42,29) →(43,28) →(44,27) →
(45,26) →(46,25) →(47,24) →(48,23) →(49,22) →(50,21) →(51,22) →(52,23) →
(53,24) →(54,25) →(55,26) →(56,27) →(57,28) →(58,27) →(59,26) →(60,25) →
(61,24) →(62,23) →(63,22) →(64,21) →(65,20) →(66,19) →(67,18) →(68,17) →
(69,16) →(70,15) →(71,14) →(72,13) →(73,12) →(73,13) →(73,14) →(73,15) →
(73,16) →(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →(73,23) →
(73,24) →(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →(73,31) →
(73,32) →(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →(73,39) →
(73,40) →(73,41) →(73,42) →(73,43) →(73,44) →(73,45) →(73,46) →(73,47) →
(73,48) →(73,49) →(73,50) →(73,51) →(73,52) →(73,53) →(73,54) →(73,55) →
(73,56) →(73,57) →(73,58) →(73,59) →(73,60) →(73,61) →(73,62) →(73,63) →
(73,64) →(73,65) →(73,66) →(73,67) →(73,68) →(73,69) →(73,70) →(73,71) →
(72,72) →(71,72) →(70,72) →(69,72) →(68,72) →(67,72) →(66,72) →(65,72) →
(64,72) →(63,72) →(62,72) →(61,72) →(60,72) →(59,72) →(58,72) →(57,72) →
(56,72) →(57,71) →(58,70) →(59,69) →(60,68) →(61,67) →(62,66) →(63,65) →
(64,64) →(63,63) = 130

**Kernel Size 7**

**Euclidean:**
(25,25) →(26,25) →(27,24) →(28,23) →(29,22) →(30,21) →(31,20) →(32,19) →
(33,18) →(34,17) →(35,16) →(36,15) →(37,16) →(38,17) →(39,18) →(40,19) →
(41,19) →(42,18) →(43,17) →(44,16) →(45,15) →(46,14) →(47,13) →(48,12) →
(49,11) →(50,10) →(51,9) →(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →
(52,13) →(52,14) →(52,15) →(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →
(52,21) →(52,22) →(52,23) →(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →
(52,29) →(52,30) →(52,31) →(52,32) →(52,33) →(52,34) →(52,35) →(52,36) →
(52,37) →(52,38) →(52,39) →(52,40) →(52,41) →(52,42) →(52,43) →(52,44) →

(51,44) →(50,44) →(49,44) →(48,44) →(47,44) →(46,44) →(45,45) = 71

**Diagonal:**

(25,25) →(25,24) →(24,23) →(24,22) →(25,22) →(26,21) →(27,20) →(28,21) →
(29,20) →(30,19) →(31,18) →(32,17) →(33,16) →(34,15) →(35,14) →(36,15) →
(37,16) →(38,17) →(39,18) →(40,19) →(41,19) →(42,18) →(43,17) →(44,16) →
(45,15) →(46,14) →(47,13) →(48,12) →(49,11) →(50,10) →(51,9) →(52,8) →
(52,9) →(52,10) →(52,11) →(52,12) →(52,13) →(52,14) →(52,15) →(52,16) →
(52,17) →(52,18) →(52,19) →(52,20) →(52,21) →(52,22) →(52,23) →(52,24) →
(52,25) →(52,26) →(52,27) →(52,28) →(52,29) →(52,30) →(52,31) →(52,32) →
(52,33) →(52,34) →(52,35) →(52,36) →(52,37) →(52,38) →(52,39) →(52,40) →
(52,41) →(52,42) →(52,43) →(52,44) →(52,45) →(51,45) →(50,45) →(49,45) →
(48,45) →(47,45) →(46,45) →(45,45) = 76

**Manhattan:**

(25,25) →(25,24) →(24,23) →(24,22) →(25,22) →(26,21) →(27,20) →(28,21) →
(29,20) →(30,19) →(31,18) →(32,17) →(33,16) →(34,15) →(35,14) →(36,15) →
(37,16) →(38,17) →(39,18) →(40,19) →(41,19) →(42,18) →(43,17) →(44,16) →
(45,15) →(46,14) →(47,13) →(48,12) →(49,11) →(50,10) →(51,9) →(52,8) →
(52,9) →(52,10) →(52,11) →(52,12) →(52,13) →(52,14) →(52,15) →(52,16) →
(52,17) →(52,18) →(52,19) →(52,20) →(52,21) →(52,22) →(52,23) →(52,24) →
(52,25) →(52,26) →(52,27) →(52,28) →(52,29) →(52,30) →(52,31) →(52,32) →
(52,33) →(52,34) →(52,35) →(52,36) →(52,37) →(52,38) →(52,39) →(52,40) →
(52,41) →(52,42) →(52,43) →(52,44) →(51,44) →(50,44) →(49,44) →(48,44) →
(47,44) →(46,44) →(45,45) = 75

**SOURCE 60,60 DESTINATION 180,315**

**Kernel Size 3**

**Euclidean:**

(20,20) →(21,20) →(22,20) →(23,20) →(24,19) →(25,18) →(26,17) →(27,16) →
(28,15) →(29,14) →(30,13) →(31,12) →(32,13) →(33,14) →(34,13) →(35,12) →
(36,11) →(37,10) →(38,9) →(39,8) →(40,7) →(41,8) →(42,7) →(43,6) → (44,5) →
(45,4) →(46,3) →(47,2) →(48,3) →(49,4) →(50,5) →(51,6) → (52,7) →(53,8) →
(54,9) →(55,10) →(56,11) →(57,12) →(58,13) →(59,14) → (60,15) →(61,16) →
(62,17) →(63,18) →(64,19) →(65,20) →(66,21) →(67,22) → (68,23) →(69,24) →
(70,25) →(71,26) →(72,27) →(73,28) →(74,29) →(75,30) → (76,31) →(77,32) →
(78,33) →(79,34) →(80,35) →(81,34) →(82,35) →(83,36) → (84,37) →(85,38) →
(86,39) →(87,40) →(88,41) →(89,42) →(90,43) →(91,44) → (92,45) →(93,46) →
(93,47) →(93,48) →(93,49) →(93,50) →(93,51) →(93,52) → (93,53) →(93,54) →
(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →(95,60) → (95,61) →(96,62) →
(96,63) →(97,64) →(98,65) →(99,64) →(100,63) →(101,62) →(102,61) →(103,60) →
(104,59) →(105,60) = 100

**Manhattan:**

(20,20) →(21,20) →(22,20) →(23,20) →(24,19) →(25,18) →(26,17) →(27,16) →
(28,15) →(29,14) →(30,13) →(31,12) →(32,13) →(33,14) →(34,15) →(35,16) →
(36,17) →(37,18) →(38,19) →(39,20) →(40,19) →(41,18) →(42,19) →(43,18) →
(44,17) →(45,16) →(46,15) →(47,14) →(48,13) →(49,12) →(50,11) →(51,12) →
(52,13) →(53,14) →(54,15) →(55,16) →(56,17) →(57,18) →(58,17) →(59,16) →
(60,17) →(61,18) →(62,19) →(63,20) →(64,21) →(65,22) →(66,21) →(67,22) →

(68,23) →(69,24) →(70,25) →(71,26) →(72,27) →(73,28) →(74,29) →(75,30) →
(76,31) →(77,32) →(78,33) →(79,34) →(80,35) →(81,34) →(82,35) →(83,36) →
(84,37) →(85,38) →(86,39) →(87,40) →(88,41) →(89,42) →(90,43) →(91,44) →
(92,45) →(93,46) →(93,47) →(93,48) →(93,49) →(93,50) →(93,51) →(93,52) →
(93,53) →(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →(95,60) →
(95,61) →(95,62) →(96,61) →(97,60) →(98,59) →(99,58) →(100,57) →(101,58) →
(102,58) →(103,59) →(104,59) →(105,60) = 100

**Diagonal:**

(20,20) →(21,21) →(22,22) →(22,23) →(21,24) →(21,25) →(20,26) →(20,27) →
(20,28) →(20,29) →(21,30) →(22,31) →(22,32) →(22,33) →(22,34) →(22,35) →
(22,36) →(23,37) →(24,38) →(24,39) →(25,40) →(25,41) →(25,42) →(26,43) →
(27,44) →(27,45) →(27,46) →(28,47) →(27,48) →(28,49) →(29,50) →(30,51) →
(31,52) →(32,53) →(33,54) →(34,55) →(35,56) →(36,57) →(37,57) →(38,57) →
(39,57) →(40,57) →(41,58) →(42,59) →(43,59) →(44,60) →(45,60) →(46,60) →
(47,60) →(48,60) →(49,60) →(50,59) →(51,58) →(52,57) →(53,56) →(54,55) →
(55,54) →(56,53) →(57,52) →(58,52) →(59,52) →(60,52) →(61,52) →(62,52) →
(63,52) →(64,52) →(65,52) →(66,52) →(67,53) →(68,52) →(69,51) →(70,51) →
(71,52) →(71,53) →(72,54) →(73,55) →(74,56) →(75,57) →(76,58) →(77,58) →
(78,59) →(79,60) →(80,59) →(81,58) →(82,57) →(83,56) →(84,55) →(85,54) →
(86,53) →(87,52) →(88,51) →(89,50) →(90,49) →(91,48) →(92,49) →(93,50) →
(93,51) →(93,52) →(93,53) →(93,54) →(94,55) →(95,56) →(96,57) →(96,58) →
(95,59) →(95,60) →(95,61) →(95,62) →(96,61) →(97,60) →(98,59) →(99,58) →
(100,57) →(101,58) →(102,58) →(103,59) →(104,59) →(105,60) = 118

**Kernel Size 5**

**Euclidean:**

(12,12) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) →(19,8) → (20,9) →
(21,8) →(22,7) →(23,6) →(24,5) →(25,4) →(26,3) →(27,4) → (28,3) →(29,2) →(30,3) →
(31,2) →(32,3) →(33,4) →(34,5) →(35,6) → (36,7) →(37,8) →(38,9) →(39,10) →
(40,9) →(41,10) →(42,11) →(43,12) → (44,13) →(45,14) →(46,15) →(47,16) →
(48,17) →(49,18) →(50,19) →(51,20) → (52,21) →(53,22) →(54,23) →(55,24) →
(56,25) →(57,26) →(58,26) →(59,25) → (60,24) →(61,23) →(62,22) →(63,21) →
(64,20) →(65,19) →(66,18) →(67,17) → (68,16) →(69,15) →(70,14) →(71,13) →
(72,12) →(73,11) →(73,12) →(73,13) → (73,14) →(73,15) →(73,16) →(73,17) →
(73,18) →(73,19) →(73,20) →(73,21) → (73,22) →(73,23) →(73,24) →(73,25) →
(73,26) →(73,27) →(73,28) →(73,29) → (73,30) →(73,31) →(73,32) →(73,33) →
(73,34) →(73,35) →(73,36) →(73,37) → (73,38) →(73,39) →(72,40) →(71,40) →
(70,40) →(69,41) →(68,41) →(67,41) → (66,41) →(65,41) →(64,41) →(63,40) →
(64,39) →(65,38) →(64,37) →(63,36) = 104

**Manhattan:**

(12,12) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) →(19,8) →(20,9) →
(21,10) →(22,11) →(23,10) →(24,11) →(25,10) →(26,9) →(27,8) →(28,7) →(29,6) →
(30,7) →(31,6) →(32,7) →(33,8) →(34,9) →(35,10) →(36,11) →(37,10) →(38,11) →
(39,12) →(40,13) →(41,14) →(42,15) →(43,16) →(44,15) →(45,16) →(46,17) →
(47,18) →(48,19) →(49,20) →(50,21) →(51,22) →(52,23) →(53,24) →(54,25) →
(55,26) →(56,27) →(57,28) →(58,27) →(59,26) →(60,25) →(61,24) →(62,23) →
(63,22) →(64,21) →(65,20) →(66,19) →(67,18) →(68,17) →(69,16) →(70,15) →

(71,14) →(72,13) →(73,12) →(73,13) →(73,14) →(73,15) →(73,16) →(73,17) →
(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →(73,23) →(73,24) →(73,25) →
(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →(73,31) →(73,32) →(73,33) →
(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →(73,39) →(72,40) →(71,40) →
(70,40) →(69,41) →(68,41) →(67,40) →(66,40) →(65,40) →(64,40) →(63,40) →
(64,39) →(65,38) →(64,37) →(63,36) = 92

**Diagonal:**

(12,12) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) →(19,8) → (20,9) →
(21,10) →(22,11) →(23,11) →(24,11) →(25,10) →(26,9) →(27,9) → (28,8) →(29,7) →
(30,7) →(31,7) →(32,8) →(33,9) →(34,10) →(35,10) → (36,11) →(37,11) →(38,12) →
(39,11) →(40,12) →(41,13) →(42,14) →(43,15) → (44,16) →(45,17) →(46,18) →
(47,19) →(48,20) →(49,21) →(50,22) →(51,23) → (52,24) →(53,25) →(54,26) →
(55,27) →(56,28) →(57,27) →(58,27) →(59,26) → (60,25) →(61,24) →(62,23) →
(63,22) →(64,21) →(65,20) →(66,19) →(67,18) → (68,17) →(69,16) →(70,15) →
(71,14) →(72,13) →(73,12) →(73,13) →(73,14) → (73,15) →(73,16) →(73,17) →
(73,18) →(73,19) →(73,20) →(73,21) →(73,22) → (73,23) →(73,24) →(73,25) →
(73,26) →(73,27) →(73,28) →(73,29) →(73,30) → (73,31) →(73,32) →(73,33) →
(73,34) →(73,35) →(73,36) →(73,37) →(73,38) → (73,39) →(72,40) →(71,40) →
(70,40) →(69,41) →(68,41) →(67,40) →(66,40) → (65,39) →(64,40) →(63,40) →
(64,39) →(65,38) →(64,37) →(63,36) = 103

**Kernel Size 7**

**Euclidean:**

(8,8) →(9,8) →(10,7) →(11,6) →(12,5) →(13,5) →(14,6) →(15,7) → (16,6) →(17,5) →
(18,4) →(19,3) →(20,2) →(21,1) →(22,2) →(23,3) → (24,4) →(25,5) →(26,6) →(27,7) →
(28,8) →(29,7) →(30,8) →(31,9) → (32,10) →(33,11) →(34,12) →(35,13) →(36,14) →
(37,15) →(38,16) →(39,17) → (40,18) →(41,19) →(42,18) →(43,17) →(44,16) →
(45,15) →(46,14) →(47,13) → (48,12) →(49,11) →(50,10) →(51,9) →(52,8) →
(52,9) →(52,10) →(52,11) → (52,12) →(52,13) →(52,14) →(52,15) →(52,16) →
(52,17) →(52,18) →(52,19) → (52,20) →(52,21) →(52,22) →(52,23) →(52,24) →
(52,25) →(52,26) →(52,27) → (52,28) →(51,28) →(50,28) →(49,29) →(48,29) →
(47,29) →(46,29) →(45,29) → (46,28) →(47,27) →(46,26) →(45,25) = 76

**Manhattan:**

(8,8) →(9,8) →(10,7) →(11,6) →(12,5) →(13,5) →(14,6) →(15,7) → (16,8) →(17,7) →
(18,6) →(19,5) →(20,4) →(21,3) →(22,4) →(23,5) → (24,6) →(25,7) →(26,8) →
(27,7) →(28,8) →(29,9) →(30,10) →(31,11) → (32,12) →(33,13) →(34,14) →
(35,15) →(36,16) →(37,17) →(38,18) →(39,19) → (40,20) →(41,19) →(42,18) →
(43,17) →(44,16) →(45,15) →(46,14) →(47,13) → (48,12) →(49,11) →(50,10) →
(51,9) →(52,8) →(52,9) →(52,10) →(52,11) → (52,12) →(52,13) →(52,14) →
(52,15) →(52,16) →(52,17) →(52,18) →(52,19) → (52,20) →(52,21) →(52,22) →
(52,23) →(52,24) →(52,25) →(52,26) →(52,27) → (52,28) →(51,28) →(50,28) →
(49,29) →(48,29) →(47,29) →(46,29) →(45,29) → (46,28) →(47,27) →(46,26) →
(45,25) = 76

**Diagonal:**

(8,8) →(9,8) →(10,7) →(11,6) →(12,5) →(13,5) →(14,6) →(15,7) → (16,8) →(17,7) →
(18,7) →(19,6) →(20,5) →(21,4) →(22,5) →(23,5) → (24,6) →(25,7) →(26,8) →
(27,8) →(28,7) →(29,8) →(30,9) →(31,10) → (32,11) →(33,12) →(34,13) →(35,14) →

(36,15) →(37,16) →(38,17) →(39,18) → (40,19) →(41,19) →(42,18) →(43,17) →
(44,16) →(45,15) →(46,14) →(47,13) → (48,12) →(49,11) →(50,10) →(51,9) →
(52,8) →(52,9) →(52,10) →(52,11) → (52,12) →(52,13) →(52,14) →(52,15) →
(52,16) →(52,17) →(52,18) →(52,19) → (52,20) →(52,21) →(52,22) →(52,23) →
(52,24) →(52,25) →(52,26) →(52,27) → (52,28) →(51,28) →(50,28) →(49,29) →
(48,29) →(47,29) →(46,29) →(45,29) → (46,28) →(47,27) →(46,26) →(45,25) = 76

**SOURCE 60,90 DESTINATION 180,210**

**Kernel 3**

**Euclidean:**
(30,20) →(31,19) →(32,18) →(33,17) →(34,16) →(33,15) →(32,14) →(31,13) →
(30,13) →(29,14) →(28,15) →(28,16) →(27,17) →(26,18) →(26,19) →(25,20) →
(24,21) →(23,22) →(22,23) →(21,24) →(21,25) →(20,26) →(20,27) →(20,28) →
(20,29) →(21,30) →(22,31) →(22,32) →(22,33) →(22,34) →(22,35) →(22,36) →
(23,37) →(24,38) →(24,39) →(25,40) →(25,41) →(25,42) →(26,43) →(27,44) →
(27,45) →(27,46) →(27,47) →(27,48) →(28,49) →(29,50) →(30,50) →(31,50) →
(32,51) →(33,51) →(34,52) →(35,52) →(36,53) →(37,53) →(38,54) →(39,55) →
(40,56) →(41,57) →(42,58) →(43,59) →(44,60) →(45,61) →(46,62) →(47,63) →
(48,64) →(49,65) →(50,66) →(51,67) →(52,68) →(52,69) →(52,70) →(52,71) →
(52,72) →(52,73) →(52,74) →(52,75) →(52,76) →(52,77) →(53,76) →(54,75) →
(55,74) →(56,73) →(57,72) →(58,71) →(59,70) →(60,69) →(61,68) →(62,67) →
(63,66) →(64,65) →(65,64) →(66,63) →(67,62) →(68,61) →(69,60) →(70,60) = 96
**Manhattan:**
(30,20) →(31,19) →(32,18) →(33,17) →(34,16) →(33,15) →(32,14) →(31,13) →
(30,13) →(29,14) →(28,15) →(28,16) →(27,17) →(26,18) →(26,19) →(25,20) →
(24,21) →(23,22) →(22,23) →(21,24) →(21,25) →(20,26) →(20,27) →(20,28) →
(20,29) →(21,30) →(22,31) →(22,32) →(22,33) →(22,34) →(22,35) →(22,36) →
(23,37) →(24,38) →(24,39) →(25,40) →(25,41) →(25,42) →(26,43) →(27,44) →
(27,45) →(27,46) →(27,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,52) →
(32,53) →(33,54) →(34,55) →(35,56) →(36,57) →(37,56) →(38,57) →(39,56) →
(40,57) →(41,58) →(42,59) →(43,58) →(44,59) →(45,60) →(46,61) →(47,62) →
(48,63) →(49,64) →(50,65) →(51,66) →(52,67) →(52,68) →(52,69) →(52,70) →
(52,71) →(52,72) →(52,73) →(52,74) →(52,75) →(52,76) →(52,77) →(53,76) →
(54,75) →(55,74) →(56,73) →(57,72) →(58,71) →(59,70) →(60,69) →(61,68) →
(62,67) →(63,66) →(64,65) →(65,64) →(66,63) →(67,62) →(68,61) →(69,60) →
(70,60) = 97
**Diagonal:**
(30,20) →(31,19) →(32,18) →(33,17) →(34,16) →(33,15) →(32,14) →(31,13) →
(30,13) →(29,14) →(28,15) →(28,16) →(27,17) →(26,18) →(26,19) →(25,20) →
(24,21) →(23,22) →(22,23) →(21,24) →(21,25) →(20,26) →(20,27) →(20,28) →
(20,29) →(21,30) →(22,31) →(22,32) →(22,33) →(22,34) →(22,35) →(22,36) →
(23,37) →(24,38) →(24,39) →(25,40) →(25,41) →(25,42) →(26,43) →(27,44) →
(27,45) →(27,46) →(27,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,52) →
(32,53) →(33,54) →(34,55) →(35,56) →(36,57) →(37,56) →(38,57) →(39,56) →
(40,57) →(41,58) →(42,59) →(43,58) →(44,59) →(45,60) →(46,61) →(47,62) →
(48,63) →(49,64) →(50,65) →(51,66) →(52,67) →(52,68) →(52,69) →(52,70) →

(52,71) →(52,72) →(52,73) →(52,74) →(52,75) →(52,76) →(52,77) →(53,76) →
(54,75) →(55,74) →(56,73) →(57,72) →(58,71) →(59,70) →(60,69) →(61,68) →
(62,67) →(63,66) →(64,65) →(65,64) →(66,63) →(67,62) →(68,61) →(69,60) →
(70,60) = 97

**Kernel 5**
**Euclidean:**
(18,12) →(19,11) →(20,10) →(21,9) →(22,8) →(23,7) →(24,6) →(25,5)→ (26,4) →
(27,3) →(28,2) →(29,1) →(30,2) →(31,3) →(32,4) →(33,5) → (34,4) →(35,5) →
(36,6) →(37,7) →(38,8) →(39,9) →(40,10) →(41,11) → (42,12) →(43,13) →(44,14) →
(45,15) →(46,16) →(47,17) →(48,18) →(49,19) → (50,20) →(51,21) →(52,22) →
(53,23) →(54,24) →(55,25) →(56,26) →(57,27) → (58,26) →(59,25) →(60,24) →
(61,23) →(62,22) →(63,21) →(64,20) →(65,19) → (66,18) →(67,17) →(68,16) →
(69,15) →(70,14) →(71,13) →(72,12) →(73,11) → (73,12) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) → (73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) → (73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) → (73,36) →(73,37) →(73,38) →
(73,39) →(73,40) →(73,41) →(73,42) →(73,43) → (73,44) →(73,45) →(73,46) →
(73,47) →(73,48) →(73,49) →(73,50) →(73,51) → (73,52) →(73,53) →(73,54) →
(73,55) →(73,56) →(73,57) →(73,58) →(73,59) → (73,60) →(73,61) →(73,62) →
(73,63) →(73,64) →(73,65) →(73,66) →(73,67) → (73,68) →(73,69) →(73,70) →
(73,71) →(72,72) →(71,72) →(70,72) →(69,72) → (68,72) →(67,72) →(66,72) →
(65,72) →(64,72) →(63,72) →(62,72) →(61,72) → (60,72) →(59,72) →(58,72) →
(57,72) →(56,72) →(55,72) →(54,72) →(53,72) → (52,72) →(51,72) →(50,72) →
(49,72) →(48,72) →(47,72) →(46,72) →(45,72) → (44,72) →(43,72) →(42,72) →
(41,72) →(40,72) →(39,72) →(38,72) →(37,72) → (36,72) →(35,72) →(34,72) →
(33,72) →(32,72) →(31,72) →(30,72) →(29,72) → (28,72) →(27,72) →(26,72) →
(25,72) →(24,72) →(23,72) →(22,72) →(21,72) → (20,72) →(19,72) →(18,72) →
(17,72) →(16,72) →(15,72) →(14,72) →(13,72) → (12,72) →(11,72) →(10,72) →
(9,72) →(10,71) →(11,70) →(12,69) →(13,68) → (14,67) →(15,66) →(16,65) →
(17,64) →(18,63) →(19,62) →(20,61) →(21,60) → (22,59) →(23,58) →(24,57) →
(25,56) →(25,55) →(26,54) →(27,53) →(27,52) → (28,51) →(29,50) →(30,49) →
(31,48) →(32,47) →(33,46) →(34,45) →(35,44) → (36,43) →(37,42) →(38,41) →
(39,40) →(40,39) →(41,38) →(42,37) →(42,36) = 216
**Manhattan:**
(18,12) →(19,11) →(20,10) →(21,11) →(22,10) →(23,11) →(24,10) →(25,9) →
(26,8) →(27,9) →(28,8) →(29,7) →(30,6) →(31,7) →(32,8) →(33,9) → (34,10) →
(35,9) →(36,10) →(37,11) →(38,12) →(39,11) →(40,12) →(41,13) → (42,14) →
(43,15) →(44,16) →(45,17) →(46,18) →(47,19) →(48,20) →(49,21) → (50,22) →
(51,23) →(52,24) →(53,25) →(54,26) →(55,27) →(56,28) →(57,27) → (58,27) →
(59,26) →(60,25) →(61,24) →(62,23) →(63,22) →(64,21) →(65,20) → (66,19) →
(67,18) →(68,17) →(69,16) →(70,15) →(71,14) →(72,13) →(73,12) → (73,13) →
(73,14) →(73,15) →(73,16) →(73,17) →(73,18) →(73,19) →(73,20) → (73,21) →
(73,22) →(73,23) →(73,24) →(73,25) →(73,26) →(73,27) →(73,28) → (73,29) →
(73,30) →(73,31) →(73,32) →(73,33) →(73,34) →(73,35) →(73,36) → (73,37) →
(73,38) →(73,39) →(73,40) →(73,41) →(73,42) →(73,43) →(73,44) → (73,45) →
(73,46) →(73,47) →(73,48) →(73,49) →(73,50) →(73,51) →(73,52) → (73,53) →
(73,54) →(73,55) →(73,56) →(73,57) →(73,58) →(73,59) →(73,60) → (73,61) →
(73,62) →(73,63) →(73,64) →(73,65) →(73,66) →(73,67) →(73,68) → (73,69) →

(73,70) →(73,71) →(72,72) →(71,72) →(70,72) →(69,72) →(68,72) → (67,72) →
(66,72) →(65,72) →(64,72) →(63,72) →(62,72) →(61,72) →(60,72) → (59,72) →
(58,72) →(57,72) →(56,72) →(55,72) →(54,72) →(53,72) →(52,72) → (51,72) →
(50,72) →(49,72) →(48,72) →(47,72) →(46,72) →(45,72) →(44,72) → (43,72) →
(42,72) →(41,72) →(40,72) →(39,72) →(38,72) →(37,72) →(36,72) → (35,72) →
(34,72) →(33,72) →(32,72) →(31,72) →(30,72) →(29,72) →(28,72) → (27,72) →
(26,72) →(25,72) →(24,72) →(23,72) →(22,72) →(21,72) →(20,72) → (19,72) →
(18,72) →(17,72) →(16,72) →(15,72) →(14,72) →(13,72) →(12,72) → (11,72) →
(10,72) →(9,72) →(10,71) →(11,70) →(12,69) →(13,68) →(14,67) → (15,66) →
(16,65) →(17,64) →(18,63) →(19,62) →(20,61) →(21,60) →(22,59) → (23,58) →
(24,57) →(25,56) →(25,55) →(26,54) →(27,53) →(27,52) →(28,51) → (29,50) →
(30,49) →(31,48) →(32,47) →(33,46) →(34,45) →(35,44) →(36,43) → (37,42) →
(38,41) →(39,40) →(40,39) →(41,38) →(42,37) →(42,36) = 215

**Diagonal:**

(18,12) →(19,11) →(20,10) →(21,11) →(22,10) →(23,11) →(24,10) →(25,9) →
(26,8) →(27,9) →(28,8) →(29,7) →(30,6) →(31,7) →(32,8) →(33,9) → (34,10) →
(35,9) →(36,10) →(37,11) →(38,12) →(39,11) →(40,12) →(41,13) → (42,14) →
(43,15) →(44,16) →(45,17) →(46,18) →(47,19) →(48,20) →(49,21) → (50,22) →
(50,23) →(51,24) →(52,25) →(53,26) →(54,27) →(55,28) →(56,29) → (57,28) →
(58,27) →(59,26) →(60,25) →(61,24) →(62,23) →(63,22) →(64,21) → (65,20) →
(66,19) →(67,18) →(68,17) →(69,16) →(70,15) →(71,14) →(72,13) → (73,12) →
(73,13) →(73,14) →(73,15) →(73,16) →(73,17) →(73,18) →(73,19) → (73,20) →
(73,21) →(73,22) →(73,23) →(73,24) →(73,25) →(73,26) →(73,27) → (73,28) →
(73,29) →(73,30) →(73,31) →(73,32) →(73,33) →(73,34) →(73,35) → (73,36) →
(73,37) →(73,38) →(73,39) →(73,40) →(73,41) →(73,42) →(73,43) → (73,44) →
(73,45) →(73,46) →(73,47) →(73,48) →(73,49) →(73,50) →(73,51) → (73,52) →
(73,53) →(73,54) →(73,55) →(73,56) →(73,57) →(73,58) →(73,59) → (73,60) →
(73,61) →(73,62) →(73,63) →(73,64) →(73,65) →(73,66) →(73,67) → (73,68) →
(73,69) →(73,70) →(73,71) →(72,72) →(71,72) →(70,72) →(69,72) → (68,72) →
(67,72) →(66,72) →(65,72) →(64,72) →(63,72) →(62,72) →(61,72) → (60,72) →
(59,72) →(58,72) →(57,72) →(56,72) →(55,72) →(54,72) →(53,72) → (52,72) →
(51,72) →(50,72) →(49,72) →(48,72) →(47,72) →(46,72) →(45,72) → (44,72) →
(43,72) →(42,72) →(41,72) →(40,72) →(39,72) →(38,72) →(37,72) → (36,72) →
(35,72) →(34,72) →(33,72) →(32,72) →(31,72) →(30,72) →(29,72) → (28,72) →
(27,72) →(26,72) →(25,72) →(24,72) →(23,72) →(22,72) →(21,72) → (20,72) →
(19,72) →(18,72) →(17,72) →(16,72) →(15,72) →(14,72) →(13,72) → (12,72) →
(11,72) →(12,71) →(13,70) →(14,69) →(15,68) →(16,67) →(17,66) → (17,65) →
(18,64) →(19,63) →(20,62) →(21,61) →(22,60) →(23,59) →(23,58) → (24,57) →
(25,56) →(25,55) →(26,54) →(27,53) →(27,52) →(28,51) →(29,50) → (30,49) →
(31,48) →(32,47) →(33,46) →(34,45) →(35,44) →(36,43) →(37,42) → (38,41) →
(39,40) →(40,39) →(41,38) →(42,37) →(42,36) = 214

**Kernel 7**

**Euclidean:**

(12,8) →(13,7) →(14,6) →(15,7) →(16,8) →(17,7) →(18,6) →(19,5) → (20,4) →
(21,3) →(22,4) →(23,5) →(24,6) →(25,7) →(26,8) →(27,7) → (28,8) →(29,9) →
(30,10) →(31,11) →(32,12) →(33,13) →(34,14) →(35,15) → (36,16) →(36,17) →

(37,18) →(36,19) →(35,20) →(34,21) →(33,22) →(32,23) → (31,24) →(30,25) = 34
**Manhattan:**
(12,8) →(11,7) →(10,8) →(9,9) →(8,9) →(8,10) →(8,11) →(8,12) → (8,13) →(8,14) →
(8,15) →(9,16) →(9,17) →(10,18) →(11,19) →(11,20) → (11,21) →(12,22) →(13,23) →
(14,22) →(15,23) →(16,24) →(17,23) →(18,24) → (19,25) →(20,26) →(21,27) →
(22,28) →(22,29) →(22,30) →(22,31) →(23,30) → (24,29) →(25,28) →(26,27) →
(27,26) →(28,25) →(29,24) →(30,25) = 39
**Diagonal:**
(12,8) →(11,7) →(10,8) →(9,9) →(8,9) →(8,10) →(8,11) →(8,12) → (8,13) →(8,14) →
(8,15) →(9,16) →(9,17) →(10,18) →(11,19) →(11,20) → (11,21) →(12,22) →(13,23) →
(14,23) →(15,24) →(16,24) →(17,24) →(18,24) → (19,25) →(20,26) →(21,27) →
(22,28) →(22,29) →(22,30) →(22,31) →(23,30) → (24,29) →(25,28) →(26,27) →
(27,26) →(28,25) →(29,24) →(30,25) = 39

**SOURCE 70,300 DESTINATION 315,180**

**Kernel 3**

**Euclidean:**
(23,100) →(24,99) →(25,98) →(26,97) →(27,96) →(28,95) →(29,94) →(30,93) →
(31,92) →(32,91) →(33,90) →(34,89) →(35,88) →(36,87) →(37,86) →(38,85) →
(39,84) →(40,83) →(41,82) →(42,81) →(43,80) →(44,79) →(45,78) →(46,77) →
(47,76) →(48,75) →(49,74) →(50,73) →(51,72) →(52,71) →(53,70) →(53,69) →
(53,68) →(53,67) →(53,66) →(53,65) →(53,64) →(53,63) →(53,62) →(53,61) →
(54,60) →(55,59) →(56,58) →(57,57) →(57,56) →(57,55) →(58,55) →(59,54) →
(60,53) →(61,52) →(62,51) →(63,50) →(64,49) →(65,48) →(66,47) →(67,46) →
(68,45) →(69,44) →(70,43) →(71,42) →(72,41) →(73,40) →(74,39) →(75,38) →
(76,37) →(77,36) →(78,35) →(79,34) →(80,35) →(81,34) →(82,35) →(83,36) →
(84,37) →(85,38) →(86,39) →(87,40) →(88,41) →(89,42) →(90,43) →(91,44) →
(92,45) →(92,46) →(92,47) →(92,48) →(93,49) →(93,50) →(93,51) →(93,52) →
(93,53) →(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →(95,60) →
(95,61) →(96,62) →(96,63) →(97,64) →(98,65) →(99,64) →(100,63) →(101,62) →
(102,61) →(103,60) →(104,59) →(105,60) = 108
**Manhattan:**
(23,100) →(24,99) →(25,98) →(26,97) →(27,96) →(28,95) →(29,94) →(30,93) →
(31,92) →(32,91) →(33,90) →(34,89) →(35,88) →(36,87) →(37,86) →(38,85) →
(39,84) →(40,83) →(41,82) →(42,81) →(43,80) →(44,79) →(45,78) →(46,77) →
(47,76) →(48,75) →(49,74) →(50,73) →(51,72) →(52,71) →(53,70) →(53,69) →
(53,68) →(53,67) →(53,66) →(53,65) →(53,64) →(52,63) →(51,62) →(50,61) →
(49,60) →(50,59) →(51,58) →(52,57) →(53,56) →(54,55) →(55,54) →(56,53) →
(57,52) →(58,53) →(59,52) →(60,53) →(61,52) →(62,53) →(63,52) →(64,51) →
(65,52) →(66,51) →(67,52) →(68,51) →(69,50) →(70,51) →(71,52) →(71,53) →
(72,54) →(73,55) →(74,56) →(75,57) →(76,58) →(77,57) →(78,58) →(79,59) →
(80,58) →(81,57) →(82,56) →(83,55) →(84,54) →(85,53) →(86,52) →(87,51) →
(88,50) →(89,49) →(90,48) →(91,47) →(92,48) →(93,49) →(93,50) →(93,51) →
(93,52) →(93,53) →(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →
(95,60) →(95,61) →(95,62) →(96,61) →(97,60) →(98,59) →(99,58) →(100,57) →
(101,58) →(102,58) →(103,59) →(104,59) →(105,60) = 109

**Diagonal:**
(23,100) →(24,99) →(25,98) →(26,97) →(27,96) →(28,95) →(29,94) →(30,93) →
(31,92) →(32,91) →(33,90) →(34,89) →(35,88) →(36,87) →(37,86) →(38,85) →
(39,84) →(40,83) →(41,82) →(42,81) →(43,80) →(44,79) →(45,78) →(46,77) →
(47,76) →(48,75) →(49,74) →(50,73) →(51,72) →(52,71) →(53,70) →(53,69) →
(53,68) →(53,67) →(53,66) →(53,65) →(53,64) →(53,63) →(53,62) →(53,61) →
(54,60) →(53,59) →(54,58) →(53,57) →(54,56) →(55,55) →(56,56) →(57,55) →
(58,55) →(58,54) →(59,53) →(60,53) →(61,53) →(62,53) →(63,52) →(64,52) →
(65,52) →(66,52) →(67,53) →(68,52) →(69,51) →(70,51) →(71,52) →(71,53) →
(72,54) →(73,55) →(74,56) →(75,57) →(76,58) →(77,58) →(78,59) →(79,60) →
(80,59) →(81,58) →(82,57) →(83,56) →(84,55) →(85,54) →(86,53) →(87,52) →
(88,51) →(89,50) →(90,49) →(91,48) →(92,49) →(93,50) →(93,51) →(93,52) →
(93,53) →(93,54) →(94,55) →(95,56) →(96,57) →(96,58) →(95,59) →(95,60) →
(95,61) →(95,62) →(96,61) →(97,60) →(98,59) →(99,58) →(100,57) →(101,58) →
(102,58) →(103,59) →(104,59) →(105,60) = 108

**Kernel Size 5**
**Euclidean:**
(14,60) →(15,59) →(16,58) →(17,57) →(18,56) →(19,55) →(20,54) →(21,53) →
(22,52) →(23,51) →(24,50) →(25,49) →(26,48) →(27,47) →(28,46) →(29,45) →
(30,44) →(31,43) →(32,42) →(31,41) →(30,40) →(31,39) →(30,38) →(31,37) →
(31,36) →(32,35) →(31,34) →(32,33) →(33,32) →(34,32) →(34,31) →(35,30) →
(36,29) →(37,28) →(38,27) →(39,26) →(40,25) →(41,24) →(42,23) →(43,22) →
(44,21) →(45,20) →(46,19) →(47,18) →(48,17) →(49,18) →(50,19) →(51,20) →
(52,21) →(53,22) →(54,23) →(55,24) →(56,25) →(57,26) →(58,27) →(59,26) →
(60,25) →(61,24) →(62,23) →(63,22) →(64,21) →(65,20) →(66,19) →(67,18) →
(68,17) →(69,16) →(70,15) →(71,14) →(72,13) →(73,12) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →
(73,39) →(72,40) →(71,40) →(70,40) →(69,41) →(68,41) →(67,41) →(66,41) →
(65,41) →(64,41) →(63,40) →(64,39) →(65,38) →(64,37) →(63,36) = 111
**Manhattan:**
(14,60) →(15,59) →(16,58) →(17,57) →(18,56) →(19,55) →(20,54) →(21,53) →
(22,52) →(23,51) →(24,50) →(25,49) →(26,48) →(27,47) →(28,46) →(29,45) →
(30,44) →(31,43) →(32,42) →(31,41) →(31,40) →(31,39) →(31,38) →(30,37) →
(29,36) →(30,35) →(31,34) →(32,33) →(33,32) →(34,32) →(34,31) →(35,30) →
(36,31) →(37,30) →(38,31) →(39,30) →(40,31) →(41,30) →(42,29) →(43,28) →
(44,27) →(45,26) →(46,25) →(47,24) →(48,23) →(49,22) →(50,21) →(51,22) →
(52,23) →(53,24) →(54,25) →(55,26) →(56,27) →(57,28) →(58,27) →(59,26) →
(60,25) →(61,24) →(62,23) →(63,22) →(64,21) →(65,20) →(66,19) →(67,18) →
(68,17) →(69,16) →(70,15) →(71,14) →(72,13) →(73,12) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →
(73,39) →(72,40) →(71,40) →(70,40) →(69,41) →(68,41) →(67,40) →(66,40) →
(65,40) →(64,40) →(63,40) →(64,39) →(65,38) →(64,37) →(63,36) = 111
**Diagonal:**
(14,60) →(15,59) →(16,58) →(17,57) →(18,56) →(19,55) →(20,54) →(21,53) →

(22,52) →(23,51) →(24,50) →(25,49) →(26,48) →(27,47) →(28,46) →(29,45) →
(30,44) →(31,43) →(32,42) →(31,41) →(31,40) →(31,39) →(31,38) →(31,37) →
(31,36) →(32,35) →(31,34) →(32,33) →(33,32) →(34,32) →(34,31) →(35,31) →
(36,31) →(37,31) →(38,31) →(39,30) →(40,31) →(41,30) →(42,30) →(43,29) →
(44,28) →(45,27) →(46,26) →(47,25) →(48,24) →(49,23) →(50,22) →(51,23) →
(52,24) →(53,25) →(54,26) →(55,27) →(56,28) →(57,27) →(58,27) →(59,26) →
(60,25) →(61,24) →(62,23) →(63,22) →(64,21) →(65,20) →(66,19) →(67,18) →
(68,17) →(69,16) →(70,15) →(71,14) →(72,13) →(73,12) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) →
(73,39) →(72,40) →(71,40) →(70,40) →(69,41) →(68,41) →(67,40) →(66,40) →
(65,39) →(64,40) →(63,40) →(64,39) →(65,38) →(64,37) →(63,36) = 111

**Kernel Size 7**

**Euclidean:**
(10,42) →(11,41) →(12,40) →(13,39) →(14,38) →(15,37) →(16,36) →(17,35) →
(18,34) →(19,33) →(20,32) →(21,31) →(22,30) →(21,29) →(22,28) →(21,27) →
(22,26) →(22,25) →(21,24) →(22,23) →(23,22) →(24,21) →(25,20) →(26,19) →
(27,18) →(28,17) →(29,16) →(30,15) →(31,14) →(32,13) →(33,12) →(34,13) →
(35,14) →(36,15) →(37,16) →(38,17) →(39,18) →(40,19) →(41,18) →(42,17) →
(43,16) →(44,15) →(45,14) →(46,13) →(47,12) →(48,11) →(49,10) →(50,9) →
(51,8) →(52,7) →(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →(52,13) → (52,14) →
(52,15) →(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →(52,21) → (52,22) →
(52,23) →(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →(51,28) → (50,28) →
(49,29) →(48,29) →(47,29) →(46,29) →(45,29) →(46,28) →(47,27) → (46,26) →(45,25)
= 82
**Manhattan:**
(10,42) →(11,41) →(12,40) →(13,39) →(14,38) →(15,37) →(16,36) →(17,35) →
(18,34) →(19,33) →(20,32) →(21,31) →(22,30) →(22,29) →(22,28) →(22,27) →
(21,26) →(20,25) →(21,24) →(22,23) →(23,22) →(24,21) →(25,20) →(26,19) →
(27,18) →(28,17) →(29,16) →(30,15) →(31,14) →(32,13) →(33,14) →(34,13) →
(35,14) →(36,15) →(37,16) →(38,17) →(39,18) →(40,19) →(41,19) →(42,18) →
(43,17) →(44,16) →(45,15) →(46,14) →(47,13) →(48,12) →(49,11) →(50,10) →
(51,9) →(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →(52,13) →(52,14) →
(52,15) →(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →(52,21) →(52,22) →
(52,23) →(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →(51,28) →(50,28) →
(49,29) →(48,29) →(47,29) →(46,29) →(45,29) →(46,28) →(47,27) →(46,26) →
(45,25) = 81
**Diagonal:**
(10,42) →(11,41) →(12,40) →(13,39) →(14,38) →(15,37) →(16,36) →(17,35) →
(18,34) →(19,33) →(20,32) →(21,31) →(22,30) →(22,29) →(22,28) →(22,27) →
(22,26) →(22,25) →(21,24) →(22,23) →(23,22) →(24,21) →(25,20) →(26,19) →
(27,18) →(28,17) →(29,16) →(30,15) →(31,14) →(32,13) →(33,14) →(34,13) →
(35,14) →(36,15) →(37,16) →(38,17) →(39,18) →(40,19) →(41,19) →(42,18) →
(43,17) →(44,16) →(45,15) →(46,14) →(47,13) →(48,12) →(49,11) →(50,10) →
(51,9) →(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →(52,13) →(52,14) →

(52,15) →(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →(52,21) →(52,22) →
(52,23) →(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →(51,28) →(50,28) →
(49,29) →(48,29) →(47,29) →(46,29) →(45,29) →(46,28) →(47,27) →(46,26) →
(45,25) = 81

**SOURCE 70,300 DESTINATION 210,300**

**Kernel 3**

**Euclidean:**
(23,100) →(24,99) →(25,98) →(26,97) →(27,96) →(28,95) →(29,94) →(30,93) →
(31,92) →(32,91) →(33,90) →(34,89) →(35,88) →(36,88) →(37,89) →(38,90) →
(39,91) →(40,92) →(41,91) →(42,92) →(43,93) →(44,94) →(45,95) →(46,94) →
(47,93) →(48,92) →(49,91) →(50,90) →(51,89) →(52,88) →(53,87) →(54,88) →
(55,87) →(56,86) →(57,87) →(58,88) →(59,89) →(60,90) →(61,91) →(62,92) →
(63,93) →(64,94) →(65,95) →(66,96) →(67,97) →(68,98) →(69,99) →(70,100) = 48
**Manhattan:**
(23,100) →(24,99) →(25,98) →(26,97) →(27,96) →(28,95) →(29,94) →(30,93) →
(31,92) →(32,91) →(33,90) →(34,89) →(35,88) →(36,89) →(37,90) →(38,91) →
(39,92) →(40,92) →(41,93) →(42,94) →(43,95) →(44,94) →(45,95) →(46,94) →
(47,93) →(48,92) →(49,91) →(50,90) →(51,89) →(52,88) →(53,87) →(54,88) →
(55,87) →(56,86) →(57,87) →(58,88) →(59,89) →(60,90) →(61,91) →(62,92) →
(63,93) →(64,94) →(65,95) →(66,96) →(67,97) →(68,98) →(69,99) →(70,100) = 48
**Diagonal:**
(23,100) →(22,101) →(21,102) →(20,103) →(19,104) →(19,105) →(20,106) →
(21,105) →(22,104) →(23,103) →(24,102) →(25,101) →(26,100) →(27,99) →
(28,98) →(29,97) →(30,96) →(31,95) →(32,94) →(33,93) →(34,92) →(35,91) →
(36,90) →(37,91) → (38,92) →(39,92) →(40,92) →(41,93) →(42,94) →(43,95) →
(44,94) →(45,95) → (46,94) →(47,93) →(48,92) →(49,91) →(50,90) →(51,89) →
(52,88) →(53,87) → (54,88) →(55,87) →(56,86) →(57,87) →(58,88) →(59,89) →
(60,90) →(61,91) → (62,92) →(63,93) →(64,94) →(65,95) →(66,96) →(67,97) →
(68,98) →(69,99) → (70,100) = 49

**Kernel Size 5**

**Euclidean:**
(14,60) →(15,59) →(16,58) →(17,57) →(18,56) →(19,55) →(20,54) →(21,53) →
(20,53) →(19,53) →(18,53) →(17,53) →(16,53) →(15,53) →(14,53) →(13,53) →
(12,52) →(11,53) →(10,54) →(9,55) →(9,56) →(9,57) →(9,58) →(9,59) →
(10,60) →(10,61) →(10,62) →(10,63) →(10,64) →(9,65) →(9,66) →(9,67) →
(9,68) →(8,69) →(7,70) →(7,71) →(7,72) →(8,72) →(9,72) →(10,72) → (11,72) →
(12,72) →(13,72) →(14,72) →(15,72) →(16,72) →(17,72) →(18,72) → (19,72) →
(20,72) →(21,72) →(22,72) →(23,72) →(24,72) →(25,72) →(26,72) → (27,72) →
(28,72) →(29,71) →(30,70) →(31,69) →(32,68) →(33,67) →(34,66) → (35,65) →
(36,64) →(37,63) →(38,62) →(39,61) →(40,60) →(41,59) →(42,60) = 72
**Manhattan:**
(14,60) →(15,59) →(16,58) →(17,57) →(18,56) →(19,55) →(20,54) →(21,53) →

(20,53) →(19,53) →(18,53) →(17,53) →(16,53) →(15,53) →(14,53) →(13,53) →
(12,52) →(11,53) →(10,54) →(9,55) →(9,56) →(9,57) →(9,58) →(10,59) →
(10,60) →(10,61) →(10,62) →(10,63) →(10,64) →(9,65) →(9,66) →(9,67) →
(9,68) →(8,69) →(7,70) →(7,71) →(7,72) →(8,72) →(9,72) →(10,72) → (11,72) →
(12,72) →(13,72) →(14,72) →(15,72) →(16,72) →(17,72) →(18,72) → (19,72) →
(20,72) →(21,72) →(22,72) →(23,72) →(24,72) →(25,72) →(26,72) → (27,72) →
(28,72) →(29,71) →(30,70) →(31,69) →(32,68) →(33,67) →(34,66) → (35,65) →
(36,64) →(37,63) →(38,62) →(39,61) →(40,60) →(41,59) →(42,60) = 72

**Diagonal:**

(14,60) →(13,60) →(12,61) →(11,62) →(11,63) →(10,64) →(9,65) →(9,66) →
(9,67) →(9,68) →(8,69) →(7,70) →(7,71) →(7,72) →(8,72) →(9,72) → (10,72) →
(11,72) →(12,72) →(13,72) →(14,72) →(15,72) →(16,72) →(17,72) → (18,72) →
(19,72) →(20,72) →(21,72) →(22,72) →(23,72) →(24,72) →(25,72) → (26,72) →
(27,72) →(28,72) →(29,71) →(30,70) →(31,69) →(32,68) →(33,67) → (34,66) →
(35,65) →(36,64) →(37,63) →(38,62) →(39,61) →(40,60) →(41,59) → (42,60) = 49

**Kernel Size 7**

**Euclidean:** (10,42) →(11,41) →(12,40) →(13,39) →(14,38) →(15,37) →(16,38) →
(16,39) → (17,40) →(18,39) →(19,40) →(20,40) →(21,39) →(22,38) →(23,37) →
(24,36) → (25,37) →(26,38) →(27,39) →(28,40) →(29,41) →(30,42) = 22

**Manhattan:**

(10,42) →(11,41) →(12,40) →(13,39) →(14,38) →(15,37) →(16,38) →(16,39) →
(17,40) →(18,39) →(19,40) →(20,40) →(21,39) →(22,38) →(23,37) →(24,36) →
(25,37) →(26,38) →(27,39) →(28,40) →(29,41) →(30,42) = 22

**Diagonal:**

(10,42) →(11,41) →(12,40) →(13,39) →(14,38) →(15,37) →(16,38) →(16,39) →
(17,40) →(18,39) →(19,40) →(20,40) →(21,39) →(22,38) →(23,37) →
(24,36) → (25,37) →(26,38) →(27,39) →(28,40) →(29,41) →(30,42) = 22

**SOURCE 60,70 DESTINATION 300,140**

**Kernel 3**

**Euclidean:**

(20,23) →(21,22) →(22,21) →(23,20) →(24,19) →(25,18) →(26,17) →(27,16) →
(28,15) →(29,14) →(30,13) →(31,12) →(32,13) →(33,14) →(34,13) →(35,12) →
(36,11) →(37,10) →(38,9) →(39,8) →(40,7) →(41,8) →(42,7) →(43,6) → (44,5) →
(45,4) →(46,3) →(47,4) →(48,3) →(49,2) →(50,3) →(51,4) → (52,5) →(53,4) →
(54,5) →(55,6) →(56,7) →(57,8) →(58,9) →(59,10) → (60,11) →(61,12) →(62,13) →
(63,12) →(64,13) →(65,14) →(66,15) →(67,16) → (68,17) →(69,18) →(70,19) →
(71,20) →(72,19) →(73,20) →(74,21) →(75,22) → (76,23) →(77,24) →(78,25) →
(79,26) →(80,27) →(81,28) →(82,29) →(83,30) → (84,31) →(85,32) →(86,33) →
(87,34) →(88,35) →(89,36) →(90,37) →(91,38) → (92,39) →(93,40) →(94,41) →
(95,42) →(96,43) →(97,44) →(98,45) →(99,45) → (100,46) = 81

**Manhattan:**

(20,23) →(21,22) →(22,21) →(23,20) →(24,19) →(25,18) →(26,17) →(27,16) →

(28,15) →(29,14) →(30,13) →(31,12) →(32,13) →(33,14) →(34,15) →(35,16) →
(36,17) →(37,18) →(38,19) →(39,20) →(40,19) →(41,18) →(42,19) →(43,18) →
(44,17) →(45,16) →(46,15) →(47,14) →(48,13) →(49,12) →(50,11) →(51,12) →
(52,13) →(53,14) →(54,15) →(55,16) →(56,17) →(57,18) →(58,17) →(59,16) →
(60,17) →(61,18) →(62,19) →(63,20) →(64,21) →(65,22) →(66,21) →(67,22) →
(68,23) →(69,24) →(70,25) →(71,26) →(72,27) →(73,28) →(74,29) →(75,30) →
(76,31) →(77,32) →(78,33) →(79,34) →(80,35) →(81,34) →(82,35) →(83,36) →
(84,37) →(85,38) →(86,39) →(87,40) →(88,41) →(89,42) →(90,43) →(91,44) →
(92,45) →(93,46) →(94,45) →(95,46) →(96,45) →(97,46) →(98,45) →(99,45) →
(100,46) = 81

**Diagonal:**

(20,23) →(21,24) →(21,25) →(20,26) →(20,27) →(20,28) →(20,29) →(21,30) →
(22,31) →(22,32) →(22,33) →(22,34) →(22,35) →(22,36) →(23,37) →(24,38) →
(24,39) →(25,40) →(25,41) →(25,42) →(26,43) →(27,44) →(27,45) →(27,46) →
(27,47) →(27,48) →(28,49) →(29,50) →(30,50) →(31,50) →(32,51) →(33,51) →
(34,52) →(35,52) →(36,53) →(37,53) →(38,53) →(39,53) →(40,53) →(41,53) →
(42,52) →(43,51) →(44,50) →(45,49) →(46,49) →(47,50) →(48,51) →(49,52) →
(50,53) →(51,54) →(52,55) →(53,54) →(54,53) →(55,52) →(56,51) →(57,50) →
(58,49) →(59,48) →(60,47) →(61,47) →(62,47) →(63,47) →(64,47) →(65,47) →
(66,47) →(67,47) →(68,47) →(69,47) →(70,48) →(71,49) →(72,50) →(73,49) →
(74,48) →(75,47) →(76,46) →(77,45) →(78,44) →(79,43) →(80,42) →(81,41) →
(82,40) →(83,39) →(84,40) →(85,41) →(86,42) →(87,43) →(88,44) →(89,45) →
(90,46) →(91,46) →(92,46) →(93,46) →(94,46) →(95,46) →(96,46) →(97,46) →
(98,45) →(99,45) →(100,46) = 99

**Kernel Size 5**

**Euclidean:**

(12,14) →(13,13) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) → (19,8) →
(20,9) →(21,8) →(22,7) →(23,6) →(24,5) →(25,4) →(26,3) → (27,4) →(28,3) →
(29,2) →(30,3) →(31,2) →(32,3) →(33,4) →(34,5) → (35,6) →(36,7) →(37,8) →
(38,9) →(39,10) →(40,9) →(41,10) →(42,11) → (43,12) →(44,13) →(45,14) →
(46,15) →(47,16) →(48,17) →(49,18) →(50,19) → (51,20) →(52,21) →(53,22) →
(54,23) →(55,24) →(56,25) →(57,26) →(58,27) → (59,26) →(60,27) →(60,28) = 51

**Manhattan:**

(12,14) →(13,13) →(14,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) → (19,8) →
(20,9) →(21,10) →(22,11) →(23,10) →(24,11) →(25,10) →(26,9) → (27,8) →(28,7) →
(29,6) →(30,7) →(31,6) →(32,7) →(33,8) →(34,9) → (35,10) →(36,11) →(37,10) →
(38,11) →(39,12) →(40,13) →(41,14) →(42,15) → (43,16) →(44,15) →(45,16) →
(46,17) →(47,18) →(48,19) →(49,20) →(50,21) → (51,22) →(52,23) →(53,24) →
(54,25) →(55,26) →(56,27) →(57,28) →(58,27) → (59,26) →(60,27) →(60,28) = 51

**Diagonal:**

(12,14) →(13,13) →(14,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) → (19,8) →
(20,9) →(21,10) →(22,11) →(23,11) →(24,11) →(25,10) →(26,9) → (27,9) →
(28,8) →(29,7) →(30,7) →(31,7) →(32,8) →(33,9) →(34,10) → (35,10) →(36,11) →
(37,11) →(38,12) →(39,12) →(40,13) →(41,14) →(42,15) → (43,16) →(44,16) →
(45,17) →(46,18) →(47,19) →(48,20) →(49,21) →(50,22) → (51,23) →(52,24) →
(53,25) →(54,26) →(55,27) →(56,28) →(57,28) →(58,27) → (59,26) →(60,27) →

(60,28) = 51

**Kernel Size 7**

**Euclidean:**

(8,10) →(9,9) →(9,8) →(10,7) →(11,6) →(12,5) →(13,5) →(14,6) → (15,7) →(16,6) →
(17,5) →(18,4) →(19,3) →(20,2) →(21,1) →(22,2) → (23,3) →(24,4) →(25,5) →
(26,6) →(27,7) →(28,8) →(29,7) →(30,8) → (31,9) →(32,10) →(33,11) →(34,12) →
(35,13) →(36,14) →(37,15) →(38,16) → (39,17) →(40,18) →(41,19) →(42,20) = 36

**Manhattan:**

(8,10) →(9,9) →(10,8) →(10,7) →(11,6) →(12,5) →(13,5) →(14,6) → (15,7) →
(16,8) →(17,7) →(18,6) →(19,5) →(20,4) →(21,3) →(22,4) → (23,5) →(24,6) →
(25,7) →(26,8) →(27,7) →(28,8) →(29,9) →(30,10) → (31,11) →(32,12) →(33,13) →
(34,14) →(35,15) →(36,16) →(37,17) →(38,18) → (39,19) →
(40,20) →(41,19) →(42,20) = 36

**Diagonal:**

(8,10) →(9,9) →(10,8) →(10,7) →(11,6) →(12,5) →(13,5) →(14,6) → (15,7) →
(16,8) →(17,7) →(18,7) →(19,6) →(20,5) →(21,4) →(22,5) → (23,5) →(24,6) →
(25,7) →(26,8) →(27,8) →(28,8) →(29,9) →(30,10) → (31,11) →(32,12) →(33,13) →
(34,14) →(35,15) →(36,16) →(37,17) →(38,18) → (39,19) →
(40,20) →(41,19) →(42,20) = 36

**SOURCE 60,70 DESTINATION 60,340**

**Kernel Size 3**

**Euclidean:**

(20,23) →(20,24) →(20,25) →(20,26) →(20,27) →(20,28) →(20,29) →(20,30) →
(20,31) →(20,32) →(20,33) →(20,34) →(20,35) →(20,36) →(21,37) →(22,38) →
(23,39) →(24,40) →(24,41) →(24,42) →(24,43) →(24,44) →(24,45) →(25,46) →
(26,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,52) →(32,53) →(33,54) →
(34,55) →(35,56) →(36,57) →(37,56) →(38,57) →(39,56) →(40,57) →(41,58) →
(42,59) →(43,59) →(44,60) →(44,61) →(45,62) →(45,63) →(46,64) →(46,65) →
(46,66) →(46,67) →(46,68) →(46,69) →(47,70) →(48,71) →(49,72) →(49,73) →
(49,74) →(50,75) →(49,76) →(49,77) →(49,78) →(49,79) →(49,80) →(49,81) →
(49,82) →(49,83) →(48,84) →(47,84) →(46,84) →(45,84) →(44,84) →(43,84) →
(42,83) →(41,84) →(40,85) →(39,86) →(38,87) →(37,88) →(36,89) →(35,89) →
(34,90) →(33,90) →(32,90) →(31,91) →(30,92) →(29,92) →(28,93) →(27,93) →
(26,94) →(25,94) →(24,95) →(24,96) →(24,97) →(24,98) →(23,99) →(23,100) →
(22,101) →(21,102) →(20,103) →(19,104) →(19,105) →(19,106) →(19,107) →
(18,108) →(17,109) →(16,110) →(17,111) →(18,112) →(19,113) →(20,113) = 110

**Manhattan:**

(20,23) →(20,24) →(20,25) →(20,26) →(20,27) →(20,28) →(20,29) →(20,30) →
(20,31) →(20,32) →(20,33) →(20,34) →(21,35) →(22,36) →(23,37) →(23,38) →
(23,39) →(24,40) →(24,41) →(24,42) →(24,43) →(24,44) →(24,45) →(25,46) →
(26,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,52) →(32,53) →(33,54) →
(34,55) →(35,56) →(36,57) →(37,56) →(38,57) →(39,56) →(40,57) →(41,58) →
(42,59) →(43,59) →(44,60) →(44,61) →(45,62) →(45,63) →(46,64) →(46,65) →

(46,66) →(46,67) →(46,68) →(46,69) →(47,70) →(48,71) →(49,72) →(49,73) →
(50,74) →(50,75) →(49,76) →(49,77) →(49,78) →(49,79) →(49,80) →(49,81) →
(49,82) →(49,83) →(48,84) →(47,85) →(46,86) →(45,87) →(44,88) →(44,89) →
(43,90) →(42,91) →(41,92) →(40,92) →(39,92) →(38,92) →(37,91) →(36,90) →
(35,89) →(34,90) →(33,90) →(32,90) →(31,91) →(30,92) →(29,92) →(28,93) →
(27,93) →(26,94) →(25,94) →(24,95) →(24,96) →(24,97) →(23,98) →(22,99) →
(22,100) →(21,101) →(20,102) →(20,103) →(19,104) →(19,105) →(19,106) →
(19,107) →(18,108) →(17,109) →(16,110) →(17,111) →(18,112) →(19,113) →
(20,113) = 111

**Diagonal:**

(20,23) →(20,24) →(20,25) →(20,26) →(20,27) →(20,28) →(20,29) →(20,30) →
(20,31) →(20,32) →(20,33) →(20,34) →(21,35) →(22,36) →(23,37) →(23,38) →
(23,39) →(24,40) →(24,41) →(24,42) →(24,43) →(24,44) →(24,45) →(25,46) →
(26,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,52) →(32,53) →(33,54) →
(34,55) →(35,56) →(36,57) →(37,56) →(38,57) →(39,56) →(40,57) →(41,58) →
(42,59) →(43,59) →(44,60) →(44,61) →(45,62) →(45,63) →(46,64) →(46,65) →
(46,66) →(46,67) →(46,68) →(46,69) →(47,70) →(48,71) →(49,72) →(49,73) →
(50,74) →(50,75) →(49,76) →(49,77) →(49,78) →(49,79) →(49,80) →(49,81) →
(49,82) →(49,83) →(48,84) →(47,85) →(46,86) →(45,87) →(44,88) →(44,89) →
(43,90) →(42,91) →(41,92) →(40,92) →(39,92) →(38,93) →(37,92) →(36,91) →
(36,90) →(35,89) →(34,90) →(33,90) →(32,90) →(31,91) →(30,92) →(29,92) →
(28,93) →(27,93) →(26,94) →(25,94) →(24,95) →(24,96) →(24,97) →(23,98) →
(22,99) →(22,100) →(21,101) →(20,102) →(20,103) →(19,104) →(19,105) →
(19,106) →(19,107) →(18,108) →(17,109) →(16,110) →(17,111) →(18,112) →
(19,113) →(20,113) = 112

**Kernel Size 5**

**Euclidean:**

(12,14) →(13,13) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) → (19,8) →
(20,9) →(21,8) →(22,7) →(23,6) →(24,5) →(25,4) →(26,3) → (27,4) →(28,3) →(29,2) →
(30,3) →(31,2) →(32,3) →(33,4) →(34,5) → (35,6) →(36,7) →(37,8) →(38,9) →
(39,10) →(40,9) →(41,10) →(42,11) → (43,12) →(44,13) →(45,14) →(46,15) →
(47,16) →(48,17) →(49,18) →(50,19) → (51,20) →(52,21) →(53,22) →(54,23) →
(55,24) →(56,25) →(57,26) →(58,27) → (59,26) →(60,25) →(61,24) →(62,23) →
(63,22) →(64,21) →(65,20) →(66,19) → (67,18) →(68,17) →(69,16) →(70,15) →
(71,14) →(72,13) →(73,12) →(73,13) → (73,14) →(73,15) →(73,16) →(73,17) →
(73,18) →(73,19) →(73,20) →(73,21) → (73,22) →(73,23) →(73,24) →(73,25) →
(73,26) →(73,27) →(73,28) →(73,29) → (73,30) →(73,31) →(73,32) →(73,33) →
(73,34) →(73,35) →(73,36) →(73,37) → (73,38) →(73,39) →(73,40) →(73,41) →
(73,42) →(73,43) →(73,44) →(73,45) → (73,46) →(73,47) →(73,48) →(73,49) →
(73,50) →(73,51) →(73,52) →(73,53) → (73,54) →(73,55) →(73,56) →(73,57) →
(73,58) →(73,59) →(73,60) →(73,61) → (73,62) →(73,63) →(73,64) →(73,65) →
(73,66) →(73,67) →(73,68) →(73,69) → (73,70) →(73,71) →(72,72) →(71,72) →
(70,72) →(69,72) →(68,72) →(67,72) → (66,72) →(65,72) →(64,72) →(63,72) →
(62,72) →(61,72) →(60,72) →(59,72) → (58,72) →(57,72) →(56,72) →(55,72) →
(54,72) →(53,72) →(52,72) →(51,72) → (50,72) →(49,72) →(48,72) →(47,72) →
(46,72) →(45,72) →(44,72) →(43,72) → (42,72) →(41,72) →(40,72) →(39,72) →

(38,72) →(37,72) →(36,72) →(35,72) → (34,72) →(33,72) →(32,72) →(31,72) →
(30,72) →(29,72) →(28,72) →(27,72) → (26,72) →(25,72) →(24,72) →(23,72) →
(22,72) →(21,72) →(20,72) →(19,72) → (18,72) →(17,72) →(16,72) →(15,72) →
(14,72) →(13,72) →(12,72) →(11,72) → (10,72) →(11,71) →(12,70) →(13,69) →
(12,68) = 189

**Manhattan:**

(12,14) →(12,13) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) → (19,8) →
(20,9) →(21,10) →(22,11) →(23,10) →(24,11) →(25,10) →(26,9) → (27,8) →(28,7) →
(29,6) →(30,7) →(31,6) →(32,7) →(33,8) →(34,9) → (35,10) →(36,11) →(37,10) →
(38,11) →(39,12) →(40,13) →(41,14) →(42,15) → (43,16) →(44,15) →(45,16) →
(46,17) →(47,18) →(48,19) →(49,20) →(50,21) → (51,22) →(52,23) →(53,24) →
(54,25) →(55,26) →(56,27) →(57,28) →(58,27) → (59,26) →(60,25) →(61,24) →
(62,23) →(63,22) →(64,21) →(65,20) →(66,19) → (67,18) →(68,17) →(69,16) →
(70,15) →(71,14) →(72,13) →(73,12) →(73,13) → (73,14) →(73,15) →(73,16) →
(73,17) →(73,18) →(73,19) →(73,20) →(73,21) → (73,22) →(73,23) →(73,24) →
(73,25) →(73,26) →(73,27) →(73,28) →(73,29) → (73,30) →(73,31) →(73,32) →
(73,33) →(73,34) →(73,35) →(73,36) →(73,37) → (73,38) →(73,39) →(73,40) →
(73,41) →(73,42) →(73,43) →(73,44) →(73,45) → (73,46) →(73,47) →(73,48) →
(73,49) →(73,50) →(73,51) →(73,52) →(73,53) → (73,54) →(73,55) →(73,56) →
(73,57) →(73,58) →(73,59) →(73,60) →(73,61) → (73,62) →(73,63) →(73,64) →
(73,65) →(73,66) →(73,67) →(73,68) →(73,69) → (73,70) →(73,71) →(72,72) →
(71,72) →(70,72) →(69,72) →(68,72) →(67,72) → (66,72) →(65,72) →(64,72) →
(63,72) →(62,72) →(61,72) →(60,72) →(59,72) → (58,72) →(57,72) →(56,72) →
(55,72) →(54,72) →(53,72) →(52,72) →(51,72) → (50,72) →(49,72) →(48,72) →
(47,72) →(46,72) →(45,72) →(44,72) →(43,72) → (42,72) →(41,72) →(40,72) →
(39,72) →(38,72) →(37,72) →(36,72) →(35,72) → (34,72) →(33,72) →(32,72) →
(31,72) →(30,72) →(29,72) →(28,72) →(27,72) → (26,72) →(25,72) →(24,72) →
(23,72) →(22,72) →(21,72) →(20,72) →(19,72) → (18,72) →(17,72) →(16,72) →
(15,72) →(14,72) →(13,72) →(12,72) →(11,72) → (10,72) →(11,71) →(12,70) →
(13,69) →(12,68) = 189

**Diagonal:**

(12,14) →(12,13) →(13,12) →(14,11) →(15,10) →(16,9) →(17,8) →(18,7) → (19,8) →
(20,9) →(21,10) →(22,11) →(23,10) →(24,11) →(25,10) →(26,9) → (27,8) →(28,7) →
(29,6) →(30,7) →(31,6) →(32,7) →(33,8) →(34,9) → (35,10) →(36,11) →(37,10) →
(38,11) →(39,12) →(40,13) →(41,14) →(42,15) → (43,16) →(44,16) →(45,17) →
(46,18) →(47,19) →(48,20) →(49,21) →(50,22) → (50,23) →(51,24) →(52,25) →
(53,26) →(54,27) →(55,28) →(55,29) →(56,30) → (55,31) →(55,32) →(55,33) →
(55,34) →(56,35) →(56,36) →(55,37) →(55,38) → (56,39) →(57,40) →(58,41) →
(58,42) →(59,43) →(60,43) →(61,42) →(62,43) → (63,43) →(64,44) →(65,43) →
(66,44) →(67,44) →(68,44) →(69,43) →(70,42) → (71,41) →(72,40) →(73,41) →
(73,42) →(73,43) →(73,44) →(73,45) →(73,46) → (73,47) →(73,48) →(73,49) →
(73,50) →(73,51) →(73,52) →(73,53) →(73,54) → (73,55) →(73,56) →(73,57) →
(73,58) →(73,59) →(73,60) →(73,61) →(73,62) → (73,63) →(73,64) →(73,65) →
(73,66) →(73,67) →(73,68) →(73,69) →(73,70) → (73,71) →(72,72) →(71,72) →
(70,72) →(69,72) →(68,72) →(67,72) →(66,72) → (65,72) →(64,72) →(63,72) →
(62,72) →(61,72) →(60,72) →(59,72) →(58,72) → (57,72) →(56,72) →(55,72) →
(54,72) →(53,72) →(52,72) →(51,72) →(50,72) → (49,72) →(48,72) →(47,72) →
(46,72) →(45,72) →(44,72) →(43,72) →(42,72) → (41,72) →(40,72) →(39,72) →
(38,72) →(37,72) →(36,72) →(35,72) →(34,72) → (33,72) →(32,72) →(31,72) →

(30,72) →(29,72) →(28,72) →(27,72) →(26,72) → (25,72) →(24,72) →(23,72) →
(22,72) →(21,72) →(20,72) →(19,72) →(18,72) → (17,72) →(16,72) →(15,72) →
(14,72) →(13,72) →(12,72) →(11,72) →(10,72) → (11,71) →(12,70) →(13,69) →
(12,68) = 172

**Kernel Size 7**

**Euclidean:**
(8,10) →(8,11) →(8,12) →(8,13) →(8,14) →(8,15) →(9,16) →(9,17) → (10,18) →
(10,19) →(10,20) →(11,21) →(12,22) →(13,23) →(14,22) →(15,23) → (16,24) →
(17,23) →(18,24) →(19,25) →(20,26) →(20,27) →(20,28) →(20,29) → (21,30) →
(21,31) →(22,32) →(21,33) →(21,34) →(20,35) →(20,36) →(19,36) → (18,35) →
(17,36) →(16,37) →(15,37) →(14,37) →(13,37) →(12,37) →(11,37) → (10,37) →
(9,37) →(8,37) →(7,38) →(6,39) →(6,40) →(6,41) →(7,42) → (7,43) →(7,44) →
(7,45) →(6,46) →(6,47) →(7,48) →(8,48) = 55
**Manhattan:**
(8,10) →(8,11) →(8,12) →(8,13) →(8,14) →(8,15) →(9,16) →(9,17) → (10,18) →
(10,19) →(10,20) →(11,21) →(12,22) →(13,23) →(14,22) →(15,23) → (16,24) →
(17,23) →(18,24) →(19,25) →(20,26) →(20,27) →(20,28) →(20,29) → (21,30) →
(21,31) →(22,32) →(21,33) →(21,34) →(20,35) →(20,36) →(19,36) → (18,35) →
(17,36) →(16,37) →(15,37) →(14,37) →(13,37) →(12,37) →(11,37) → (10,37) →
(9,37) →(8,37) →(7,38) →(6,39) →(6,40) →(6,41) →(7,42) → (7,43) →(7,44) →
(7,45) →(6,46) →(6,47) →(7,48) →(8,48) = 55
**Diagonal:**
(8,10) →(8,11) →(8,12) →(8,13) →(8,14) →(8,15) →(9,16) →(9,17) → (10,18) →
(10,19) →(10,20) →(11,21) →(12,22) →(13,23) →(14,22) →(15,23) → (16,24) →
(17,23) →(18,24) →(19,25) →(20,26) →(20,27) →(20,28) →(20,29) → (21,30) →
(21,31) →(22,32) →(21,33) →(21,34) →(20,35) →(20,36) →(19,36) → (18,35) →
(17,36) →(16,37) →(15,37) →(14,38) →(13,38) →(12,39) →(11,38) → (10,37) →
(9,38) →(8,37) →(7,38) →(6,39) →(6,40) →(6,41) →(7,42) → (7,43) →(7,44) →
(7,45) →(6,46) →(6,47) →(7,48) →(8,48) = 55

**SOURCE 320,40 DESTINATION 60,340**

**Kernel Size 3**

**Euclidean:**
(106,13) →(106,14) →(106,15) →(105,16) →(104,17) →(103,18) →(102,19) →
(101,20) →(100,21) →(99,22) →(98,23) →(97,24) →(96,25) →(95,26) →(94,27) →
(93,28) → (92,29) →(91,30) →(90,31) →(89,32) →(88,33) →(87,33) →(86,33) →
(85,33) → (84,33) →(83,33) →(82,33) →(81,33) →(80,33) →(79,33) →(78,33) →
(77,32) → (76,31) →(75,30) →(74,29) →(73,28) →(72,28) →(71,27) →(70,26) →
(69,25) → (68,24) →(67,23) →(66,22) →(65,22) →(64,22) →(63,21) →(62,20) →
(61,19) → (60,18) →(59,17) →(58,17) →(57,17) →(56,17) →(55,16) →(54,15) →
(53,14) → (52,13) →(51,12) →(50,12) →(49,12) →(48,12) →(47,12) →(46,12) →
(45,12) → (44,12) →(43,12) →(42,12) →(41,12) →(40,12) →(39,12) →(38,12) →
(37,12) → (36,13) →(35,13) →(34,13) →(33,13) →(32,13) →(31,13) →(30,13) →

(29,14) → (28,15) →(28,16) →(27,17) →(26,18) →(26,19) →(25,20) →(24,21) →
(23,22) → (22,23) →(21,24) →(21,25) →(20,26) →(20,27) →(20,28) →(20,29) →
(20,30) → (20,31) →(20,32) →(20,33) →(20,34) →(20,35) →(20,36) →(21,37) →
(22,38) → (23,39) →(24,40) →(24,41) →(24,42) →(24,43) →(24,44) →(24,45) →
(25,46) → (26,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,50) →(32,51) →
(33,52) → (34,53) →(35,52) →(36,53) →(37,53) →(38,54) →(39,55) →(40,56) →
(41,57) → (42,58) →(43,59) →(44,60) →(44,61) →(45,62) →(45,63) →(46,64) →
(46,65) → (46,66) →(46,67) →(46,68) →(46,69) →(47,70) →(48,71) →(49,72) →
(49,73) → (49,74) →(50,75) →(50,76) →(50,77) →(50,78) →(50,79) →(50,80) →
(50,81) → (50,82) →(49,83) →(48,84) →(47,84) →(46,84) →(45,84) →(44,84) →
(43,84) → (42,83) →(41,83) →(40,83) →(39,83) →(38,83) →(37,83) →(36,83) →
(35,84) → (34,85) →(33,86) →(32,87) →(31,88) →(30,89) →(29,90) →(28,91) →
(27,92) → (26,93) →(25,94) →(24,95) →(24,96) →(24,97) →(24,98) →(23,99) →
(23,100) → (22,101) →(21,102) →(20,103) →(19,104) →(19,105) →(19,106) →
(19,107) →(18,108) →(17,109) →(16,110) →(16,111) →(17,112) →(18,113) →
(19,112) →(20,113) = 199

**Manhattan:**

(106,13) →(106,14) →(106,15) →(105,16) →(104,17) →(103,18) →(102,19) →
(101,20) →(100,21) →(99,22) →(98,23) →(97,24) →(96,25) →(95,26) →(94,27) →
(93,28) → (92,29) →(91,30) →(90,31) →(89,32) →(88,33) →(87,33) →(86,33) →
(85,33) → (84,33) →(83,33) →(82,33) →(81,33) →(80,33) →(79,33) →(78,33) →
(77,32) → (76,31) →(75,30) →(74,29) →(73,28) →(72,28) →(71,27) →(70,26) →
(69,25) → (68,24) →(67,23) →(66,22) →(65,22) →(64,22) →(63,21) →(62,20) →
(61,19) → (60,18) →(59,17) →(58,17) →(57,17) →(56,17) →(55,16) →(54,15) →
(53,14) → (52,13) →(51,12) →(50,12) →(49,13) →(48,13) →(47,13) →(46,13) →
(45,13) → (44,13) →(43,13) →(42,13) →(41,13) →(40,13) →(39,13) →(38,13) →
(37,13) → (36,13) →(35,13) →(34,13) →(33,13) →(32,13) →(31,13) →(30,13) →
(29,14) → (28,15) →(27,16) →(26,17) →(25,18) →(24,19) →(23,20) →(22,21) →
(21,22) → (20,23) →(20,24) →(20,25) →(20,26) →(20,27) →(20,28) →(20,29) →
(20,30) → (20,31) →(20,32) →(20,33) →(20,34) →(20,35) →(20,36) →(21,37) →
(22,38) → (23,39) →(24,40) →(24,41) →(24,42) →(24,43) →(24,44) →(24,45) →
(25,46) → (26,47) →(27,48) →(28,49) →(29,50) →(30,51) →(31,52) →(32,53) →
(33,54) → (34,55) →(35,56) →(36,57) →(37,56) →(38,57) →(39,56) →(40,57) →
(41,58) → (42,59) →(43,59) →(44,60) →(44,61) →(45,62) →(45,63) →(46,64) →
(46,65) → (46,66) →(46,67) →(46,68) →(46,69) →(47,70) →(48,71) →(49,72) →
(49,73) → (49,74) →(50,75) →(49,76) →(49,77) →(49,78) →(49,79) →(49,80) →
(49,81) → (49,82) →(49,83) →(48,84) →(47,85) →(46,86) →(45,87) →(44,88) →
(44,89) → (43,90) →(42,91) →(41,92) →(40,92) →(39,92) →(38,92) →(37,91) →
(36,90) → (35,89) →(34,90) →(33,90) →(32,90) →(31,91) →(30,92) →(29,92) →
(28,93) → (27,93) →(26,94) →(25,94) →(24,95) →(24,96) →(24,97) →(23,98) →
(22,99) → (22,100) →(21,101) →(20,102) →(20,103) →(19,104) →(19,105) →
(19,106) →(19,107) →(18,108) →(17,109) →(16,110) →(17,111) →(18,112) →
(19,113) →(20,113) = 199

**Diagonal:**

(106,13) →(106,14) →(106,15) →(105,16) →(104,17) →(103,18) →(102,19) →
(101,20) →(100,21) →(99,22) →(98,23) →(97,24) →(96,25) →(95,26) →(94,27) →
(93,28) → (92,29) →(91,30) →(90,31) →(89,32) →(88,33) →(87,34) →(86,35) →
(85,36) → (84,37) →(83,38) →(82,37) →(82,36) →(81,35) →(80,35) →(79,35) →
(78,34) → (77,34) →(76,33) →(75,32) →(75,31) →(74,30) →(74,29) →(73,28) →

(72,28) → (71,27) →(70,26) →(69,25) →(68,25) →(67,24) →(67,23) →(66,22) →
(65,22) → (64,22) →(63,21) →(62,20) →(61,20) →(60,19) →(60,18) →(59,17) →
(58,18) → (57,18) →(56,17) →(55,17) →(54,16) →(54,15) →(53,14) →(52,13) →
(51,12) → (50,12) →(49,13) →(48,14) →(47,15) →(46,15) →(45,16) →(44,17) →
(43,18) → (42,19) →(41,19) →(40,20) →(39,21) →(38,20) →(37,20) →(36,19) →
(35,19) → (34,20) →(33,20) →(32,20) →(31,19) →(31,18) →(32,17) →(32,16) →
(32,15) → (32,14) →(31,13) →(30,13) →(29,14) →(28,15) →(27,16) →(26,17) →
(25,18) → (24,19) →(23,20) →(22,21) →(21,22) →(20,23) →(20,24) →(20,25) →
(20,26) → (20,27) →(20,28) →(20,29) →(20,30) →(20,31) →(20,32) →(20,33) →
(20,34) → (20,35) →(20,36) →(21,37) →(22,38) →(23,39) →(24,40) →(24,41) →
(24,42) → (24,43) →(24,44) →(24,45) →(25,46) →(26,47) →(27,48) →(28,49) →
(29,50) → (30,51) →(31,52) →(32,53) →(33,54) →(34,55) →(35,56) →(36,57) →
(37,56) → (38,57) →(39,56) →(40,57) →(41,58) →(42,59) →(43,59) →(44,60) →
(44,61) → (45,62) →(45,63) →(46,64) →(46,65) →(46,66) →(46,67) →(46,68) →
(46,69) → (47,70) →(48,71) →(49,72) →(49,73) →(49,74) →(50,75) →(49,76) →
(49,77) → (49,78) →(49,79) →(49,80) →(49,81) →(49,82) →(49,83) →(48,84) →
(47,85) → (46,86) →(45,87) →(44,88) →(44,89) →(43,90) →(42,91) →(41,92) →
(40,92) → (39,92) →(38,93) →(37,92) →(36,91) →(36,90) →(35,89) →(34,90) →
(33,90) → (32,90) →(31,91) →(30,92) →(29,92) →(28,93) →(27,93) →(26,94) →
(25,94) → (24,95) →(24,96) →(24,97) →(23,98) →(22,99) →(22,100) →(21,101) →
(20,102) →(20,103) →(19,104) →(19,105) →(19,106) →(19,107) →(18,108) →
(17,109) →(16,110) →(17,111) →(18,112) →(19,113) →(20,113) = 212

**Kernel Size 5**

**Euclidean:**
(60,14) →(61,13) →(62,12) →(63,11) →(64,10) →(65,9) →(66,8) →(67,7) → (68,6) →
(69,5) →(70,4) →(71,3) →(72,2) →(73,1) →(73,2) →(73,3) → (73,4) →(73,5) →(73,6) →
(73,7) →(73,8) →(73,9) →(73,10) →(73,11) → (73,12) →(73,13) →(73,14) →(73,15) →
(73,16) →(73,17) →(73,18) →(73,19) → (73,20) →(73,21) →(73,22) →(73,23) →
(73,24) →(73,25) →(73,26) →(73,27) → (73,28) →(73,29) →(73,30) →(73,31) →
(73,32) →(73,33) →(73,34) →(73,35) → (73,36) →(73,37) →(73,38) →(73,39) →
(73,40) →(73,41) →(73,42) →(73,43) → (73,44) →(73,45) →(73,46) →(73,47) →
(73,48) →(73,49) →(73,50) →(73,51) → (73,52) →(73,53) →(73,54) →(73,55) →
(73,56) →(73,57) →(73,58) →(73,59) → (73,60) →(73,61) →(73,62) →(73,63) →
(73,64) →(73,65) →(73,66) →(73,67) → (73,68) →(73,69) →(73,70) →(73,71) →
(72,72) →(71,72) →(70,72) →(69,72) → (68,72) →(67,72) →(66,72) →(65,72) →
(64,72) →(63,72) →(62,72) →(61,72) → (60,72) →(59,72) →(58,72) →(57,72) →
(56,72) →(55,72) →(54,72) →(53,72) → (52,72) →(51,72) →(50,72) →(49,72) →
(48,72) →(47,72) →(46,72) →(45,72) → (44,72) →(43,72) →(42,72) →(41,72) →
(40,72) →(39,72) →(38,72) →(37,72) → (36,72) →(35,72) →(34,72) →(33,72) →
(32,72) →(31,72) →(30,72) →(29,72) → (28,72) →(27,72) →(26,72) →(25,72) →
(24,72) →(23,72) →(22,72) →(21,72) → (20,72) →(19,72) →(18,72) →(17,72) →
(16,72) →(15,72) →(14,72) →(13,72) → (12,72) →(11,72) →(10,72) →(11,71) →
(12,70) →(13,69) →(12,68) = 151
**Manhattan:**
(60,14) →(61,13) →(62,12) →(63,11) →(64,10) →(65,9) →(66,8) →(67,7) → (68,6) →
(69,5) →(70,4) →(71,3) →(72,2) →(73,1) →(73,2) →(73,3) → (73,4) →(73,5) →(73,6) →

(73,7) →(73,8) →(73,9) →(73,10) →(73,11) → (73,12) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) → (73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) → (73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) → (73,36) →(73,37) →(73,38) →
(73,39) →(73,40) →(73,41) →(73,42) →(73,43) → (73,44) →(73,45) →(73,46) →
(73,47) →(73,48) →(73,49) →(73,50) →(73,51) → (73,52) →(73,53) →(73,54) →
(73,55) →(73,56) →(73,57) →(73,58) →(73,59) → (73,60) →(73,61) →(73,62) →
(73,63) →(73,64) →(73,65) →(73,66) →(73,67) → (73,68) →(73,69) →(73,70) →
(73,71) →(72,72) →(71,72) →(70,72) →(69,72) → (68,72) →(67,72) →(66,72) →
(65,72) →(64,72) →(63,72) →(62,72) →(61,72) → (60,72) →(59,72) →(58,72) →
(57,72) →(56,72) →(55,72) →(54,72) →(53,72) → (52,72) →(51,72) →(50,72) →
(49,72) →(48,72) →(47,72) →(46,72) →(45,72) → (44,72) →(43,72) →(42,72) →
(41,72) →(40,72) →(39,72) →(38,72) →(37,72) → (36,72) →(35,72) →(34,72) →
(33,72) →(32,72) →(31,72) →(30,72) →(29,72) → (28,72) →(27,72) →(26,72) →
(25,72) →(24,72) →(23,72) →(22,72) →(21,72) → (20,72) →(19,72) →(18,72) →
(17,72) →(16,72) →(15,72) →(14,72) →(13,72) → (12,72) →(11,72) →(10,72) →
(11,71) →(12,70) →(13,69) →(12,68) = 151

**Diagonal:**

(60,14) →(61,13) →(62,12) →(63,11) →(64,10) →(65,9) →(66,8) →(67,7) → (68,6) →
(69,5) →(70,4) →(71,3) →(72,2) →(73,1) →(73,2) →(73,3) → (73,4) →(73,5) →
(73,6) →(73,7) →(73,8) →(73,9) →(73,10) →(73,11) → (73,12) →(73,13) →(73,14) →
(73,15) →(73,16) →(73,17) →(73,18) →(73,19) → (73,20) →(73,21) →(73,22) →
(73,23) →(73,24) →(73,25) →(73,26) →(73,27) → (73,28) →(73,29) →(73,30) →
(73,31) →(73,32) →(73,33) →(73,34) →(73,35) → (73,36) →(73,37) →(73,38) →
(73,39) →(72,40) →(73,41) →(73,42) →(73,43) → (73,44) →(73,45) →(73,46) →
(73,47) →(73,48) →(73,49) →(73,50) →(73,51) → (73,52) →(73,53) →(73,54) →
(73,55) →(73,56) →(73,57) →(73,58) →(73,59) → (73,60) →(73,61) →(73,62) →
(73,63) →(73,64) →(73,65) →(73,66) →(73,67) → (73,68) →(73,69) →(73,70) →
(73,71) →(72,72) →(71,72) →(70,72) →(69,72) → (68,72) →(67,72) →(66,72) →
(65,72) →(64,72) →(63,72) →(62,72) →(61,72) → (60,72) →(59,72) →(58,72) →
(57,72) →(56,72) →(55,72) →(54,72) →(53,72) → (52,72) →(51,72) →(50,72) →
(49,72) →(48,72) →(47,72) →(46,72) →(45,72) → (44,72) →(43,72) →(42,72) →
(41,72) →(40,72) →(39,72) →(38,72) →(37,72) → (36,72) →(35,72) →(34,72) →
(33,72) →(32,72) →(31,72) →(30,72) →(29,72) → (28,72) →(27,72) →(26,72) →
(25,72) →(24,72) →(23,72) →(22,72) →(21,72) → (20,72) →(19,72) →(18,72) →
(17,72) →(16,72) →(15,72) →(14,72) →(13,72) → (12,72) →(11,72) →(10,72) →
(11,71) →(12,70) →(13,69) →(12,68) = 151

### Kernel Size 7

**Euclidean:**

(42,10) →(43,9) →(44,8) →(45,7) →(46,6) →(47,5) →(48,4) →(49,3) → (50,2) →
(51,1) →(52,0) →(52,1) →(52,2) →(52,3) →(52,4) →(52,5) → (52,6) →(52,7) →
(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →(52,13) → (52,14) →(52,15) →
(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →(52,21) → (52,22) →(52,23) →
(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →(51,28) → (50,28) →(49,29) →
(48,29) →(47,29) →(46,29) →(45,29) →(44,29) →(43,29) → (42,29) →(41,28) →
(40,27) →(40,26) →(40,25) →(40,24) →(39,24) →(38,24) → (37,24) →(36,24) →

(35,24) →(34,24) →(33,24) →(32,24) →(31,23) →(30,23) → (31,22) →(30,21) →
(29,21) →(28,21) →(27,21) →(26,22) →(25,22) →(24,22) → (23,23) →(22,24) →
(22,25) →(22,26) →(22,27) →(22,28) →(22,29) →(22,30) → (22,31) →(22,32) →
(21,33) →(21,34) →(20,35) →(20,36) →(19,36) →(18,35) → (17,36) →(16,37) →
(15,37) →(14,37) →(13,37) →(12,37) →(11,37) →(10,37) → (9,37) →(8,37) →
(7,38) →(6,39) →(6,40) →(6,41) →(7,42) →(7,43) → (7,44) →(7,45) →(6,46) →
(6,47) →(7,48) →(8,48) = 110

**Manhattan:**

(42,10) →(43,9) →
(44,8) →(45,7) →(46,6) →(47,5) →(48,4) →(49,3) → (50,2) →(51,1) →
(52,0) →(52,1) →(52,2) →(52,3) →(52,4) →(52,5) → (52,6) →(52,7) →
(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →(52,13) → (52,14) →(52,15) →
(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →(52,21) → (52,22) →(52,23) →
(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →(51,28) → (50,28) →(49,29) →
(48,29) →(47,29) →(46,29) →(45,29) →(44,29) →(43,29) → (42,29) →(41,29) →
(40,28) →(39,27) →(40,26) →(40,25) →(40,24) →(39,24) → (38,24) →(37,24) →
(36,24) →(35,24) →(34,24) →(33,24) →(32,24) →(31,23) → (30,23) →(31,22) →
(30,21) →(29,21) →(28,21) →(27,21) →(26,22) →(25,22) → (24,22) →(23,23) →
(22,24) →(22,25) →(22,26) →(22,27) →(22,28) →(22,29) → (22,30) →(22,31) →
(22,32) →(21,33) →(21,34) →(20,35) →(20,36) →(19,36) → (18,35) →(17,36) →
(16,37) →(15,37) →(14,37) →(13,37) →(12,37) →(11,37) → (10,37) →(9,37) →
(8,37) →(7,38) →(6,39) →(6,40) →(6,41) →(7,42) → (7,43) →(7,44) →(7,45) →
(6,46) →(6,47) →(7,48) →(8,48) = 111

**Diagonal:**

(42,10) →(43,9) →(44,8) →(45,7) →(46,6) →(47,5) →(48,4) →(49,3) → (50,2) →
(51,1) →(52,0) →(52,1) →(52,2) →(52,3) →(52,4) →(52,5) → (52,6) →(52,7) →
(52,8) →(52,9) →(52,10) →(52,11) →(52,12) →(52,13) → (52,14) →(52,15) →
(52,16) →(52,17) →(52,18) →(52,19) →(52,20) →(52,21) → (52,22) →(52,23) →
(52,24) →(52,25) →(52,26) →(52,27) →(52,28) →(52,29) → (51,29) →(50,29) →
(49,29) →(48,29) →(47,29) →(46,29) →(45,29) →(44,29) → (43,29) →(42,29) →
(41,29) →(40,28) →(39,27) →(40,26) →(40,25) →(40,24) → (39,24) →(38,24) →
(37,24) →(36,24) →(35,24) →(34,24) →(33,24) →(32,24) → (31,23) →(30,23) →
(31,22) →(30,21) →(29,21) →(28,21) →(27,21) →(26,22) → (25,22) →(24,22) →
(23,23) →(22,24) →(22,25) →(22,26) →(22,27) →(22,28) → (22,29) →(22,30) →
(22,31) →(22,32) →(21,33) →(21,34) →(20,35) →(20,36) → (19,36) →(18,35) →
(17,36) →(16,37) →(15,37) →(14,38) →(13,38) →(12,39) → (11,38) →(10,37) →
(9,38) →(8,37) →(7,38) →(6,39) →(6,40) →(6,41) → (7,42) →(7,43) →(7,44) →
(7,45) →(6,46) →(6,47) →(7,48) →(8,48) = 112

**SOURCE 320,40 DESTINATION 320,300**

**Kernel Size 3**

**Euclidean:**

(106,13) →(106,14) →(106,15) →(105,16) →(104,17) →(103,18) →(102,19) →
(101,20) →(100,21) →(99,22) →(98,23) →(97,24) →(96,25) →(95,26) →(94,27) →
(93,28) → (92,29) →(91,30) →(90,31) →(89,32) →(88,33) →(88,34) →(88,35) →
(88,36) → (88,37) →(88,38) →(88,39) →(88,40) →(88,41) →(88,42) →(88,43) →

(88,44) → (89,45) →(90,46) →(91,47) →(91,48) →(91,49) →(92,50) →(93,51) →
(93,52) → (93,53) →(93,54) →(93,55) →(93,56) →(93,57) →(93,58) →(94,59) →
(94,60) → (94,61) →(94,62) →(95,63) →(96,64) →(97,65) →(98,66) →(99,67) →
(100,68) → (101,67) →(102,68) →(103,69) →(104,70) →(105,71) →(106,72) →
(107,73) →(107,74) →(107,75) →(107,76) →(107,77) →(107,78) →(107,79) →
(107,80) →(107,81) →(107,82) →(107,83) →(107,84) →(107,85) →(107,86) →
(107,87) →(107,88) →(107,89) →(107,90) →(107,91) →(107,92) →(107,93) →
(107,94) →(107,95) →(107,96) →(107,97) →(107,98) →(107,99) →(106,100) = 90

**Manhattan:**

(106,13) →(106,14) →(106,15) →(105,16) →(104,17) →(103,18) →(102,19) →
(101,20) →(100,21) →(99,22) →(98,23) →(97,24) →(96,25) →(95,26) →(94,27) →
(93,28) → (92,29) →(91,30) →(90,31) →(89,32) →(88,33) →(89,34) →(90,35) →
(91,36) → (91,37) →(92,38) →(93,39) →(93,40) →(93,41) →(93,42) →(93,43) →
(93,44) → (93,45) →(93,46) →(93,47) →(93,48) →(93,49) →(93,50) →(93,51) →
(93,52) → (93,53) →(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →
(95,60) → (95,61) →(96,62) →(96,63) →(97,64) →(98,65) →(98,66) →(99,67) →
(100,68) → (101,69) →(102,70) →(103,69) →(104,70) →(105,71) →(106,72) →
(107,73) →(107,74) →(107,75) →(107,76) →(107,77) →(107,78) →(107,79) →
(107,80) →(107,81) →(107,82) →(107,83) →(107,84) →(107,85) →(107,86) →
(107,87) →(107,88) →(107,89) →(107,90) →(107,91) →(107,92) →(107,93) →
(107,94) →(107,95) →(107,96) →(107,97) →(107,98) →(107,99) →(106,100) = 90

**Diagonal:**

(106,13) →(106,14) →(106,15) →(106,16) →(105,17) →(104,18) →(103,19) →
(102,20) →(101,21) →(100,22) →(100,23) →(100,24) →(99,25) →(98,26) →(97,27) →
(96,28) →(95,29) →(94,30) →(93,31) →(92,32) →(91,33) →(90,34) →(90,35) →
(91,36) → (91,37) →(92,38) →(93,39) →(93,40) →(93,41) →(93,42) →(93,43) →
(93,44) → (93,45) →(93,46) →(93,47) →(93,48) →(93,49) →(93,50) →(93,51) →
(93,52) → (93,53) →(93,54) →(94,55) →(95,56) →(95,57) →(95,58) →(95,59) →
(95,60) → (95,61) →(96,62) →(96,63) →(97,64) →(98,65) →(98,66) →(99,67) →
(100,68) → (101,69) →(102,70) →(103,69) →(104,70) →(105,71) →(106,72) →
(107,73) →(107,74) →(107,75) →(107,76) →(107,77) →(107,78) →(107,79) →
(107,80) →(107,81) →(107,82) →(107,83) →(107,84) →(107,85) →(107,86) →
(107,87) →(107,88) →(107,89) →(107,90) →(107,91) →(107,92) →(107,93) →
(107,94) →(107,95) →(107,96) →(107,97) →(107,98) →(107,99) →(106,100) = 90

**Kernel Size 5**

**Euclidean:**

(64,8) →(65,7) →(66,6) →(67,5) →(68,4) →(69,3) →(70,2) →(71,1) → (72,0) →
(73,0) →(73,1) →(73,2) →(73,3) →(73,4) →(73,5) →(73,6) → (73,7) →(73,8) →
(73,9) →(73,10) →(73,11) →(73,12) →(73,13) →(73,14) → (73,15) →(73,16) →
(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) → (73,23) →(73,24) →
(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) → (73,31) →(73,32) →
(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) → (73,39) →(73,40) →
(73,41) →(73,42) →(73,43) →(73,44) →(73,45) →(73,46) → (73,47) →(73,48) →
(73,49) →(73,50) →(73,51) →(73,52) →(73,53) →(73,54) → (73,55) →(73,56) →
(73,57) →(73,58) →(73,59) →(73,60) →(73,61) →(73,62) → (73,63) →(73,64) →
(73,65) →(73,66) →(73,67) →(73,68) →(73,69) →(73,70) → (73,71) →(72,72) →

(71,72) →(70,72) →(69,72) →(68,72) →(67,72) →(66,72) → (65,72) →(64,72) →
(63,72) →(62,72) →(61,72) →(60,72) →(59,72) →(58,72) → (57,72) →(56,72) →
(55,72) →(54,72) →(53,72) →(54,71) →(55,70) →(56,69) → (57,68) →(58,67) →
(59,66) →(60,65) →(61,64) →(62,63) →(63,62) →(63,61) → (64,60) = 113

**Manhattan:**

(64,8) →(65,7) →(66,6) →(67,5) →(68,4) →(69,3) →(70,2) →(71,1) → (72,0) →
(73,0) →(73,1) →(73,2) →(73,3) →(73,4) →(73,5) →(73,6) → (73,7) →(73,8) →
(73,9) →(73,10) →(73,11) →(73,12) →(73,13) →(73,14) → (73,15) →(73,16) →
(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) → (73,23) →(73,24) →
(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) → (73,31) →(73,32) →
(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) → (73,39) →(73,40) →
(73,41) →(73,42) →(73,43) →(73,44) →(73,45) →(73,46) → (73,47) →(73,48) →
(73,49) →(73,50) →(73,51) →(73,52) →(73,53) →(73,54) → (73,55) →(73,56) →
(73,57) →(73,58) →(73,59) →(73,60) →(73,61) →(73,62) → (73,63) →(73,64) →
(73,65) →(73,66) →(73,67) →(73,68) →(73,69) →(73,70) → (73,71) →(72,72) →
(71,72) →(70,72) →(69,72) →(68,72) →(67,72) →(66,72) → (65,72) →(64,72) →
(63,72) →(62,72) →(61,72) →(60,72) →(59,72) →(58,72) → (57,72) →(56,72) →
(55,72) →(54,72) →(53,72) →(54,71) →(55,70) →(56,69) → (57,68) →(58,67) →
(59,66) →(60,65) →(61,64) →(62,63) →(63,62) →(63,61) → (64,60) = 113

**Diagonal:**

(64,8) →(65,7) →(66,6) →(67,5) →(68,4) →(69,3) →(70,2) →(71,1) → (72,0) →
(73,0) →(73,1) →(73,2) →(73,3) →(73,4) →(73,5) →(73,6) → (73,7) →(73,8) →
(73,9) →(73,10) →(73,11) →(73,12) →(73,13) →(73,14) → (73,15) →(73,16) →
(73,17) →(73,18) →(73,19) →(73,20) →(73,21) →(73,22) → (73,23) →(73,24) →
(73,25) →(73,26) →(73,27) →(73,28) →(73,29) →(73,30) → (73,31) →(73,32) →
(73,33) →(73,34) →(73,35) →(73,36) →(73,37) →(73,38) → (73,39) →(73,40) →
(73,41) →(73,42) →(73,43) →(73,44) →(73,45) →(73,46) → (73,47) →(73,48) →
(73,49) →(73,50) →(73,51) →(73,52) →(73,53) →(73,54) → (73,55) →(73,56) →
(73,57) →(73,58) →(73,59) →(73,60) →(73,61) →(73,62) → (73,63) →(73,64) →
(73,65) →(73,66) →(73,67) →(73,68) →(73,69) →(73,70) → (73,71) →(72,72) →
(71,72) →(70,72) →(69,72) →(68,72) →(67,72) →(66,72) → (65,72) →(64,72) →
(63,72) →(62,72) →(61,72) →(60,72) →(59,72) →(58,72) → (57,72) →(56,72) →
(55,72) →(54,72) →(55,71) →(56,70) →(57,69) →(58,68) → (59,67) →(60,66) →
(60,65) →(61,64) →(62,63) →(63,62) →(63,61) →(64,60) = 113

**Kernel Size 7**

**Euclidean:**

(45,5) →(46,4) →(47,3) →(48,2) →(49,1) →(50,0) →(51,1) →(52,0) → (52,1) →
(52,2) →(52,3) →(52,4) →(52,5) →(52,6) →(52,7) →(52,8) → (52,9) →(52,10) →
(52,11) →(52,12) →(52,13) →(52,14) →(52,15) →(52,16) → (52,17) →(52,18) →
(52,19) →(52,20) →(52,21) →(52,22) →(52,23) →(52,24) → (52,25) →(52,26) →
(52,27) →(52,28) →(52,29) →(52,30) →(52,31) →(52,32) → (52,33) →(52,34) →
(52,35) →(52,36) →(52,37) →(52,38) →(52,39) →(52,40) → (52,41) →(51,41) →
(50,41) →(49,41) →(48,41) →(47,41) →(46,41) →(45,42) = 56

**Manhattan:**

(45,5) →(46,4) →(47,3) →(48,2) →(49,1) →(50,0) →(51,1) →(52,0) → (52,1) →
(52,2) →(52,3) →(52,4) →(52,5) →(52,6) →(52,7) →(52,8) → (52,9) →(52,10) →

(52,11) →(52,12) →(52,13) →(52,14) →(52,15) →(52,16) → (52,17) →(52,18) →
(52,19) →(52,20) →(52,21) →(52,22) →(52,23) →(52,24) → (52,25) →(52,26) →
(52,27) →(52,28) →(52,29) →(52,30) →(52,31) →(52,32) → (52,33) →(52,34) →
(52,35) →(52,36) →(52,37) →(52,38) →(52,39) →(52,40) → (52,41) →(51,41) →
(50,41) →(49,41) →(48,41) →(47,41) →(46,41) →(45,42) = 56

**Diagonal:**

(45,5) →(46,4) →(47,3) →(48,2) →(49,1) →(50,0) →(51,1) →(52,0) → (52,1) →
(52,2) →(52,3) →(52,4) →(52,5) →(52,6) →(52,7) →(52,8) → (52,9) →(52,10) →
(52,11) →(52,12) →(52,13) →(52,14) →(52,15) →(52,16) → (52,17) →(52,18) →
(52,19) →(52,20) →(52,21) →(52,22) →(52,23) →(52,24) → (52,25) →(52,26) →
(52,27) →(52,28) →(52,29) →(52,30) →(52,31) →(52,32) → (52,33) →(52,34) →
(52,35) →(52,36) →(52,37) →(52,38) →(52,39) →(52,40) → (52,41) →(52,42) →
(51,42) →(50,42) →(49,42) →(48,42) →(47,42) →(46,42) → (45,42) = 57