Федеральное государственное бюджетное образовательное учреждение   
высшего профессионального образования

**“Санкт-Петербургский государственный электротехнический университет “ЛЭТИ” им. В.И. Ульянова (Ленина)”**

Кафедра: МО ЭВМ

**Курсовая работа**

По дисциплине

**Системы реального времени**

Выполнили:

Ужегов А.Т., группа 1304

Вахтель А.В., группа 1304

#### Санкт-Петербург

2014

# Содержание

[Содержание 2](#_Toc471416180)

[1 Постановка задания 3](#_Toc471416181)

[2 Исходные данные 3](#_Toc471416182)

[3 Симуляция 3](#_Toc471416183)

[4 Описание алгоритма 4](#_Toc471416184)

[5 Реализация алгоритма на языке С++ 6](#_Toc471416185)

[6 Лабиринт 9](#_Toc471416186)

[7 Выполнение 12](#_Toc471416187)

[1 Заключение 13](#_Toc471416188)

# Постановка задания

Необходимо средствами, предоставляемыми Фреймворком для программирования роботов ROS (Robot Operating System) сделать так чтобы робот нашел выход на неизвестной карте. Карту запоминать нельзя.

# Исходные данные

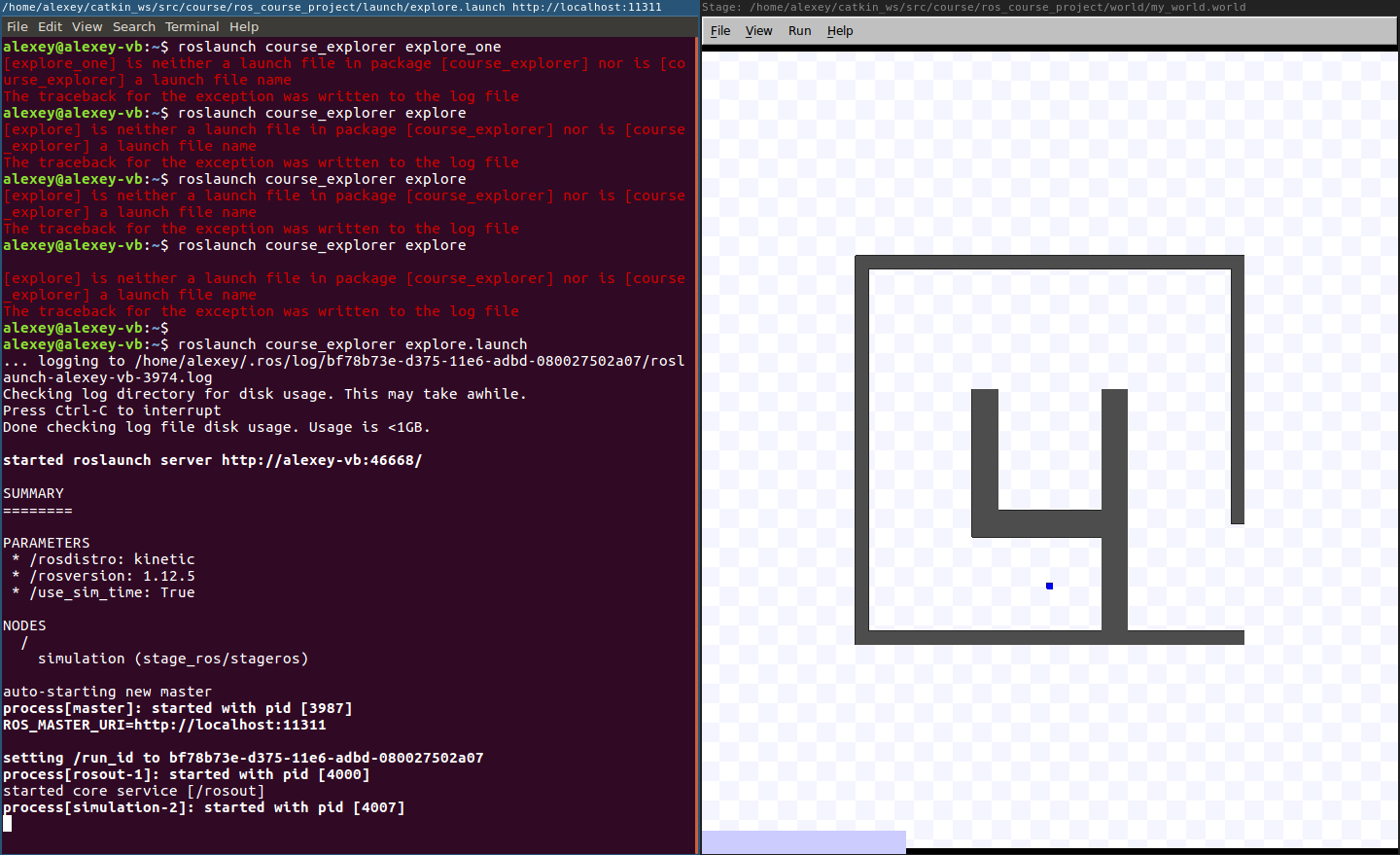
На неизвестной карте необходимо найти выход. Строить и запоминать карту нельзя. За собой остаётся след из хлебных крошек.

Ограничения на исходные данные:

* Есть скан местности, по которому можно определить, в какую сторону можно двигаться.
* Один раз в некоторый интервал времени за вами остаётся хлебная крошка, символизирующая, что в этом участке карты робот уже был.
* Карту запоминать нельзя.
* Выход определяется сразу, как только он оказывается в радиусе видимости скана.

# Симуляция

Симуляция работы алгоритма проводилось в симуляторе Player/Stage. Выбор этого симулятора обуславливается низкими потребностями к производительности и простотой использования.



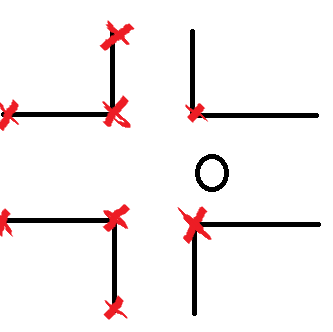
Робот представляет собой квадратик с лазером, который бегает по лабиринту.

# Описание алгоритма

Важной составляющей алгоритма поиска выхода из лабиринта является алгоритм фильтрации, точек которые робот получает в результате работы его датчиков. Основным используемым датчиком в данном случае является лазер, который определяет расстояние до препятствия. Также в алгоритме используются так называемые “Crumb Points” или крошки ориентируясь по которым робот определяет куда он уже ходил и куда ему следует пойти.

Используя данные пришедшие с лазерного датчика находятся точки, которые представляют препятствия, окружающие робота.

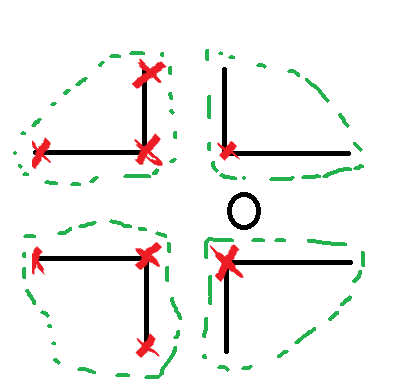
Далее требуется найти угловые точки:



Нахождение угловых точек:

Данные полеченные от лазерного датчика усредняются это делается для уменьшения выборки и погрешности в расположении точек, которая может повлиять на вычисление.

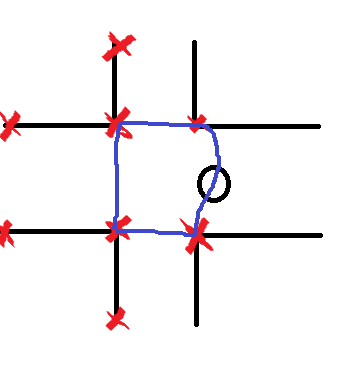
Далее вычисляются сегменты (области с точками, которые являются частью одной и той же стены (на рисунке ниже выделены зеленым).



Сегменты вычисляются путем обхода всех усреднённых точек по часовой стрелке и если расстояние между текущей точкой и следующей превышает некое глобальное расстояния, то это один из концов сегмента.

После этого строятся линии и вычисляется тип построенной линии, есть всего 2 типа вертикальная и горизонтальная в сегментах они обязательно чередуются. Построение линий проходит следующим образом: внутри сегмента обходятся точки рассматривается текущая точка и последующая далее вычисляется угол в прямоугольном треугольнике который образуют эти две точки, если угол больше какого-то определённого угла (в данном случае ПИ делённое на два и умноженное на некоторую константу) в случае если например на данный момент тип линии был горизонтальным, то он меняется на вертикальный. В моменты смены типа линий с вертикального на горизонтальный или наоборот появляются точки угла.

Полученные точки также следует отфильтровать так-как для построения Crumb Points нужны только ближайшие точки углов. Для определения таких точек строится минимальная выпуклая оболочка для точек которые непосредственно окружают робота. Для этого делается обход точек по часовой стрелке, если можно сделать правый поворот к следующей точке, то точка добавляется в результирующий стек в противном случае точка игнорируется. Перед добавлением точки также делается обход по стеку чтобы определить можно ли сделать правый поворот к текущей точке от уже добавленных, чтобы исключить лишние точки из-стека и тем самым уменьшить выпуклую оболочку.



Завершающим этапом построения Crumb Points является вычисление точек, находящихся между точкам входящими в множество минимальной оболочки. Эти точки и есть крошки которые робот ставит в проходах лабиринта.

Суть алгоритма хождения по лабиринту заключается в том, что робот выбирает любую Crumb Point через которую он ещё не проезжал дважды, подъезжая к этой точке он приближается к выходу из сегмента на котором он сейчас находится и в конце концов пересекает его. Таким образом робот переходит из одного сегмента в другой пока не досягнет финальной точки маршрута, то есть нужного сегмента.

# Реализация алгоритма на языке С++

Ниже представлена реализация ключевых частей алгоритма на языке С++

Разделение на сегменты:

|  |
| --- |
| std::vector<BreakPointsExtractor::Mask>  BreakPointsExtractor::divideIntoSegmentsWithFiltering(const std::vector<Point2D>& points) const  {  std::vector<Mask> masks(1);  masks[0].push\_back(0);  size\_t previous = 0;  size\_t current = 0;  while (++current < points.size())  {  float distance = GeomUtils::distance(points[current], points[current - 1]);  if (distance > m\_globalDistance)  {  if (masks.back().back() != current - 1)  {  distance = GeomUtils::distance(points[masks.back().back()], points[current - 1]);  if (distance > m\_localDistance / 5.f)  {  masks.back().push\_back(current - 1);  }  else  {  masks.back().back() = current - 1;  }  }  masks.emplace\_back();  masks.back().push\_back(current);  previous = current;  }  else  {  distance = GeomUtils::distance(points[current], points[previous]);  if (distance > m\_localDistance)  {  masks.back().push\_back(current);  previous = current;  }  }  }  return masks;  } |

Определение типа линии:

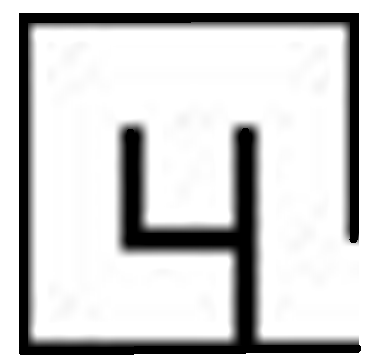
|  |
| --- |
| BreakPointsExtractor::LineType  BreakPointsExtractor::calculateAngles(float dy, float dx, const float MAXIMUM\_K) const  {  float k;  if (dx == 0.f)  {  k = MAXIMUM\_K;  }  else  {  k = dy / dx;  }  float angle = std::abs(std::atan(k));  if (angle > M\_PI\_2 \* m\_threshold)  {  return LineType::VERTICAL;  }  else  {  return LineType::HORIZONTAL;  }  } |

Вычисление минимальной оболочки:

|  |
| --- |
| LaserScanProcessor::calculateCrumbSegments(const std::vector<Point2D>& breakPoints, size\_t closestPointIndex) const  {  Point2D center = {0, 0};  std::vector<size\_t> stack = {closestPointIndex,  (closestPointIndex == breakPoints.size() - 1) ? 0 : closestPointIndex + 1};  for (size\_t i = 2; i < breakPoints.size(); ++i)  {  size\_t index = (closestPointIndex + i) % breakPoints.size();  const Point2D& pointA = breakPoints[stack[stack.size() - 2]];  const Point2D& pointB = breakPoints[stack[stack.size() - 1]];  const Point2D& pointC = breakPoints[i];  if (cross\_product(pointA, pointB, pointC) >= 0.f)  {  stack.emplace\_back(i);  }  else  {  if (cross\_product(pointB, pointC, center) >= 0.f)  {  for (size\_t j = stack.size() - 3; j != std::numeric\_limits<size\_t>::max(); --j)  {  const Point2D& pointNewA = breakPoints[stack[j]];  if (cross\_product(pointNewA, pointB, pointC) >= 0.f &&  cross\_product(pointNewA, pointB, center) >= 0.f)  {  std::swap(stack[stack.size() - 1], stack[j + 1]);  stack.resize(j + 2);  stack.emplace\_back(i);  break;  }  }  }  }  }  return calculateCrumbSegmentsImpl(breakPoints, stack);  } |

# Лабиринт

В работе используется генератор лабиринтов написанный на языке Питон. Пример построенного лабиринта представлен ниже:



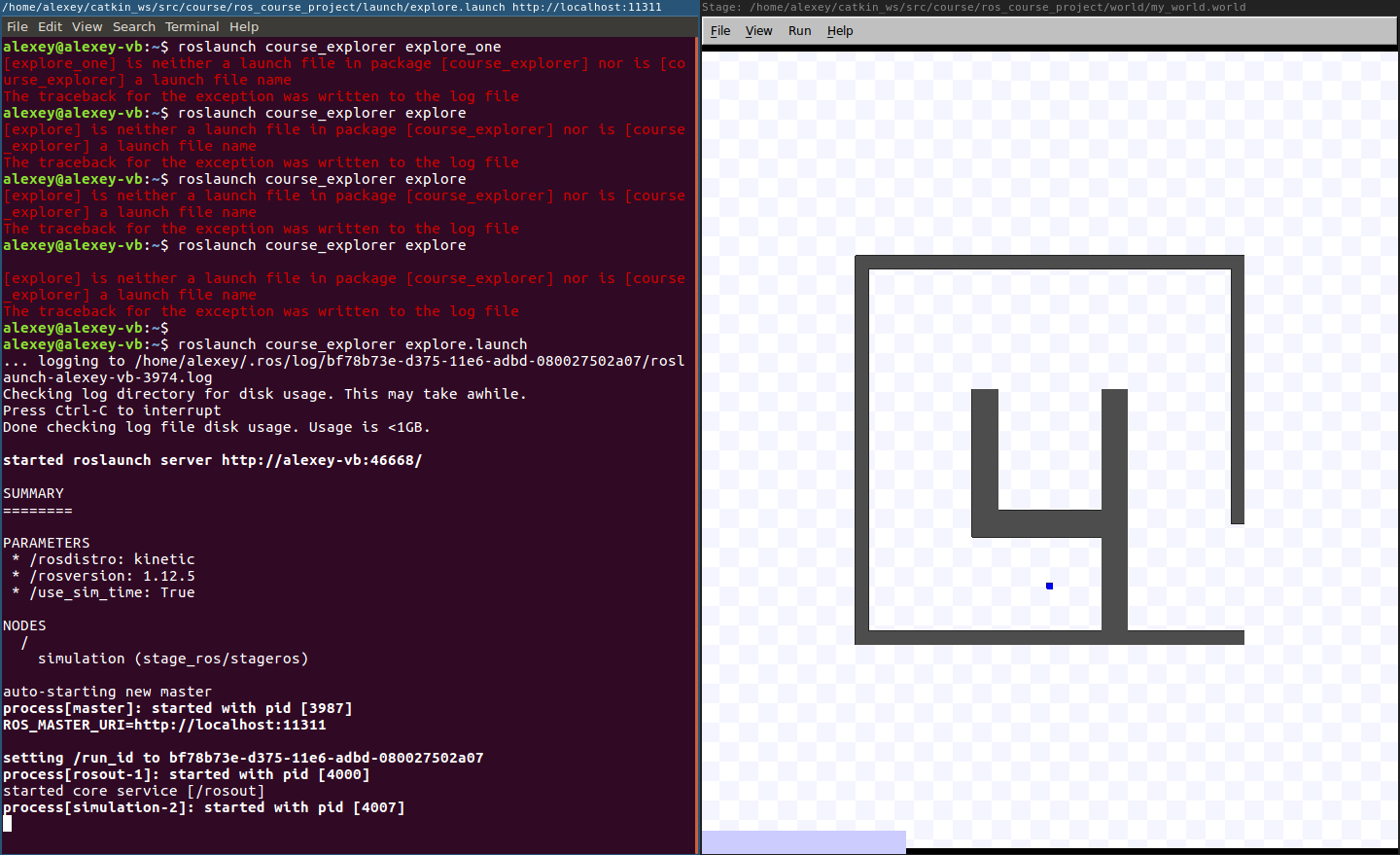
Код генератора:

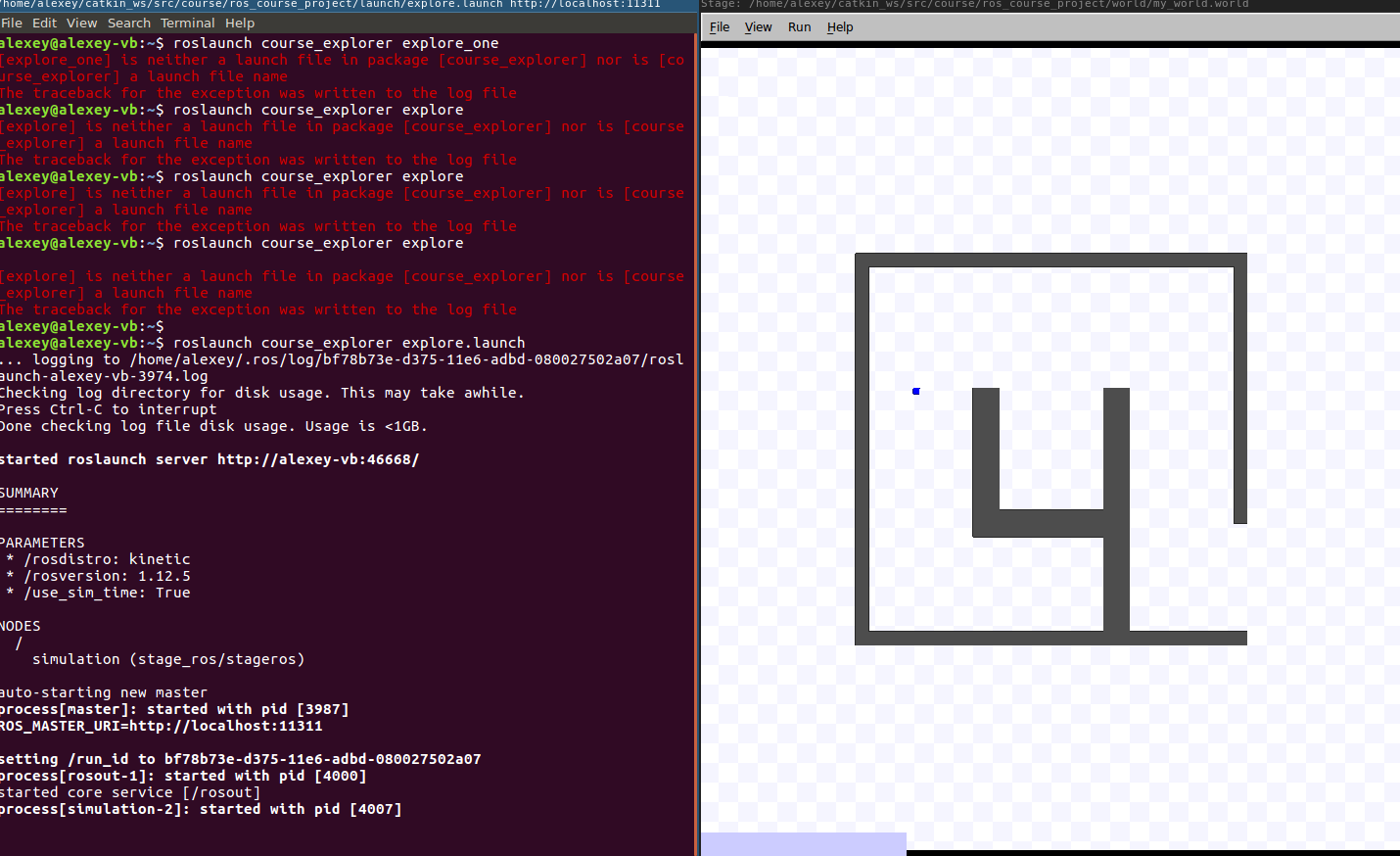
|  |
| --- |
| import random  import sys  import numpy as np  from PIL import Image  from matplotlib import pyplot as plt  def add\_horizontal\_wall(world\_map, row, col\_range, passages=()):  col\_begin, col\_end = col\_range  world\_map[row, range(col\_begin, col\_end), 3] = 1  world\_map[row + 1, range(col\_begin, col\_end), 1] = 1  for passage in passages:  world\_map[row, passage, 3] = 0  world\_map[row + 1, passage, 1] = 0  def add\_vertical\_wall(world\_map, col, row\_range, passages=()):  row\_begin, row\_end = row\_range  world\_map[range(row\_begin, row\_end), col, 2] = 1  world\_map[range(row\_begin, row\_end), col + 1, 0] = 1  for passage in passages:  world\_map[passage, col, 2] = 0  world\_map[passage, col + 1, 0] = 0  def \_recursive\_maze\_generator\_impl(world\_map, row\_range, col\_range):  start\_row, end\_row = row\_range  start\_col, end\_col = col\_range  pivot\_row = random.randrange(start\_row + 1, end\_row)  pivot\_col = random.randrange(start\_col + 1, end\_col)  x\_passages = [random.randrange(start\_col, pivot\_col)]  y\_passages = [random.randrange(start\_row, pivot\_row)]  if random.randint(0, 1):  x\_passages.append(random.randrange(pivot\_col, end\_col))  else:  y\_passages.append(random.randrange(pivot\_row, end\_row))  add\_horizontal\_wall(world\_map, pivot\_row - 1, (start\_col, end\_col), x\_passages)  add\_vertical\_wall(world\_map, pivot\_col - 1, (start\_row, end\_row), y\_passages)  if (pivot\_row - start\_row) > 1 and (pivot\_col - start\_col) > 1:  \_recursive\_maze\_generator\_impl(  world\_map,  (start\_row, pivot\_row),  (start\_col, pivot\_col))  if (end\_row - pivot\_row) > 1 and (pivot\_col - start\_col) > 1:  \_recursive\_maze\_generator\_impl(  world\_map,  (pivot\_row, end\_row),  (start\_col, pivot\_col))  if (pivot\_row - start\_row) > 1 and (end\_col - pivot\_col) > 1:  \_recursive\_maze\_generator\_impl(  world\_map,  (start\_row, pivot\_row),  (pivot\_col, end\_col))  if (end\_row - pivot\_row) > 1 and (end\_col - pivot\_col) > 1:  \_recursive\_maze\_generator\_impl(  world\_map,  (pivot\_row, end\_row),  (pivot\_col, end\_col))  def recursive\_maze\_generator(rows, cols):  world\_map = np.zeros((rows, cols, 4), dtype=np.uint8)  world\_map[range(0, rows), 0, 0] = 1  world\_map[0, range(0, cols), 1] = 1  world\_map[range(0, rows), cols - 1, 2] = 1  world\_map[rows - 1, range(0, cols), 3] = 1  world\_map[rows - 1, cols - 1, 2] = 0  \_recursive\_maze\_generator\_impl(world\_map, (0, rows), (0, cols))  return world\_map  def convert\_map\_to\_image(world\_map, width=10, wall\_width=1):  rows, cols, \_ = world\_map.shape  image = np.zeros((rows \* width, cols \* width), dtype=np.uint8)  image.fill(255)  for row in range(rows):  for col in range(cols):  cell\_data = world\_map[row, col]  if cell\_data[0] == 1:  image[np.ix\_(range(row \* width, (row + 1) \* width),  range(col \* width, col \* width + wall\_width))] = 0  if cell\_data[1] == 1:  image[np.ix\_(range(row \* width, row \* width + wall\_width),  range(col \* width, (col + 1) \* width))] = 0  if cell\_data[2] == 1:  image[np.ix\_(range(row \* width, (row + 1) \* width),  range((col + 1) \* width - wall\_width, (col + 1) \* width))] = 0  if cell\_data[3] == 1:  image[np.ix\_(range((row + 1) \* width - wall\_width, (row + 1) \* width),  range(col \* width, (col + 1) \* width))] = 0  return image  def show\_maze(image):  if sys.platform == 'darwin':  interpolation = 'nearest'  else:  interpolation = 'none'  plt.imshow(image, cmap='Greys', interpolation=interpolation)  plt.show()  def save\_image(image, name):  img = Image.fromarray(image, "L")  img.save("{}.png".format(name))  def main():  num\_rows = int(input("Rows: ")) # number of rows  num\_cols = int(input("Columns: ")) # number of columns  maze = recursive\_maze\_generator(num\_rows, num\_cols)  image = convert\_map\_to\_image(maze)  save\_image(image, "map")  show\_maze(image)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

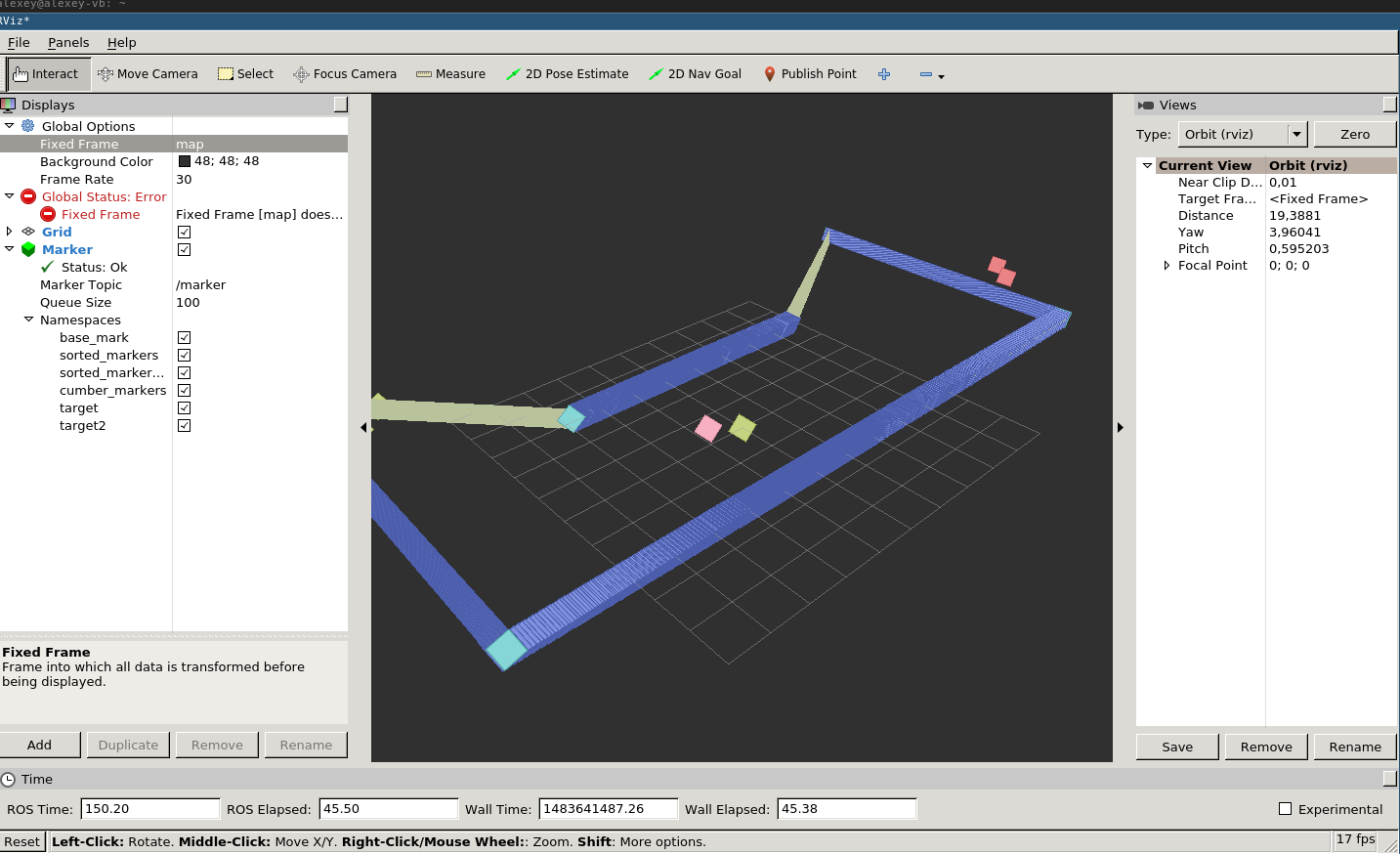
# Выполнение

**Общий сценарий запуска:**

|  |
| --- |
| *source . /devel/setup.sh*  *roslaunch cource\_explorer explore.launch* |

**Пример выполнения:** Реализованная система симуляции была протестирована на тестовом лабиринте. Ниже приведены результаты работы алгоритма в определенные промежутки времени. 





# Заключение

Разработка робота является достаточно сложной задачей, однако благодаря существующим в ROS инструментам можно значительно упростить реализацию алгоритмов передвижения, локализации и построения карты. Симуляция роботов в пакете Gazebo позволяет отладить алгоритмы без необходимости в аппаратной реализации робота. В ходе этой работы используя инструменты предоставляемые ROS был написан алгоритм позволяющие роботу не запоминая карту, ориентируясь по хлебным крошкам найти выход из лабиринта. Для этого были реализованы алгоритм фильтрации точек, алгоритм хождения по лабиринту и алгоритм генерации лабиринтов.