Командный практический тур

В командной части заключительного этапа участники должны спроектировать автономную систему управления для решения классической робототехнической задачи SLAM — одновременной картографии и локализации с использованием алгоритмов компьютерного зрения.

Командная часть заключительного этапа проходит в течении 3 дней (всего 21 астрономический час), которые включают работу по оснащению роботов необходимыми датчиками, программированию, пробные заезды на макете логистического центра, зачетные попытки.

Легенда

В будущем в автоматических логистических центрах почти всю работу будут выполнять роботы, а не люди. Таким образом, будет минимизироваться человеческий фактор, а вслед за этим повышаться скорость и точность работы. Этими роботами будет управлять интеллектуальное программное обеспечение по постановке и распределению задач.

В таких логистических центрах в любой момент может произойти нештатная ситуация и система из роботов под управлением программного обеспечения должна уметь справляться с такими ситуациями. Для того, чтобы такая «красивая картинка» сложилась в реальности, нужны команды опытных разработчиков.

Поэтому задача заключительного этапа профиля «Интеллектуальные робототехнические системы» основывается на робототехнической задаче SLAM — одновременной локализации и картографировании.

Она будет требовать от участников применения таких навыков, как определение объектов окружающей обстановки робота с помощью камеры, построение карты с помощью дальномеров, выбор стратегии по сбору недостающих данных об окружающем пространстве через планирование перемещений робота.

Несмотря на то, что, по легенде, описанные выше задачи ставятся перед складским роботом, подобные алгоритмы используются также беспилотными автомобилями для успешного перемещения в потоке машин и прибытия в пункт назначения, роботами-курьерами, роботизированной сельскохозяйственной техникой, роботами — пылесосами для качественной чистки обслуживаемых помещений.

Задача заключительного этапа будет следующей: В логистическом центре произошла перезагрузка всех систем, что привело к сбросу информации о местоположении работающего в данный момент робота-погрузчика, а также карты данной части логистического центра.

В начале выполнения задания считается, что робототехническое устройство активируется в правом нижнем углу логистического центра. Робот должен локализоваться, найти ближайшую к стене тумбу и определить свое глобальное местоположение. Дальше он должен найти ARTag метку, по ней узнать местоположение сервисного обслуживания и приехать на эти координаты. Структура логистического центра

неизвестна заранее. При перемещении робот не должен повреждать логистический центр.

Задача участников Олимпиады — разработать программу управления робототехническим устройством для выполнения задания, описанного выше.

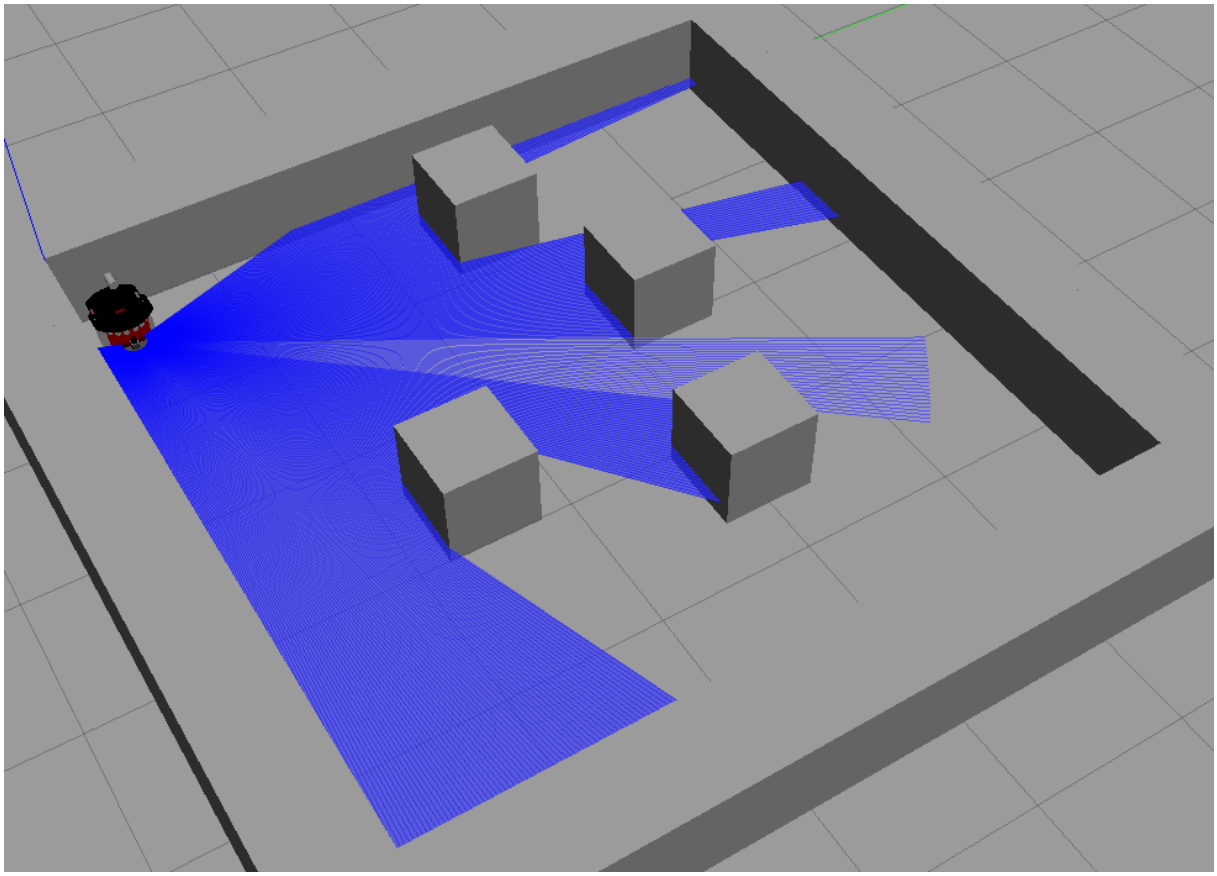

Рис. III.2.1: Полигон для запуска робототехнических устройств на финале Олимпиады НТИ

Набор заданий

Решение командной задачи разбито на три этапа. Первые два этапа итеративно подводят участников к решению полной финальной задачи, осуществляемому во время последнего третьего этапа. На каждом этапе в проверку решения заданий данного этапа входят:

- способность проверить гипотезу о работоспособности алгоритма через демонстрацию решения в симуляторе;
- полнота решения задания конкретного этапа;
- воспроизводимость результатов — робототехническое устройство участников должно неоднократно выполнить требуемые действия.

Первый этап

Задача: Робот начинает свое движение в правом нижнем углу полигона с неизвестной ориентацией. На полигоне имеются четыре тумбы различных цветов. Цвет

тумбы, наименее удаленной от стены полигона, задает направления осей координат и, соответственно, начальные координаты. Тумбы располагаются параллельно стенкам и координаты их центра задаются следующим образом: $(0,5\text{i};\ 0,5\text{j})$, где i,j — целые числа (т. е. тумбы могут быть смещены на пол сектора). В процессе работы роботу необходимо определить свои глобальные координаты. *Включая содержательные задачи:*

- Построение карт местности робототехническими системами.
- Реализация алгоритмов вычисления усредненного цвета графического поля с помощью камеры.
- Реализация алгоритмов определения координат положения.

Второй этап

Задача: Робот с неизвестной ориентацией начинает свое движение рядом с желтой тумбой с случайной ее стороны. Необходимо найти ARTag маркер и расшифровать глобальные координаты, закодированные на нем.

Включая содержательные задачи:

- Реализация поиска ключевых объектов на изображении с камеры.
- Реализация алгоритмов перемещения до ключевого кадра на камере.
- Реализация алгоритмов исключения лишних элементов из кадра.
- Реализация алгоритмов компьютерного зрения: считывание ARTag меток.
- Декодирование бинарного кода с использованием кода Хэмминга.

Третий этап

Задача: Робот начинает свое движение в правом нижнем углу полигона с неизвестной ориентацией. На полигоне имеются четыре тумбы различных цветов. Цвет тумбы, наименее удаленной от стены полигона, задает направления координат и, соответственно, начальные координаты. На желтой тумбе с неизвестной стороны располагается ARTag метка, кодирующая координаты конечного сектора. Тумбы располагаются параллельно стенкам и координаты их центра задаются следующим образом: $(0,5\text{i};\ 0,5\text{j})$, где i,j — целые числа (т. е. тумбы могут быть смещены на пол сектора). Роботу необходимо доехать до конечного сектора.

Включая содержательные задачи:

- Калибровка камеры робототехнического устройства;
- Реализация алгоритмов составления карты неизвестной местности и локализация на данной местности;
- Реализация алгоритмов перемещения на пустой местности без опорных сооружений;
- Реализация алгоритмов компьютерного зрения: считывание ARTag меток без использования дополнительных библиотек;
- Декодирование бинарного кода, с использованием кода Хэмминга;

Описание модели логистического центра

Полигон — квадратное поле 4800×4800 мм., разделенное на квадратные сектора 600×600 мм. Некоторые места полигона недоступны для посещения робототехническим устройством и представляют из себя тумбу. Тумба имеет размер $600 \times 600 \times 400$ мм (Д $\times$ Ш $\times$ В), может находиться либо в центре сектора, либо со смещением в половину сектора. Координаты центра тумбы можно представить следующим образом: $(0,5\text{i};\ 0,5\text{j})$, где i,j — целые числа. На полигоне располагается 4 тумбы.

Полигон окружен бортом высотой 400 мм. Конфигурация полигона изменяется и определяется непосредственно перед каждым запуском робота.

Тумба желтого цвета, расположенная на полигоне, содержит ARTag маркер, расположенный на одной из боковых сторон, на высоте 250-350 мм от основания. Горизонтально маркер расположен по центру тумбы.

Он определяет глобальные координаты X и Y — координаты местоположения сектора сервисного обслуживания, служащего финальной точкой. Размер маркера — 50×50 мм. Конкретная высота расположения маркеров определяется в первый день финала и остается постоянной на все дни финального этапа. При этом допустимая погрешность установки маркеров ± 5 мм относительно их первичной установки. Пример расположения маркера на стеллаже представлен на рисунке III.2.2

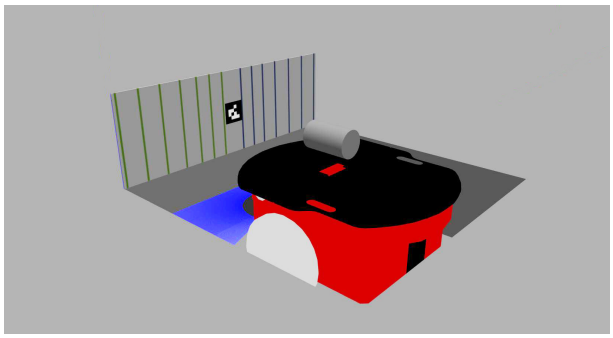

Рис. III.2.2: Тумба с установленным ARTag маркером

ArTag метка состоит из 6×6 квадратов, кодирующих биты. Черный квадрат считается равным 1, белый квадрат считается равным 0. Биты читаются слева направо сверху вниз, при этом самый первый элемент — старший бит закодированного числа. Картинка с подробным описанием приведена ниже.

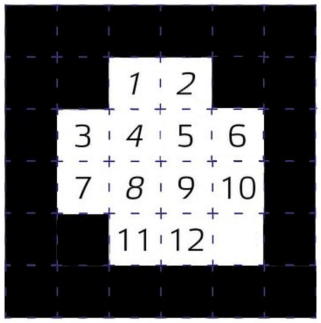

Рис. III.2.3: Нумерация элементов маркера относительно ориентационных элементов

Элементы маркера, расположенные по его границе — всегда черные. Четыре элемента, находящиеся в углах внутреннего 4×4 квадрата определяют ориентацию маркера таким образом, что только один из них — белый. Нумерация элементов относительно ориентационных элементов обозначена на рисунке III.2.3.

Для кодирования информации используется код с детекцией и коррекцией ошибок — код Хэмминга (https://habr.com/ru/post/140611/). Для кода Хэмминга используется 12 бит информации. После декодирования кода Хэмминга первые 3 бита — координата X в little endian порядке битов, следующие 3 бита — координата Y в little endian порядке битов, последние два бита равны 0.

1 Первый контрольный бит;
2 Второй контрольный бит;
3 Младший бит координаты X;
4 Третий контрольный бит;
5 Средний бит координаты X;
6 Старший бит координаты X;
7 Младший бит координаты Y;
8 Четвертый контрольный бит;
9 Средний бит координаты Y;
10 Старший бит координаты Y;
11 0;
12 0;

Данные биты задают координаты X и Y, которые дальше будет использовать робот-погрузчик:

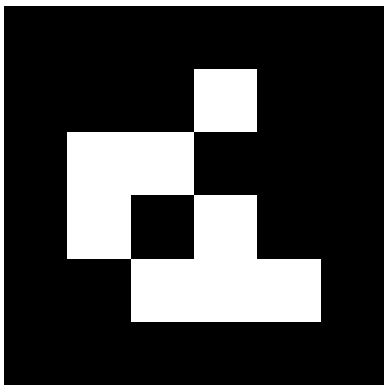

Рис. III.2.4: Маркер с закодированным значением — 100011010100_2

Маркер на рисунке III.2.4 кодирует число 100011010100_2 — $X = 110_2 = 6_{10}$, $Y = 100_2 = 4_{10}$.

Робот начинает свое движение в правом нижнем секторе. Гарантируется, что координаты сервисного обслуживания достижимы. Сектора активации робота (старта) никак не обозначаются на поле и определяются непосредственно перед каждым заездом робота.

В некоторых задачах используются локальные (относительные) координаты. Эти координаты зависят от направления старта робота и определяются следующим образом: сектор $(0, 0)$ находится в точке начального расположения робота, ось Y направлена вдоль начального направления робота, ось X — направлена вправо отно-

сительно направления старта робота.

Также существует 4 вида глобальных координат: желтые, красные, синие, зеленые. Во время попытки активным может быть только один вид, это определяется цветом тумбы, ближайшей к стенке. Гарантируется, что глобальные координаты можно определить однозначно в процессе движения. Ориентация и начало отсчета координат представлены на рис. III.2.5

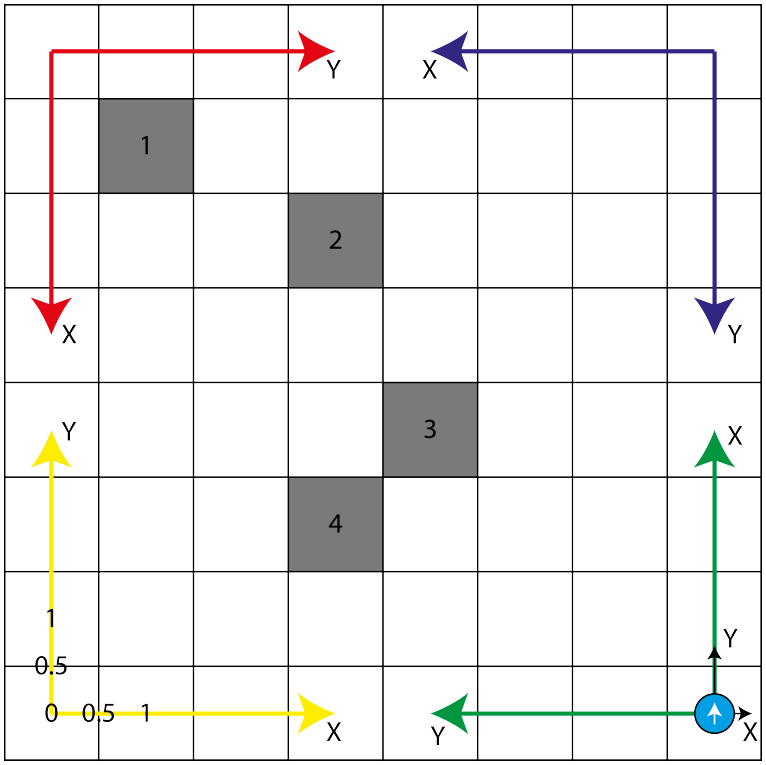

Рис. III.2.5: Расположение и ориентация глобальных координат

Также на рис. III.2.5 представлен пример расположения тумб и робота. Робот — голубой круг, белая стрелка — направление старта робота. Черная ось координат означает относительные координаты. Например, тумба 1 имеет следующие локальные координаты: $(-6; 6)$. Также эта тумба является ближайшей к стенке, следовательно, ее цвет кодирует вид глобальных координат. Рассмотрим следующие варианты, если цвет тумбы 1:

1. Желтый, то глобальные координаты данной тумбы: $(1; 6)$;
2. Красный, то глобальные координаты данной тумбы: $(1; 1)$;
3. Синий, то глобальные координаты данной тумбы: $(6; 1)$;
4. Зеленый, то глобальные координаты данной тумбы: $(6; 6)$;

Описание робота

На финальном туре команды используют робота в симуляторе. Робот, собранный по дифференциальной схеме, оснащен камерой и лазерным дальномером. Дальномер позволяет получить массив данных, который содержит 683 показания, полученные в диапазоне (-120°;120°). Показания распределены равномерно в пределах указанного диапазона. Камера направлена «вперед» по направлению движения робота. Разреше-

ние камеры: 1280×720. Робот, расположенный на полигоне, управляется при помощи Python-скриптов. Для обеспечения работы робототехническое устройство использует библиотеку **«robot»**, имеющую следующие команды:

- `sleep(sec)` — пауза в sec секунд, возможно использование дробных чисел.
- `time()` -возвращает текущее время в симуляции (simulation time в Gazebo) в секундах.
- `getDirection()` — возвращает направление робота в радианах.
- `getEncoders()` — возвращает показания экнодеров в радианах в формате dict.
- `getLaser()` — возвращает показания лазера, включая время кадра, угол лазера, изменение угла между соседними показаниями и сами показания в формате dict.
- `setVelosities(linear, angular)` — подать заданные угловые и линейные скорости на робота.
- `getImage()` — возвращает RGB-изображение с камеры робота в формате OpenCV image.

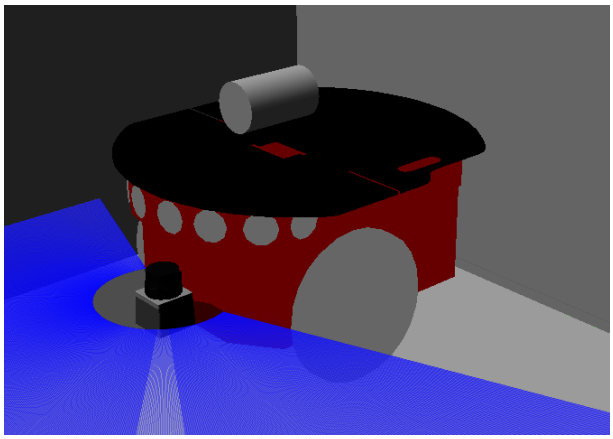

Рис. III.2.6: Мобильная платформа для симулятора

Условия проведения

1. Из имеющегося набора датчиков команды могут выбирать те, с помощью которых, по мнению участников, можно решить задачу наиболее эффективным способом.
2. Участники во время командного этапа финального тура могут использовать интернет и заранее подготовленные библиотеки для решения задачи.
3. Участники могут использовать предустановленные библиотеки numpy и openCV. Использование иных библиотек, помимо стандартных, не допускается. Их использование может повлечь аннулирование баллов за задачу.
4. Использование ros функций и библиотек, включая rospy не допускается. Исключением является библиотека **«robot»** и иные команды созданные организаторами.
5. Для проведения финала подготовлен образ системы (доступный по ссылке: http://bit.ly/gazebo-image-2), с настроенным программным обеспечением. Участники должны использовать данный образ.

6. Допускается использование и настройка иных систем. В этом случае вся ответственность за настройку и работоспособность лежит на участниках. Также необходимо убедиться в работоспособности решения в виртуальной системе, выданной организаторами.
7. Проверка решений будет происходить в виртуальной системе или в системе с подобными зависимостями, например, с такими же версиями используемых библиотек.
8. Участники не могут использовать помощь тренера, сопровождающего лица или привлекать третьих лиц для решения задачи.
9. Финальная задача формулируется участникам в первый день финального тура, но участники выполняют решение задачи поэтапно. Критерии прохождения каждого этапа формулируются для каждого дня финального тура. За подзадачи, решенные в конкретном этапе, начисляются баллы. Баллы за подзадачи можно получить только в день, закрепленный за конкретным этапом.
10. Во время рабочего времени команды могут проводить испытания в симуляторе Gazebo: команда получает не менее 3 тестовых виртуальных полигонов с соответствующими наборами входных данных для подготовки решения, в то время как прием решений происходит на иных полигонах, составленных для проверки универсальности управляющей программы. Для запуска данных тестов рекомендуется использовать следующую команду в терминале: `./sh/run_simulation.sh` N, где N – номер теста.
11. Также в директории `open_worlds` вашего репозитория могут находиться комментарии к открытым тестам, содержащие ответ на задачу или подсказки к ним.
12. Каждый день финального тура в 15:00 по МСК (может варьироваться в зависимости от расписания) команда должна загрузить свое решение на gitlab.com в соответствующий репозиторий дня, внутри своей группы, доступ к которой участники получат в начале олимпиады. Время может изменяться и зависит от количества команд и сложности подзадач, принимаемых в конкретный этап.
13. До истечения указанного времени команды могут изменять файл с решением сколько угодно раз. Проверяться будет всегда только последняя доступная версия.
14. Необходимо зафиксировать последнее решение - создать Merge Request (MR).
15. При создании Merge Request в качестве ответственного (assignee) укажите того, кто будет отвечать за приемку результатов.
16. После момента, когда все отправили свои решения на проверку, судьи приступают к проверке отправленных решений и подзадач, закрепленных за этапом конкретного дня финального тура.
17. Может быть предусмотрено до двух попыток сдачи решения одной и той же подзадачи в симуляторе. Конкретное количество попыток определяется в конкретных подзадачах.
18. После прохождения приемочных запусков, баллы, набранные командой, заносятся судьями в протокол.
19. Робот должен выполнять задание полностью автономно. Удаленное управление не допускается.
20. Не допускается вывод в консоль ничего, кроме информации, требуемой в задаче. В случае вывода иной информации она может быть расценена как часть

ответа.

21. Если во время приемочных запусков у судьи возникает ситуация, когда он не может однозначно решить выполняются ли критерии решения подзадачи, он вправе принять решение не в пользу команды.
22. В случае, если возникает техническая проблема, независящая от участников, то по решению судей может быть предоставлена возможность перезапуска.
23. Результат тестирования участникам будет сообщен на следующий день после сдачи решения.
24. Команды могут до 10:00 МСК обратиться с апелляцией. Жюри вправе провести тестирование или видео воспроизведение попытки (если таковая имеется) и назначить баллы за соответствующие подзадачи.

Процедура проведения приемочных запусков и критерии оценки

Первый этап

1. Командам необходимо подготовить следующую задачу для симулятора:
   1.1. Робот начинает свое движение в правом нижнем углу полигона с неизвестной ориентацией. На полигоне имеются четыре тумбы различных цветов. Цвет тумбы, наименее удаленной от стены полигона, задает направления осей координат и, соответственно, начальные координаты. Тумбы располагаются параллельно стенкам и координаты их центра задаются следующим образом: $(0,5\text{i};\ 0,5\text{j})$, где i,j — целые числа (т. е. тумбы могут быть смещены на пол сектора). В процессе работы роботу необходимо определить свои глобальные координаты.
2. Основной файл с управляющей программой для проверки решения должен называться `main.py` и находиться в корне репозитория.
3. Merge Request необходимо назвать следующим образом: “код команды_day1”. Например: “irs202100_day1”.
4. Максимальное время выполнения одной попытки для задачи — 5 минут.
5. Решение будет проверяться на 2 проверочных полигонах.
6. Баллы за решение подзадач этапа:
   6.1. Проекция всего робота покинула сектор старта — 4 балла.
   6.2. Робот смог определить расстояние между стенкой и ближайшей тумбой и вывел его в консоль в формате: `distance` X — 11 баллов.
   6.3. Робот определил цвет ближайшей к стенке тумбы и вывел его в консоль в формате: `color` X, где X — цвет (red, green, blue, yellow) — 13 баллов.
   6.4. После определения цвета или расстояния робот остановился и вывел в консоль относительные коодинаты в формате: `temp coordinates` X Y и не менее чем через 10 сек продолжил движение — 15 баллов.
   6.5. Робот смог определить свои точные координаты и вывел их в консоль в формате: `coordinates` X Y, где X и Y — координаты в секторах. Робот продолжил движение не менее чем через 10 сек (если необходимо). — 22 балла.

7. Значение координат необходимо вычислять для центра робота.
8. Значение координат необходимо вывести с точностью до целых.
9. Баллы за все попытки в каждой подзадаче суммируются.
10. Выполнение всех критериев в каждой из двух попыток всех подзадач дает дополнительные 10 баллов.
11. Максимальное количество баллов за этап — 140.

Второй этап

1. Командам необходимо обновить библиотеку для корректной работы `robot.sleep` функции в симуляторе. Это позволит командам работать с временем из симулятора, а не реальным. Проверка решений будет происходить на обновленном рабочем окружении. Для обновления следует воспользоваться следующей командой: `./sh/update.sh`
2. Командам необходимо подготовить следующую задачу для симулятора:
   2.1. Робот с неизвестной ориентацией начинает свое движение рядом с желтой тумбой с случайной ее стороны. Необходимо найти ArTag маркер и расшифровать глобальные координаты, закодированные на нем.
3. Основной файл с управляющей программой для проверки решения должен называться `main.py` и находиться в корне репозитория.
4. Merge Request необходимо назвать следующим образом: “код команды_day2”. Например: “irs202100_day2”.
5. Максимальное время выполнения одной попытки для задачи — 3 минуты.
6. Решение будет проверяться на 2 проверочных полигонах.
7. Баллы за решение подзадач этапа:
   7.1. Робот смог найти желтую тумбу, остановился, повернулся в направлении данной тумбы, вывел в консоль `found` и через 10 сек при необходимости продолжил движение — 4 балла.
   7.2. Робот смог найти сторону с маркером, остановился и вывел в консоль: `found marker` и через 10 сек при необходимости продолжил движение — 11 баллов.
   7.3. Робот смог распознать сырое значение маркера (все биты в двоичном виде) и вывел в консоль данные значения в формате: `binary` X — 16 баллов.
   7.4. Робот смог распознать значения координат и вывел в консоль координаты в формате: `coordinates` X Y — 20 баллов.
8. Значение координат необходимо вывести с точностью до целых.
9. Баллы за все попытки в каждой подзадаче суммируются.
10. Выполнение всех критериев в каждой из двух попыток всех подзадач дает дополнительные 8 баллов.
11. Максимальное количество баллов за этап — 110.

Третий этап

1. Командам необходимо подготовить следующую задачу для симулятора:

1.1. Робот начинает свое движение в правом нижнем углу полигона с неизвестной ориентацией. На полигоне имеются четыре тумбы различных цветов. Цвет тумбы, наименее удаленной от стены полигона, задает направления координат и соответственно начальные координаты. На желтой тумбе с неизвестной стороны располагается arTag метка, кодирующая координаты финального сектора. Тумбы располагаются параллельно стенкам и координаты их центра задаются следующим образом: $(0,5\text{i};\ 0,5\text{j})$, где i,j — целые числа (т. е. тумбы могут быть смещены на пол сектора). Роботу необходимо доехать до точки финиша.

2. Основной файл с управляющей программой для проверки решения должен называться `main.py` и находиться в корне репозитория.
3. Merge Request необходимо назвать следующим образом: “код команды_day3”. Например: “irs202100_day3”.
4. Максимальное время выполнения одной попытки для задачи — 5 минут.
5. Решение будет проверяться на 2 проверочных полигонах.
6. Баллы за решение подзадач этапа:
   6.1. Робот смог определить расстояние между стенкой и ближайшей тумбой в метрах с точностью до десятых и вывел его в консоль в формате: `distance` X — 5 баллов.
   6.2. Робот нашел ближайшую к стенке тумбу, остановился и вывел в консоль ее цвет в формате: `color` X, где X — цвет (red, green, blue, yellow) — 10 баллов.
   6.3. Робот нашел желтую тумбу, смог определить сторону с маркером. Остановился в направлении данного маркера и вывел в консоль: `found marker` и через 10 сек продолжил движение — 5 баллов.
   6.4. Робот нашел желтую тумбу, распознал маркер и вывел значения маркера в консоль в формате: `marker` X Y — 15 баллов.
   6.5. Робот определил свои глобальные координаты, находясь возле желтой тумбы, остановился, вывел координаты в консоль в формате: `coordinates` X Y и через 10 сек продолжил движение — 18 баллов.
   6.6. Проекция робота частично оказалась в секторе финиша, робот остановился и вывел в консоль `finish` — 12 баллов.
   6.7. Проекция робота полностью оказалась в секторе финиша, робот остановился и вывел в консоль `finish` — 5 баллов.
7. Значение координат необходимо вычислять для центра робота.
8. Значение координат необходимо вывести с точностью до целых.
9. Баллы за все попытки в каждой подзадаче суммируются.
10. Выполнение всех критериев в каждой из двух попыток всех подзадач дает дополнительные 10 баллов.
11. Максимальное количество баллов за этап — 150.

Решение

hamming.py

```
from functools import reduce
import operator as op
import numpy as np


class Hamming:
    def __init__(self, block_size: int, include_0_bit=False) -> None:
        self.block_size = block_size
        self.include_0_bit = include_0_bit

        if not self.include_0_bit:
            # the code will still calculate the 0 bit but won't show it to the user
            self.block_size += 1

        self.message_len = self.block_size - \
            int(np.ceil(np.log2(self.block_size))) - 1

    def _is_control_bit(self, num: int) -> bool:
        """
        :returns: whether the `num` is a power of `2` or `== 0`
        """
        assert num >= 0

        return (num & (num - 1) == 0)

    def _hamming_xor(self, block: np.array) -> int:
        """
        :returns: calculated checksum bits if the checksum is empty or wrong bit. 0 ==
        ↪  no wrong bit is identified
        """
        assert len(block) == self.block_size

        return reduce(op.xor, (i for i, bit in enumerate(block) if bit), 0)

    def _extract_message(self, block: np.array) -> np.array:
        assert len(block) == self.block_size

        return np.array([bit for i, bit in enumerate(block) if not
        ↪  self._is_control_bit(i)], dtype=np.byte)

    def decode_block(self, encoded: np.array) -> np.array:
        """
        :encoded: the array of *bits*
        """
        if not self.include_0_bit:
            encoded = np.hstack([0, encoded])

        assert len(encoded) == self.block_size

        mistake_num = self._hamming_xor(encoded)
        if mistake_num:
            encoded[mistake_num] = not encoded[mistake_num]

        if self.include_0_bit:
            oddness = reduce(op.xor, encoded[1:], 0)
```

```
            if (oddness != encoded[0]):
                raise ValueError("Unable to decode")

        return self._extract_message(encoded)

    def encode_block(self, message: np.array) -> np.array:
        assert len(message) <= self.message_len

        ret = np.zeros(self.block_size, dtype=np.byte)
        check_bits = []
        i = 0
        for bit in message:
            while self._is_control_bit(i):
                check_bits.append(i)
                ret[i] = 0
                i += 1
            ret[i] = bit
            i += 1

        checksum = self._hamming_xor(ret)
        for bit_n in check_bits:
            ret[bit_n] = (checksum & bit_n) != 0

        ret[0] = reduce(op.xor, ret[1:], 0)

        if not self.include_0_bit:
            ret = ret[1:]

        return ret
```

grand_finale_full_solution.py

```
import sys
from robot_library.robot import *
import cv2
import numpy as np
import math
import time as tm
import hamming

# Shortcut methods
pi = math.pi
robot = Robot()
laser = robot.getLaser()

# This list will be used after detection of all boxes to store their positions and
↪  colors
# Format: ((x,y),color)
box_list_colored = []

# This list will store a 2D map of the field with resiolution equal to one semisector
# 0 - empty space
# 2 - walls
# 3 - box centers
# Every element of this map represents a cross of grid lines, NOT A SECTOR!
# (= 0.3 in Gazebo > GUI > Grid)!
map = []

# Fill the map with zeros
```

```
for i in range(0, 19):
    map.append([])
    for j in range(0, 19):
        map[i].append(0)

# Build walls around the area
for i in range(0, 19):
    map[0][i] = 2
    map[17][i] = 2
    map[i][0] = 2
    map[i][17] = 2
    map[1][i] = 2
    map[18][i] = 2
    map[i][1] = 2
    map[i][18] = 2

# Robot's start position in map coordinates
# measured in semisectors
# To use in local coordinates(as in the NTI task day 2), subtract from 16
locX = 16
locY = 16
# Rotation
# 0-UP, 1-RIGHT 2-DOWN 3-LEFT
rotation = 3
# Global coordinates calculated after all detections
X = -1
Y = -1
# Global finish coordinates after ARTAG reading
XArTag = -1
YArTag = -1


# Update coordinates after forward movement
def updateCoordinates(dist):
    global locX
    global locY
    global rotation
    if rotation == 3:
        locX -= dist
    elif rotation == 1:
        locX += dist
    elif rotation == 0:
        locY -= dist
    elif rotation == 2:
        locY += dist


# Read the color of the box
def colorDecoder():
    # retreive the frame from robot's camera
    frame = robot.getImage()
    # Translate the frame into HSV colorspace
    hsv = cv2.cvtColor(frame, cv2.COLOR_RGB2HSV)
    # Get the pixel in the middle of the frame
    pix = hsv[360][640]
    # Compare the HUE with all the four colors
    redp = min((abs(pix[0] - 0)), (abs(pix[0] - 179)))
    greenp = (abs(pix[0] - 60))
    bluep = (abs(pix[0] - 120))
    yellowp = (abs(pix[0] - 30))
```

```
    # Return the right color
    if (redp < greenp and redp < bluep and redp < yellowp):
        return "red"
    elif (greenp < redp and greenp < bluep and greenp < yellowp):
        return "green"
    elif (bluep < redp and bluep < greenp and bluep < yellowp):
        return "blue"
    elif (yellowp < redp and yellowp < bluep and yellowp < greenp):
        return "yellow"


# function for detecting the ARTag on the picture
def cameraDecode():
    global XArTag
    global YArTag
    robot.sleep(0.1)
    # retreive the frame from robot's camera
    frame = robot.getImage()
    # Translate the frame into HSV colorspace
    hsv = cv2.cvtColor(frame, cv2.COLOR_RGB2HSV)

    # define range of color in HSV
    lower = np.array([0, 127, 0])
    upper = np.array([255, 255, 255])

    # ARTag is the darkest entity on the map. To find it, it is enough to threshold
    ↪  the HSV image

    # Threshold the HSV image to get only specific colors
    mask = cv2.inRange(hsv, lower, upper)
    hsv[mask > 0] = (0, 0, 255)

    lower = np.array([0, 0, 10])
    upper = np.array([255, 255, 255])

    # Threshold the HSV image to get only specific colors
    mask = cv2.inRange(hsv, lower, upper)
    mask = mask[:600][:]

    # Here we find the corners of ARTag
    top = 10000
    bottom = 0
    left = 10000
    right = 0
    for i in range(len(mask)):
        for j in range(len(mask[1])):
            if (mask[i][j] < 100):
                if j < left:
                    left = j
                if j > right:
                    right = j
                if i < top:
                    top = i
                if i > bottom:
                    bottom = i

    # If we have found anything
    if (top != 10000 or bottom != 0):

        # Extract Region of Interest from original
```

```
        artagart = mask[top:bottom, left:right]

        w = abs(right - left)
        h = abs(bottom - top)
        semistepw = w / 12
        semisteph = h / 12
        artag = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

        # From the extracted ARTag image, get the martix 4x4
        for i in range(0, 4):
            for j in range(0, 4):
                if artagart[math.floor(i * semisteph * 2 + 3 * semisteph)][
                    math.floor(j * semistepw * 2 + 3 * semistepw)] > 0:
                    artag[i][j] = 1
        # Rotate the matrix until we get a right orientation
        rots = 0
        while artag[3][3] != 1:
            rots += 1
            artag = list(zip(*artag[::-1]))
            if (rots > 4):
                # Error in finding ARTag
                break

        #Empty line of bits
        artagLine = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
        #Fill the list of bits
        index = 0
        for z in range(0, 4):
            for z2 in range(0, 4):
                if not ((z == 0 or z == 3) and (z2 == 0 or z2 == 3)):
                    artagLine[index] = artag[z][z2]
                    index += 1

        # applying hamming code
        h = hamming.Hamming(12)
        detectedData = h.decode_block(np.array(artagLine))

        XArTag = detectedData[0] * 1 + detectedData[1] * 2 + detectedData[2] * 4
        YArTag = detectedData[3] * 1 + detectedData[4] * 2 + detectedData[5] * 4
        # Return 2 - ARTag was found!
        return 2
    else:
        # Return 1 - ARTag was NOT found =(
        return 1

# This method translates cartesian coordinates to polar
def cart2pol(x, y):
    rho = np.sqrt(x ** 2 + y ** 2)
    phi = np.arctan2(y, x)
    return (rho, phi)


# This method translates polar coordinates to cartesian
def pol2cart(rho, phi):
    x = rho * np.cos(phi)
    y = rho * np.sin(phi)
    return (x, y)

# This method translates HSV to RGB (Thanks StackOverflow!!!)
def hsv_to_rgb(h, s, v):
```

```
    if s == 0.0: return (v, v, v)
    i = int(h * 6.)
    f = (h * 6.) - i;
    p, q, t = v * (1. - s), v * (1. - s * f), v * (1. - s * (1. - f));
    i %= 6
    if i == 0: return (v, t, p)
    if i == 1: return (q, v, p)
    if i == 2: return (p, v, t)
    if i == 3: return (p, q, v)
    if i == 4: return (t, p, v)
    if i == 5: return (v, p, q)

# This method receives an index of laser sensor cropped [40:-40] and returns the angle
# of this sensor relative to robot's rotation angle
def get_deg_by_index(index):
    return math.radians((index + 40) * 240 / 684 - 240 / 2)

# This method finds distance from line to point.
# PT1, PT2 -- line points
# PT3 -- point to calculate the distance to
def distance_line_to_point(pt1, pt2, pt3):
    p1 = np.array(pt1)
    p2 = np.array(pt2)
    p3 = np.array(pt3)
    return np.linalg.norm(np.cross(p2 - p1, p1 - p3)) / np.linalg.norm(p2 - p1)

# Standard method of finding distance between two points
# pt1=(x,y) pt2=(x,y)
def distance(pt1, pt2):
    d1 = pt1[0] - pt2[0]
    d2 = pt1[1] - pt2[1]
    return math.sqrt(d1 * d1 + d2 * d2)

# A wrapper method for the previous one, used to calculate the line length
def line_length(l):
    return distance(l[0], l[1])

# This method uses laser sensors to find boxes on the field
# Returns a list of center points of boxes in cm
def raw_laser_scan():
    robot_direction = robot.getDirection()

    # Debug Image
    img = np.zeros((1000, 1000, 3), np.uint8)

    laser = robot.getLaser()
    lv = laser.get('values')[40:len(laser.get('values')) - 40]
    # First step: find lines among all the received points
    lines_det = []
    cur_line_x = []
    cur_line_y = []
    # This 'for' cycle works like that:
    # We iterate through all the points, transform their coordinates to (x,y)
    # While doing this we populate the list of points which will represent a line
    ↪  later
    # If this list has more than 2 points, start checking when the next point in this
    ↪  list
    # will not fit on a line we built using least squares method using all previous
    ↪  points in this list
    # When that happens, we put the first and the last point on this list in a tuple
    ↪  and save this
```

```
    # tuple as a line
    for i in range(len(lv)):
        # Get the distance and the absolute angle of a point
        c_dir = get_deg_by_index(i) + robot_direction
        c_val = lv[i]
        # Translate to cartesian coordinates
        x, y = pol2cart(c_val, c_dir - math.pi / 2)
        # To prevent shift of the picture while robot rotates,
        # Subtract the laser sensor location on the robot relative to robot's wheels,
        ↪  18cm
        # Later, in our experiments using a very small grid we found that actual
        ↪  sensor position
        # is 15.5 cm, but... Don't touch if works
        xc, yc = pol2cart(0.18, robot_direction - math.pi / 2 + math.pi)
        x -= xc
        y -= yc
        # Meters to centimeters
        x = x * 100
        y = -y * 100
        # if we see anything
        if (c_val != float('inf')):
            #draw a debug circle on a debug image
            cv2.circle(img, (int(x) + 500
                             , int(y) + 500), 1, (255, 255, 255), thickness=2,
                             ↪  lineType=8, shift=0)

            if len(cur_line_x) < 3:
                cur_line_x.append(x)
                cur_line_y.append(y)
            else:
                if len(cur_line_x) >= 3:
                    # least squares method
                    m, b = np.polyfit(cur_line_x, cur_line_y, 1)
                    p1 = (0, b)
                    p2 = (10, 10 * m + b)
                    # if the new point is on the same line, append it to this line
                    # else, save the current line's first and last point and start a
                    ↪  new line
                    if distance_line_to_point(p1, p2, (x, y)) < 1:
                        cur_line_x.append(x)
                        cur_line_y.append(y)
                    else:
                        save = True
                        for i in range(1, len(cur_line_x)):
                            if distance((cur_line_x[i], cur_line_y[i]), (cur_line_x[i
                            ↪  - 1], cur_line_y[i - 1])) > 30:
                                save = False
                        if save:
                            fin_line = ((cur_line_x[0], cur_line_y[0]),
                            ↪  (cur_line_x[-1], cur_line_y[-1]))
                            lines_det.append(fin_line)
                        cur_line_x = []
                        cur_line_y = []
    if len(cur_line_x) > 0:
        fin_line = ((cur_line_x[0], cur_line_y[0]), (cur_line_x[-1], cur_line_y[-1]))
        lines_det.append(fin_line)

    # Find boxes among these lines
    boxes = []
    # Here, we iterete throug all the lines we found
```

```
    for line in lines_det:
        # Draw a line on the debug image
        cv2.line(img, (int(line[0][0]) + 500, int(line[0][1]) + 500),
                 (int(line[1][0]) + 500, int(line[1][1]) + 500), (0, 0, 255),
                 thickness=2)
        # If it seems like this line has a length of N cubes,
        # we split the line on N equal segments
        subcubes = line_length(line) / 60
        if abs((subcubes + 0.5) % 1 - 0.5) < 0.2:
            subcubes = round(subcubes)
            for sub in range(1, subcubes + 1):
                # There we split the line on segments
                d1 = line[1][0] - line[0][0]
                d2 = line[1][1] - line[0][1]
                p1 = (line[0][0] + d1 * (sub - 1) / subcubes, line[0][1] + d2 * (sub -
                ↪  1) / subcubes)
                p2 = (line[0][0] + d1 * (sub) / subcubes, line[0][1] + d2 * (sub) /
                ↪  subcubes)
                # draw some debug shapes
                cv2.circle(img, (int(p1[0]) + 500, int(p1[1]) + 500), 10, (255, 255,
                ↪  255), thickness=2, lineType=8,
                           shift=0)
                cv2.circle(img, (int(p2[0]) + 500, int(p2[1]) + 500), 10, (255, 255,
                ↪  255), thickness=2, lineType=8,
                           shift=0)
                lt = (p1, p2)
                cv2.line(img, (int(lt[0][0]) + 500, int(lt[0][1]) + 500),
                         (int(lt[1][0]) + 500, int(lt[1][1]) + 500), (0, 255, 0),
                         thickness=2)
                ltd1 = p2[0] - p1[0]
                ltd2 = p2[1] - p1[1]
                # Here we check is the line vertical or horizontal
                if abs(lt[0][0] - lt[1][0]) > abs(lt[0][1] - lt[1][1]):
                    # Horizontal line
                    # If the line is above us, the center of the box is above the line
                    if (lt[0][1] < -15):
                        boxes.append((p1[0] + ltd1 / 2, p1[1] + ltd2 / 2 - 30))
                    else:
                        boxes.append((p1[0] + ltd1 / 2, p1[1] + ltd2 / 2 + 30))
                else:
                    # Vertical line
                    # If the line is on the right side of us, the center of the box is
                    ↪  on the right too
                    if (lt[0][0] < -15):
                        boxes.append((p1[0] + ltd1 / 2 - 30, p1[1] + ltd2 / 2))
                    else:
                        boxes.append((p1[0] + ltd1 / 2 + 30, p1[1] + ltd2 / 2))
                if len(boxes) > 1 and distance(boxes[-1], boxes[-2]) < 20:
                    del boxes[-1]
    # Draw all the boxes on debug image
    for box in boxes:
        cv2.rectangle(img, (int(box[0]) - 30 + 500, int(box[1]) - 30 + 500),
                      (int(box[0]) + 30 + 500, int(box[1]) + 30 + 500), (255, 0, 0),
                      ↪  3)
    # Show the debug image
    # uncomment to enable
    #for i in range(10):
    #    cv2.imshow('image', img)
    #    cv2.waitKey(1)
    return boxes
```

```
# Get the distance using 22 lasers in front of our robot
# The result is the nearest measure
# Also, it adds 0.155. This is a position of the laser sensor relative to robot wheels
def get_forward_distance_min():
    laser = robot.getLaser()
    lv = laser.get('values')[40:len(laser.get('values')) - 40]
    min_dist = 10000
    for i in range(len(lv) // 2 - 11, len(lv) // 2 + 11):
        if lv[i] < min_dist:
            min_dist = lv[i]
    return min_dist + 0.155

# This class's name is kinda understandable
class Movement:
    # Dist is the amount of semisectors we gonna move forward
    def forward(self, dist):
        target_dir = 0
        if rotation == 0:
            target_dir = np.pi
        if rotation == 1:
            target_dir = np.pi / 2
        if rotation == 2:
            target_dir = 0
        if rotation == 3:
            target_dir = -np.pi / 2
        current_dir = robot.getDirection()
        current_dist = get_forward_distance_min()
        dist_in_semisectors = current_dist / 0.3

        # Used to correct robot's position knowing that the distance is always a
        ↪  multiple of 0.3
        dist_correction = 0.3 * round(dist_in_semisectors) - current_dist

        target_dist = current_dist - 0.3 * dist + dist_correction
        speed_limit = 0
        dist_diff = 10000
        while abs(dist_diff) > 0.001:
            current_dir = robot.getDirection()

            # This thing descended to us from above
            # nobody knows how this works, but it does!!!
            # It calculates the difference in rotation ignoring the sensor value jumps
            e = (target_dir - current_dir + np.pi * 5) % (np.pi * 2) - (np.pi)

            # Slow acceleration to prevent drifting
            speed_limit += 0.05

            # never go faster than 1
            speed_limit = min(speed_limit, 1)
            # This is the distance left
            dist_diff = get_forward_distance_min() - target_dist
            # This is the linear speed calculated using speed limit and proportional
            ↪  control
            lin_speed = max(-speed_limit, min(speed_limit, dist_diff * 2.5))
            # This is a proportional controller used to correct our direction during
            ↪  the movement
            rot_speed = -e * 2

            robot.setVelosities(lin_speed, rot_speed)
```

```
            robot.sleep(0.01)
        robot.setVelosities(0, 0)
        updateCoordinates(dist)

    # Dir is the direction of movement
    def turn(self, dir):
        global rotation
        if dir == 1:
            rotation += 1
            rotation %= 4
        elif dir == -1:
            rotation += 3
            rotation %= 4
        target_dir = 0
        if rotation == 0:
            target_dir = np.pi
        if rotation == 1:
            target_dir = np.pi / 2
        if rotation == 2:
            target_dir = 0
        if rotation == 3:
            target_dir = -np.pi / 2

        # Same way as in forward function, but we use the speed limit with our
        ↪  direction control
        current_dir = robot.getDirection()
        speed_limit = 0
        while (abs(target_dir - current_dir) > 0.0001):
            current_dir = robot.getDirection()
            e = (target_dir - current_dir + np.pi * 5) % (np.pi * 2) - (np.pi)
            speed_limit += 0.1
            speed_limit = min(speed_limit, 1.5)
            speed = max(-speed_limit, min(speed_limit, -e * 2.5))
            robot.setVelosities(0, speed)
            robot.sleep(0.01)
        robot.setVelosities(0, 0)

# Initalize our class
mov = Movement()

# This function puts boxes list retrieved from raw_laser_scan() to our 2D map
def raw_to_map(raw_boxes_list):
    # We iterate through boxes
    for it in raw_boxes_list:
        # If this is not a part of a wall
        if int(round(it[0] / 30) + locX) >= 2 and int(round(it[1] / 30) + locY) >= 2
        ↪  and int(
                round(it[0] / 30) + locX) <= 16 and int(round(it[1] / 30) + locY) <=
                ↪  16:
            # Update the map!
            map[int(round(it[0] / 30) + locX)][int(round(it[1] / 30) + locY)] = 3
            map[int(round(it[0] / 30) + locX) - 1][int(round(it[1] / 30) + locY) - 1]
            ↪  = 2
            map[int(round(it[0] / 30) + locX) - 1][int(round(it[1] / 30) + locY)] = 2
            map[int(round(it[0] / 30) + locX) - 1][int(round(it[1] / 30) + locY) + 1]
            ↪  = 2
            map[int(round(it[0] / 30) + locX) + 1][int(round(it[1] / 30) + locY) - 1]
            ↪  = 2
            map[int(round(it[0] / 30) + locX) + 1][int(round(it[1] / 30) + locY)] = 2
            map[int(round(it[0] / 30) + locX) + 1][int(round(it[1] / 30) + locY) + 1]
            ↪  = 2
```

```
            map[int(round(it[0] / 30) + locX)][int(round(it[1] / 30) + locY) - 1] = 2
            map[int(round(it[0] / 30) + locX)][int(round(it[1] / 30) + locY) + 1] = 2

# Counts the amount of boxes on the map
def get_box_amount():
    box_count = 0
    for line in map:
        for val in line:
            if val == 3:
                box_count += 1
    return box_count

# Stores the location and direction of an entity
class Position:
    x = 0
    y = 0
    rotation = 0

    def __init__(self, x, y, rotation):
        self.x = x
        self.y = y
        self.rotation = rotation

# finding the shortest path - Dijkstra algorithm
def dijkstra(finX, finY):
    global locX
    global locY
    global rotation
    infinity = 100000
    s = list()
    #initializing parents of a previous positions
    # and minLength to the positions on map
    pos = Position(locX, locY, rotation)
    parentPos = Position(-1, -1, -1)
    s.append(pos)
    minLenghts = []
    for i in range(0, 19):
        minLenghts.append([])
        for j in range(0, 19):
            minLenghts[i].append(infinity)
    parents = []
    for i in range(0, 19):
        parents.append([])
        for j in range(0, 19):
            parents[i].append(parentPos)
    minLenghts[locX][locY] = 0
    #finding a shortest path
    pathWord = ""
    if int(finX) > 0 and int(finX) < 19 and int(finY) > 0 and int(finY) < 19:
        if map[int(finX)][int(finY)] == 0:
            while len(s):
                #popping current position
                newPos = s.pop()

                #variable add_more_weight_to_turns used to add more weight to turns
                #to create a better path
                add_more_weight_to_turns = 0

                #adding position in left side
                if newPos.rotation == 1:
```

```
            add_more_weight_to_turns = 2
        elif newPos.rotation == 3:
            add_more_weight_to_turns = 0
        else:
            add_more_weight_to_turns = 1
        if map[newPos.x - 1][newPos.y] == 0 and minLenghts[newPos.x -
        ↪  1][newPos.y] > minLenghts[newPos.x][
            newPos.y] + add_more_weight_to_turns:
            minLenghts[newPos.x - 1][newPos.y] =
            ↪  minLenghts[newPos.x][newPos.y] + add_more_weight_to_turns
            s.append(Position(newPos.x - 1, newPos.y, 3))
            parents[newPos.x - 1][newPos.y] = newPos
        if newPos.rotation == 1:
            add_more_weight_to_turns = 0
        elif newPos.rotation == 3:
            add_more_weight_to_turns = 2
        else:
            add_more_weight_to_turns = 1
        # adding position in right side
        if map[newPos.x + 1][newPos.y] == 0 and minLenghts[newPos.x +
        ↪  1][newPos.y] > minLenghts[newPos.x][
            newPos.y] + add_more_weight_to_turns:
            minLenghts[newPos.x + 1][newPos.y] =
            ↪  minLenghts[newPos.x][newPos.y] + add_more_weight_to_turns
            s.append(Position(newPos.x + 1, newPos.y, 1))
            parents[newPos.x + 1][newPos.y] = newPos
        if newPos.rotation == 2:
            add_more_weight_to_turns = 0
        elif newPos.rotation == 0:
            add_more_weight_to_turns = 2
        else:
            add_more_weight_to_turns = 1
        #adding position in "down" side
        if map[newPos.x][newPos.y + 1] == 0 and minLenghts[newPos.x][newPos.y
        ↪  + 1] > minLenghts[newPos.x][
            newPos.y] + add_more_weight_to_turns:
            minLenghts[newPos.x][newPos.y + 1] =
            ↪  minLenghts[newPos.x][newPos.y] + add_more_weight_to_turns
            s.append(Position(newPos.x, newPos.y + 1, 2))
            parents[newPos.x][newPos.y + 1] = newPos
        if newPos.rotation == 0:
            add_more_weight_to_turns = 0
        elif newPos.rotation == 2:
            add_more_weight_to_turns = 2
        else:
            add_more_weight_to_turns = 1
        # adding position in "up" side
        if map[newPos.x][newPos.y - 1] == 0 and minLenghts[newPos.x][newPos.y
        ↪  - 1] > minLenghts[newPos.x][
            newPos.y] + add_more_weight_to_turns:
            minLenghts[newPos.x][newPos.y - 1] =
            ↪  minLenghts[newPos.x][newPos.y] + add_more_weight_to_turns
            s.append(Position(newPos.x, newPos.y - 1, 0))
            parents[newPos.x][newPos.y - 1] = newPos

    #if we found a path - distance should not be infinity
    if minLenghts[finX][finY] != infinity:
        # take all parents to make a path
        path = list()
        tempPos = Position(finX, finY, 0)
```

```
                while tempPos.x != -1 and tempPos.y != -1:
                    path.append(tempPos)
                    tempPos = parents[tempPos.x][tempPos.y]
                prevPoint = path.pop()
                temp_rot = rotation
                prev_temp_rot = rotation
                #creating a path by parents
                while len(path):
                    newPoint = path.pop()
                    if newPoint.x > prevPoint.x:
                        temp_rot = 1
                    if newPoint.x < prevPoint.x:
                        temp_rot = 3
                    if newPoint.y > prevPoint.y:
                        temp_rot = 2
                    if newPoint.y < prevPoint.y:
                        temp_rot = 0
                    if temp_rot - prev_temp_rot == 1 or temp_rot - prev_temp_rot ==
                    ↪  -3:
                        pathWord += "R"
                    elif temp_rot - prev_temp_rot == -1 or temp_rot - prev_temp_rot ==
                    ↪  3:
                        pathWord += "L"
                    elif abs(temp_rot - prev_temp_rot) == 2:
                        pathWord += "RR"
                    pathWord += "F"
                    prevPoint = newPoint
                    prev_temp_rot = temp_rot
    return pathWord

# Find the box which is nearest to the wall and return it's coordinates using our 2D
↪  map
def get_side_box():
    res_box = None
    minDist = 10000
    # Iterate through map
    for i in range(0, 19):
        for j in range(0, 19):
            if map[i][j] == 3:
                if i - 2 >= 0 and i - 2 < minDist:
                    minDist = i - 2
                    res_box = (i, j)
                elif j - 2 >= 0 and j - 2 < minDist:
                    minDist = j - 2
                    res_box = (i, j)
                elif 18 - i >= 0 and 18 - i < minDist:
                    minDist = 18 - i
                    res_box = (i, j)
                elif 18 - j >= 0 and 18 - j < minDist:
                    minDist = 18 - j
                    res_box = (i, j)

    return res_box

# Move the robot to specified coordinates finding the path using Dijkstra's algorithm
# 'rebuild' value shows if we need to stop, scan the area and recalculate the path
↪  every turn
# used, for example, when we still didn't find all the boxes
# 'start' value shows if we need to stop moving when we detect all the 4 boxes on the
↪  map
```

```
def move_by_path(coordinates_temp, rebuild, start=False):
    x = coordinates_temp[0]
    y = coordinates_temp[1]
    path = dijkstra(x, y)
    #if we found a path we go through it
    while len(path):
        #if we in a start and found 4 boxes then return
        if start and get_box_amount() >= 4:
            return
        #if we not rebuild a map then we can move forward to the different count of
        ↪  semisectors
        if not rebuild:
            forward_count = 0
            while len(path) and (path[0] == 'F'):
                forward_count += 1
                path = path[1:]

            if forward_count:
                mov.forward(forward_count)
        else:
            #else we can only go by one semisector
            if len(path):
                if path[0] == 'F':
                    mov.forward(1)
        #if we cannot go forward then it can be maybe turns to the left or right
        if len(path):
            if path[0] == 'L':
                mov.turn(-1)
            elif path[0] == 'R':
                mov.turn(1)
        path = path[1:]
        # if we rebuild then we should update our map boxes if we found new boxes
        if rebuild:
            raw_to_map(raw_laser_scan())
            print_map()
            path = dijkstra(x, y)

# This function is used to print the map
def print_map():
    for i in range(0, 19):
        std = ""
        for j in range(0, 19):
            std += (str(map[j][i]) + " ")
        print(std)

# This function sets the robot direction to a specific value
# Used only when the robot is near the box
def rotate_to(rot):
    if rot - rotation == -1 or rot - rotation == 3:
        mov.turn(-1)
    elif rot - rotation == 1 or rot - rotation == -3:
        mov.turn(1)
    elif rot - rotation == -2 or rot - rotation == 2:
        mov.turn(-1)
        mov.turn(-1)

# This method moves the robot to all the boxes un the list, detects and writes
# their color, and returns the position of the yellow box or a message that ARTag was
↪  found
def findYellow():
```

```
    for i, box in enumerate(box_list_colored):
        if box[1] is None:
            rot_coord = next_to(box[0])
            move_by_path(rot_coord[0], False)
            rotate_to(rot_coord[1])
            box_color = colorDecoder()
            artag = cameraDecode()
            if artag == 1:
                box_list_colored[i] = (box_list_colored[i][0], box_color)
                if box_color == "yellow":
                    return box[0]
            else:
                return "found"

# This function returns the nearest position next to the box side
def next_to(coordinates_temp):
    # Find the distance to the left, right, top and bottom sides of a box
    l = len(dijkstra(coordinates_temp[0] - 2, coordinates_temp[1]))
    r = len(dijkstra(coordinates_temp[0] + 2, coordinates_temp[1]))
    t = len(dijkstra(coordinates_temp[0], coordinates_temp[1] - 2))
    b = len(dijkstra(coordinates_temp[0], coordinates_temp[1] + 2))
    # If we cannot go to this side, reset the distance
    if l == 0:
        l = 10000
    if r == 0:
        r = 10000
    if t == 0:
        t = 10000
    if b == 0:
        b = 10000
    # Return the best variant
    min_len = min(l, r, t, b)
    if min_len == l:
        return ((coordinates_temp[0] - 2, coordinates_temp[1]), 1)
    if min_len == r:
        return ((coordinates_temp[0] + 2, coordinates_temp[1]), 3)
    if min_len == t:
        return ((coordinates_temp[0], coordinates_temp[1] - 2), 2)
    if min_len == b:
        return ((coordinates_temp[0], coordinates_temp[1] + 2), 0)

# This function moves along the sides of a box to find an ARTag on it
def find_ARTag(coordinates_temp):
    # Here we find the nearest available sides of this box
    l = len(dijkstra(coordinates_temp[0] - 2, coordinates_temp[1]))
    r = len(dijkstra(coordinates_temp[0] + 2, coordinates_temp[1]))
    t = len(dijkstra(coordinates_temp[0], coordinates_temp[1] - 2))
    b = len(dijkstra(coordinates_temp[0], coordinates_temp[1] + 2))
    if l == 0:
        l = 10000
    if r == 0:
        r = 10000
    if t == 0:
        t = 10000
    if b == 0:
        b = 10000

    # Here we go to these sides chosing the optimal path
    min_len = min(l, r, t, b)
    while min_len < 10000:
```

```
        min_len = min(l, r, t, b)
        if min_len == l:
            l = 10000
            move_by_path((coordinates_temp[0] - 2, coordinates_temp[1]), False)
            rotate_to(1)
        elif min_len == t:
            t = 10000
            move_by_path((coordinates_temp[0], coordinates_temp[1] - 2), False)
            rotate_to(2)
        elif min_len == r:
            r = 10000
            move_by_path((coordinates_temp[0] + 2, coordinates_temp[1]), False)
            rotate_to(3)
        elif min_len == b:
            b = 10000
            move_by_path((coordinates_temp[0], coordinates_temp[1] + 2), False)
            rotate_to(0)
        artag = cameraDecode()
        # If we have found the ARTag, return
        if artag == 2:
            return


# main function
if __name__ == "__main__":
    laser = robot.getLaser()
    laser = laser.get('values')[40:len(laser.get('values')) - 40]

    rotation = 3

    # checking in which orientation we are
    # to turn our robot to the right rotation
    sumLeft = 0
    sumRight = 0
    for z in range(100, 110):
        sumRight += laser[z]
    for z in range(492, 502):
        sumLeft += laser[z]
    if (sumLeft < 3.5 and laser[math.ceil(len(laser) / 2)] < 0.5):
        rotation = 2
        mov.turn(1)
    elif (sumLeft > 3.5 and laser[math.ceil(len(laser) / 2)] > 0.5):
        rotation = 0
        mov.turn(-1)
    elif (sumLeft > 3.5 and laser[math.ceil(len(laser) / 2)] < 0.5):
        rotation = 1
        mov.turn(1)
        mov.turn(1)

    #initial scan of the map
    raw_to_map(raw_laser_scan())

    #go to the coordinates 2,2 to explore all map
    while get_box_amount() < 4:
        move_by_path((2, 2), True, True)

    #inserting boxes to the box_list_colored to go through them in the future
    for i in range(0, 19):
        for j in range(0, 19):
            if map[i][j] == 3:
```

```
            box_list_colored.append(((i, j), None))

raw_to_map(raw_laser_scan())

# if we founf 4 boxes
if get_box_amount() >= 4:
    # go to the closest to the border box
    box_coord = get_side_box()
    next_to_point = next_to(box_coord)
    move_by_path(next_to_point[0], False)
    rotate_to(next_to_point[1])
    box_color = colorDecoder()
    artag = cameraDecode()
    # and update box color
    for i, box in enumerate(box_list_colored):
        if box[0][0] == box_coord[0] and box[0][1] == box_coord[1]:
            box_list_colored[i] = (box[0], box_color)

    # calculating global coordinates
    # and starting Yellow box finding or ARTag finding
    returnInfo = "found"
    if artag == 1:
        if box_color == "red":
            X = locY
            Y = locX
            returnInfo = findYellow()
        elif box_color == "yellow":
            Y = 18 - locY
            X = locX
            returnInfo = box_coord
        elif box_color == "blue":
            X = 18 - locX
            Y = locY
            returnInfo = findYellow()
        elif box_color == "green":
            X = 18 - locY
            Y = 18 - locX
            returnInfo = findYellow()
        if returnInfo != "found":
            find_ARTag(returnInfo)
    else:
        box_color = "yellow"

    # calculating final coordinates to go to the finish position
    final_coordinates = (0, 0)
    if box_color == "red":
        final_coordinates = (YArTag * 2 + 2, XArTag * 2 + 2)
    elif box_color == "yellow":
        final_coordinates = (XArTag * 2 + 2, 16 - YArTag * 2)
    elif box_color == "blue":
        final_coordinates = (16 - XArTag * 2, YArTag * 2 + 2)
    elif box_color == "green":
        final_coordinates = (16 - YArTag * 2, 16 - XArTag * 2)

    move_by_path(final_coordinates, False)
    exit(1)
```