ELSIROS

Выпуск 0.0.1

Azer Babaev, Egor Davydenko, Ilya Ryakin, Vladimir Litvinenko,

сент. 27, 2021

Содержание:

Глава 1

Введение в ELSIROS

Игра в футбол человекоподобными роботами (гуманоидами) становится популярным в университетском и научном сообществе. Этот вид соревнований является источником для большого количества научно – прикладных исследований. Для учеников школ, колледжей и студентов в общем можно сформулировать 3 основных препятствия для создания и поддержания команды из роботов-гуманоидов-футболистов:

1. Слишком высокая цена роботов – гуманоидов,
2. Слишком сложные алгоритмические задачи, которые необходимо решить на начальном этапе включения робота в игру,
3. Маленькое число команд в одном населенном пункте – означает, что каждаигра возможна только в комбинации с поездкой на дальние расстояния с относительно большими расходами на командировку.

Все 3 препятствия можно избежать, если новая команда начинает приобретать опыт футбола гуманоидов с ELSIROS. Используя ELSIROS, можно воспользоваться следующими преимуществами:

1. ELSIROS полностью бесплатный и с открытым кодом.
2. Самые трудные части освоения гуманоидных роботов, которые выходят за рамки программы школы и колледжей, такие как обратная кинематика, генератор ходьбы, планирование траекторий, детекция мяча и препятствий и локализация обеспечены в готовом виде с предоставлением исходного кода. Пример стратегии игры, который был доказан как успешный на играх реальных роботов предоставляется для изучения и дальнейшего совершенствования.
3. Для участия в соревнованиях и товарищеских играх нет необходимости в командировках. Команды – участники могут скомпилировать свой исходный код в исполняемый бинарный код, который обеспечен защитой от копирования алгоритмов, и выгрузить его на сервер судьи.
4. Команды не испытывают проблем с люфтами сервомоторов, разрегулировкой, раскалибровкой, так как виртуальная модель предоставляется без люфтов, отрегулированной и откалиброванной.
5. Модули стратегии, которые будут создавать команды в дальнейшем готовы к использованию на физических роботах, которые команды могут захотеть построить сами или приобрести готовыми.

6. Для симуляции тренировочных игр не требуется мощных серверов, симуляцию игры можно запустить даже на портативном компьютере.

ELSIROS – бесплатная с открытым кодом платформа, претендующая с одной стороны помочь новым исследовательским кружкам войти в мир футбола гуманоидных роботов, с другой стороны стать основой для виртуальных соревнований

1. Симулятор Webots (скачивается с сайта разработчика);
2. Первоначальная модель робота, которая является виртуальной моделью физически существующего робота – победителя международных соревнований футбола гуманоидных роботов в 2019, 2020 и 2021 годах, использовавшийся командой «Robokit»;
3. Виртуальное окружение симулятора в виде футбольного поля;
4. Программа судьи, действующего Автономно или под управлением человека, а также программа управления игрой (game controller);
5. Пакет программ управления виртуальным роботом, обеспечивающий игру в футбол;

Вновь образованные команды могут использовать для участия в соревнованиях виртуальную модель робота «Robokit-1» а также использовать его для изучения основ программирования стратегии игры в футбол роботами-гуманоидами. Это удобный инструмент для изучения Искусственного Интеллекта в школах, колледжах и университетах

Структура пакета управляющей программы построена для 4 уровней программирования робота: Начальный, Средний, Продвинутый и Экспертный

```
https://youtu.be/AmfKpkL2MUc
```

Программисты среднего уровня могут пробовать изменять модуль launcher.py. Этот модуль отвечает за детекцию состояния игры, состояния игрока, управление ролями игроков и их начальных позиций.

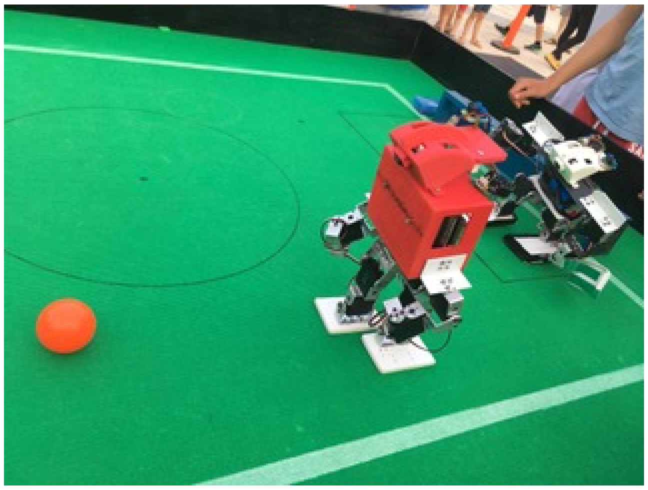
Специалисты экспертного уровня могут собрать свою модель робота и использовать свои программы управления роботом, решая сами пользоваться или не пользоваться исходным кодом, который предоставлен разработчиками ELSIROS. Для допуска к соревнованиям команда, предоставляющая свою собственную виртуальную модель робота, должна пройти квалификацию модели робота. Основное требование к модели робота состоит в том, что модель должна иметь такие же технические характеристики, как и физический робот. Специальные требования к PROTO модели, которые следуют исходя из свойств виртуальной среды, могут быть высланы по запросу.

Если вы руководитель потенциальной команды, которая хочет играть в футбол гуманоидными роботами, то вы можете самостоятельно начинать без особых трудностей. Необходимо скачать и установить ELSIROS на ваш компьютер. На выбор есть версия в исполняемых файлах для Windows 10 или версия для самостоятельной сборки для Linux. Следуйте инструкциям по установке и программа, приведенная как пример, начнет играть в футбол сразу после установки. Используйте вашу предпочитаемую среду разработки программ Python 3 для улучшения исходного кода игры игрока и вы уже готовы к виртуальным соревнованиям в футбол гуманоидных роботов в симуляции. Вы можете проводить тренировочные игры на своем переносном компьютере.

Глава 2

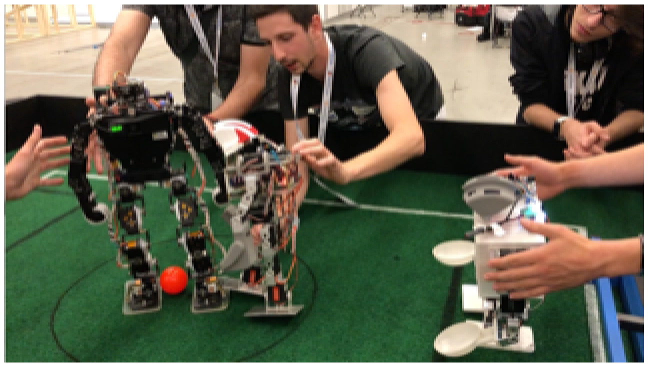
История юношеских гуманоидных футбольных игр

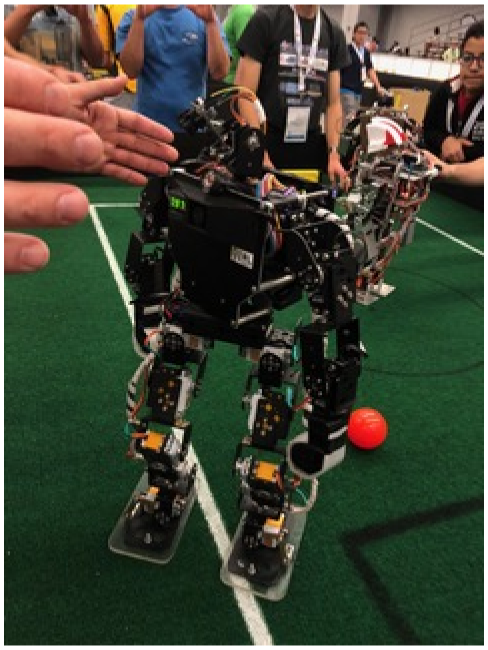
Known early examples of humanoid robot soccer built and programmed by junior students for Robocup refers to year 2016.



Some attepmts to play soccer by humanoid robot built by team from Israel were made at 2016 - 2018

Teams from Israel and Italy have played first International match as demo game at Robocup-2018 in Montreal





In parallel 2 teams from Tomsk and from Moscow have played first game at Robofinist tournament in St.Petersburg at 2018

Same 2 teams have repeated demo game in Moscow at 2019

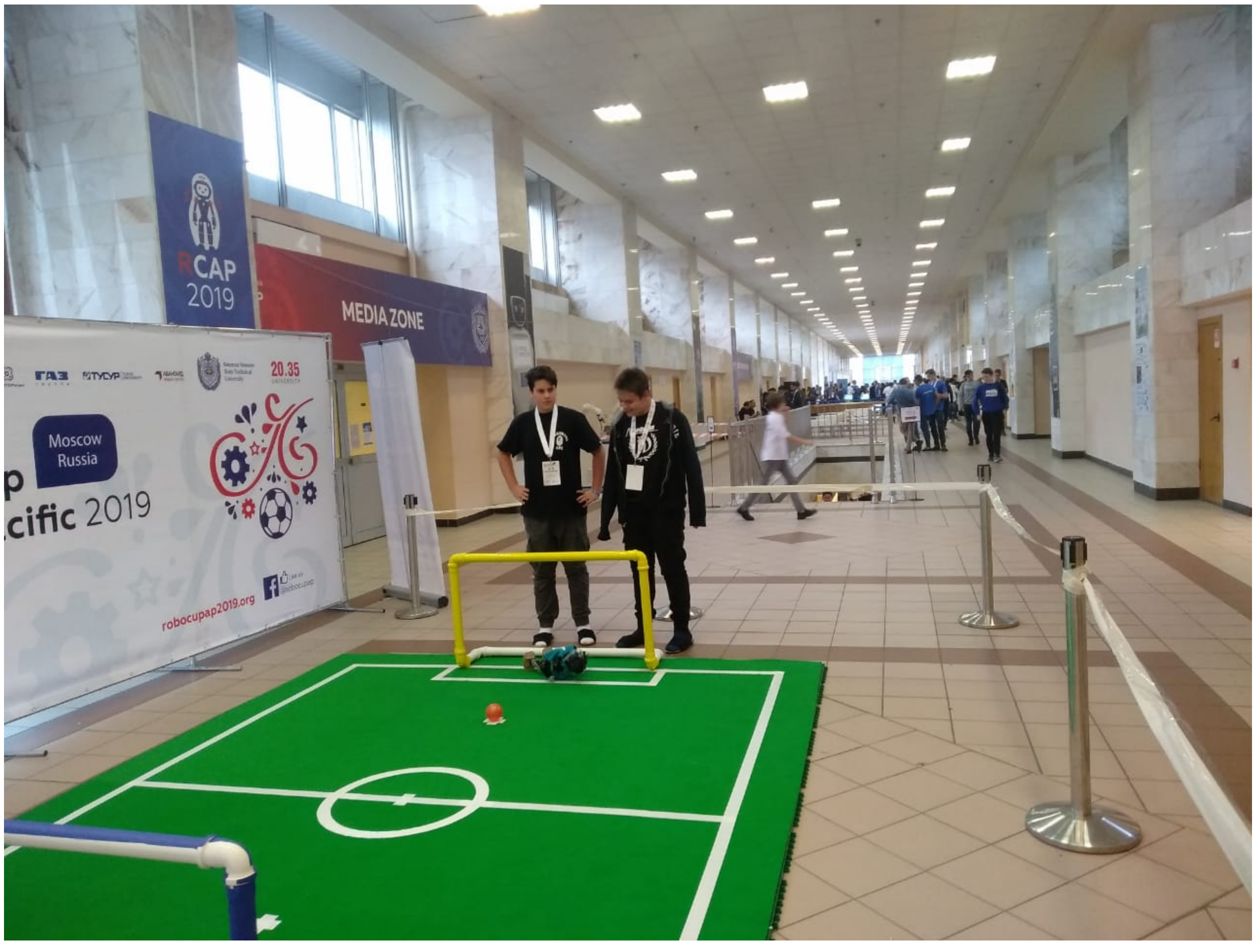
First full function tournament was held in Tomsk-2019 with 3 teams participating in Russian Nathinonal Robocup

ый Российский этап чемпионата
ssia Op 2019
KILLER
HUMANOID SOCCER

Since 2019 the game was included into program of Robocup Asia - Pacific.

2.1 The field

The field is made from 3mm carpet with total size of 3 x 4 m,



with goals made from plumbing tubes, colored to yellow ant blue color.

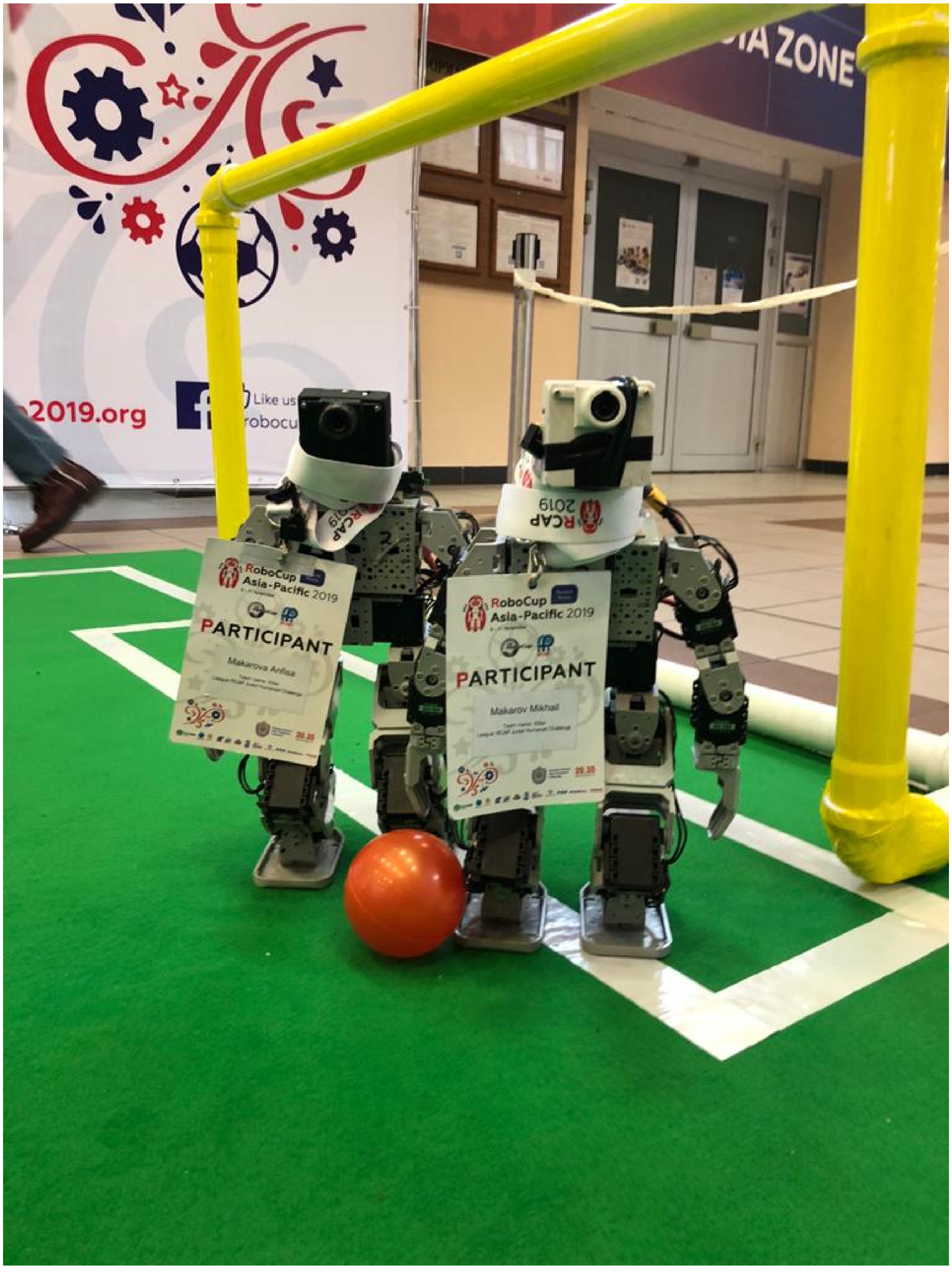
2.2 The ball

The ball is orange color sponge ball with 80mm diameter.

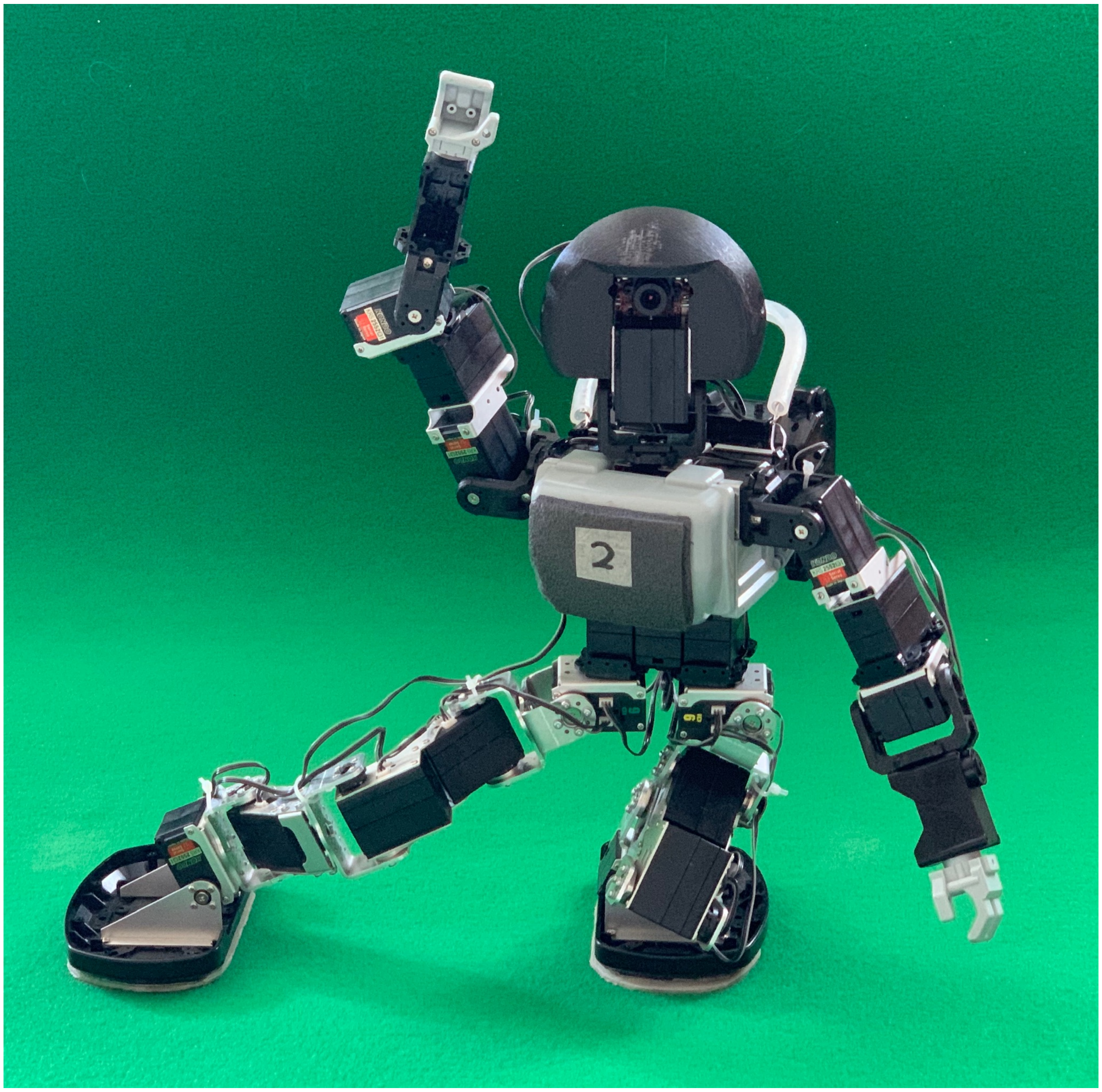
2.3 The Robots

Currently 2 types of robots mostly used by teams:

2.3.1 Bioloid type with added camera head and computer



2.3.2 Robokit - a robot designed in MIPT with using Kondo servomotors from Japan



Last robot which performed best results in soccer game being champion of Asia-Pacific games of 2019 and 2020 is used as prototype for virtual model of standart robot in ELSIROS platform.

Глава 3

Руководство программирования Робота Robokit-1

1. Не рекомендуется играть игру между полностью одинаковыми командами. В большинстве случаем вы будете наблюдать неинтересную игру с малым количеством голов и безрезультатной борьбой. Поэтому мы вам предоставляем 2 разных алгоритма игры: нормальный и старого стиля, который использовался в 2020 году.

2. Реальные роботы используют смарт камеру OpenMV H7 в качестве визуального сенсора и в качестве вычислительного модуля. Это одноядерный контроллер со средой программирования Micropython без операционной системы. Это означает, что контроллер не позволяет одновременно использовать функцию хождения и получения видеокадра. Для обновления информации о нахождении мяча и само-локализации робот должен остановиться в стабилизированной вертикальной позиции и поворачивать голову в разные сектора обзора для осмотра окрестностей. При поворотах головы положение мяча может быть считано, если мяч оказался в видимом секторе камеры. Апертура камеры 46 градусов.

3. В процессе обзора камерой в вертикальной позиции робот также получает информацию, полезную для локализации, такую как стойки ворот, разметка поля и граница зеленого поля. Чем больше кадров обработано с разных ракурсов, тем точнее информация о локализации.

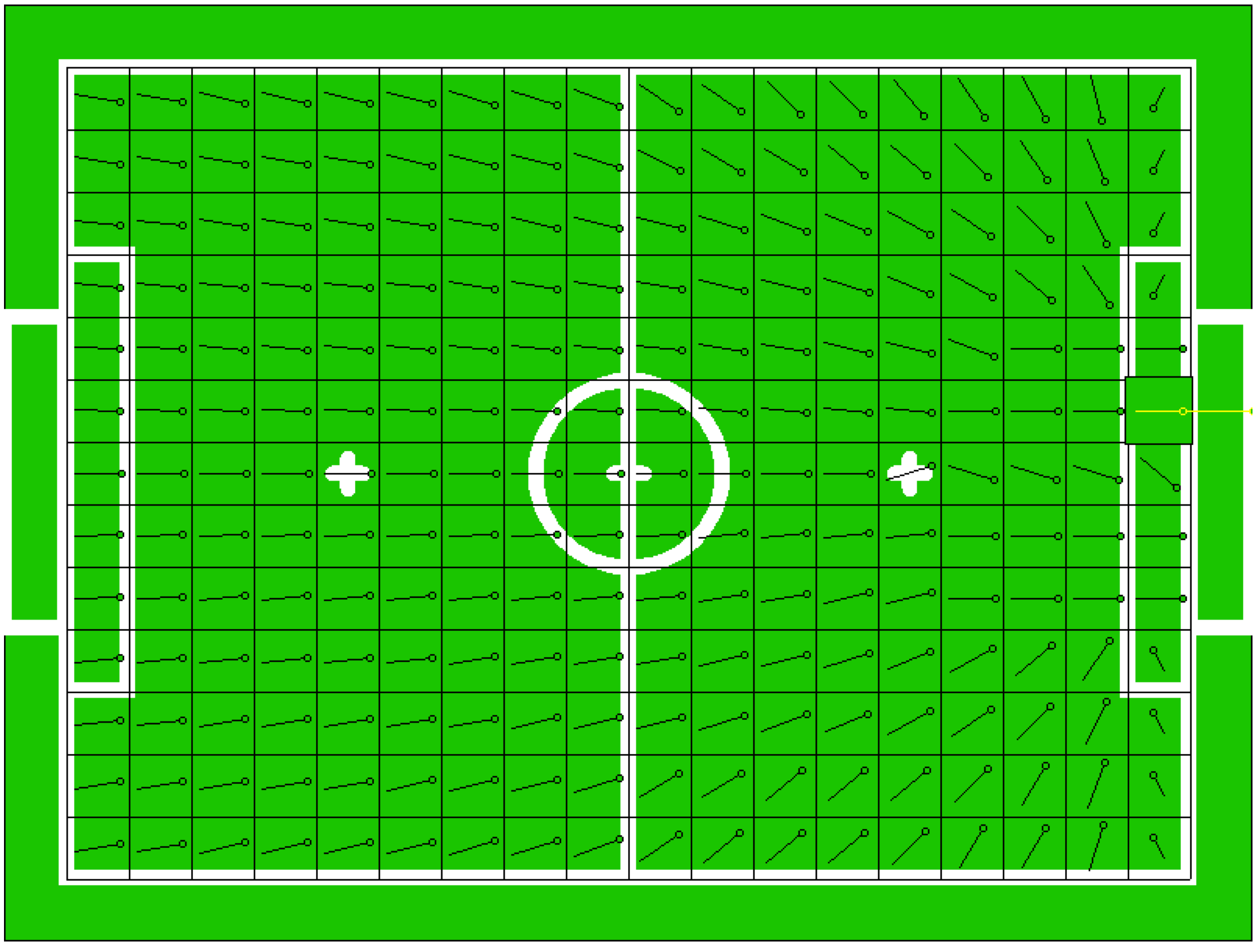
4. Робот может обновлять информацию о препятствиях из полученных кадров. В алгоритм построения траекторий включен алгоритм обхода препятствий, который не совсем совершенен. Нет планирования направления удара с учетом препятствий на пути мяча. Список self.glob.obstacles содержит информацию об обнаруженных и обновленных данных о препятствиях.

5. Коммуникация между игроками разрешена по правилам через UDP сообщения WIFI. Коммуникация не реализована в текущей стратегии, но команды могут её разрабатывать. Коммуникация внутри команды может помочь организовать более эффективную игру. ELSIROS API содержит средства коммуникации между членами команды.

6. Координатное пространство, используемое в стратегии, отличается от абсолютного координатного пространства в симуляции. Для целей стратегии собственные ворота располагаются в части поля с отрицательными значениями координаты X, ворота противника располагаются в части поля с положительными значениями координаты X. Положительные значения Y расположены на левом фланге атаки, отрицательные значения Y расположены на правом фланге атаки. Азимут yaw имеет нулевое значение, если направлен из центра поля к воротам противника. Yaw

изменяется от 0 к math.pi при вращении вокруг вертикальной оси влево от нулевого направления. Yaw изменяется от 0 к -math.pi при вращении вокруг вертикальной оси вправо от нулевого направления. Координата Z направлена вверх с нулём на поверхности поля.

7. Нормальный игрок 'forward' использует предварительно запрограммированную стратегию, заложенную в матрицу векторов. Матрица закодирована в файле strategy_data.json . Этот файл является читаемым и редактируемым так же как и обычный текстовый файл. В файле содержится словарь с одним ключом 'strategy_data'. Под ключом 'strategy_data' содержится список с числом элементов по умолчанию 234. Каждый элемент списка представляет квадратный сектор поля размерами 20cmX20cm. Каждому сектору назначен вектор представляющий из себя направление удара в ситуации когда мяч оказался в этом секторе. Сила удара закодирована в виде ослабляющего коэффициента: 1 – стандартный удар, 2 – удар ослабленный в 2 раза, 3- удар ослабленный в 3 раза. Каждый элемент списка закодирован следующим: [столбец, ряд, сила, yaw]. Футбольное поле разбито на 13 рядов и 18 столбцов. Столбец 0 находится возле своих ворот, столбец 17 находится возле ворот противника. Ряд 0 находится в области с положительной Y координатой, ряд 12 находится в области с отрицательной Y координатой.

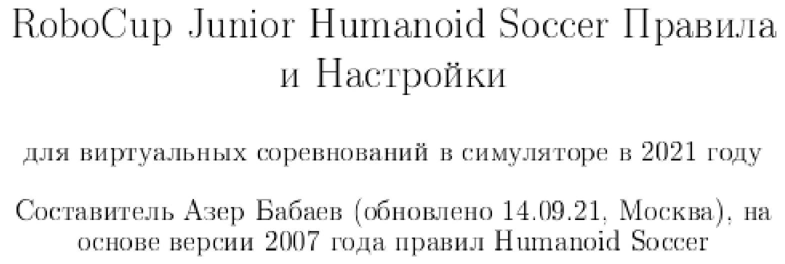


1. В процессе игры игрок может принимать 4 разные роли: 'forward', 'goalkeeper', 'penalty_Shooter', 'penlaty_Goalkeeper'. Для каждой роли программа стратегии разная. Модуль launcher выбирает роль игрока для запуска в зависимости от номера игрока и вторичного состояния игры. В случае если номер игрока 1 назначаемая роль будет 'goalkeeper' или 'penalty_Goalkeeper'. В случае, если номер игрока отличается от 1, тогда назначаемая роль 'forward' или 'penalty_Shooter'. В случае secondary game state = 'STATE_PENALTYSHOOT' тогда игрок с номером 1 будет назначен 'penlaty_Goalkeeper', а игрок с номером, отличающимся от 1 будет назначен на роль 'penalty_Shooter'. Котроллер игры, поставляемый по умолчанию будет

назначать игроку с номером 1 ‘goalkeeper’, а игроку с номером, отличающимся от 1 роль ‘forward’ во всех остальных значениях secondary game state. Команды могут менять стратегию игры путем назначения различных ролей в зависимости от величины secondary game state, поставляемой геймконтроллером. Текущий геймконтроллер может поставлять следующие значения secondary game state: STATE_NORMAL=0, STATE_PENALTYSHOOT=1, STATE_OVERTIME=2, STATE_TIMEOUT=3, STATE_DIRECT_FREEKICK=4, STATE_INDIRECT_FREEKICK=5, STATE_PENALTYKICK=6, STATE_CORNERKICK=7, STATE_GOALKICK=8, STATE_THROWIN=9, DROPBALL=128, UNKNOWN=255

Глава 4

Правила



RoboCup Junior Humanoid Soccer Правила и Настройки

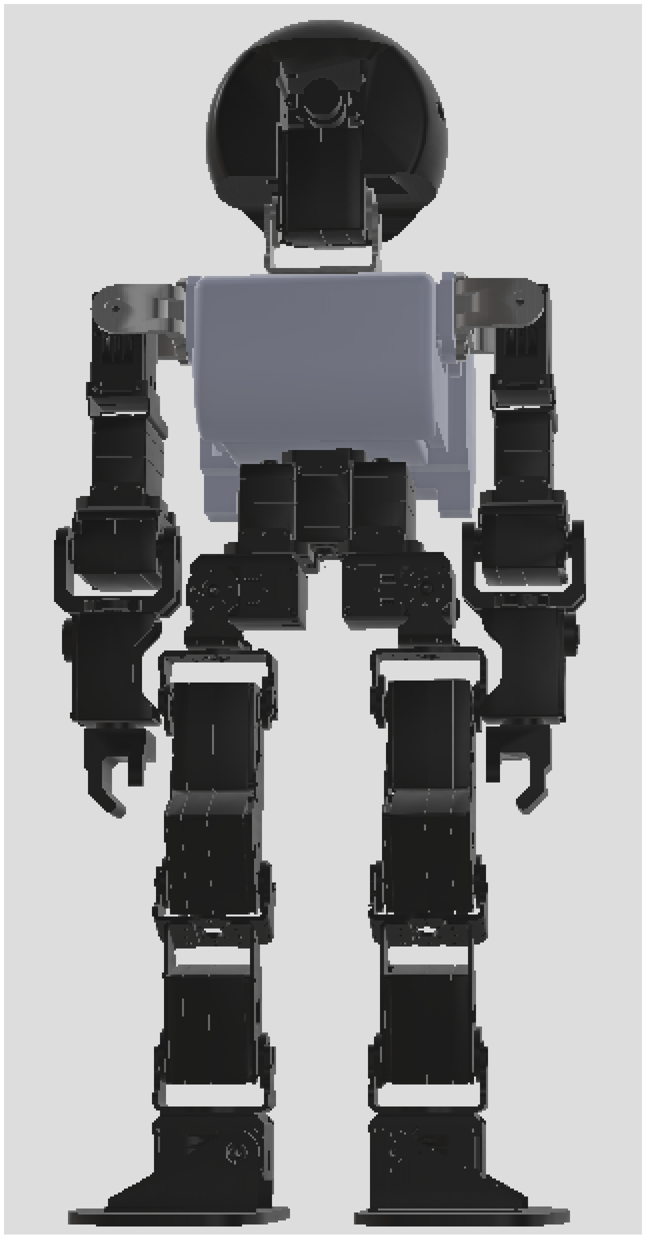
для виртуальных соревнований в симуляторе в 2021 году

Составитель Азер Бабаев (обновлено 14.09.21, Москва), на основе версии 2007 года правил Humanoid Soccer

Глава 5

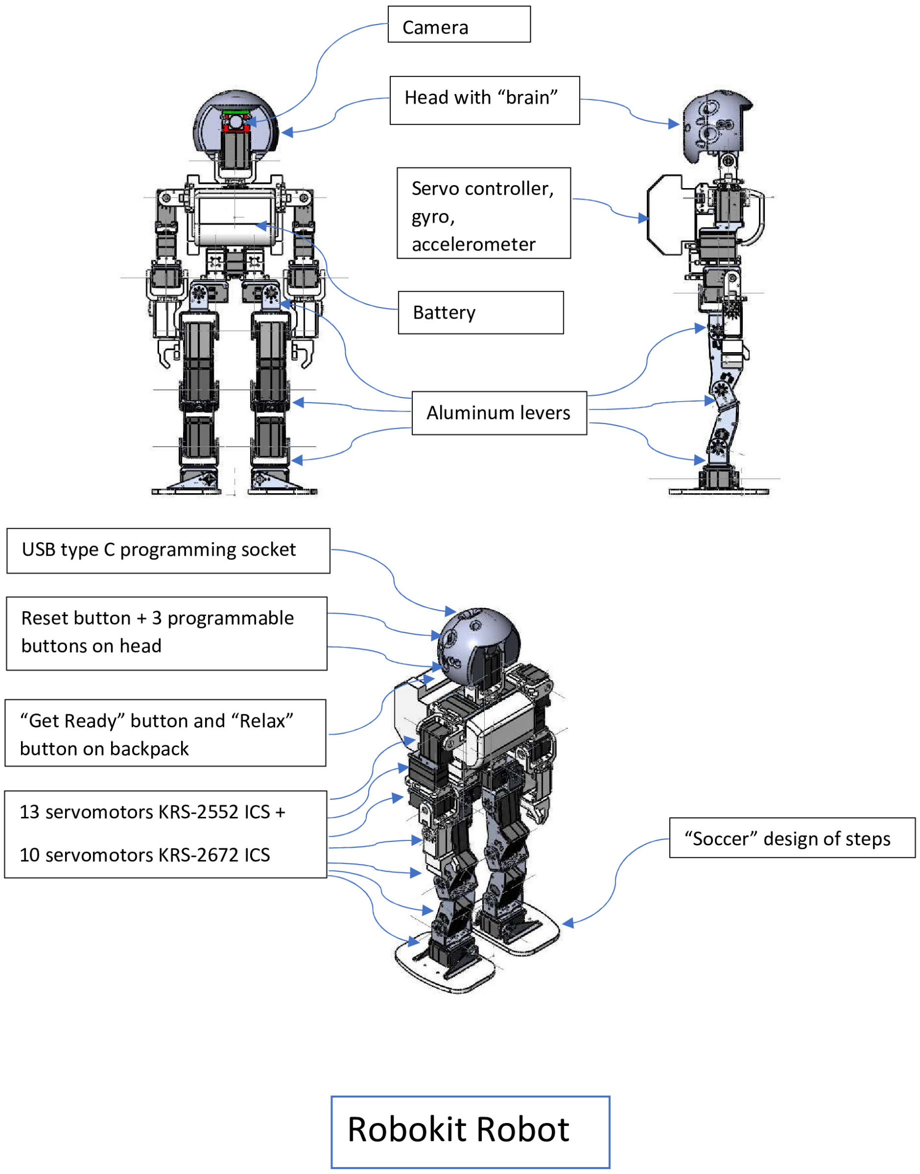
Дизайн робота

Proto стандартного робота, используемого для симуляции взят Robokit1.proto.proto модель создана в качестве виртуальной модели реального робота. Для достоверного отражения поведения реального робота все технические свойства виртуального робота скопированы со спецификаций реального робота, включая такие, как моменты вращения сервомоторов, максимальные скорости сервомоторов, масса сервомоторов, расположение камеры, разрешение камеры, апертура линзы камеры, число и расположение акселерометров, распределение веса по частям тела.



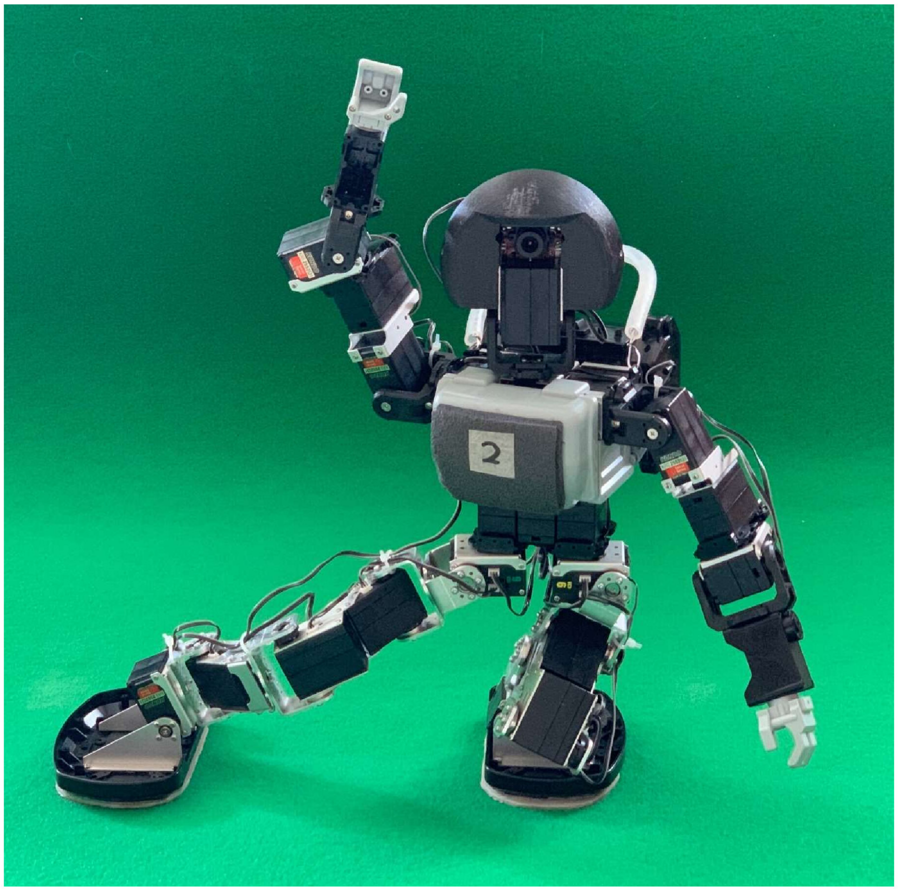
Virtual model has 2 IMUs named «imu_head» and «imu_body». Real robot is capable to recognize coordinate of itself at soccer field with accuracy ±15cm, it is capable to recognize obstacles and ball. Recognition of distance and course to ball and obstacles is made through machine vision algorithms. Robot detectds distance and course to object with accuracy mainly dependent on camera accuracy. Good calibrated robot detects course to object with accuracy ± 0.01 Radian. Accuracy of distance recognition non-linearly depends on distance to object, with best accuracy at minimum distances which is ± 5mm. Distance measurement accuracy at distance 1m is ± 25mm, at distance 2m is ± 100mm High accuracy of distance and course measurement are provided due to equipment and smart algotihms used for direct measurements. Localization on soccer field is provided by measurements of cource and distence to goal posts, field marking, green field border, odometry, measurement of IMU. All localization data is accumulated with historical data and processed by patricle filter algorithm in order to achieve better accuracy than measurement of individual object. Simulation procedure in Webots is orgaised in way when simulation of physics and motion is provided together with camera image scanning. Procedure of image scanning and transfer to outside from Webots greatly reduces simulation speed. In order to keep simulation speed at acceptable rate it was decided to skip procedure of scanning images by camera in simulation and replace it with direct request- report procedure. Following data are reported to Robokit1 robot from simulation by request: distance and course from robot to ball, distance and course from robot to other players, self coordinate and orientation on field. Requested data before reporting to robot are mixed with random values in order to provide same accuracy which usually real robot detects from camera image. Above procedure is called «blurrer». Above procedure provide increasing is simulation speed up to 10 times. This know-how of ELSIROS provides games to be played at laptop computer with nearly realtime speed.

Below is technical description of real Robokit robot.


Camera
Head with "brain"
Servo controller, gyro, accelerometer
Battery
Aluminum levers
USB type C programming socket
Reset button + 3 programmable buttons on head
"Get Ready" button and "Relax" button on backpack
13 servomotors KRS-2552 ICS + 10 servomotors KRS-2672 ICS
"Soccer" design of steps
Robokit Robot

ROBOKIT Robot Specification:

- Height 45 cm
- Weight 1.9 kg
- Battery voltage 12 V
- 23 DOF: 13 servomotors KRS-2552 ICS +10 servomotors KRS-2672 ICS
- Main controller: OpenMV H7 with 32-Bit Arm Cortex-M7 operating at 400MHz with 1Mb SRAM
- Motion controller: Kondo RCB-4HV
- Programming language: Micropython.
- Sensors: OV7725 640x480 camera, 6D digital IMU BNO055, 2D analogue Gyro, 3D analogue Accelerometer



Глава 6

Building elsiros on Ubuntu

6.1 Installing dependencies

1. Install GameController - https://github.com/elsiros/GameController
2. Install Webots R2021b - https://cyberbotics.com/

6.2 Installing elsiros

6.2.1 Clone code

```
git clone https://github.com/elsiros/elsiros_webots
```

6.2.2 Install libs from requirements.txt

```
cd /path/to/elsiros_webots
pip install -r requirements.txt
```

6.2.3 Install protobuf-compiler

```
sudo apt-get install libprotobuf-dev protobuf-compiler
```

6.2.4 Build controller and protobuf messages

```
cd /path/to/elsiros_webots/controllers/player
make
```

6.2.5 Test player controller

```
    cd /path/to/webots
    ./webots /path/to/elsiros_webots/worlds/elsiros_training.wbt
```

```
    cd /path/to/elsiros_webots/controllers/player
    python communication_manager_test.py
```

Глава 7

API

7.1 `SAMPLE_TEAM.Soccer.Motion`

7.1.1 Subpackages

`SAMPLE_TEAM.Soccer.Motion.motion_slots`

7.1.2 Submodules

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq`

Module Contents

Functions

uprint (*text)	
normalize_rotation (yaw)	
steps (motion, x1, y1, u1, x2, y2, u2)	
ball_Approach (motion, local, glob, ball_coord)	

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.uprint(`**text*`)`

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.normalize_rotation(`*yaw*`)`

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.steps(`*motion, x1, y1, u1, x2, y2, u2*`)`

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.ball_Approach(`*motion*, *local*, *glob*, *ball_coord*`)`

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_calc`

Module Contents

Functions

`uprint` (*text)	
`ball_Approach_Calc` (glob, ball_coord)	

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_calc.uprint(`**text*`)`

`SAMPLE_TEAM.Soccer.Motion.ball_Approach_calc.ball_Approach_Calc(`*glob*, *ball_coord*`)`

`SAMPLE_TEAM.Soccer.Motion.class_Motion`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

This module contains walking engine

Module Contents

Classes

`Motion1`

`class SAMPLE_TEAM.Soccer.Motion.class_Motion.Motion1(`*glob*`)`

> `imu_body_yaw(`*self*`)`
>
> `norm_yaw(`*self*, *yaw*`)`
>
> `quaternion_to_euler_angle(`*self*, *quaternion*`)`
>
> `play_Soft_Motion_Slot(`*self*, *name=''*`)`
>
> `computeAlphaForWalk(`*self*, *sizes*, *limAlpha*, *hands_on=True*`)`
>
> `activation(`*self*`)`
>
> `walk_Initial_Pose(`*self*`)`
>
> `walk_Cycle(`*self*, *stepLength*, *sideLength*, *rotation*, *cycle*, *number_Of_Cycles*`)`
>
> `walk_Final_Pose(`*self*`)`
>
> `kick(`*self*, *first_Leg_Is_Right_Leg*, *small=False*`)`

`refresh_Orientation`(*self*)

`SAMPLE_TEAM.Soccer.Motion.class_Motion_Webots_PB`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev. The module is designed to provide communication from motion controller to simulation

Module Contents

Classes

`Motion_sim`

`class SAMPLE_TEAM.Soccer.Motion.class_Motion_Webots_PB.Motion_sim`(*glob*, *robot*, *gcreceiver*, *pause*, *logger*)

Bases: *`SAMPLE_TEAM.Soccer.Motion.class_Motion_real.Motion_real`*

`game_time`(*self*)

`game_time_ms`(*self*)

`pause_in_ms`(*self*, *time_in_ms*)

`sim_Trigger`(*self*, *time*)

`wait_for_step`(*self*, *step*)

`imu_activation`(*self*)

`read_head_imu_euler_angle`(*self*)

`read_imu_body_yaw`(*self*)

`falling_Test`(*self*)

`send_angles_to_servos`(*self*, *angles*, *use_step_correction=False*)

`move_head`(*self*, *pan*, *tilt*)

`simulateMotion`(*self*, *number=0*, *name=''*)

`sim_Get_Ball_Position`(*self*)

`sim_Get_Obstacles`(*self*)

`sim_Get_Robot_Position`(*self*)

`sim_Start`(*self*)

`sim_Progress`(*self*, *simTime*)

`SAMPLE_TEAM.Soccer.Motion.class_Motion_real`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

The module is a part of motion generating functions

Module Contents

Classes

Motion_real

`class SAMPLE_TEAM.Soccer.Motion.class_Motion_real.Motion_real(`*glob*`)`

Bases: *SAMPLE_TEAM.Soccer.Motion.class_Motion.Motion1*

`seek_Ball_In_Pose(`*self, fast_ Reaction_ On, penalty_ Goalkeeper=False, with_ Localization=True*`)`

`watch_Ball_In_Pose(`*self, penalty_ Goalkeeper=False*`)`

`seek_Ball_In_Frame(`*self, with_ Localization=True*`)`

`detect_Ball_Speed(`*self, with_ Localization=False*`)`

`see_ball_confirmation(`*self*`)`

`turn_To_Course(`*self, course, accurate=False*`)`

`head_Up(`*self*`)`

`head_Return(`*self, old_ neck_ pan, old_ neck_ tilt*`)`

`localisation_Motion(`*self*`)`

`normalize_rotation(`*self, yaw*`)`

`near_distance_omni_motion(`*self, dist_ mm, napravl*`)`

`near_distance_ball_approach_and_kick(`*self, kick_ direction, strong_ kick=False, small_ kick=False*`)`

`far_distance_ball_approach(`*self, ball_ coord*`)`

`far_distance_plan_approach(`*self, ball_ coord, target_ yaw, stop_ Over=False*`)`

`SAMPLE_TEAM.Soccer.Motion.compute_Alpha_v3`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. Module can be used for Inverted Kinematics for legs of Robokit-1 robot. Advantage of module against other IK implementations is fast and repeatable calculation benchmark. Result is achieved due to mixed analytic/numerical calculation method. Module is designed for 6 DOF robot leg. From 6 angles one angle is calculated using numerical iterations, other 5 angles are obtained through polynom roots formula calculation. This way prowides fast benchmark and repeatability. Algorithm being implemented in C language with integration into firnware of OpenMV is capable to calculated angles for robot legs within time less than 1ms. Multiple IK solutions are filtered through applying of angle limits within calculation. This yields less time for calculation. usage: create class Alpha instance and call method compute_Alpha_v3 with arguments. Returns list of 0, 1 or 2 lists of servo angles. List of 0 elements means that IK was not solved. List of 1 list means 1 possible solition is detected. List of 2 lists means that plurality of solutions was not filtered by provided arguments.

Module Contents

Classes

Alpha

Attributes

a5

`class SAMPLE_TEAM.Soccer.Motion.compute_Alpha_v3.Alpha`

`compute_Alpha_v3`(*self*, *xt*, *yt*, *zt*, *x*, *y*, *z*, *w*, *sizes*, *limAlpha*)

usage: list: angles = self.compute_Alpha_v3(float: xt, float: yt, float: zt, float: x, float: y, float: z, float: w, list: sizes, list: limAlpha)

angles: list of floats angles in radians of servos which provide target positioning and orientation of robots“ foot

xt: target x coordinate of foots“ center point yt: target y coordinate of foots“ center point zt: target z coordinate of foots“ center point x: x coordinate of vector of orientation of foot y: y coordinate of vector of orientation of foot z: z coordinate of vector of orientation of foot w: rotation in radians of foot around vector of orientation sizes: list of sizes defining distances between servo axles in biped implementation limAlpha: list of limits [minimum, maximum] of servomotors measured in number of encoder ticks

of Kondo series 2500 servomotors.

Target coordinates are measured in local robot coordinate system XYZ with ENU orientation. [0,0,0] point of coordinate system is linked to pelvis of robot. Foot orientation vector has length 1. Base of vector is at bottom of foot and tip of vector is directed down when foot is on floor.

`SAMPLE_TEAM.Soccer.Motion.compute_Alpha_v3.a5 = 21.5`

`SAMPLE_TEAM.Soccer.Motion.path_planning`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev. module can be used for optimized path planing of Robokit-1 robot. usage: create class PathPlan type object instance and call method path_calc_optimum. Optionally module can be launched stand alone for purpose of tuning and observing result of path planing. Being launched stand alone module draws soccer field with player (white circle), ball (orange circle), obstacles (black circles). Circles are movable by mouse dragging. After each stop of mouse new path is drawing.

Module Contents

Classes

PathPlan	Plans optimized path of humanoid robot from start coordinate to target coordinate.
Glob	

Attributes

goalPostRadius
ballRadius
uprightRobotRadius
roundAboutRadiusIncrement

`SAMPLE_TEAM.Soccer.Motion.path_planning.goalPostRadius = 0.15`

`SAMPLE_TEAM.Soccer.Motion.path_planning.ballRadius = 0.1`

`SAMPLE_TEAM.Soccer.Motion.path_planning.uprightRobotRadius = 0.2`

`SAMPLE_TEAM.Soccer.Motion.path_planning.roundAboutRadiusIncrement = 0.15`

`class SAMPLE_TEAM.Soccer.Motion.path_planning.PathPlan(`*glob*`)`

Plans optimized path of humanoid robot from start coordinate to target coordinate. Coordinates are taken together with orientation. Path is composed from initial arc, final arc and connecting line. Connecting line must be tangent to arcs. In case of obstacles on path line additional arc is added in order to go around obstacle. Only one obstacle can be avoided reliably. Avoiding of second obstacle is not guaranteed. Therefore there are used evaluations of prices of variants of path. The Path with cheaper price is returned. Collision with obstacle in far distance is cheaper than collision with obstacle in near distance. During Path heuristic various radiuses of arcs are considered. Arc with zero radius means turning without changing coordinate.

`coord2yaw(`*self*, *x*, *y*`)`

`intersection_line_segment_and_line_segment(`*self*, *x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*`)`

Checks if 2 line segments have common point. :returns: True - if there is common point

False - if not.

x = x1 + (x2 - x1) * t1 t1 - paramentric coordinate y = y1 + (y2 - y1) * t1 x = x3 + (x4 - x3) * t2 t2 - paramentric coordinate y = y3 + (y4 - y3) * t2 x1 + (x2 - x1) * t1 = x3 + (x4 - x3) * t2 y1 + (y2 - y1) * t1 = y3 + (y4 - y3) * t2 t1 = (x3 + (x4 - x3) * t2 - x1) / (x2 - x1) t1 = (y3 + (y4 - y3) * t2 - y1) / (y2 - y1) y1 + (y2 - y1) * (x3 + (x4 - x3) * t2 - x1) / (x2 - x1) = y3 + (y4 - y3) * t2 (y2 - y1) * (x4 - x3)/ (x2 - x1) * t2 - (y4 - y3) * t2 = y3 - y1 - (y2 - y1) * (x3 - x1) / (x2 - x1) t2 = (y3 - y1 - (y2 - y1) * (x3 - x1) / (x2 - x1)) /((y2 - y1) * (x4 - x3)/ (x2 - x1) - (y4 - y3)) if t1 == 0:

t2 = (y1 - y3)/ (y4 - y3) t2 = (x1 - x3)/ (x4 - x3)

`intersection_line_segment_and_circle`(*self*, *x1*, *y1*, *x2*, *y2*, *xc*, *yc*, *R*)
: Checks if line segment and circle have common points. :returns: True - if there is common point

  > False - if not.

  x = x1 + (x2 - x1) * t t - paramentric coordinate y = y1 + (y2 - y1) * t R**2 = (x - xc)**2 + (y - yc)**2 (x1 + (x2 - x1) * t - xc)**2 + (y1 + (y2 - y1) * t - yc)**2 - R**2 = 0 ((x2 - x1) * t)**2 + (x1 - xc)**2 + 2 * (x2 - x1) * (x1 - xc) * t + ((y2 - y1) * t)**2 + (y1 - yc)**2 + 2 * (y2 - y1) * (y1 - yc) * t - R**2 = 0 ((x2 - x1)**2 + (y2 - y1)**2) * t**2 + (2 * (x2 - x1) * (x1 - xc) + 2 * (y2 - y1) * (y1 - yc)) * t + (x1 - xc)**2 + (y1 - yc)**2 - R**2 = 0 a * t**2 + b * t + c = 0 a = (x2 - x1)**2 + (y2 - y1)**2 b = 2 * (x2 - x1) * (x1 - xc) + 2 * (y2 - y1) * (y1 - yc) c = (x1 - xc)**2 + (y1 - yc)**2 - R**2

`intersection_circle_segment_and_circle`(*self*, *x1*, *y1*, *x2*, *y2*, *x0*, *y0*, *CW*, *xc*, *yc*, *R*)

`norm_yaw`(*self*, *yaw*)

`delta_yaw`(*self*, *start_yaw*, *dest_yaw*, *CW*)

`path_calc_optimum`(*self*, *start_coord*, *target_coord*)
: Returns optimized humanoid robot path. usage:

  > list: dest, list: centers, int: number_Of_Cycles = self.path_calc_optimum(list: start_coord, list: target_coord) dest: list of destination point coordinates. Each coordinate is list or tuple of floats [x,y].
  >
  > > Each coordinate is starting or end point of path segment. Path comprises of following segments: circle segment, line segment, n*(circle segment, line segment), circle segment. Where n - iterable.
  >
  > centers: list of coordinates of circle centers of circle segments of path. Each coordinate is list or tuple of floats [x,y]. number_Of_Cycles: integer which represents price of path. In case if value is >100 then collision with second obstacle on path
  >
  > > is not verified.
  >
  > start_coord: list or tuple of floats [x, y, yaw] target_coord: list or tuple of floats [x, y, yaw]

`path_calc`(*self*, *start_coord*, *target_coord*)

`arc_path_external`(*self*, *x1*, *y1*, *yaw1*, *x2*, *y2*, *yaw2*)

`check_Obstacle`(*self*, *xp1*, *yp1*, *xp2*, *yp2*)

`check_Limits`(*self*, *x1*, *y1*, *x2*, *y2*, *xp1*, *yp1*, *xp2*, *yp2*, *xc1*, *yc1*, *CW1*, *xc2*, *yc2*, *CW2*, *dest*)

`check_Price`(*self*, *x1*, *y1*, *x2*, *y2*, *xp1*, *yp1*, *xp2*, *yp2*, *xc1*, *yc1*, *CW1*, *xc2*, *yc2*, *CW2*, *dest*, *centers*)

`number_Of_Cycles_count`(*self*, *dest*, *centers*, *yaw1*, *yaw2*)

`external_tangent_line`(*self*, *start*, *R1*, *R2*, *x1*, *y1*, *xc1*, *yc1*, *xc2*, *yc2*, *CW*)

`arc_path_internal`(*self*, *x1*, *y1*, *yaw1*, *x2*, *y2*, *yaw2*)

`internal_tangent_line`(*self*, *start*, *R1*, *R2*, *x1*, *y1*, *xc1*, *yc1*, *xc2*, *yc2*, *CW*)

`square_equation`(*self*, *a*, *b*, *c*)

`class SAMPLE_TEAM.Soccer.Motion.path_planning.Glob`

`import_strategy_data`(*self*, *current_work_directory*)

7.2 `SAMPLE_TEAM.Soccer.Localisation`

7.2.1 Submodules

`SAMPLE_TEAM.Soccer.Localisation.class_Glob`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

This module is used to store variables which are used in many classes

Module Contents

Classes

`Glob`

`class SAMPLE_TEAM.Soccer.Localisation.class_Glob.Glob(`*simulation, current_work_directory*`)`

`import_strategy_data(`*self, current_work_directory*`)`

`SAMPLE_TEAM.Soccer.Localisation.class_Local`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

This module is assisting localization

Module Contents

Classes

`Local`

`class SAMPLE_TEAM.Soccer.Localisation.class_Local.Local(`*logger, motion, glob, coord_odometry=[0.0, 0.0, 0.0]*`)`

`coordinate_fall_reset(`*self*`)`

`coordinate_trust_estimation(`*self*`)`

`normalize_yaw(`*self, yaw*`)`

`correct_yaw_in_pf(`*self*`)`

`coordinate_record(`*self*`)`

`localisation_Complete(`*self*`)`

`group_obstacles(self)`

`read_Localization_marks(self)`

7.3 SAMPLE_TEAM.Soccer.strategy

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. The module is designed for strategy of soccer game for forward and goalkeeper.

7.3.1 Module Contents

Classes

GoalKeeper	class GoalKeeper is designed to define goalkeeper's play according to style developed by
Forward	The class Forward is designed for definition of strategy of play for 'forward' role of player
Forward_Vector_Matrix	The class Forward_Vector_Matrix is designed for definition of strategy of play for 'forward' role of player
Player	class Player is designed for implementation of main cycle of player.

class SAMPLE_TEAM.Soccer.strategy.GoalKeeper(*logger, motion, local, glob*)

class GoalKeeper is designed to define goalkeeper's play according to style developed by Matvei Ivaschenko - a student of Phystech Lyceum in 2020. Idea of style was in dividing of home half of shoccer field to 8 sectors according to distance from home goals. When ball is in 4 sectors closest to goals A1, A2, A3, A4 goalkeeper attacks ball with purpose transfer it to side of opponent. When ball is in 4 sectors B1, B2, B3, B4 which are in longer distance from goals goalkeeper just slide to better position from current without attempt to attack ball. In case if ball didn't go longer than 1m after kick of goalkeeper, he will undertake another attempt up to 10 times in total. In case if ball goes to distance longer than 1m or ball can't be seen by goalkeeper then goalkeeper returns to center of goals.

turn_Face_To_Guest(*self*)

The method is designed to define kick direction and load it into self.direction_To_Guest direction is measured in radians of yaw. After definition of kick direction robot turns to this direction. In case if robots" own coordinate self.glob.pf_coord shows lication on own half of field i.e. self.glob.pf_coord[0] $<$ 0 then direction of shooting is 0 if robots" x coordinate $>$ 0.8 and abs(y coordinate) $>$ 0.6 then direction of kick is to center of opponents" goals. if robots" x $<$ 1.5 and abs(y) $<$ 0.25 kick direction will be to corner of opponents" goals, with left or right corner is defined randomly. in all other positions of robot kick direction is defined as ditection to target point with coordinates x = 0, y = 2.8

goto_Center(*self*)

Goalkeeper returns to duty position 0.4m in front of own goals. before returning robot checks trustability of localization. If localization is poor then robot undertake special motions by head and by turning to goals with purpose to improve localization. In case if distance to duty position is more than 0.5m then far_distance_plan_approach will be used, else near_distance_omni_motion will be used. after returning to duty position robot turns to kick direction, for which yaw=0 in front of own goals.

`find_Ball`(*self*)
: Before using motion method seek_Ball_In_Pose goalkeeper define usage of method in quick mode or in accurate mode. In case if localization is trustable quick mode is used means fast_Reaction_On=True. seek_Ball_In_Pose method moves head of robot to 15 positions covering all visible areas in front and in sides of robot. this way seeking of ball is not single task. Robot improves localization through observing localization markers on obtained pictures. In case if fast_Reaction_On=True then observation of surroundings will be interrupted as soon as ball appear in visible sector. Speed of ball is also detected.

`scenario_A1`(*self*, *dist*, *napravl*)
: This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents" goals before seeking ball. usage:

  None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

  method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

`scenario_A2`(*self*, *dist*, *napravl*)
: This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from math.pi/4 to math.pi/2. Supposed that goalkeeper stands on duty position faced to opponents" goals before seeking ball. usage:

  None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

  method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

`scenario_A3`(*self*, *dist*, *napravl*)
: This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from 0 to -math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents" goals before seeking ball. usage:

  None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

  method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

`scenario_A4`(*self*, *dist*, *napravl*)
: This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from -math.pi/4 to -math.pi/2. Supposed that goalkeeper stands on duty position faced to opponents" goals before seeking ball. usage:

  None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

  method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

`scenario_B1`(*self*)
: This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands

on duty position faced to opponents“ goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls“ Y coordinate. In case if balls“ Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

`scenario_B2`(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents“ goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls“ Y coordinate. In case if balls“ Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

`scenario_B3`(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents“ goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls“ Y coordinate. In case if balls“ Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

`scenario_B4`(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents“ goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls“ Y coordinate. In case if balls“ Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

`class SAMPLE_TEAM.Soccer.strategy.Forward`(*logger*, *motion*, *local*, *glob*)

The class Forward is designed for definition of strategy of play for „forward“ role of player in year 2020. usage:

Forward(object: motion, object: lical, object: glob)

`dir_To_Guest`(*self*)

The method is designed to define kick direction and load it into self.direction_To_Guest, direction is measured in radians of yaw. In case if robots“ own coordinate self.glob.pf_coord shows lication on own half of field i.e. self.glob.pf_coord[0] < 0 then direction of shooting is 0 if robots“ x coordinate > 0.8 and abs(y coordinate) > 0.6 then direction of kick is to center of opponents“ goals. if robots“ x < 1.5 and abs(y) < 0.25 kick direction will be to corner of opponents“ goals, with left or right corner is defined randomly. in all other positions of robot kick direction is defined as ditection to target point with coordinates x = 0, y = 2.8 returns float: self.direction_To_Guest

`turn_Face_To_Guest`(*self*)

`class SAMPLE_TEAM.Soccer.strategy.Forward_Vector_Matrix`(*logger*, *motion*, *local*, *glob*)

The class Forward_Vector_Matrix is designed for definition of strategy of play for „forward“ role of player in year 2021. Matrix is coded in file strategy_data.json This file is readable and editable as well as normal text file. There is a dictionary with one key "strategy_data". Value of key "strategy_data" is a list with default number of elements 234. Each element of list represents rectangular sector of soccer field with size 20cmX20cm. For each sector there assigned a vector representing yaw direction of shooting when ball is positioned in this sector. Power of shot is coded by attenuation value: 1 – standard power, 2 – power reduced 2 times, 3- power reduced 3 times. Each element of list is coded as follows: [column, row, power, yaw]. Soccer field is split to sectors in 13 rows and 18 columns. Column 0 is near own goals, column 17 is near opposed goals. Row 0 is in positive Y coordinate, row 12 is in negative Y coordinate. Strategy data is imported from strategy_data.json file into self.glob.strategy_data list. usage:

Forward_Vector_Matrix(object: motion, object: local, object: glob)

`dir_To_Guest`(*self*)
The method is designed to define kick direction and load it into self.direction_To_Guest. Direction is measured in radians of yaw. usage:

int: row, int: col = self.dir_To_Guest() row, col - row and column of matrix attributing rectangular sector of field where ball coorinate self.glob.ball_coord fits.

`turn_Face_To_Guest`(*self*)

`class SAMPLE_TEAM.Soccer.strategy.Player`(*logger*, *role*, *second_pressed_button*, *glob*, *motion*, *local*)
class Player is designed for implementation of main cycle of player. Real robot have 3 programmable buttons. Combination of button pressing can transmit to programm pressed button code from 1 to 9. At initial button pressing role of player is selected. With second pressed button optional playing mode is selected depending on role. For „forward" and „forward_old_style" role second_pressed_button can take value 1 or value 4. With value 1 player starts game as kick-off player, with value 4 player stars as non-kick-off player, which means player starts moving 10 seconds later. For „run_test" role second_pressed_button can take values from 1 or value 9 with following optional modes: 1 - 10 cycle steps walk forward 2 - 20 cycle side step walk to right 3 - 20 cycle side step walk to left 4 - 20 cycle steps walk forward 5 - 20 cycle steps with rotation to right side 6 - 20 cycle steps with rotation to left side 9 - 20 cycle steps of spot walk All modes of run test are used with purpose to calibrate walking. After calibration is completed results of calibration must be input to file Sim_params.json. Motion module is used calibration data for planning motions and odometry correction into localization. usage:

Player(str: role, int: second_pressed_button, object: glob, object: motion, object: local)

`play_game`(*self*)

`rotation_test_main_cycle`(*self*, *pressed_button*)

`run_test_main_cycle`(*self*, *pressed_button*)
For „run_test" role second_pressed_button can take values from 1 or value 9 with following optional modes: 1 - 10 cycle steps walk forward 2 - 20 cycle side step walk to right 3 - 20 cycle side step walk to left 4 - 20 cycle steps walk forward 5 - 20 cycle steps with rotation to right side 6 - 20 cycle steps with rotation to left side 9 - 20 cycle steps of spot walk All modes of run test are used with purpose to calibrate walking. After calibration is completed results of calibration must be input to file Sim_params.json. Motion module is used calibration data for planning motions and odometry correction into localization. usage:

self.run_test_main_cycle(int: pressed_button)

`sidestep_test_main_cycle`(*self*, *pressed_button*)

`norm_yaw`(*self*, *yaw*)
This module normalizes yaw according to internal rule: -pi <= yaw <= pi usage:

float: yaw = self.norm_yaw(float: yaw) yaw - orientation on horizontal surface in radians,

zero value orientation is directed along X axis

`forward_main_cycle`(*self*, *pressed_button*)
Main cycle method for „forward" role of player. usage:

self.forward_main_cycle(int: pressed_button)

`forward_old_style_main_cycle`(*self*, *pressed_button*)
Main cycle method for „forward_old_style" role of player. usage:

self.forward_main_cycle(int: pressed_button)

`goalkeeper_main_cycle`(*self*)
: goalkeeper main cycle method is based on vector matrix strategy. Goalkeeper doesn't leave goals too far. Supposed that goalkeeper starts game at point on middle of goal line. After 10 seconds from game start goalkeeper moves to duty position which depends on detected ball position. In case if ball appears in dangetous position goalkeeper attcks ball.

`goalkeeper_old_style_main_cycle`(*self*)
: main cycle for old style goalkeeper strategy

`penalty_Shooter_main_cycle`(*self*)
: main cycle for penalty striker

`penalty_Goalkeeper_main_cycle`(*self*)

`dance_main_cycle`(*self*)

7.4 `SAMPLE_TEAM.launcher_pb`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. The module is designed for strategy of soccer game by forward and goalkeeper.

7.4.1 Module Contents

Functions

`init_gcreceiver`(team, player, is_goalkeeper)	The function creates and object receiver of Game Controller messages. Game Controller messages are broadcasted to
`player_super_cycle`(falling, team_id, robot_color, player_number, SIMULATION, current_work_directory, robot, pause, logger)	The function is called player_super_cycle because during game player can change several roles. Each role

`SAMPLE_TEAM.launcher_pb.init_gcreceiver`(*team*, *player*, *is_goalkeeper*)
: The function creates and object receiver of Game Controller messages. Game Controller messages are broadcasted to teams and to referee. Format of messages can be seen in module gamestate.py. Messages from Game Controller contains Robot info, Team info and Game state info. usage of function:

object: receiver = init_gcreceiver(int: team, int: player, bool: is_goalkeeper)

team - number of team id. For junior competitions it is recommended to use unique id
for team in range 60 - 127

player - number of player displayed at his trunk is_goalkeeper - True if player is appointed to play role of goalkeeper

`SAMPLE_TEAM.launcher_pb.player_super_cycle`(*falling*, *team_id*, *robot_color*, *player_number*, *SIMULATION*, *current_work_directory*, *robot*, *pause*, *logger*)
: The function is called player_super_cycle because during game player can change several roles. Each role appointed to player put it into cycle connected to playing it's role. Cycles of roles are defined in strategy.py module. player_super_cycle is cycle of cycles. For example player playing role of „forward" can change role to „penalty_shooter" after main times and extra times of game finished. In some situations you may decide to switch roles between forward player and goalkeeper. Usage:

player_super_cycle(object: falling, int: team_id, str: robot_color, int: player_number, int: SIMU
Path_object: current_work_directory, object: robot, object: pause)

falling - class object which contains int: falling.Flag which is used to deliver information about fall
low level logic to high level logic. falling.Flag can take 0 - nothing happend, 1 -falling on stomach, -1 - falling face up, 2 - falling to left, -2 - falling to right, 3 - exit from playing fase

team_id - can take value from 60 to 127 robot_color - can be „red" or „blue" player_number - can be from 1 to 5, with 1 to be assigned to goalkeeper SIMULATION - used for definition of simulation enviroment. value 4 is used for Webots simulation,

value 2 is used for playing in real robot

current_work_directory - is Path type object robot - object of class which is used for communication between robot model in simulation and controller

program. In case of external controller program „ProtoBuf" communication manager is used. „ProtoBuf" - is protocol developed by Google.

pause - object of class Pause which contains pause.Flag boolean variable. It is used to transfer pre
pause button on player's dashboard event to player's high level logic.

7.5 `SAMPLE_TEAM.main_pb`

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. The module is designed for creating players" dashboard and alternating between team game with Game Controller or individual play without Game Controller.

7.5.1 Module Contents

Classes

`Log`
`Falling`
`Pause`
`RedirectText`
`Main_Panel`

Functions

main_procedure()	
main()	

Attributes

LOGGING_LEVEL	
SIMULATION	
current_work_directory	
game_data	
team_1_data	
team_2_data	
pause	

SAMPLE_TEAM.main_pb.LOGGING_LEVEL

SAMPLE_TEAM.main_pb.SIMULATION = 4

SAMPLE_TEAM.main_pb.current_work_directory

SAMPLE_TEAM.main_pb.game_data

SAMPLE_TEAM.main_pb.team_1_data

SAMPLE_TEAM.main_pb.team_2_data

class SAMPLE_TEAM.main_pb.Log(*filename*)

get_file_handler(*self*)

get_stream_handler(*self*)

get_logger(*self*, *name*)

class SAMPLE_TEAM.main_pb.Falling

class SAMPLE_TEAM.main_pb.Pause

SAMPLE_TEAM.main_pb.pause

SAMPLE_TEAM.main_pb.main_procedure()

class SAMPLE_TEAM.main_pb.RedirectText(*aWxTextCtrl*)

Bases: object

write(*self*, *string*)

`class SAMPLE_TEAM.main_pb.Main_Panel(`**args*, ***kwargs*`)`

Bases: `wx.Frame`

`main_procedure(`*self*`)`

`InitUI(`*self*`)`

`ShowMessage1(`*self*, *event*`)`

`ShowMessage2(`*self*, *event*`)`

`SAMPLE_TEAM.main_pb.main()`

7.6 communication_manager_robokit

Class that provides communication with simulator Webots.

7.6.1 Module Contents

Classes

CommunicationManager

`class communication_manager_robokit.CommunicationManager(`*maxsize=1*, *host='127.0.0.1'*, *port=10001*, *logger=logging*, *team_color='RED'*, *player_number=1*, *time_step=15*`)`

`enable_sensors(`*self*, *sensors*`)` → None

`__get_sensor(`*self*, *name*`)` → dict

`__send_message(`*self*`)`

`__update_history(`*self*, *message*`)`

`__procces_object(`*self*, *name*`)`

`time_sleep(`*self*, *t*`)` → None

Emulate sleep according to simulation time.

Параметры `t` (*float*) – time

`get_imu_body(`*self*`)` → dict

Provide last measurement from imu located in body. Can be empty if „imu body“ sensor is not enabled or webots does not sent us any measurement. Also contains simulation time of measurement.

Результат {«position»: [roll, pitch, yaw]}

Тип результата dict

`get_imu_head(`*self*`)` → dict

Provide last measurement from imu located in head. Can be empty if „imu_head“ sensor is not enabled or webots does not send us any measurement. Also contains simulation time of measurement.

Результат {«position»: [roll, pitch, yaw], «time»: time}

Тип результата dict

`get_localization(`*self*`)` → dict

Provide blurred position of the robot on the field and confidence in this position („consistency“ - where 1 fully confident and 0 - have no confidence). Can be empty if „gps_body“ sensor is not enabled or webots does not send us any measurement. Also contains simulation time of measurement.

Результат {«position»: [x, y, consistency], «time»: time}

Тип результата dict

`get_ball(`*self*`)` → dict

Provide blurred position of the ball relative to the robot. Can be empty if: 1. „recognition“, „gps_body“ or „imu_body“ sensors are not enabled 2. webots did not send us any measurement. 3. robot does not stand upright position 4. ball is not in the camera field of view (fov)

Also contains simulation time of measurement.

Результат {«position»: [x, y], «time»: time}

Тип результата dict

`get_opponents(`*self*`)` → list

Provide blurred positions of the opponents relative to the robot. Can be empty if:

1. „recognition“, „gps_body“ or „imu_body“ sensors are not enabled
2. webots did not send us any measurement.
3. robot does not stand upright position
4. opponent is not in the camera field of view (fov)

Also contains simulation time of measurement.

Результат [{«position»: [x1, y1], «time»: time}, {«position»: [x2, y2], «time»: time}]

Тип результата list

`get_mates(`*self*`)` → dict

Provide blurred position of the mate relative to the robot. Can be empty if:

1. „recognition“, „gps_body“ or „imu_body“ sensors are not enabled
2. webots did not send us any measurement.
3. robot does not stand upright position
4. mate is not in the camera field of view (fov)

Also contains simulation time of measurement.

Результат {«position»: [x, y], «time»: time}

Тип результата list

`get_time(`*self*`)` → float

Provide latest observed simulation time.

Результат simulation time

Тип результата float

`send_servos`(*self*, *data*) → None

Add to message queue dict with listed servo names and angles in radians. List of posible servos: [«right_ankle_roll», «right_ankle_pitch», «right_knee», «right_hip_pitch», «right_hip_roll», «right_hip_yaw», «right_elbow_pitch», «right_shoulder_twirl», «right_shoulder_roll», «right_shoulder_pitch», «pelvis_yaw», «left_ankle_roll», «left_ankle_pitch», «left_knee», «left_hip_pitch», «left_hip_roll», «left_hip_yaw», «left_elbow_pitch», «left_shoulder_twirl», «left_shoulder_roll», «left_shoulder_pitch», «head_yaw», «head_pitch»]

Параметры `data` (*dict*) – {servo_name: servo_angle, …}

`run`(*self*)

Infinity cycle of sending and receiving messages. Should be launched in sepparet thread. Communication manager launch this func itself in constructor

7.7 `blurrer`

7.7.1 Module Contents

Classes

Blurrer	Simulate localization and vision noize.

`class blurrer.Blurrer`(*object_angle_noize=0.0*, *object_distance_noize=0.0*, *observation_bonus=0.0*, *step_cost=0.0*, *constant_loc_noize=0.0*, *loc_noize_meters=0.0*)

Simulate localization and vision noize. Params is placed in the blurrer.json file. :param object_angle_noize: Noize for angle in radians.

Blurrer will uniformly random value from -object_angle_noize to object_angle_noize and add it to the ground truth course. Defaults to 0.

Параметры

- `object_distance_noize` (*float, optional*) – Noize for distance in percents divided by 100. Blurrer will uniformly random value from -object_distance_noize to object_distance_noize and multiply difference of 1 and this value with ground truth distance. Defaults to 0..
- `observation_bonus` (*float, optional*) – Blurrer will increase the consistency for every good observation (successfuly processed image). Defaults to 0..
- `step_cost` (*float, optional*) – Blurrer will decrease the consistency for every simulation step. Defaults to 0..
- `constant_loc_noize` (*float, optional*) – Constant localization noize. Defaults to 0..
- `loc_noize_meters` (*float, optional*) – Multiplier for consistency, in meters. Defaults to 0. Defaults to 0..

`load_json`(*self*, *filename*)

`course`(*self*, *angle*)

`distance`(*self*, *distance*)

`objects`(*self*, *course=course*, *distance=distance*) `loc`(*self*, *x*, *y*)

`coord`(*self*, *p*)

`step`(*self*)

`observation`(*self*)

`update_consistency`(*self*, *value*)

7.8 `message_manager`

Class operates with protobuff messages. used to create and parse messages.

7.8.1 Module Contents

Classes

MessageManager

`class message_manager.MessageManager`(*logger=logging*, *head_ buffer_ size=4*)

`get_size`(*self*)

Результат Value of heder byte buffer.

Тип результата int

`static create_requests_message()`
Create Empty protobuf class instance for request message.

Результат Empty protobuf class instance.

Тип результата messages_pb2

`static create_answer_message()`
Create Empty protobuf class instance for answer message.

Результат Empty protobuf class instance.

Тип результата messages_pb2

`static build_request_from_file`(*path*)
Parsing data from message file to protobuf message instance.

Параметры `path` (*string*) – path to filename.txt with message.

Результат protobuf class instance of filled message from file.

Тип результата messages_pb2

`build_request_positions`(*self*, *positions*)
Creating an instance of the protobuff class and fills it with the values of the actuators

Параметры `positions` (*dict*) – key - servo name and values - position.

Результат protobuf class instance of filled message with servos.

Тип результата messages_pb2

`static generate_message(`*message*`)`
Generate bytes string for sending message.

Параметры

- `message` (*messages_pb2*) – protobuf class instance of filled
- `message.` –

Результат bytes string of message.

Тип результата bytes

`message_from_file(`*self*, *path*`)`
Function process the protobuff message. Measurement values of sensors, messages from player.exe and webots. Received messages are placed in the dictionary :param path: path to filename.txt with default message. :type path: [string]

Результат protobuf class instance, with values from file.

Тип результата messages_pb2

`get_answer_size(`*self*, *content_size*`)`
Calculating message size from header bytes

Параметры `content_size` (*bytes*) – Byte size of answer message.

Результат Size of answer message.

Тип результата int

`add_initial_request(`*self*, *sensor_name*, *sensor_time*`)`
Generate bytes string for sending message.

Параметры

- `sensor_name` (*string*) – protobuf class instance of filled
- `message.` –

Результат bytes string of message.

Тип результата bytes

`build_initial_request(`*self*`)`
Generate bytes string for initialization message.

Результат bytes string of message.

Тип результата bytes

`parse_answer_message(`*self*, *data*`)`
Parsing answer message from byte array to dict with measurements

Параметры `data` (*[type]*) – [description]

Результат [description]

Тип результата [type]

`static parse_message(`*message*`)` → dict
Function process the protobuff message. Measurement values of sensors, messages from player.exe and webots. Received messages are placed in the dictionary :param message: protobuf class instance :type message: messages_pb2 :param of new message with filled or unfilled.:

Результат dict with keys of names sensors

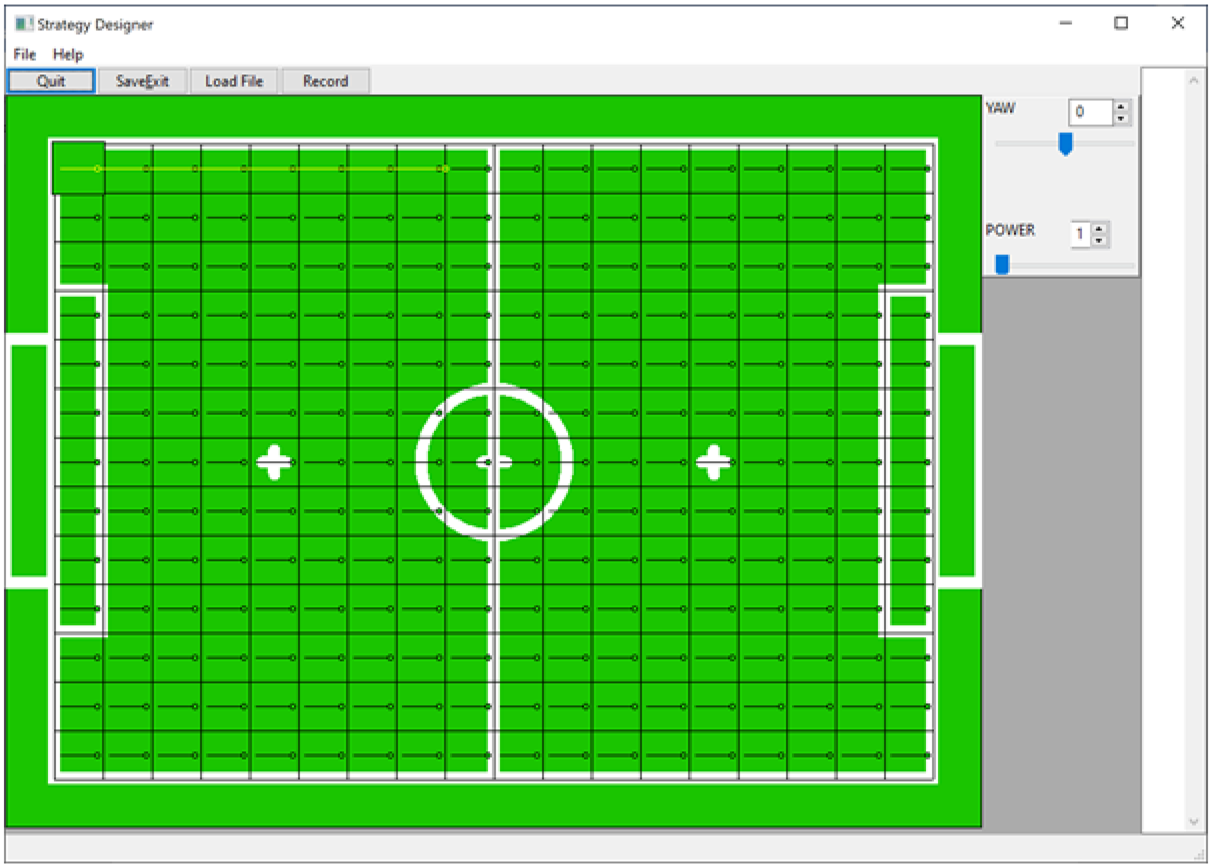
Тип результата dict

Глава 8

TUTORIAL 1

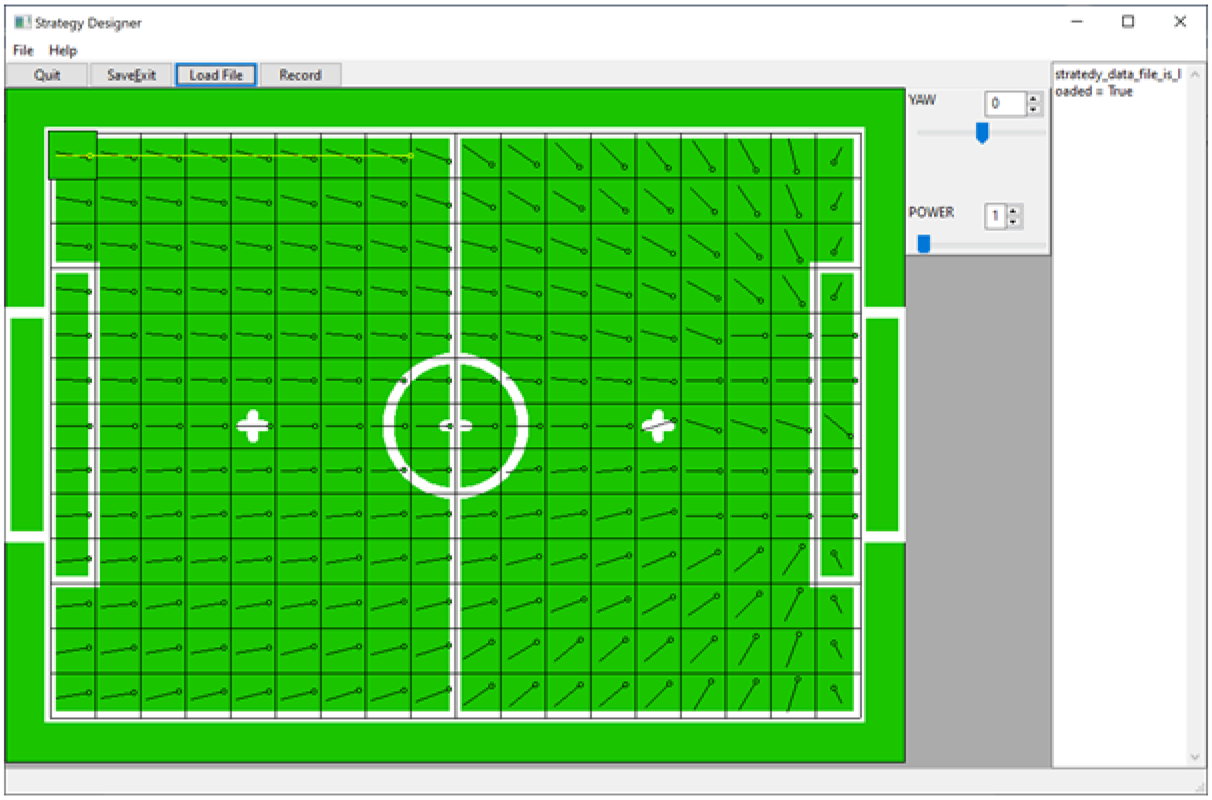
В этом уроке мы изменим стратегию игры игрока «forward» используя матрицу, которая хранится в файле strategy_data.json. Если вы запустите первую игру, которая может быть запущена после установки, то сможете наблюдать такую сцену: красному игроку с номером 2 предоставляется право первого удара. Согласно правиламигрок команды, которой не предоставлено право вводить мяч в игру не имеет право входить в центральный круг на протяжении 10 секунд, поэтому синий игрок стоит за пределами круга в ожидании. Красный игрок наносит удар по мячу прямо в ноги синему игроку. Такое направление удара не является выгодным. Имеет смысл изменить направление удара. Есть несколько способов изменения направление удара. Здесь мы приеним один способ, который связан с изменением матрицы стратегии

1. Откройте консоль операционной системы командой «cmd».
2. Введите с клавиатуры в командную строку "cd C:\Elsiros\controllers\SAMPLE_TEAM". Это изменит текущую директорию на "C:\Elsiros\controllers\SAMPLE_TEAM>"
3. Введите в командную строку "python strategy_designer.py"
4. Когда запустится программа strategy_designer.py вы увидите следующее окно



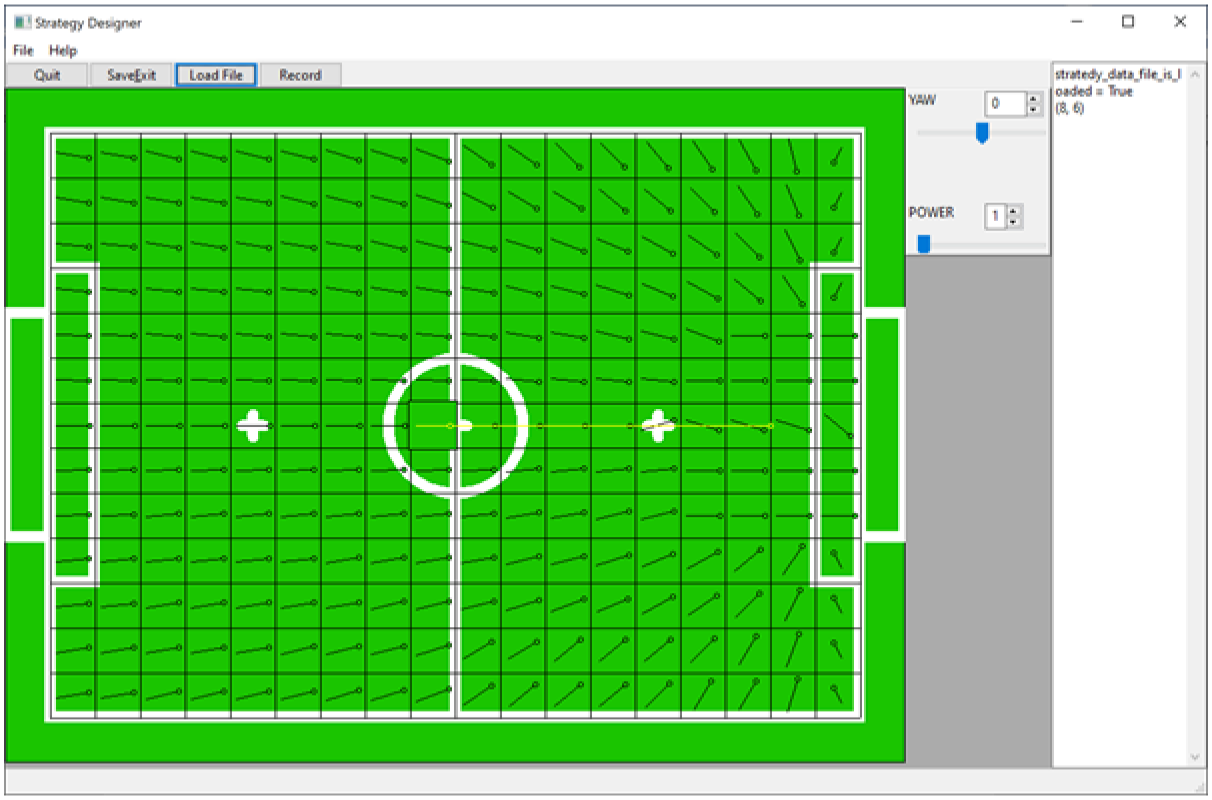
Футбольное поле разделено на сектора. В каждом секторе короткий отрезок с кружочком, который отображает направление и силу удара.

1. Нажмите кнопку “Load File” и загрузите следующий файл: «C:\Elsiros\контроллеры\SAMPLE_TEAM\Init_params\ strategy_data.json». Появится следующее окно:



```
0
1
stratedy_data_file_is_loaded = True
```

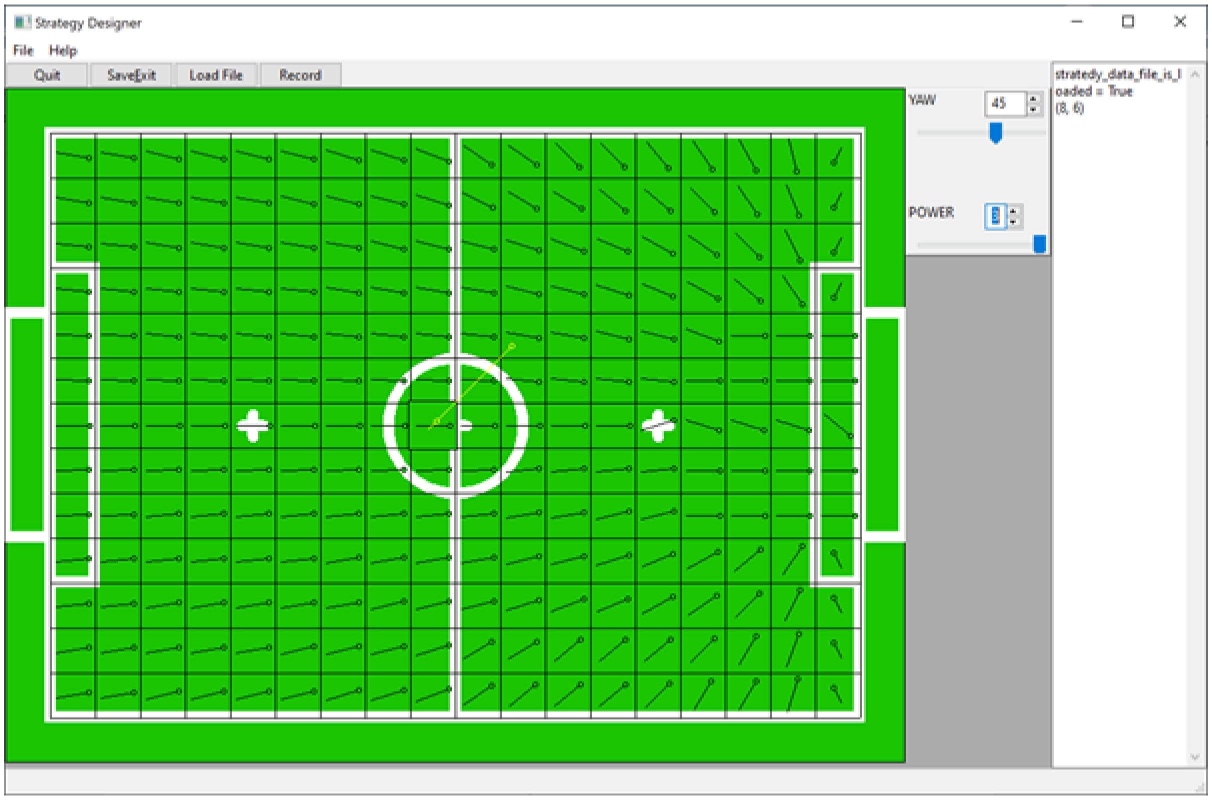
1. Кликните по сектору поля, которое 7-е сверху и 9-е слева.



```
0
1
stratedy_data_file_is_loaded = True
(8, 6)
```

1. Желтая линия отображает предполагаемую дистанцию удара.

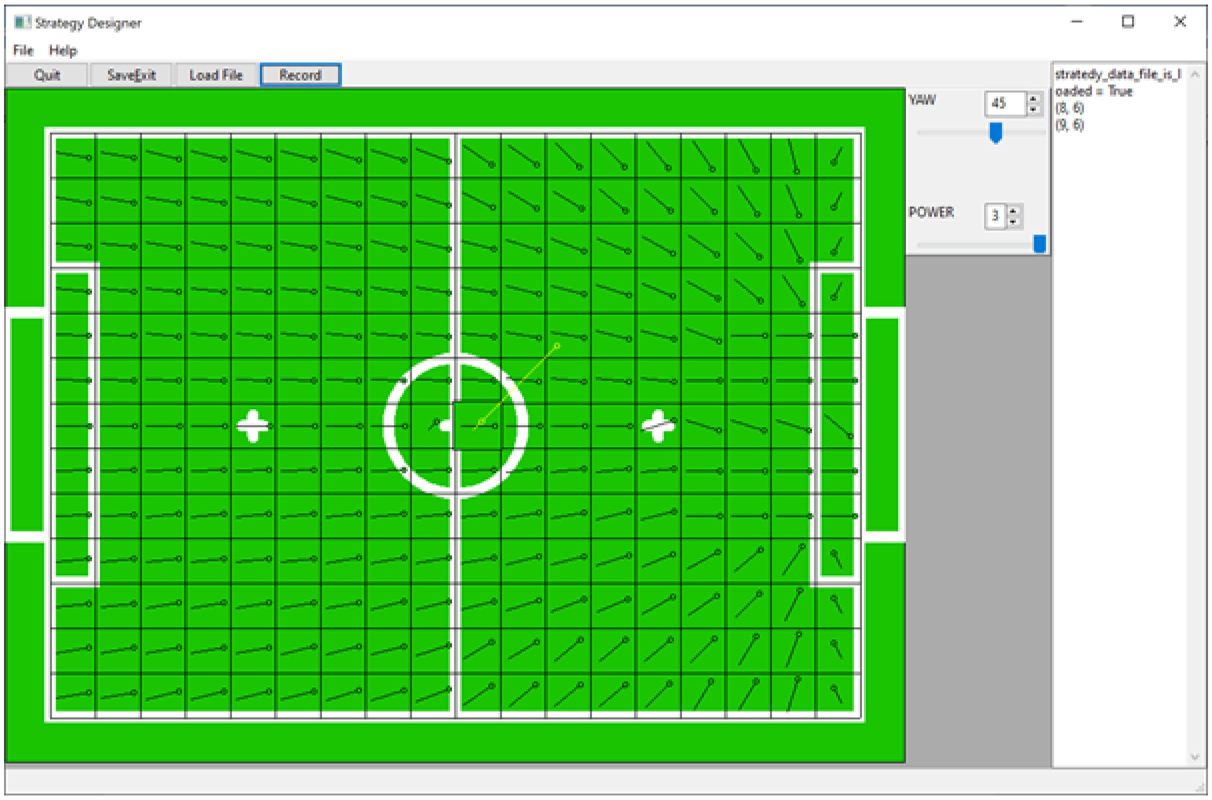
2. Потяните вправо ползунок «YAW», или введите в окне рядом с ним 45 для того чтобы выбрать угол 45 градусов. Потяните ползунок «POWER» или введите в окно рядом с ним цифру 3 для назначения степени ослабления удара.



stratedy_data_file_is_loaded = True
(8, 6)

1. Прежде чем переходить к другой клетке сохраните текущие значения в матрице. Для этого надо нажать на кнопку "Record".

2. Кликните на соседнюю клетку, расположенную 7-й сверху и 10-й слева



stratedy_data_file_is_loaded = True
(8, 6)
(9, 6)

1. Кликните на кнопку "Record" ещё раз для записи такого же значения направления и силы удара в эту клетку.
2. Для сохранения изменений в файле strategy_data.json кликните на кнопку"SaveExit"
3. Запустите игру и посмотрите результат корректировки стратегии.

Содержание модулей Python

Алфавитный указатель