МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Моделирование и симуляция автономных систем» для студентов направления подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Инженерия сложных программных систем»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Методические указания предназначены для студентов направления подготовки 09.03.04 «Программная инженерия» (направленность «Разработка и сопровождение программного обеспечения»). Дисциплина «Моделирование и симуляция автономных систем» изучается в рамках подготовки специалистов в области разработки программного обеспечения автономных систем.

Дисциплина формирует у студентов знания и навыки в области моделирования и симуляции автономных систем: создание 3D-моделей роботов (URDF/SDF), работа с симулятором Gazebo Harmonic совместно с ROS 2, SIL-тестирование навигационных алгоритмов, автоматизация прогонов симуляций, моделирование шумов сенсоров и финальный интеграционный проект.

Практикум построен по принципу последовательного усложнения: от базовой установки и запуска готовой модели через создание собственного URDF с сенсорами к SIL-тестированию и автоматизированному анализу качества алгоритмов.

Каждая лабораторная работа включает цель, теоретическое обоснование, пример выполнения, варианты индивидуальных заданий, требования к отчёту и вопросы для защиты.

Лабораторная работа 1.

УСТАНОВКА GAZEBO И ROS. ЗАПУСК ГОТОВОЙ МОДЕЛИ TURTLEBOT3

Цель и содержание работы: освоить базовую установку связки ROS 2 Jazzy Jalisco и Gazebo Harmonic в Ubuntu 24.04 LTS, развернуть пакеты симуляции мобильного робота TurtleBot3, запустить готовую модель робота в стандартном мире Gazebo, изучить интерфейс симулятора, исследовать список ROS-топиков (/scan, /odom, /cmd_vel, /imu) и научиться управлять моделью робота через клавиатурный teleop-узел.

Теоретическое обоснование

1. Симулятор Gazebo – это трёхмерный симулятор с открытым исходным кодом, предназначенный для моделирования мобильных и манипуляционных роботов. Симулятор поддерживает физические движки (DART, Bullet через gz-physics), рендеринг через ogre2, обширную библиотеку моделей и сенсоров (камеры, лидары, IMU, GPS, контактные датчики), а также набор системных плагинов (gz-sim-physics-system, gz-sim-sensors-system и др.). Исторически существуют две ветви: устаревшая Gazebo Classic (gazebo1 – gazebo11), официально прекратившая поддержку в январе 2025 года, и новая Gazebo (релизы Citadel, Edifice, Fortress, Garden, Harmonic, Ionic, Jetty). Для дисциплины используется Gazebo Harmonic – долгосрочный (LTS) релиз с поддержкой до сентября 2028 года, штатно работающий в Ubuntu 24.04 и официально рекомендованный командой ROS 2 Jazzy.

2. Архитектура связки ROS 2 + Gazebo Harmonic. ROS 2 (Robot Operating System версии 2) – это middleware-фреймворк для построения распределённых робототехнических приложений на базе DDS. Gazebo Harmonic – самостоятельное приложение со своим транспортом gz-transport; связь с ROS 2 выполняется через пакет ros_gz, ключевыми компонентами

которого являются ros_gz_sim (launch-обёртки для gz sim), ros_gz_bridge (узел parameter_bridge, переводящий сообщения между gz.msgs и соответствующими ROS-типами по YAML-конфигурации) и ros_gz_image (специализированный мост для изображений). Типичная конфигурация запуска включает четыре процесса: gz sim -s – серверная часть симулятора без графики, gz sim -g – графический клиент GUI, parameter_bridge – мост сообщений, robot_state_publisher – узел, публикующий преобразования TF из URDF-модели.

3. Платформа TurtleBot3. TurtleBot3 – это серия дифференциальных двухколёсных мобильных роботов, разрабатываемая компанией ROBOTIS совместно с Open Robotics. В серию входят три модели, сводка по которым приведена в таблице 1.

Таблица 1 – Сравнение моделей серии TurtleBot3

4. ROS-топики, публикуемые моделью робота. После запуска симуляции в Gazebo и поднятия системных плагинов gz::sim::systems::DiffDrive, gz::sim::systems::JointStatePublisher и gz::sim::systems::Sensors симулятор начинает публиковать данные во внутренние gz-топики, а узел parameter_bridge зеркалит их в ROS 2. Перечень ключевых топиков приведён в таблице 2.

Таблица 2 – Стандартные ROS-топики модели TurtleBot3 в Gazebo Harmonic

5. Управление через teleop. Узел teleop_keyboard из пакета turtlebot3_teleop публикует сообщения geometry_msgs/TwistStamped в топик /cmd_vel в ответ на нажатия клавиш; мост ros_gz_bridge переводит их в gz.msgs.Twist, который потребляет плагин gz::sim::systems::DiffDrive. Конвенция управления TurtleBot3 совпадает с принятой в ROS:

положительная linear.x – движение вперёд, положительная angular.z – поворот против часовой стрелки. Максимальные скорости для модели Burger составляют 0.22 м/с по линейной и 2.84 рад/с по угловой компоненте; на уровень управления значения подаются дискретными приращениями по 0.01 м/с и 0.1 рад/с (клавиши w / x / a / d, s – полная остановка).

6. Графический интерфейс gz sim -g. Основное окно симулятора содержит трёхмерную сцену, левую панель Entity Tree, правую панель Component Inspector и нижнюю строку World Control. Кнопки строки World Control обеспечивают паузу симуляции (Space), пошаговое продвижение, сброс времени и моделей. Меню три точки в правом верхнем углу позволяет добавить дополнительные Insertable Plugins: Transform Control, Component Inspector, Visualize Lidar, Plotting и др. Прокрутка колёсика мыши – зум; средняя кнопка – вращение камеры; правая – перемещение. Дополнительные модели вставляются через панель Resource Spawner; источники моделей задаются переменной окружения GZ_SIM_RESOURCE_PATH и берутся из локальных каталогов либо из онлайн-репозитория Fuel (app.gazebosim.org).

Пример

Эталонный сценарий рассчитан на чистую установку Ubuntu 24.04 LTS (кодовое имя noble). Все команды выполняются от обычного пользователя; административные команды используют sudo. Пример разбит на семь шагов: подготовка репозитория ROS 2, установка ROS 2 Jazzy, установка Gazebo Harmonic, установка пакетов TurtleBot3 и моста ros_gz, запуск симуляции, изучение топиков, клавиатурный teleop.

Шаг 1. Подключение репозитория ROS 2. С 2024 года Open Robotics распространяет конфигурацию apt-репозитория в виде deb-пакета ros2-apt-source, который сам прописывает ключ и источник. Команды приведены в листинге 1.

Листинг 1 – Подключение apt-репозитория ROS 2 через ros2-apt-source

```
sudo apt update && sudo apt install -y curl
export ROS_APT_SOURCE_VERSION=$(curl -s \
  https://api.github.com/repos/ros-infrastructure/\
ros-apt-source/releases/latest \
  | grep -F 'tag_name' | awk -F'"' '{print $4}')
curl -L -o /tmp/ros2-apt-source.deb \
  "https://github.com/ros-infrastructure/ros-apt-source/\
releases/download/${ROS_APT_SOURCE_VERSION}/\
ros2-apt-source_${ROS_APT_SOURCE_VERSION}.\
$(. /etc/os-release && echo ${UBUNTU_CODENAME})_all.deb"
sudo dpkg -i /tmp/ros2-apt-source.deb
sudo apt update
```

Шаг 2. Установка ROS 2 Jazzy Jalisco и сопутствующих утилит. Берётся профиль desktop, содержащий RViz2, демонстрационные пакеты и инструменты разработчика.

Листинг 2 – Установка пакетов ROS 2 Jazzy

```
sudo apt install -y ros-jazzy-desktop \
  ros-dev-tools python3-colcon-common-extensions \
  python3-rosdep python3-argcomplete
sudo rosdep init || true
rosdep update
echo 'source /opt/ros/jazzy/setup.bash' >> ~/.bashrc
source ~/.bashrc
```

Шаг 3. Установка Gazebo Harmonic. Метапакет gz-harmonic ставится из официального репозитория Open Source Robotics Foundation; в стандартных репозиториях Ubuntu 24.04 идёт более старый Garden, поэтому используется внешний источник packages.osrfoundation.org.

Листинг 3 – Установка Gazebo Harmonic

```
sudo apt install -y lsb-release gnupg
sudo curl https://packages.osrfoundation.org/gazebo.gpg \
  --output /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg] \
https://packages.osrfoundation.org/gazebo/ubuntu-stable \
$(lsb_release -cs) main" | sudo tee \
  /etc/apt/sources.list.d/gazebo-stable.list > /dev/null
sudo apt update
```

```
sudo apt install -y gz-harmonic
gz sim --versions
```

Ожидаемый отклик последней команды – строка вида «8.x.x», подтверждающая корректную установку Harmonic (gz-sim 8 входит в состав Harmonic).

Шаг 4. Установка пакетов TurtleBot3 и моста ros_gz. Пакеты turtlebot3, turtlebot3_msgs, turtlebot3_simulations, turtlebot3_teleop загружаются из бинарного репозитория ROS 2 Jazzy; одновременно ставится ros_gz_bridge / ros_gz_sim / ros_gz_image для интеграции с Harmonic. В ~/.bashrc прописывается переменная TURTLEBOT3_MODEL и путь GZ_SIM_RESOURCE_PATH.

Листинг 4 – Установка пакетов TurtleBot3 и моста ros_gz

```
sudo apt install -y ros-jazzy-turtlebot3 \
  ros-jazzy-turtlebot3-msgs \
  ros-jazzy-turtlebot3-simulations \
  ros-jazzy-turtlebot3-gazebo \
  ros-jazzy-turtlebot3-teleop \
  ros-jazzy-ros-gz
echo 'export TURTLEBOT3_MODEL=burger' >> ~/.bashrc
echo 'export GZ_SIM_RESOURCE_PATH=$GZ_SIM_RESOURCE_PATH:'\
'/opt/ros/jazzy/share/turtlebot3_gazebo/models' >> ~/.bashrc
source ~/.bashrc
```

Шаг 5. Запуск симуляции в пустом мире. Launch-файл empty_world.launch.py поднимает gz sim -s (сервер) и gz sim -g (клиент) через ros_gz_sim/gz_sim.launch.py, спавнит модель TurtleBot3 в начало координат и стартует robot_state_publisher вместе с parameter_bridge.

Листинг 5 – Запуск TurtleBot3 в пустом мире Gazebo Harmonic

```
ros2 launch turtlebot3_gazebo empty_world.launch.py
```

После запуска открывается окно Gazebo Sim; в центре сцены – модель робота. Симуляция стартует автоматически (флаг -r в gz_args); пауза/возобновление – клавиша Space или кнопка Play / Pause в нижней

панели World Control. Для добавления препятствия используется панель Resource Spawner: выбирается, например, модель Construction Cone и перетаскивается в нескольких метрах от робота.

Шаг 6. Изучение списка ROS-топиков. В новом терминале (с обновлённым окружением ROS 2) выполняется опрос графа узлов и вывод одного сообщения /scan. Параллельно через gz topic можно просмотреть внутренние gz-топики симулятора.

Листинг 6 – Перечень опубликованных топиков и одно сообщение лидара

```
ros2 topic list
ros2 topic info /scan
ros2 topic echo --once /scan
ros2 topic hz /odom
gz topic -l
```

Ожидаемый фрагмент вывода команды ros2 topic list приведён в листинге 7.

Листинг 7 – Фрагмент вывода ros2 topic list для TurtleBot3 в Gazebo Harmonic

```
/clock
/cmd_vel
/imu
/joint_states
/odom
/parameter_events
/robot_description
/rosout
/scan
/tf
/tf_static
```

Шаг 7. Управление роботом через teleop. В отдельном терминале запускается клавиатурный узел из пакета turtlebot3_teleop; в его окне нажатие клавиш w / x / a / d увеличивает или уменьшает линейную и угловую скорость, s – полная остановка.

Листинг 8 – Запуск клавиатурного teleop-узла TurtleBot3

```
ros2 run turtlebot3_teleop teleop_keyboard
```

В третьем терминале можно подписаться на /cmd_vel и убедиться, что сообщения geometry_msgs/TwistStamped действительно публикуются с ожидаемыми значениями twist.linear.x и twist.angular.z. Робот должен перемещаться по сцене Gazebo в соответствии с введёнными командами; после прохождения по любой замкнутой траектории следует вернуть его в исходную точку и остановить симуляцию по Ctrl+C в каждом из терминалов.

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	**Содержание индивидуального задания**
1	Модель Burger, мир empty_world. Через ros2 topic hz измерить фактическую частоту публикации /scan, /odom и /imu; объяснить, почему они отличаются от значений update_rate в SDF. Через gz topic -e /scan записать пять последовательных сообщений во внутреннем gz-транспорте; сверить структуру с ROS-сообщением sensor_msgs/msg/LaserScan. Сделать вывод о роли parameter_bridge в задержке доставки данных от симулятора до ROS-подписчика.

2	Модель Waffle, мир turtlebot3_world. Через ros2 topic echo --once /scan измерить число лучей в сообщении и угловой шаг между лучами; сверить с конфигурацией лидара LDS-02 в SDF-модели turtlebot3_common. Изменить параметр update_rate сенсора hls_lfcd_lds с 5 до 10 в локальной копии SDF; пересобрать рабочее пространство и проверить, что /scan теперь публикуется чаще. Оценить, как увеличение частоты сказывается на real-time factor (поле в правом нижнем углу gz sim).
3	Модель Waffle Pi, мир turtlebot3_house. Через ros2 topic echo --no-arr /camera/image_raw определить разрешение и формат пикселей; найти соответствующие поля width / height / encoding в сообщении. Запустить отдельно image_bridge из пакета ros_gz_image для топика /camera/image_raw; объяснить, почему он работает эффективнее, чем generic parameter_bridge. Измерить пропускную способность через ros2 topic bw /camera/image_raw.
4	Модель Burger, мир empty_world. Через панель Resource Spawner поставить три цилиндра на расстоянии 0.5, 1.0 и 2.0 м перед роботом; снять расстояния по соответствующим лучам /scan и сверить с известными расстояниями. Подсчитать индексы массива ranges, в которые попадают эти цилиндры, через формулу i = (angle − angle_min) / angle_increment. Объяснить расхождение между фактической и измеренной дистанцией исходя из конфигурации шума gaussian в SDF лидара.
5	Модель Burger, мир turtlebot3_world. Через teleop провезти робота вдоль одной из стен по прямой 2 м; сравнить смещение по /odom с фактическим в Gazebo (через gz model -m burger -p). Оценить накопленную ошибку одометрии в процентах. Повторить эксперимент при выключенной коррекции через TF и объяснить,

	как меняется поведение rviz2.
6	Модель Waffle, мир empty_world. Через ros2 topic pub --rate 10 /cmd_vel geometry_msgs/msg/TwistStamped подать постоянную команду linear.x = 0.1 м/с в течение 10 с; оценить пройденное расстояние по /odom. Сравнить с теоретическим расстоянием v · t = 1.0 м; объяснить разницу. Повторить с linear.x = 0.2 м/с и angular.z = 0.5 рад/с – построить траекторию по записи /odom.
7	Модель Burger, мир turtlebot3_house. Запустить ros2 bag record -a -s mcap на 30 с; вывести объём bag-каталога и список записанных топиков через ros2 bag info. Воспроизвести запись через ros2 bag play -r 2.0 (двойное ускорение) и подтвердить, что rviz2 отображает движение. Объяснить разницу между бэкендами sqlite3 и mcap для rosbag2.
8	Модель Waffle Pi, мир empty_world. Через ros2 topic echo /imu вывести десять сообщений; измерить фактическую частоту публикации через ros2 topic hz /imu (ожидаемая 200 Гц). Определить ориентацию робота в покое по полю orientation (кватернион); пересчитать в углы Эйлера. Подать команду angular.z = 1.0 рад/с в течение 3 с; снять интеграл по angular_velocity.z и сравнить с фактическим поворотом yaw.
9	Модель Burger, мир turtlebot3_world. Через teleop выполнить разворот на 360° по часовой стрелке; проверить согласованность изменения yaw в /odom. Записать траекторию через ros2 bag record /odom; построить график yaw(t) средствами matplotlib после ros2 bag info. Объяснить, в какой момент возникает скачок yaw на ±π и как его сгладить.
10	Модель Burger, мир empty_world. Через gz topic -l перечислить все внутренние gz-топики симулятора; найти топики, не

	пробрасываемые в ROS через bridge. Через gz service -l вывести список сервисов Gazebo; вызвать /world/default/control с pause: true и проверить эффект. Объяснить разницу между топиками и сервисами в gz-transport.
11	Модель Waffle, мир turtlebot3_house. Через Resource Spawner поставить модель Bookshelf из локальной библиотеки; замерить дальность до неё одновременно по /scan и по линейке (плагин Distance Measurement в gz sim). Сверить два значения с точностью 0.05 м. Объяснить, как параметр range_resolution сенсора влияет на точность.
12	Модель Burger, мир turtlebot3_world. Запустить две копии rviz2: одну с Fixed Frame = odom, другую с base_link; объяснить разницу визуализации /scan. Через rqt_tf_tree вывести дерево tf2; убедиться, что цепочка map → odom → base_footprint → base_link замкнута. Назвать, какой узел публикует каждый из перечисленных переходов.
13	Модель Waffle, мир empty_world. Через gz sim --verbose 4 запустить симулятор; снять реальный коэффициент real-time factor под нагрузкой (одновременно запустить rviz2 и rqt). Сократить max_step_size с 0.001 до 0.0005 в файле мира; пересчитать RTF. Объяснить связь между шагом физики, частотой публикации /clock и устойчивостью DDS-обмена.
14	Модель Burger, мир turtlebot3_house. С помощью ros2 node list определить полный список узлов; построить графовое представление через rqt_graph. Выделить на графе три области: узлы Gazebo-моста, узлы TurtleBot3, системные узлы ROS 2. Объяснить, почему parameter_bridge виден как один узел, хотя перенаправляет несколько топиков.

15	Модель Waffle Pi, мир turtlebot3_world. Через ros2 topic pub /cmd_vel geometry_msgs/msg/TwistStamped опубликовать команду, при которой робот движется по окружности радиуса 1.0 м (linear.x = 0.2, angular.z = 0.2); подтвердить /odom. Записать траекторию в rosbag2; построить (x, y)-график. Оценить, насколько фактический радиус отклоняется от заданного и почему.

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Что такое Gazebo и зачем он нужен для разработки автономных систем?
2. Чем Gazebo Harmonic отличается от Gazebo Classic?
3. Как запустить TurtleBot3 в Gazebo через launch-файл?
4. Какие топики ROS 2 публикует симуляция TurtleBot3?
5. Как управлять роботом через teleop_twist_keyboard?
6. Что такое ros_gz_bridge и зачем он нужен?
7. Как добавить объект в сцену через интерфейс Gazebo?
8. Что такое real-time factor (RTF) и как он влияет на симуляцию?

Лабораторная работа 2.

СОЗДАНИЕ URDF-МОДЕЛИ ПРОСТОГО РОБОТА (ДВА КОЛЕСА + КОРПУС)

Цель и содержание работы: освоить формат URDF (Unified Robot Description Format). Научиться описывать кинематическую структуру простого мобильного робота из корпуса и двух ведущих колёс, корректно задавать инерционные и коллизионные свойства каждого link, формировать дерево joints, проверять валидность модели утилитой check_urdf и визуализировать её в RViz2 с публикацией состояний суставов через joint_state_publisher_gui и robot_state_publisher (среда ROS 2 Jazzy на Ubuntu 24.04).

Теоретическое обоснование

1. Назначение формата URDF. URDF – это XML-формат описания кинематической схемы робота, утверждённый сообществом ROS и поддерживаемый всеми основными инструментами экосистемы: rviz2, robot_state_publisher, ros_gz_sim, MoveIt 2, Nav2. Файл описывает робот как ориентированный ациклический граф жёстких тел (link), соединённых сочленениями (joint). Структура графа – дерево с одним корневым звеном; кинематические циклы прямо не поддерживаются и обрабатываются через расширение URDF или формат SDF (Simulation Description Format), нативный для Gazebo Harmonic.

2. Элемент link. Корневой тег <link> описывает одно жёсткое тело и содержит до трёх вложенных секций: <visual> – геометрия для отрисовки в RViz и других визуализаторах; <collision> – геометрия для определения столкновений физическим движком; <inertial> – масса и тензор инерции, необходимые для интегрирования уравнений движения. Геометрия задаётся либо примитивами (<box>, <cylinder>, <sphere>), либо ссылкой на внешний

mesh-файл (<mesh filename="package://…/wheel.dae"/>). Каждая секция имеет независимое смещение <origin>, что позволяет разнести центр масс и центр визуализации.

3. Тензор инерции. Тензор инерции для тела массы m задаётся симметричной матрицей 3 × 3 и в URDF записывается шестью независимыми компонентами ixx, iyy, izz, ixy, ixz, iyz. Для однородных примитивов с осями симметрии вдоль X, Y, Z диагональные элементы вычисляются по аналитическим формулам. Для сплошного цилиндра с радиусом r и высотой h, осью симметрии вдоль Z, элементы вычисляются по формулам (1) и (2):

$$Ixx = Iyy = (1/12) \cdot m \cdot (3 \cdot r^2 + h^2) \qquad (1)$$

$$Izz = (1/2) \cdot m \cdot r^2 \qquad (2)$$

Для прямоугольного параллелепипеда со сторонами a × b × c (вдоль осей X, Y, Z) тензор инерции относительно центра масс вычисляется по формуле (3):

$$Ixx = (1/12) \cdot m \cdot (b^2+c^2);\ Iyy = (1/12) \cdot m \cdot (a^2+c^2);\ Izz = (1/12) \cdot m \cdot (a^2+b^2) \qquad (3)$$

4. Элемент joint. Сочленение описывает связь между двумя links: родителем (<parent link="…"/>) и ребёнком (<child link="…"/>). Атрибут type определяет кинематический тип сочленения; поддерживаемые в URDF типы перечислены в таблице 1.

Таблица 1 – Типы joint в URDF и их применение

5. Структура дерева URDF-модели представлена на рисунке 1.

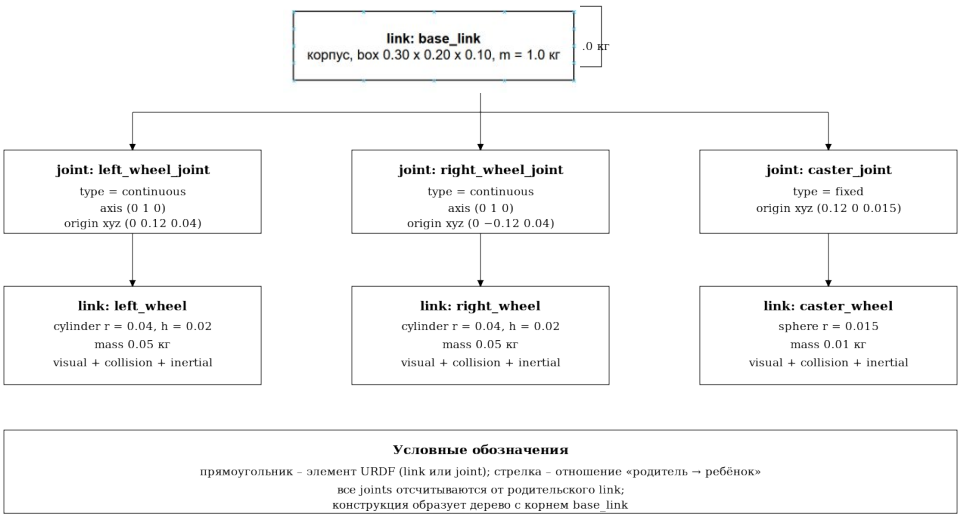

Рисунок 1 – Структура дерева URDF-модели двухколёсного робота

6. Утилита check_urdf. После записи XML-файла его обязательно проверяют утилитой check_urdf из пакета liburdfdom-tools. Утилита парсит модель и сообщает о найденных дефектах: недостающие link, несоответствие parent/child, рекурсия, отсутствие <inertial>, неположительная масса. Корректная модель печатается в виде иерархического дерева с указанием корневого link и списка детей.

7. Подсистема публикации состояния. Для визуализации URDF в RViz2 используются три узла. robot_state_publisher принимает содержимое URDF через параметр robot_description, подписывается на /joint_states (sensor_msgs/JointState) и при каждом сообщении вычисляет преобразования tf2 между всеми links, публикуя их в топик /tf. joint_state_publisher (или его GUI-версия joint_state_publisher_gui) генерирует фиктивные значения положений суставов: фиксированные нули либо подвижные слайдеры. Узел rviz2 отображает геометрию, опираясь на /tf и URDF.

Пример

Эталонная модель называется simple_robot и состоит из корпуса (box 0.30 × 0.20 × 0.10 м, масса 1.0 кг), двух ведущих колёс (cylinder r = 0.04 м, h =

0.02 м, масса 0.05 кг каждое) и одного переднего опорного шарика-каста (sphere r = 0.015 м, масса 0.01 кг). Дерево модели изображено на рисунке 1. Структура пакета и команды сборки приведены ниже.

Шаг 1. Создание ament-пакета с описанием робота. Пакет содержит каталог urdf для XML-файлов, каталог launch для launch-сценариев и каталог rviz для конфигурации визуализатора.

Листинг 1 – Создание пакета simple_robot_description

```
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_cmake \
  --license Apache-2.0 simple_robot_description
cd simple_robot_description
mkdir -p urdf launch rviz
```

Шаг 2. Файл simple_robot.urdf. Корневой link – base_link, две цилиндрические колёсные пары прикреплены через continuous-сочленения по оси Y; опорный шарик – через fixed. Все геометрии продублированы в visual и collision; для каждого link задан <inertial> с аналитически вычисленным тензором инерции.

Листинг 2 – URDF-описание двухколёсного робота simple_robot

```
<?xml version="1.0"?>
<robot name="simple_robot">


 <material name="grey">
  <color rgba="0.5 0.5 0.5 1.0"/>
 </material>
 <material name="black">
  <color rgba="0.1 0.1 0.1 1.0"/>
 </material>


 <link name="base_link">
  <visual>
   <origin xyz="0 0 0.05" rpy="0 0 0"/>
   <geometry><box size="0.30 0.20 0.10"/></geometry>
   <material name="grey"/>
  </visual>
  <collision>
   <origin xyz="0 0 0.05" rpy="0 0 0"/>
```

```
    <geometry><box size="0.30 0.20 0.10"/></geometry>
  </collision>
  <inertial>
    <origin xyz="0 0 0.05" rpy="0 0 0"/>
    <mass value="1.0"/>
    <inertia ixx="0.0042" iyy="0.0083" izz="0.0108"
          ixy="0" ixz="0" iyz="0"/>
  </inertial>
</link>


<link name="left_wheel">
  <visual>
    <origin xyz="0 0 0" rpy="1.5708 0 0"/>
    <geometry><cylinder radius="0.04" length="0.02"/></geometry>
    <material name="black"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="1.5708 0 0"/>
    <geometry><cylinder radius="0.04" length="0.02"/></geometry>
  </collision>
  <inertial>
    <mass value="0.05"/>
    <inertia ixx="2.4e-5" iyy="2.4e-5" izz="4.0e-5"
          ixy="0" ixz="0" iyz="0"/>
  </inertial>
</link>


<link name="right_wheel">
  <visual>
    <origin xyz="0 0 0" rpy="1.5708 0 0"/>
    <geometry><cylinder radius="0.04" length="0.02"/></geometry>
    <material name="black"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="1.5708 0 0"/>
    <geometry><cylinder radius="0.04" length="0.02"/></geometry>
  </collision>
  <inertial>
    <mass value="0.05"/>
    <inertia ixx="2.4e-5" iyy="2.4e-5" izz="4.0e-5"
          ixy="0" ixz="0" iyz="0"/>
  </inertial>
</link>


<link name="caster_wheel">
  <visual>
    <geometry><sphere radius="0.015"/></geometry>
    <material name="black"/>
  </visual>
```

```
  <collision>
    <geometry><sphere radius="0.015"/></geometry>
  </collision>
  <inertial>
    <mass value="0.01"/>
    <inertia ixx="9e-7" iyy="9e-7" izz="9e-7"
             ixy="0" ixz="0" iyz="0"/>
  </inertial>
</link>


<joint name="left_wheel_joint" type="continuous">
  <parent link="base_link"/>
  <child link="left_wheel"/>
  <origin xyz="0 0.12 0.04" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
</joint>


<joint name="right_wheel_joint" type="continuous">
  <parent link="base_link"/>
  <child link="right_wheel"/>
  <origin xyz="0 -0.12 0.04" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
</joint>


<joint name="caster_joint" type="fixed">
  <parent link="base_link"/>
  <child link="caster_wheel"/>
  <origin xyz="0.12 0 0.015" rpy="0 0 0"/>
</joint>


</robot>
```

Шаг 3. Проверка модели утилитой check_urdf. Утилита парсит файл и выводит дерево link/joint; любые ошибки описания будут распечатаны как сообщения parser error.

Листинг 3 – Запуск check_urdf на эталонной модели

```
check_urdf urdf/simple_robot.urdf
```

Ожидаемый вывод приведён в листинге 3.

Листинг 4 – Дерево модели simple_robot, напечатанное check_urdf

```
robot name is: simple_robot
---------- Successfully Parsed XML ---------------
root Link: base_link has 3 child(ren)
  child(1):  left_wheel
  child(2):  right_wheel
  child(3):  caster_wheel
```

Шаг 4. Launch-файл для визуализации в RViz2. Файл display.launch.py запускает три узла: robot_state_publisher с robot_description, joint_state_publisher_gui для слайдеров и rviz2 с заранее подготовленной конфигурацией.

Листинг 5 – Launch-файл display.launch.py для визуализации URDF в RViz2

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
  pkg = get_package_share_directory('simple_robot_description')
  urdf = os.path.join(pkg, 'urdf', 'simple_robot.urdf')
  rviz = os.path.join(pkg, 'rviz', 'display.rviz')
  with open(urdf, 'r') as fh:
    robot_desc = fh.read()
  return LaunchDescription([
    Node(
      package='robot_state_publisher',
      executable='robot_state_publisher',
      parameters=[{'robot_description': robot_desc}],
    ),
    Node(
      package='joint_state_publisher_gui',
      executable='joint_state_publisher_gui',
    ),
    Node(
      package='rviz2', executable='rviz2',
      arguments=['-d', rviz],
    ),
  ])
```

Шаг 5. Сборка пакета и запуск визуализации. Из корня рабочего пространства выполняется colcon build и source install/setup.bash, после чего запускается launch-файл.

Листинг 6 – Сборка пакета и запуск display.launch.py

```
cd ~/ros2_ws
colcon build --packages-select simple_robot_description
source install/setup.bash
ros2 launch simple_robot_description display.launch.py
```

После запуска открывается окно RViz2; в дереве Displays выставляется Fixed Frame = base_link, добавляются RobotModel (параметр Description Topic = /robot_description) и TF. В отдельном окне joint_state_publisher_gui слайдеры вращают колёса; соответствующие links поворачиваются в трёхмерной сцене, что подтверждает корректную работу пары robot_state_publisher + joint_state_publisher.

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	**Содержание индивидуального задания**
1	Изменить корпус на цилиндр (radius 0.10, length 0.20); пересчитать тензор инерции по формулам (1) и (2), указать новые

	значения ixx, iyy, izz. Сохранить файл как simple_robot_v1.urdf и проверить через check_urdf. Визуально подтвердить изменение формы корпуса в RViz2.
2	Добавить четыре опорных каста по углам корпуса вместо одного переднего; каждый – sphere r 0.012, fixed-joint. Скорректировать высоту корпуса так, чтобы он остался параллельно полу при равной длине касторов. Объяснить, как изменилась устойчивость и почему дифференциальная база с четырьмя касторами становится недетерминированной по повороту.
3	Увеличить радиус колёс до 0.06 м; поднять корпус соответственно так, чтобы дно корпуса оставалось на высоте 0.03 м над полом. Скорректировать origin всех трёх joints (left, right, caster). Пересчитать тензор инерции колёс под новый радиус и подтвердить check_urdf.
4	Добавить пятый link – mast (cylinder r 0.01, h 0.30) вертикально на корпусе через fixed joint, переименовать робот в watchtower_bot. Задать массу мачты 0.05 кг и вычислить тензор инерции по формулам (1)–(2). Проверить, что центр масс корпуса по-прежнему находится внутри его геометрии (через ros2 run urdf_launch_python display.py с включённой опцией Show Inertia).
5	Реализовать поворотную башню: добавить link tower с revolute joint, осью Z и limits ±π/2, effort 1.0 Н·м, velocity 1.5 рад/с. Подтвердить в joint_state_publisher_gui наличие слайдера tower и его пределы. Объяснить разницу между revolute и continuous для поворотной башни.
6	Описать робот в xacro: вынести параметры (радиус колеса, ширина базы, масса корпуса) в <xacro:property>; сгенерировать URDF через ros2 run xacro xacro. Реализовать макрос

	<xacro:macro name="wheel"> и применить его к обоим колёсам. Сравнить размер исходного XML и сгенерированного URDF в байтах.
7	Заменить визуальную геометрию колеса на STL-mesh из package://simple_robot_description/meshes/wheel.stl; оставить collision как cylinder (так быстрее работает физика). Подготовить простой mesh-файл (например, экспортированный из FreeCAD) и положить в каталог meshes. Объяснить, почему collision-меши обычно делают грубее visual-мешей.
8	Сделать асимметричную базу: правое колесо на 0.05 м впереди левого по оси X. Пройти check_urdf и убедиться, что предупреждений нет. Объяснить, как такое смещение повлияет на одометрию и точку вращения робота при чистом развороте на месте.
9	Поднять массу корпуса до 5 кг; пересчитать тензор инерции по формуле (3) для тех же размеров. Проверить, что check_urdf не выдаёт предупреждений о слишком малых элементах тензора. Объяснить, почему масса базы существенно влияет на устойчивость оценки RTF в Gazebo.
10	Добавить два дополнительных пассивных колеса (revolute, axis xyz 0 1 0, limit ±10π) с осями, перпендикулярными ведущим – четырёхколёсная версия. Подобрать положение пассивных колёс так, чтобы машина оставалась на четырёх точках опоры. Объяснить, как наличие пассивных колёс изменяет дифференциальную кинематику и тип робота с двухколёсного на квази-omnidirectional.
11	Реализовать манипуляторное звено: link arm + revolute joint с limit (–π/4, π/4), effort 2 Н·м, velocity 1 рад/с, осью Y; разместить на

	верхней грани корпуса. Подтвердить появление третьего слайдера в joint_state_publisher_gui. Указать, какой joint станет «третьим звеном» при добавлении схвата на конец arm.
12	Сменить материалы: красный корпус, белые колёса, чёрный каст; сохранить цвета в едином блоке <material> в начале файла. Применить материалы только к visual (не к collision). Объяснить, почему дублировать <material> внутри <link> не рекомендуется.
13	Описать робот с одним приводным колесом и двумя пассивными роликами; разместить приводное колесо в центре корпуса, пассивные – по углам передней грани. Подтвердить check_urdf. Объяснить, почему такая база является неголономной и какие движения остаются невозможны без поворота приводного колеса.
14	Добавить link bumper (тонкий box 0.31 × 0.02 × 0.05) перед корпусом через fixed joint, центр в xyz 0.16 0 0.05. Назначить bumper материал «красный»; пройти check_urdf. Проверить ориентацию через urdf_to_graphiz и приложить полученное дерево к отчёту.
15	Заменить continuous joints на revolute с limit (-31.4159, 31.4159) и effort 1.0 Н·м, velocity 5 рад/с. Проследить разницу в выводе check_urdf по сравнению с continuous. Объяснить, к каким артефактам приведёт превышение лимита при длительном движении (revolute «упрётся» в стенку лимита).

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Что такое URDF и для чего он используется?
2. Что описывают теги <link> и <joint> в URDF?
3. Какие типы joint существуют в URDF?
4. Как задать тензор инерции для цилиндра?
5. Как визуализировать URDF в RViz без Gazebo?
6. Что такое robot_state_publisher и joint_state_publisher?
7. Как проверить корректность URDF командой check_urdf?
8. Как добавить цвет к визуальному элементу в URDF?

Лабораторная работа 3.

ДОБАВЛЕНИЕ СЕНСОРОВ В URDF: ЛИДАР И КАМЕРА

Цель и содержание работы: научиться расширять описание робота (SDF либо URDF с тегом <gazebo>) сенсорными блоками <sensor type="gpu_lidar"> и <sensor type="camera">, корректно настраивать параметры сценирования лидара (число лучей, угловой сектор, диапазон дальностей, шум) и оптические параметры камеры (разрешение, поле зрения, частота кадров, плоскости отсечения), активировать системные плагины gz::sim::systems::Sensors в файле мира, прокидывать сенсорные данные из gz-транспорта в ROS 2 через ros_gz_bridge и ros_gz_image и убеждаться в работе сенсоров через ROS-топики /scan и /camera/image_raw и их визуализацию в RViz2.

Теоретическое обоснование

1. Расширение URDF тегом <gazebo>. Базовый URDF описывает кинематику и инерцию, но не содержит ни сенсоров, ни приводов, ни параметров материалов поверхности, поскольку эти понятия выходят за рамки стандарта. Для интеграции с симулятором Gazebo URDF расширяется тегом <gazebo>, который имеет необязательный атрибут reference, указывающий, к какому link относится содержимое. Внутри тега располагают <sensor>, <plugin>, а также переопределения коэффициентов трения (mu1, mu2), упругости (kp, kd) и собственного цвета поверхности. При разборе URDF пакетом robot_state_publisher теги <gazebo> игнорируются; их разбирает только sdformat внутри gz sim. В Gazebo Harmonic рекомендуемой практикой остаётся либо писать модель сразу в SDF (нативный формат симулятора), либо использовать xacro + URDF с вкладыванием SDF-сниппетов в <gazebo>.

2. Лидар. Тег <sensor type="gpu_lidar"> описывает лидар как набор виртуальных лучей, испускаемых одновременно из заданного link. В Gazebo Harmonic тип "gpu_lidar" пришёл на смену Classic-варианту "ray" / "gpu_ray" и обязательно опирается на системный плагин gz::sim::systems::Sensors, подключаемый в файле мира. Базовые параметры лидара перечислены в таблице 1.

Таблица 1 – Параметры тега <sensor type="gpu_lidar"> в Gazebo Harmonic

3. Модель шума лидара. Шум дальности предполагается аддитивным гауссовым с нулевым средним и заданным стандартным отклонением σ, поэтому измеренная дальность ri описывается формулой (1):

$$ri = ri,true + \eta,\ \eta \sim N(0, \sigma^2) \qquad (1)$$

В реальных лидарах σ слабо зависит от дальности; в Gazebo Harmonic значение постоянно и берётся из тега <noise><stddev>. Лучи, для которых истинная дальность ri,true превышает range/max, возвращают значение +∞, что должны корректно обрабатывать узлы-подписчики (Nav2 и rviz2 такие лучи просто игнорируют).

4. Камера. Тег <sensor type="camera"> моделирует обычную RGB-камеру через ray-cast рендеринг (Harmonic использует движок ogre2). Поле зрения по горизонтали horizontal_fov задаётся в радианах. Высота поля зрения пересчитывается автоматически по соотношению сторон image/width и image/height. Формат изображения задаётся строкой R8G8B8 (8 бит на канал), L8 (градации серого), R16G16B16 (для HDR). Плоскости отсечения clip/near и clip/far ограничивают глубину сцены: слишком близкая near приводит к z-fighting, слишком далёкая far – к потере точности глубины. Параметры камеры перечислены в таблице 2.

Таблица 2 – Параметры тега <sensor type="camera">

5. Связь сенсорной системы с ROS-графом представлена на рисунке 1: каждое описание сенсора в SDF/URDF разворачивается в Gazebo Harmonic в виртуальное устройство, gz-топик которого затем зеркалится в ROS 2 узлом parameter_bridge из пакета ros_gz_bridge.

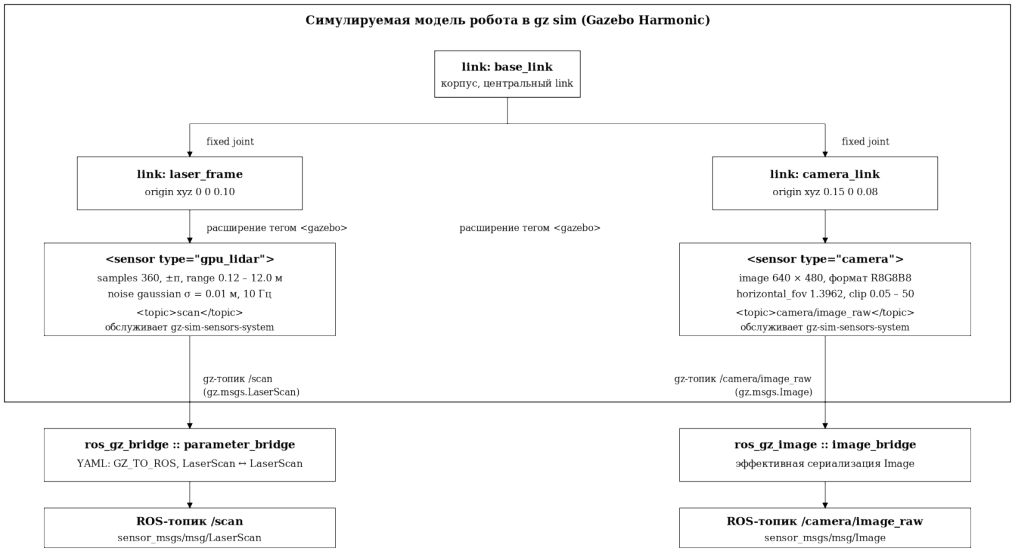

Рисунок 1 – Связь сенсорных тегов SDF/URDF с ROS-топиками через ros_gz_bridge

6. Публикация сенсорных данных в ROS 2. В Gazebo Harmonic загрузка отдельных публикующих плагинов в каждый <sensor> больше не требуется: системный плагин gz::sim::systems::Sensors (filename = gz-sim-sensors-system), включаемый один раз в файле мира, автоматически создаёт gz-топики /scan, /imu, /camera/image_raw и т. д. Зеркалирование в ROS 2 выполняется узлом ros_gz_bridge::parameter_bridge, конфигурируемым YAML-файлом, в котором для каждой пары gz-топик ↔ ROS-топик указываются gz_type_name (например, gz.msgs.LaserScan), ros_type_name (sensor_msgs/msg/LaserScan) и направление (GZ_TO_ROS, ROS_TO_GZ, BIDIRECTIONAL). Для изображений рекомендуется специализированный image_bridge из пакета ros_gz_image – он использует более эффективную сериализацию.

7. Визуализация в RViz2. Для лидара в RViz2 добавляется дисплей LaserScan с подпиской на /scan; в качестве Fixed Frame следует указать одометрический кадр odom или базовый base_link, а в поле Topic – /scan. Для камеры добавляется дисплей Image либо Camera с топиком /camera/image_raw; второй вариант поверх изображения накладывает 3D-объекты из сцены.

Пример

За основу взята модель simple_robot из предыдущей лабораторной работы. К ней добавлены два link, два fixed-joint и две <gazebo>-секции с сенсорами. Файл сохраняется под именем simple_robot_sensors.urdf. Параллельно в файле мира мир world/empty.sdf активируется плагин gz::sim::systems::Sensors.

Шаг 1. Добавление новых links и joints. Лидар крепится в центре корпуса на высоте 0.10 м, камера – перед корпусом на высоте 0.08 м. Обе геометрии – небольшие boxes / cylinders с заглушечной массой и минимальной инерцией.

Листинг 1 – Дополнительные links и joints для лидара и камеры

```
<link name="laser_frame">
  <visual>
    <geometry><cylinder radius="0.025" length="0.04"/></geometry>
    <material name="black"/>
  </visual>
  <collision>
    <geometry><cylinder radius="0.025" length="0.04"/></geometry>
  </collision>
  <inertial>
    <mass value="0.05"/>
    <inertia ixx="1e-5" iyy="1e-5" izz="1e-5"
          ixy="0" ixz="0" iyz="0"/>
  </inertial>
</link>


<link name="camera_link">
  <visual>
```

```
   <geometry><box size="0.03 0.05 0.03"/></geometry>
   <material name="grey"/>
  </visual>
  <collision>
   <geometry><box size="0.03 0.05 0.03"/></geometry>
  </collision>
  <inertial>
   <mass value="0.02"/>
   <inertia ixx="1e-6" iyy="1e-6" izz="1e-6"
         ixy="0" ixz="0" iyz="0"/>
  </inertial>
 </link>


 <joint name="laser_joint" type="fixed">
  <parent link="base_link"/>
  <child link="laser_frame"/>
  <origin xyz="0 0 0.10" rpy="0 0 0"/>
 </joint>


 <joint name="camera_joint" type="fixed">
  <parent link="base_link"/>
  <child link="camera_link"/>
  <origin xyz="0.15 0 0.08" rpy="0 0 0"/>
 </joint>
```

Шаг 2. Секция <gazebo> для лидара. Сенсор имеет 360 лучей, круговой обзор от $-\pi$ до $+\pi$, диапазон 0.12 – 12.0 м, гауссов шум со стандартным отклонением 0.01 м, частоту обновления 10 Гц. Тег <topic>scan</topic> указывает имя выходного gz-топика, который позже зеркалится в ROS 2 как /scan.

Листинг 2 – Конфигурация лидара через <sensor type="gpu_lidar">

```
<gazebo reference="laser_frame">
  <sensor name="lds_lidar" type="gpu_lidar">
   <pose>0 0 0 0 0 0</pose>
   <visualize>true</visualize>
   <always_on>true</always_on>
   <update_rate>10</update_rate>
   <topic>scan</topic>
   <gz_frame_id>laser_frame</gz_frame_id>
   <lidar>
    <scan>
     <horizontal>
      <samples>360</samples>
```

```
        <resolution>1</resolution>
        <min_angle>-3.14159</min_angle>
        <max_angle>3.14159</max_angle>
      </horizontal>
    </scan>
    <range>
      <min>0.12</min>
      <max>12.0</max>
      <resolution>0.01</resolution>
    </range>
    <noise>
      <type>gaussian</type>
      <mean>0.0</mean>
      <stddev>0.01</stddev>
    </noise>
  </lidar>
 </sensor>
</gazebo>
```

Шаг 3. Секция <gazebo> для камеры. Камера имеет разрешение 640 × 480, поле зрения 80° (1.3962 рад), плоскости отсечения 0.05 – 50.0 м, частоту кадров 30 Гц. Выходной gz-топик camera/image_raw зеркалится в ROS 2 узлом image_bridge.

Листинг 3 – Конфигурация камеры через <sensor type="camera">

```
<gazebo reference="camera_link">
  <sensor name="front_cam" type="camera">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <always_on>true</always_on>
    <update_rate>30.0</update_rate>
    <topic>camera/image_raw</topic>
    <gz_frame_id>camera_link</gz_frame_id>
    <camera name="head">
      <horizontal_fov>1.3962</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.05</near>
        <far>50.0</far>
      </clip>
      <noise>
        <type>gaussian</type>
```

```
        <mean>0.0</mean>
        <stddev>0.007</stddev>
       </noise>
      </camera>
     </sensor>
    </gazebo>
```

Шаг 4. Активация системного плагина Sensors в файле мира. В Harmonic публикацию данных лидара и камеры обеспечивает плагин gz::sim::systems::Sensors, который подключается в SDF мира один раз. Файл worlds/empty_with_sensors.sdf приведён в листинге 4.

Листинг 4 – Подключение системных плагинов в SDF-файле мира

```
<sdf version="1.10">
 <world name="default">
  <plugin filename="gz-sim-physics-system"
        name="gz::sim::systems::Physics"/>
  <plugin filename="gz-sim-scene-broadcaster-system"
        name="gz::sim::systems::SceneBroadcaster"/>
  <plugin filename="gz-sim-user-commands-system"
        name="gz::sim::systems::UserCommands"/>
  <plugin filename="gz-sim-sensors-system"
        name="gz::sim::systems::Sensors">
   <render_engine>ogre2</render_engine>
  </plugin>
  <light name="sun" type="directional">
   <pose>0 0 10 0 0 0</pose>
   <diffuse>1 1 1 1</diffuse>
  </light>
  <include>
   <uri>model://ground_plane</uri>
  </include>
 </world>
</sdf>
```

Шаг 5. Конфигурация моста ros_gz_bridge. YAML-файл config/sensors_bridge.yaml сопоставляет gz-топики симулятора ROS-топикам и типам сообщений; узел parameter_bridge запускается с параметром config_file.

Листинг 5 – Файл sensors_bridge.yaml для ros_gz_bridge

```
- ros_topic_name: "scan"
```

```
  gz_topic_name: "scan"
  ros_type_name: "sensor_msgs/msg/LaserScan"
  gz_type_name: "gz.msgs.LaserScan"
  direction: GZ_TO_ROS


- ros_topic_name: "clock"
  gz_topic_name: "clock"
  ros_type_name: "rosgraph_msgs/msg/Clock"
  gz_type_name: "gz.msgs.Clock"
  direction: GZ_TO_ROS
```

Шаг 6. Launch-файл и запуск. Файл sensors.launch.py поднимает gz sim с миром empty_with_sensors.sdf через ros_gz_sim, спавнит URDF в мир, запускает robot_state_publisher, parameter_bridge и image_bridge.

Листинг 6 – Сборка пакета и запуск симуляции с сенсорами

```
cd ~/ros2_ws && colcon build \
  --packages-select simple_robot_description
source install/setup.bash
ros2 launch simple_robot_description sensors.launch.py
# в новом терминале:
ros2 topic list | grep -E 'scan|camera'
ros2 topic hz /scan
ros2 topic hz /camera/image_raw
```

Ожидаемый отклик ros2 topic list содержит как минимум четыре новых топика и приведён в листинге 7.

Листинг 7 – Фрагмент вывода ros2 topic list после добавления сенсоров

```
/camera/camera_info
/camera/image_raw
/clock
/scan
```

Шаг 7. Визуализация в RViz2. В отдельном терминале запускается RViz2; в дереве Displays добавляются RobotModel (Description Topic /robot_description), TF, LaserScan (Topic /scan), Image (Topic /camera/image_raw). Fixed Frame – odom либо base_link. Перед роботом в

Gazebo через панель Resource Spawner ставится кубик; соответствующая дуга должна появиться в окне LaserScan и стена куба – в окне Image.

Листинг 8 – Запуск RViz2 с готовой конфигурацией

```
ros2 run rviz2 rviz2 -d \
  $(ros2 pkg prefix simple_robot_description)\
/share/simple_robot_description/rviz/sensors.rviz
```

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Лидар gpu_lidar: 180 лучей, сектор ±π/2, диапазон 0.10 – 8.0 м, σ = 0.02 м. Сравнить плотность точек в окне LaserScan rviz2 с эталонной конфигурацией (360 лучей, ±π). Объяснить, почему angle_increment изменился в 4 раза при том же числе лучей.
2	Лидар: 720 лучей, сектор ±π, частота 40 Гц. Оценить нагрузку на CPU и GPU через gz sim --verbose 4 и системный монитор; зафиксировать падение real-time factor. Сравнить с конфигурацией 360 лучей / 10 Гц по числу точек в секунду.
3	Камера: 1280 × 720, fov 1.05 рад, формат L8 (градации серого). Подтвердить разрешение через ros2 topic echo --no-arr

	/camera/image_raw (поля width / height / encoding). Объяснить, почему L8 даёт в три раза меньший трафик, чем R8G8B8.
4	Лидар: сектор ±π/4, дальность до 30 м, σ = 0.05 м. Поставить через Resource Spawner стену на 25 м – проверить, видна ли она. Объяснить влияние шума на воспроизводимость карты, строимой slam_toolbox: насколько ухудшится качество при σ = 0.05 vs 0.01.
5	Камера: 320 × 240, частота 60 Гц, fov 0.8 рад. Замерить реальный rate через ros2 topic hz /camera/image_raw; сравнить с заявленным. Объяснить, при каких условиях фактическая частота отстаёт от update_rate (загрузка GPU, расходы ogre2).
6	Два лидара gpu_lidar – передний и задний (по 180 лучей каждый), сектор ±π/2; каждый со своим <topic>: scan_front и scan_back. Через ros_gz_bridge пробросить оба топика в ROS 2. Написать узел-агрегатор на rclpy, объединяющий два LaserScan-сообщения в одно с сектором ±π.
7	Лидар gpu_lidar с 1080 лучами и частотой 20 Гц. Сравнить производительность (RTF, кадры/с) с эталонной конфигурацией 360 / 10 на одном и том же компьютере. Сделать вывод о масштабируемости gpu_lidar в Harmonic.
8	Камера-стерео: два <sensor type="camera"> на одном link, разнесённые по оси Y на 0.06 м. Выходные gz-топики stereo/left и stereo/right пробросить в ROS 2 через ros_gz_image (image_bridge). В rviz2 одновременно отобразить оба Image-дисплея; визуально оценить горизонтальный диспаритет на ближнем объекте.
9	Лидар с range/min = 1.0 м (увеличенная мёртвая зона). Поставить препятствие на 0.5 м – подтвердить, что луч возвращает +∞ для этой ячейки. Объяснить, как это повлияет на навигацию рядом с

	препятствиями и почему в Nav2 нужен расширенный inflation_radius.
10	Заменить камеру на depth-сенсор: <sensor type="depth_camera">, опубликовать gz-топик camera/depth_image. Через ros_gz_bridge зеркалить как /camera/depth/image_raw с типом sensor_msgs/msg/Image. Проверить в rviz2 через дисплей DepthCloud.
11	Лидар: 360 лучей, диапазон 0.15 – 4.0 м, σ = 0.03 м. Поставить через Resource Spawner стену на 4.5 м – подтвердить, что значения превышают range/max и потому возвращают +∞. Записать через ros2 bag record /scan 10 с; вывести распределение валидных и невалидных лучей.
12	Камера с шумом σ = 0.05 (сильно зашумлённая). Сравнить визуально с σ = 0.0 при одинаковой сцене (один и тот же кубик перед роботом); сохранить два кадра. Оценить SNR изображения по приближённой формуле $10 \lg(\mu^2/\sigma^2)$.
13	Изменить gz_frame_id лидара на base_link (вместо laser_frame). Объяснить, почему преобразование TF становится тривиальным и в чём опасность такой замены при наклонах робота. Подтвердить разницу через rviz2: точки скана теперь рисуются в другом кадре.
14	Лидар с visualize = true. Через GUI Gazebo Harmonic (Visualize Lidar plugin) показать виртуальные лучи; снять фактическую длину каждого луча. Привести скриншот окна LaserScan в RViz при пустом мире (все ri = +∞) и объяснить, что увидит подписчик.
15	Передвинуть камеру на 0.20 м выше корпуса и наклонить вниз на rpy = (0, 0.3, 0). Подтвердить изменение картины

	/camera/image_raw в rviz2 (тот же кубик теперь смещён к нижнему краю кадра). Объяснить, как изменение позы сенсора отражается через TF и не требует перепрошивки самой камеры.

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Как добавить сенсор в URDF через тег <gazebo>?
2. Какие параметры настраиваются для симулированного лидара?
3. Что такое шум сенсора (<noise type="gaussian">) и какие параметры он задаёт?
4. Как добавить камеру в URDF-модель?
5. Какой топик публикует данные лидара? Камеры?
6. Как визуализировать LaserScan в RViz?
7. Что такое gz_ros2_bridge и как его настроить?
8. Как проверить, что сенсоры работают в симуляции?

Лабораторная работа 4.

SIL-ТЕСТИРОВАНИЕ: УЗЕЛ АВТОНОМНОГО ДВИЖЕНИЯ В СИМУЛЯЦИИ

Цель и содержание работы: освоить методику Software-in-the-Loop тестирования стека автономной навигации: запустить URDF-робот из ЛР 3 в Gazebo Harmonic под ROS 2 Jazzy, поднять навигационный стек Nav2 с глобальным планировщиком NavFn (A*) и локальным контроллером DWB (DWA), задать через RViz2 цель движения, провести запись траектории и сенсорных данных в rosbag2 (бэкенд mcap), а затем проанализировать журнал по трём метрикам: длине и времени пройденной траектории и количеству столкновений.

Теоретическое обоснование

1. Понятие SIL-тестирования. Software-in-the-Loop – это режим верификации программного обеспечения системы управления, при котором тот же исполняемый код узлов управления, что планируется к запуску на реальном роботе, выполняется на персональном компьютере и взаимодействует не с физическим объектом, а с его симуляционной моделью. Симулятор играет роль объекта управления и источника сенсорных данных: его выход публикуется в те же ROS-топики (/scan, /odom, /imu, /camera/image_raw), что и в реальной системе, а вход (/cmd_vel) принимается без изменений. За счёт совпадения интерфейсов код проходит SIL и попадает на робота без перекомпиляции; различаются только launch-файлы и параметр use_sim_time.

2. Архитектура навигационного стека Nav2. Стек Nav2 (Navigation 2) для ROS 2 заменил собой move_base из ROS 1. Стек собран из lifecycle-узлов, переход которых в активное состояние выполняет узел-оркестратор

lifecycle_manager. Ключевые серверы и их функции перечислены в таблице 1. В релизе Nav2 для ROS 2 Jazzy набор серверов и их параметров стабилен.

Таблица 1 – Узлы навигационного стека Nav2

3. Глобальный планировщик NavFn (A*). По умолчанию planner_server загружает плагин NavfnPlanner. Плагин строит на global_costmap двумерное поле потенциалов и проходит по нему алгоритмом A* в восьмисвязной решётке. Эвристика – манхэттенское расстояние до цели. Стоимость прохода через ячейку определяется значением costmap: проходимая ячейка – 0, граница препятствия – 253 (LETHAL), inflated-зона – линейно убывающая по экспоненте функция от расстояния до препятствия с коэффициентом inflation_radius. Параметр use_astar = true включает A*; при false работает Dijkstra.

4. Локальный контроллер DWB (DWA). controller_server загружает плагин dwb_core::DWBLocalPlanner – реализацию динамического оконного подхода (Dynamic Window Approach). На каждом шаге 20 Гц контроллер выполняет три действия:

сэмплирует пары (v, ω) линейной и угловой скорости в пределах динамически ограниченного окна, задаваемого формулой (1);

прогнозирует для каждой пары траекторию на горизонт sim_time, обычно 1.7 с;

оценивает каждую траекторию суммой критиков (PathAlign, GoalAlign, Oscillation, ObstacleFootprint, PathDist, GoalDist) и публикует /cmd_vel с наилучшей парой.

$$Vd = \{ (v, \omega) \mid v \in [v_a - a_v \cdot \Delta t,\ v_a + a_v \cdot \Delta t],\ \omega \in [\omega_a - a_\omega \cdot \Delta t,\ \omega_a + a_\omega \cdot \Delta t] \} \qquad (1)$$

Параметры min/max_vel_x, min/max_vel_theta, acc_lim_x, acc_lim_theta задают физические ограничения базы; vx_samples, vy_samples, vtheta_samples

– дискретизацию окна. Веса критиков (PathDist.scale, GoalDist.scale, ObstacleFootprint.scale) позволяют сместить поведение между «строго по линии» и «агрессивный объезд».

5. Запись rosbag2 и метрики качества. rosbag2 пишет указанные топики в каталог с метаданными metadata.yaml и одним или несколькими файлами .mcap (рекомендуемый бэкенд в ROS 2 Jazzy) либо .db3 (SQLite3). Воспроизведение возможно как в реальном времени, так и с ускорением. По записи /odom вычисляется длина пройденного пути S как сумма евклидовых расстояний между соседними кадрами позиций, а время в пути T – как разность временных меток первого и последнего сообщения. Число столкновений N оценивается по сообщениям контактного сенсора Gazebo /bumper (тип gz.msgs.Contacts, прокинутый через ros_gz_bridge), если он включён, либо вручную по эпизодам обнуления линейной скорости в ситуации, когда цель ещё не достигнута. Базовые метрики SIL-теста сведены в таблице 2.

Таблица 2 – Метрики SIL-теста автономной навигации

Пример

В качестве эталонного теста запускается робот simple_robot_sensors из ЛР 3 в мире turtlebot3_world, для которого предзаписан файл карты map.yaml + map.pgm (формат nav2_map_server). Цель ставится в точке (2.0, 1.5) в кадре map; робот стартует в (0, 0, 0). Запись ведётся 60 с в bag с именем sil_run_0 (формат mcap).

Шаг 1. Подготовка карты статического окружения. Карта сохраняется через узел slam_toolbox в режиме offline или берётся из пакета turtlebot3_navigation2.

Листинг 1 – Копирование готовой карты turtlebot3_world

```
cp $(ros2 pkg prefix turtlebot3_navigation2)/share/\
```

```
turtlebot3_navigation2/map/map.yaml ~/maps/
cp $(ros2 pkg prefix turtlebot3_navigation2)/share/\
turtlebot3_navigation2/map/map.pgm ~/maps/
```

Шаг 2. Параметры Nav2 для эталонной модели. Файл nav2_params.yaml содержит секции planner_server, controller_server, global_costmap, local_costmap. Ключевые значения приведены в листинге 2.

Листинг 2 – Фрагмент nav2_params.yaml – planner и controller

```
planner_server:
  ros__parameters:
    expected_planner_frequency: 1.0
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.20
      use_astar: true
      allow_unknown: true


controller_server:
  ros__parameters:
    use_sim_time: true
    controller_frequency: 20.0
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      min_vel_x: 0.0
      max_vel_x: 0.22
      max_vel_theta: 1.8
      acc_lim_x: 2.5
      acc_lim_theta: 3.2
      vx_samples: 20
      vtheta_samples: 40
      sim_time: 1.7
      critics: ["RotateToGoal", "Oscillation",
              "BaseObstacle", "GoalAlign",
              "PathAlign", "PathDist", "GoalDist"]
```

Шаг 3. Запуск Gazebo Harmonic с моделью и Nav2. В терминале № 1 поднимается gz sim через launch-файл sensors.launch.py из ЛР 3, в терминале № 2 – Nav2 со включённым use_sim_time, в терминале № 3 – RViz2.

Листинг 3 – Запуск Gazebo Harmonic с моделью и навигационного стека

```
# Terminal 1:
ros2 launch simple_robot_description sensors.launch.py


# Terminal 2:
ros2 launch nav2_bringup bringup_launch.py \
  map:=$HOME/maps/map.yaml \
  params_file:=$HOME/ros2_ws/src/\
simple_robot_description/config/nav2_params.yaml \
  use_sim_time:=true


# Terminal 3:
ros2 launch nav2_bringup rviz_launch.py
```

Шаг 4. Постановка цели через RViz2. В верхней панели RViz2 имеется кнопка 2D Pose Estimate для начальной локализации (установить курсор на стартовой позиции и оттащить в сторону ориентации) и кнопка Nav2 Goal для постановки цели. После клика на Nav2 Goal и указания точки (2.0, 1.5) в окне 3D-сцены bt_navigator принимает action NavigateToPose, planner_server строит путь, controller_server начинает публиковать /cmd_vel, и робот в Gazebo Harmonic приходит в движение.

Шаг 5. Запись rosbag2. В терминале № 4 параллельно с движением запускается ros2 bag record на интересующих топиках с бэкендом mcap.

Листинг 4 – Запись rosbag2 в формате mcap

```
ros2 bag record -o sil_run_0 -s mcap \
  /odom /cmd_vel /scan /tf /tf_static \
  /plan /local_costmap/costmap
```

По завершении движения (около 30 – 45 с в эталонном сценарии) запись останавливается комбинацией Ctrl + C. В каталоге sil_run_0 появляются файлы metadata.yaml и sil_run_0_0.mcap.

Шаг 6. Анализ записи rosbag2 на Python. Скрипт читает bag через rosbag2_py, извлекает сообщения /odom и /cmd_vel и считает метрики S, T,

v_avg. /cmd_vel в Jazzy имеет тип geometry_msgs/msg/TwistStamped (новый дефолт, согласованный с ros_gz_bridge).

Листинг 5 – Скрипт analyze_bag.py – подсчёт метрик SIL-прогона

```
from rosbag2_py import SequentialReader, StorageOptions, \
                       ConverterOptions
from rclpy.serialization import deserialize_message
from nav_msgs.msg import Odometry
from geometry_msgs.msg import TwistStamped
import math, sys


def read(path):
    r = SequentialReader()
    r.open(StorageOptions(uri=path, storage_id='mcap'),
           ConverterOptions('', ''))
    odom, cmd = [], []
    while r.has_next():
        topic, data, ts = r.read_next()
        if topic == '/odom':
            m = deserialize_message(data, Odometry)
            p = m.pose.pose.position
            odom.append((ts * 1e-9, p.x, p.y))
        elif topic == '/cmd_vel':
            m = deserialize_message(data, TwistStamped)
            cmd.append((ts * 1e-9, m.twist.linear.x))
    return odom, cmd


def metrics(odom, cmd):
    s = sum(math.hypot(b[1]-a[1], b[2]-a[2])
            for a, b in zip(odom, odom[1:]))
    t = odom[-1][0] - odom[0][0]
    v = sum(abs(x[1]) for x in cmd) / max(1, len(cmd))
    return s, t, v


odom, cmd = read(sys.argv[1])
s, t, v = metrics(odom, cmd)
print(f'S = {s:.2f} m, T = {t:.1f} s, v_avg = {v:.3f} m/s')
```

Ожидаемый вывод для эталонного прогона приведён в листинге 7.

Листинг 6 – Метрики эталонного SIL-прогона

```
$ python3 analyze_bag.py sil_run_0
S = 2.84 m, T = 38.2 s, v_avg = 0.184 m/s
```

Полученная длина траектории S = 2.84 м превышает кратчайшее евклидово расстояние (≈ 2.50 м) на 14 %, что характерно для DWB с включёнными критиками PathAlign и ObstacleFootprint при выбранном inflation_radius. Среднее значение скорости v_avg = 0.184 м/с – около 84 % от максимума 0.22 м/с, столкновений по /bumper не зафиксировано.

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Цель (1.0, 0.0) в кадре map. Запустить эталонный прогон – записать S, T, v_avg. Изменить inflation_radius в global_costmap с 0.55 до 0.25 м. Повторить прогон, оценить, как сократилась длина пути S и не появились ли касания препятствий.
2	Цель (2.0, 2.0). Уменьшить max_vel_x в FollowPath с 0.22 до 0.10 м/с. Сравнить T и v_avg с эталоном (увеличение T примерно в 2 раза ожидается). Проверить, что DWB не выходит за заданный лимит – построить временной ряд cmd.twist.linear.x.
3	Цель (−1.5, 1.0). Поменять use_astar с true на false – включить Dijkstra. Сравнить пути A* и Dijkstra по длине S и числу узлов (через /plan). Объяснить, почему Dijkstra даёт оптимальный по

	длине путь, но медленнее A* в обсчёте.
4	Цель (3.0, −1.0). Через панель Resource Spawner поставить цилиндр-препятствие посередине прямой между стартом и целью. Оценить, обходит ли DWB препятствие, и измерить отклонение Δ от исходного плана. Если препятствие появилось после старта плана – проверить, перепланирует ли planner_server траекторию.
5	Цель (0.0, 2.5). Увеличить vtheta_samples в DWB с 40 до 80. Снять временной ряд cmd.twist.angular.z – оценить, изменилась ли плавность поворотов. Сравнить нагрузку CPU controller_server (top -p $(pgrep controller_server)) с эталонной.
6	Цель (1.5, 1.5). Сократить sim_time DWB с 1.7 до 1.0 с. Зафиксировать момент проявления «близоруких» манёвров: сравнить число резких рывков /cmd_vel за прогон. Объяснить, почему слишком короткий sim_time мешает DWB избегать препятствий.
7	Цель (2.0, 0.0). В SDF лидара увеличить шум noise/stddev до 0.05 м. Проследить, как изменилось дрожание /cmd_vel и плотность точек на local_costmap. Оценить, появились ли отказы локализации amcl при такой зашумлённости.
8	Цель (1.0, 1.0). Сменить локальный планировщик с DWB на TEB (Timed Elastic Band, плагин teb_local_planner_ros::TebLocalPlannerROS). Сравнить S, T, v_avg на той же цели. Сделать вывод, какой планировщик лучше держит длину пути и какой – время.
9	Цель (2.5, 2.5). Уменьшить controller_frequency с 20 до 5 Гц. Зафиксировать число столкновений и среднее отклонение от плана. Объяснить, при каких физических скоростях такая частота

	становится опасной (формула $v \cdot \Delta t >$ inflation_radius).
10	Цель (3.0, 0.5). Поднять acc_lim_x с 2.5 до 5.0 м/с². Оценить, как изменилось время T и максимум |a| из численного дифференцирования cmd.twist.linear.x. Сравнить с физическими ограничениями колёс TurtleBot3 Burger – укажет ли DWB фактически бóльшие ускорения, чем DiffDrive-плагин может реализовать.
11	Цель (−2.0, 0.0). Запустить три прогона подряд (между прогонами – перезапуск gz sim и Nav2). Оценить разброс S, T, v_avg между запусками: среднее и стандартное отклонение. Объяснить источники недетерминизма в SIL-прогоне (DDS, ogre2-рендеринг, многопоточность).
12	Цель (1.5, −1.5). Изменить вес критика PathAlign с 32 до 8. Сравнить отклонение Δ от глобального плана с эталоном. Дополнительно повысить вес GoalDist с 24 до 48 – проверить, уменьшится ли S за счёт «срезания угла».
13	Цель (0.0, 3.0). Через Resource Spawner поставить два узких прохода (стенки на расстоянии 0.35 м друг от друга). Оценить, проходит ли робот при footprint radius = 0.15 м и inflation_radius = 0.20 м. Уменьшить inflation_radius до 0.10 м – повторить, объяснить трейд-офф «проходимость vs запас безопасности».
14	Цель (2.0, 2.0). Запустить ros2 bag record -s mcap только /odom. Проверить, можно ли по одной /odom восстановить S и T (достаточные данные). Дополнить bag /cmd_vel и посчитать v_avg – оценить разницу с оценкой из одного /odom.
15	Цель (1.0, 2.0). Снизить tolerance планировщика с 0.20 до 0.05 м. Оценить, появляются ли отказы NavigateToPose (статус ABORTED). Объяснить, почему слишком жёсткий tolerance

	несовместим с дрожанием одометрии и сферическим footprint.

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Что такое SIL-тестирование и зачем оно нужно?
2. Как запустить Nav2 в симуляции Gazebo?
3. Как задать цель навигации через RViz (2D Nav Goal)?
4. Как записать rosbag для анализа?
5. Какие метрики оценивают качество навигационного алгоритма?
6. Как построить траекторию из rosbag на Python?
7. Что такое «коллизия» в контексте SIL?
8. Какие параметры DWB влияют на скорость навигации?

Лабораторная работа 5.

АВТОМАТИЗИРОВАННЫЙ ПРОГОН СИМУЛЯЦИЙ С РАНДОМИЗАЦИЕЙ СЦЕНАРИЕВ

Цель и содержание работы: освоить методику автоматизированного массового тестирования алгоритмов автономного движения в среде Gazebo Harmonic, реализовать Python-обвязку для запуска симулятора через subprocess с рандомизацией начальных условий (положение робота, цель, препятствия), научиться собирать статистику успешности и времени выполнения по сотне прогонов и интерпретировать полученные распределения для оценки качества алгоритма навигации.

Теоретическое обоснование

1. Зачем нужен автоматизированный прогон. Один сеанс работы робота в симуляторе мало о чём говорит – случайно сошедшаяся траектория не гарантирует, что алгоритм навигации будет устойчиво работать при иных начальных условиях. Принятой практикой в индустрии беспилотного транспорта является массовое тестирование (scenario-based testing): один и тот же стек запускается на десятках и сотнях случайно сгенерированных сцен, после чего по агрегированным метрикам (процент успеха, среднее время, число коллизий) делается вывод о готовности алгоритма.

Минимальный набор метрик, который имеет смысл собирать в учебной постановке, приведён в таблице 1.

Таблица 1 – Метрики массового тестирования автономного движения

2. Структура цикла рандомизированного тестирования. Цикл состоит из пяти повторяющихся шагов: генерация набора случайных параметров сцены; запуск симуляции (Gazebo Harmonic + стек навигации) подпроцессом;

ожидание условия завершения (достижение цели, столкновение, таймаут); чтение результата из лог-файла либо из ROS-топика; запись строки в CSV-таблицу результатов. После всех прогонов запускается отдельный скрипт визуализации, который строит гистограмму времени и столбчатую диаграмму процента успеха.

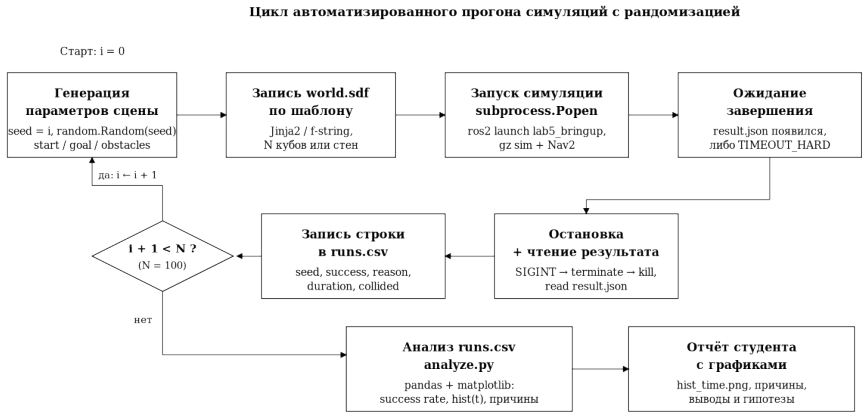

Рисунок 1 – Цикл автоматизированного прогона симуляций с рандомизацией

3. Управление подпроцессом Gazebo. С января 2025 года Gazebo Classic (gazebo11) официально не поддерживается – вместо него используется Gazebo Harmonic (LTS до 2028 года) с командной утилитой gz. В ROS 2 Jazzy Jalisco запуск симулятора и стека навигации выполняется командой ros2 launch <пакет> <файл>.launch.py; интеграция с симулятором обеспечивается пакетами ros_gz_sim (запуск gz sim) и ros_gz_bridge (мост топиков gz transport ↔ ROS 2). Из Python удобнее всего использовать модуль subprocess: подпроцесс стартует через subprocess.Popen, ожидание ограничивается методом wait(timeout=…), а принудительная остановка – send_signal(signal.SIGINT) с последующим terminate() и kill(), если процесс не отреагировал. Такая каскадная остановка нужна потому, что gz sim и спавненные ноды должны успеть корректно освободить порты gz transport и временные файлы; жёсткий kill без предварительного SIGINT приводит к «висящим» процессам gz sim и блокировке следующего запуска.

4. Источник случайных параметров. Студенту достаточно трёх семейств случайных величин:

Начальная поза робота (x, y, theta) – x и y равномерно из квадрата стороной A метров, theta равномерно из [0, 2π).

Целевая точка (xg, yg) – равномерно из того же квадрата с ограничением на минимальное расстояние до старта (не ближе D = 2 м), чтобы задача была содержательной.

Препятствия (N кубов либо отрезков стен) – координаты центров равномерно по квадрату, повороты произвольные, проверка непересечения со стартом и целью.

Для воспроизводимости генератор инициализируется фиксированным зерном seed = run_index, где run_index – порядковый номер прогона. Это позволяет повторно прогнать конкретный неудачный сценарий, передав его номер в качестве аргумента командной строки.

5. Внесение препятствий в Gazebo. Препятствие в среде Gazebo Harmonic – это сущность модели (model), описанная в формате SDF версии 1.10. Динамически вставить модель в работающий симулятор можно тремя способами:

Сгенерировать SDF-файл мира перед стартом каждого прогона (самый надёжный путь для учебной задачи – Gazebo читает финальный мир один раз при старте).

Использовать сервис gz service /world/<name>/create – запросом SDF-фрагмента (полезно, если требуется добавлять препятствия по ходу прогона; в ROS 2 удобно обёрнут узлами пакета ros_gz_sim).

Шаблонизация через xacro либо Jinja2: единый шаблон мира с подстановкой списка препятствий из Python.

В рамках лабораторной работы применяется первый способ – перед каждым прогоном Python-скрипт пишет world.sdf в файл и передаёт путь к нему через аргумент launch.

6. Условие завершения прогона и таймаут. Прогон считается успешным, если в течение времени T_max евклидово расстояние от текущей позиции робота до цели стало меньше радиуса R_goal (типичные значения R_goal = 0.25 м, T_max = 60 с). Прогон считается неуспешным, если истёк T_max или зафиксирована коллизия. Само условие проверяет отдельная нода-монитор, подписанная на /odom, /goal и /bumper, и пишущая итоговую строку в файл result.json в момент выхода. Python-обвязка после завершения подпроцесса читает этот файл и добавляет запись в общий runs.csv.

7. Анализ результатов. После накопления N строк в runs.csv запускается скрипт analyze.py: он загружает таблицу через pandas, считает процент успеха (доля строк с success = True), медиану и интерквартильный размах времени до цели для успешных прогонов, строит гистограмму времени с помощью matplotlib (savefig в PNG, число корзин ~ √N) и столбчатую диаграмму распределения причин неудачи (таймаут / коллизия / застревание). Полученные графики прикладываются к отчёту.

Пример

В качестве «нулевого» варианта рассматривается следующая постановка: робот TurtleBot3 burger в пустой комнате 8×8 м, 5 случайных кубических препятствий со стороной 0.5 м, 100 прогонов, таймаут 60 с, радиус достижения цели 0.25 м, стек навигации Nav2 со стандартным DWB-локальным планировщиком и NavfnPlanner-глобальным. Целевая ROS-дистрибуция – Jazzy Jalisco на Ubuntu 24.04, симулятор – Gazebo Harmonic, пакет моделей робота – turtlebot3_simulations.

Шаг 1. Генератор сценариев. Скрипт scenario.py отвечает за случайную генерацию набора параметров сцены и формирование SDF-файла мира. На каждом вызове генератор инициализируется переданным seed, поэтому сценарий с номером i всегда восстановим. Реализация показана в листинге 1.

Листинг 1 – Генерация случайного сценария и запись world.sdf

```
import math
import random
from pathlib import Path


AREA = 4.0          # полуразмер квадрата сцены, м
MIN_DIST = 2.0      # минимальное расстояние старт-цель, м
OBSTACLE_SIDE = 0.5  # сторона куба-препятствия, м


WORLD_HEADER = '''<?xml version="1.0"?>
<sdf version="1.10">
  <world name="randomized">
    <plugin filename="gz-sim-physics-system"
            name="gz::sim::systems::Physics"/>
    <plugin filename="gz-sim-sensors-system"
            name="gz::sim::systems::Sensors"/>
    <plugin filename="gz-sim-scene-broadcaster-system"
            name="gz::sim::systems::SceneBroadcaster"/>
    <light type="directional" name="sun">
      <direction>-0.5 0.1 -0.9</direction>
    </light>
    <model name="ground_plane"><static>true</static>
      <link name="link"><collision name="c"><geometry>
        <plane><normal>0 0 1</normal></plane>
      </geometry></collision></link></model>
'''
WORLD_FOOTER = '  </world>\n</sdf>\n'


OBSTACLE_TPL = '''    <model name="obs_{i}">
      <pose>{x} {y} 0.25 0 0 {yaw}</pose>
      <static>true</static>
      <link name="l">
        <collision name="c"><geometry><box>
          <size>{s} {s} 0.5</size></box></geometry></collision>
        <visual name="v"><geometry><box>
          <size>{s} {s} 0.5</size></box></geometry></visual>
      </link>
    </model>
'''


def _far_enough(p, items, d):
    return all(math.hypot(p[0]-q[0], p[1]-q[1]) > d for q in items)


def generate(seed, n_obstacles=5):
    rng = random.Random(seed)
    start = (rng.uniform(-AREA, AREA), rng.uniform(-AREA, AREA),
```

```
            rng.uniform(0, 2*math.pi))
    while True:
        goal = (rng.uniform(-AREA, AREA), rng.uniform(-AREA, AREA))
        if math.hypot(goal[0]-start[0], goal[1]-start[1]) > MIN_DIST:
            break
    obstacles = []
    while len(obstacles) < n_obstacles:
        p = (rng.uniform(-AREA, AREA), rng.uniform(-AREA, AREA))
        if _far_enough(p, [start[:2], goal] + obstacles, 0.8):
            obstacles.append(p)
    yaws = [rng.uniform(0, 2*math.pi) for _ in obstacles]
    return {'start': start, 'goal': goal,
            'obstacles': list(zip(obstacles, yaws))}


def write_world(scn, out_path):
    body = WORLD_HEADER
    for i, ((x, y), yaw) in enumerate(scn['obstacles']):
        body += OBSTACLE_TPL.format(i=i, x=x, y=y, yaw=yaw,
                                    s=OBSTACLE_SIDE)
    body += WORLD_FOOTER
    Path(out_path).write_text(body, encoding='utf-8')
```

Шаг 2. Нода-монитор завершения. Монитор подписывается на одометрию и контакт-сенсор, проверяет условие достижения цели, фиксирует столкновения и по любому исходу пишет JSON-результат, после чего вызывает rclpy.shutdown(). Реализация приведена в листинге 2.

Листинг 2 – ROS 2-нода monitor.py – детектирует исход прогона и пишет result.json

```
import json
import math
import sys
import time
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from std_msgs.msg import Bool


GOAL_RADIUS = 0.25
TIMEOUT = 60.0


class Monitor(Node):
    def __init__(self, goal_xy, out_path):
```

```
        super().__init__('run_monitor')
        self.goal = goal_xy
        self.out_path = out_path
        self.t0 = time.time()
        self.collided = False
        self.create_subscription(Odometry, '/odom',
                                 self.on_odom, 10)
        self.create_subscription(Bool, '/bumper',
                                 self.on_bumper, 10)
        self.create_timer(0.5, self.on_tick)


    def on_bumper(self, msg):
        if msg.data:
            self.finish(success=False, reason='collision')


    def on_odom(self, msg):
        p = msg.pose.pose.position
        if math.hypot(p.x - self.goal[0],
                      p.y - self.goal[1]) < GOAL_RADIUS:
            self.finish(success=True, reason='goal_reached')


    def on_tick(self):
        if time.time() - self.t0 > TIMEOUT:
            self.finish(success=False, reason='timeout')


    def finish(self, success, reason):
        result = {'success': success, 'reason': reason,
                  'duration': time.time() - self.t0,
                  'collided': self.collided}
        with open(self.out_path, 'w') as f:
            json.dump(result, f)
        rclpy.shutdown()


def main():
    rclpy.init()
    gx, gy, out = float(sys.argv[1]), float(sys.argv[2]), sys.argv[3]
    rclpy.spin(Monitor((gx, gy), out))


if __name__ == '__main__':
    main()
```

Шаг 3. Обвязка – запуск прогона. Функция run_once собирает сценарий, пишет world.sdf, стартует Gazebo Harmonic и стек Nav2 одним launch-файлом (передавая мир и начальную позу аргументами), параллельно стартует

монитор. По завершении (или таймауту) корректно гасит подпроцесс каскадом сигналов и читает result.json. Реализация показана в листинге 3.

Листинг 3 – Запуск одного прогона: сборка сценария, старт gz sim + Nav2, чтение result.json

```
import json
import signal
import subprocess
import tempfile
import time
from pathlib import Path
from scenario import generate, write_world


LAUNCH_PKG = 'lab5_bringup'
LAUNCH_FILE = 'sim_with_nav2.launch.py'
TIMEOUT_HARD = 90  # на ~30 с больше, чем мониторовский


def _stop(proc):
    if proc.poll() is not None:
        return
    proc.send_signal(signal.SIGINT)
    try:
        proc.wait(timeout=10)
        return
    except subprocess.TimeoutExpired:
        proc.terminate()
    try:
        proc.wait(timeout=5)
    except subprocess.TimeoutExpired:
        proc.kill()


def run_once(seed):
    scn = generate(seed)
    tmp = Path(tempfile.mkdtemp(prefix=f'run{seed}_'))
    world = tmp / 'world.sdf'
    result = tmp / 'result.json'
    write_world(scn, world)
    sx, sy, sth = scn['start']
    gx, gy = scn['goal']
    args = ['ros2', 'launch', LAUNCH_PKG, LAUNCH_FILE,
            f'world:={world}', f'x_pose:={sx}', f'y_pose:={sy}',
            f'yaw:={sth}', f'goal_x:={gx}', f'goal_y:={gy}',
            f'result_path:={result}']
    proc = subprocess.Popen(args)
    t0 = time.time()
    while time.time() - t0 < TIMEOUT_HARD:
```

```
        if result.exists():
            break
        time.sleep(0.5)
    _stop(proc)
    if result.exists():
        data = json.loads(result.read_text())
    else:
        data = {'success': False, 'reason': 'no_result',
                'duration': TIMEOUT_HARD, 'collided': False}
    data['seed'] = seed
    return data
```

Шаг 4. Главный цикл и сохранение runs.csv. Скрипт run_all.py последовательно вызывает run_once(seed) для seed = 0…N – 1, пишет каждую запись в CSV и периодически выводит промежуточную сводку, чтобы преподаватель видел ход эксперимента. Параллельный запуск нескольких прогонов в учебной постановке не рассматривается – Gazebo Harmonic требует значительных ресурсов и плохо переносит сосуществование нескольких gz sim на одном хосте.

Листинг 4 – Главный цикл массового тестирования с записью в CSV

```
import csv
import sys
from runner import run_once


FIELDS = ['seed', 'success', 'reason', 'duration', 'collided']


def main(n=100, out='runs.csv'):
    with open(out, 'w', newline='') as f:
        w = csv.DictWriter(f, fieldnames=FIELDS)
        w.writeheader()
        succ = 0
        for seed in range(n):
            r = run_once(seed)
            w.writerow({k: r[k] for k in FIELDS})
            f.flush()
            succ += int(r['success'])
            print(f'[{seed+1}/{n}] {r["reason"]:<14} '
                  f'success_rate={succ/(seed+1):.2%}')


if __name__ == '__main__':
    main(int(sys.argv[1]) if len(sys.argv) > 1 else 100)
```

Шаг 5. Анализ runs.csv и визуализация. Сводка по итогам ста прогонов приведена в таблице 2; на её основе студент в отчёте делает выводы и формулирует гипотезы об источниках неудач.

Таблица 2 – Пример сводки по 100 прогонам базового сценария

Гистограмма времени до цели строится скриптом analyze.py стандартными средствами pandas + matplotlib (листинг 5). На полученной диаграмме обычно видно одно-два модальных значения, соответствующих типичным траекториям обхода препятствий, и длинный «хвост» от прогонов, в которых локальный планировщик уходил в локальный минимум и перепланировался несколько раз.

Листинг 5 – Скрипт построения гистограммы времени и сводной таблицы

```
import pandas as pd
import matplotlib.pyplot as plt


df = pd.read_csv('runs.csv')
succ = df[df.success]
print('success rate:', df.success.mean())
print('median t:', succ.duration.median())
print(df.reason.value_counts())


ax = succ.duration.plot.hist(bins=int(len(succ) ** 0.5),
                             edgecolor='black')
ax.set_xlabel('Время до цели, с')
ax.set_ylabel('Число прогонов')
plt.savefig('hist_time.png', dpi=150, bbox_inches='tight')
```

Ожидаемые числовые ориентиры. Для исправного стека Nav2 на пустой сцене с пятью кубическими препятствиями типичный процент успеха – 75…90 %, медианное время – 15…25 с, большинство неудач приходится на таймауты (локальный минимум), а не на столкновения. Если процент успеха

ниже 60 %, в отчёте необходимо разобраться, ограничен ли результат плохой настройкой планировщика, нереалистичной плотностью препятствий или ошибкой в самом сценарии.

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	**Содержание индивидуального задания**
1	Собрать сцену: робот TurtleBot3 burger в комнате 8×8 м с 5 кубическими препятствиями (сторона 0.5 м), 100 прогонов, таймаут 60 с, стандартный Nav2 (NavfnPlanner + DWB). Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Построить гистограмму времени до цели и столбчатую диаграмму причин неудач (goal_reached / timeout / collision). Вычислить процент успеха и его 95-%-доверительный интервал по формуле Уилсона; обосновать достаточность 100 прогонов. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.

2	Собрать сцену: TurtleBot3 burger в комнате 6×6 м с 8 кубами стороной 0.4 м, 100 прогонов, таймаут 60 с, Nav2 по умолчанию. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Сравнить процент успеха и долю коллизий с базовым сценарием варианта 1; объяснить влияние плотности препятствий на работу локального планировщика. Снять зависимость доли коллизий от inflation_radius (0.30, 0.40, 0.55 м), по 50 прогонов на значение. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
3	Собрать сцену: TurtleBot3 waffle в комнате 10×10 м с 6 цилиндрами радиуса 0.3 м, 100 прогонов, таймаут 90 с. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Снять распределение длины пройденного пути; вычислить медианное отношение длина / евклидова дистанция между стартом и целью. Сравнить TurtleBot3 burger и waffle на одних и тех же seed (по 50 прогонов каждый); объяснить разницу по медианному времени габаритами робота. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
4	Собрать сцену: TurtleBot3 burger, сцена 8×8 м, рандомизация только начальной позы (цель фиксирована (3, 3)), 5 препятствий, 100 прогонов. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев

	scenario.py (write_world + generate) и обвязку run_once. Построить тепловую карту 8×8 успешных и неуспешных стартовых позиций (бин 1 м, цвет по доле успеха). Идентифицировать секторы старта с долей успеха < 50 %; предложить, как изменить inflation_radius или вес критика PathAlign, чтобы их устранить. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
5	Собрать сцену: TurtleBot3 burger, сцена 8×8 м, рандомизация только цели (старт фиксирован (–3, –3)), 5 препятствий, 100 прогонов. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Построить тепловую карту успешных целей. Выделить «мёртвые зоны» цели (бины с долей успеха < 50 %) и объяснить геометрически, почему именно туда планировщик не приводит робота. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
6	Собрать сцену: TurtleBot3 burger, сцена 8×8 м, число препятствий варьируется $n \in \{0, 3, 6, 9\}$, по 50 прогонов на каждое значение. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Построить кривую «процент успеха – n» с доверительными интервалами Уилсона. Определить критическое n*, при котором процент успеха падает ниже 60 %; обосновать выбранную плотность. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная

	таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
7	Собрать сцену: TurtleBot3 burger, сцена 8×8 м, препятствия в виде отрезков стен длиной 2 м (по 4 стены случайно развёрнутых), 100 прогонов. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Сравнить долю коллизий и долю таймаутов с базовым сценарием варианта 1. Объяснить, почему стены проигрывают/выигрывают у кубов по числу локальных минимумов DWB. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
8	Собрать сцену: TurtleBot3 burger, базовая сцена, 150 прогонов при двух значениях max_vel_x (0.22 м/с и 0.40 м/с). Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Сравнить процент успеха, медианное время и долю коллизий при двух скоростях. Построить гистограммы времени для обеих скоростей на одной плоскости; определить, при какой скорости разброс меньше. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
9	Собрать сцену: TurtleBot3 burger, базовая сцена, 100 прогонов при двух локальных планировщиках (DWB и MPPI). Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world +

	generate) и обвязку run_once. Сравнить распределения времени до цели для DWB и MPPI (median, p95). Подсчитать число коллизий в каждом случае; обосновать выбор контроллера для последующих лабораторных. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
10	Собрать сцену: TurtleBot3 burger, сцена 12×12 м, 10 препятствий, 100 прогонов, таймаут 120 с. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Оценить, как меняется доля таймаутов с увеличением масштаба сцены (сравнить с базовым сценарием варианта 1). Снять зависимость медианного времени от площади сцены (8×8, 10×10, 12×12, по 50 прогонов). Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
11	Собрать сцену: робот собственного URDF из ЛР 2 (дифпривод, колея 0.20 м, лидар 360°), сцена 8×8 м, 5 препятствий, 100 прогонов. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Сравнить с TurtleBot3 burger на одних и тех же seed (по 100 прогонов) по проценту успеха и средней длине пути. Определить, какие параметры собственного робота (колея, максимальная скорость, расположение лидара) ограничивают его в сравнении с burger. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент),

	сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
12	Собрать сцену: TurtleBot3 burger, базовая сцена, 100 прогонов; цель задаётся в виде узкого коридора шириной 0.6 м между двумя параллельными стенами. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Оценить долю успешных проходов; сравнить с долей успеха в широком коридоре (1.2 м) на тех же seed. Подобрать минимальную ширину коридора, при которой доля успеха остаётся выше 80 % при стандартных параметрах Nav2. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
13	Собрать сцену: TurtleBot3 burger, сцена 8×8 м, 5 препятствий, 200 прогонов. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Сравнить устойчивость оценки процента успеха при n = 50, n = 100 и n = 200 (ширина доверительного интервала Уилсона). Обосновать минимальное число прогонов, при котором ширина интервала не превышает ±5 %. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
14	Собрать сцену: TurtleBot3 burger, 100 прогонов в двух мирах: empty (как в примере) и turtlebot3_house из пакета turtlebot3_simulations. Запустить Gazebo Harmonic + Nav2 одним

	launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Сравнить распределения времени и долю таймаутов в двух мирах. Объяснить, какие особенности дома (узкие проёмы, разветвлённость) дают основной вклад в задержку. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.
15	Собрать сцену: TurtleBot3 waffle_pi, сцена 8×8 м, 6 препятствий со случайной формой (куб / цилиндр / сфера, выбирается по seed), 100 прогонов. Запустить Gazebo Harmonic + Nav2 одним launch-файлом через ros_gz_sim; подготовить генератор сценариев scenario.py (write_world + generate) и обвязку run_once. Построить столбчатую диаграмму причин неудачи в разрезе формы препятствия. Идентифицировать наиболее «опасную» форму и предложить, как изменить критики DWB, чтобы снизить коллизии именно с ней. Подготовить отчёт: схема сцены, конфиг Nav2 (yaml-фрагмент), сводная таблица runs.csv, основная гистограмма, выводы об узких местах и предложение по их устранению.

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Зачем нужен автоматизированный прогон симуляций?
2. Как запустить Gazebo из Python через subprocess?
3. Как рандомизировать начальную позицию робота и цели?
4. Что такое гистограмма времени выполнения?
5. Как задать таймаут для одного прогона?
6. Как сохранить статистику в CSV?
7. Что такое процент успешных завершений?
8. Как визуализировать результаты 100 прогонов?

Лабораторная работа 6.

ГЕНЕРАЦИЯ И ДОБАВЛЕНИЕ ШУМОВ В СЕНСОРЫ (ИМИТАЦИЯ РЕАЛЬНОСТИ)

Цель и содержание работы: освоить методику внесения реалистичных моделей шума в сенсоры робота (лидар, камера, IMU), описанные в формате URDF/SDF для Gazebo Harmonic; повторно запустить алгоритм автономной навигации из лабораторной работы №4 при разных уровнях шума; оценить, как именно деградирует поведение робота (число столкновений, время выполнения, форма траектории) по сравнению с идеализированным случаем и сделать практические выводы о требованиях к сенсорной подсистеме реального робота.

Теоретическое обоснование

1. Зачем добавлять шум в симуляции. По умолчанию сенсоры в Gazebo Harmonic возвращают «идеальные» значения: лидар – точное расстояние до ближайшего пересечения луча с геометрией мира, камера – пиксели, отрендеренные движком ogre2, IMU – производные от ground-truth-кинематики тела. Алгоритм навигации, отлаженный на таких данных, в реальности может вести себя хуже на порядок: подавится мультиэховыми отражениями лидара, уйдёт в накопленную ошибку одометрии из-за дрейфа IMU, пропустит препятствие из-за пересвеченной камеры. Добавление модели шума в симуляцию – обязательный этап валидации, без которого SIL-тестирование не даёт защиты от класса «реальных» отказов.

2. Гауссова модель шума и её параметры. Аддитивный гауссов шум описывается двумя числами: математическим ожиданием μ (чаще всего $\mu = 0$, шум симметричный относительно истинного значения) и стандартным отклонением σ. Для одиночного наблюдения z истинного значения z^* выполняется:

$$z = z^* + \varepsilon,\ \ \varepsilon \sim N(\mu, \sigma^2) \qquad (1)$$

Параметр σ имеет физический смысл: для лидара это разброс измерения дальности в метрах, для гироскопа IMU – СКО угловой скорости в рад/с, для акселерометра – СКО ускорения в м/с². Реальные паспортные значения берутся из datasheet сенсора (например, для Velodyne VLP-16 σ ≈ 0.03 м, для MPU-6050 σ_gyro ≈ 0.005 рад/с). Сводка типовых порядков приведена в таблице 1.

Таблица 1 – Типовые СКО шума бытовых и индустриальных сенсоров

3. Описание шума в SDF для Gazebo Harmonic. В формате SDF 1.10, используемом Gazebo Harmonic, шум объявляется внутри тега <sensor> через подтег <noise>. Для лидара (<sensor type="gpu_lidar"> либо type="lidar") поддерживается аддитивный гауссов шум на каждой измеренной дальности; шаблон фрагмента показан в листинге 1. Обратите внимание: в Harmonic для интеграции с ROS 2 Jazzy достаточно объявить топик и подключить мост через ros_gz_bridge – собственного gazebo_ros-плагина в URDF больше не пишут.

Листинг 1 – Фрагмент SDF: лидар gpu_lidar с гауссовым шумом дальности

```
<sensor name="laser" type="gpu_lidar">
 <pose>0 0 0.15 0 0 0</pose>
 <topic>scan</topic>
 <update_rate>10</update_rate>
 <lidar>
  <scan>
   <horizontal>
    <samples>360</samples>
    <resolution>1.0</resolution>
    <min_angle>-3.1415</min_angle>
    <max_angle> 3.1415</max_angle>
   </horizontal>
  </scan>
  <range>
   <min>0.12</min>
```

```
    <max>3.5</max>
    <resolution>0.01</resolution>
   </range>
   <noise>
    <type>gaussian</type>
    <mean>0.0</mean>
    <stddev>0.02</stddev>
   </noise>
  </lidar>
</sensor>
```

4. Шум IMU. Для тега <sensor type="imu"> поддерживается раздельная настройка четырёх компонент: noise + bias_mean + bias_stddev для угловой скорости и для линейного ускорения. Параметр noise описывает аддитивный гауссов шум измерения, а bias_mean / bias_stddev задают случайно выбираемое значение начального смещения (drift), которое сохраняется постоянным в течение всего прогона. Это аккуратно моделирует реальную ситуацию: каждый сенсор IMU при включении имеет свой собственный bias, который медленно меняется со временем. Фрагмент показан в листинге 2.

Листинг 2 – Фрагмент SDF: IMU с шумом и дрейфом (Gazebo Harmonic)

```
<sensor name="imu" type="imu">
 <always_on>true</always_on>
 <update_rate>200</update_rate>
 <topic>imu</topic>
 <imu>
  <angular_velocity>
   <x><noise type="gaussian">
    <mean>0.0</mean>
    <stddev>0.005</stddev>
    <bias_mean>0.0</bias_mean>
    <bias_stddev>0.001</bias_stddev>
   </noise></x>
   <y><noise type="gaussian">
    <mean>0.0</mean><stddev>0.005</stddev>
    <bias_mean>0.0</bias_mean><bias_stddev>0.001</bias_stddev>
   </noise></y>
   <z><noise type="gaussian">
    <mean>0.0</mean><stddev>0.005</stddev>
    <bias_mean>0.0</bias_mean><bias_stddev>0.001</bias_stddev>
   </noise></z>
  </angular_velocity>
```

```
   <linear_acceleration>
    <x><noise type="gaussian">
     <mean>0.0</mean><stddev>0.05</stddev>
     <bias_mean>0.0</bias_mean><bias_stddev>0.01</bias_stddev>
    </noise></x>
    <y><noise type="gaussian">
     <mean>0.0</mean><stddev>0.05</stddev>
     <bias_mean>0.0</bias_mean><bias_stddev>0.01</bias_stddev>
    </noise></y>
    <z><noise type="gaussian">
     <mean>0.0</mean><stddev>0.05</stddev>
     <bias_mean>0.0</bias_mean><bias_stddev>0.01</bias_stddev>
    </noise></z>
   </linear_acceleration>
  </imu>
</sensor>
```

5. Шум камеры. Для камеры в SDF также есть тег <noise>, но его выразительности обычно не хватает (он поддерживает только гауссов аддитивный шум на яркость пикселя). Поэтому более реалистичные искажения (размытие движения, бинаризация, пересвет, потеря фокуса) принято моделировать на уровне ROS 2-обвязки: отдельная нода подписывается на исходный топик /camera/image_raw, применяет к каждому кадру OpenCV 4.x-функции (cv2.GaussianBlur, cv2.threshold, cv2.add) и публикует результат в новый топик /camera/image_noisy. Алгоритм навигации переключается на новый топик через ремап в launch-файле. Сводка наиболее употребительных «шумов» камеры приведена в таблице 2.

Таблица 2 – Модели искажений камеры и их типичные параметры

6. Запись и сравнение траекторий. Стандартный инструмент сохранения данных в ROS 2 Jazzy – rosbag2 в формате mcap (по умолчанию с релиза Jazzy). Запись запускается командой ros2 bag record -o run_<i> /odom /scan /tf /tf_static /cmd_vel. Для каждого прогона сохраняется отдельный rosbag-каталог, в дальнейшем читаемый библиотекой rosbag2_py из Python

либо утилитой ros2 bag info. Для сравнения траектории с «идеальной» (записанной без шумов) применяется одна из двух методик:

Послужная метрика: усреднённая по времени евклидова разность позиций робота в двух прогонах с одним seed-ом, после выравнивания временной шкалы (Mean Trajectory Error, MTE).

Геометрическая метрика: посчитанная по точкам Хаусдорфова дистанция между двумя пространственными ломаными (без учёта времени) – она более устойчива к замедлению робота, чем MTE.

7. Связь с лабораторной работой №4. Базовый стек навигации, заимствуемый из лабораторной работы №4 (Gazebo Harmonic + URDF робота с лидаром и камерой + Nav2 с NavfnPlanner и DWB), остаётся без изменений. Меняется только описание сенсоров: исходный SDF дублируется в три отдельных файла – clean.sdf (без шумов, эталон), noisy_partial.sdf (шум только на одном выбранном сенсоре) и noisy_full.sdf (шумы на всех трёх сенсорах). Все три прогона выполняются с одинаковыми стартом, целью и набором препятствий (seed фиксирован), что позволяет приписать различие в поведении именно влиянию сенсорного шума.

Пример

В качестве «нулевого» варианта рассматривается следующая постановка: робот TurtleBot3 burger в мире из ЛР 4 (8×8 м, 4 неподвижных препятствия), стек Nav2 (NavfnPlanner + DWB), три уровня шума: clean (без шума), lidar_only (только лидар, $\sigma = 0.05$ м), full (лидар $\sigma = 0.05$ м + IMU $\sigma_{gyro} = 0.01$, $\sigma_{acc} = 0.1$, bias_stddev = 0.005 + камера Gaussian $\sigma = 15 / 255$). По каждому уровню выполняется 20 прогонов с фиксированным набором seed = 0…19. Целевая связка: ROS 2 Jazzy + Gazebo Harmonic + ros_gz_bridge.

Шаг 1. Подготовка трёх SDF робота. Из исходного turtlebot3_burger.sdf (пакет turtlebot3_simulations для Gazebo Harmonic) копируется три файла; в

каждом ставится свой набор <noise>-тегов в блоке лидара и IMU. Фрагмент целевого блока лидара в файле noisy_full.sdf приведён в листинге 3.

Листинг 3 – SDF: лидар TurtleBot3 burger с увеличенным σ дальности

```
<link name="base_scan">
 <sensor name="lds_lfcd_sensor" type="gpu_lidar">
  <pose>0 0 0.15 0 0 0</pose>
  <topic>scan</topic>
  <update_rate>5</update_rate>
  <gz_frame_id>base_scan</gz_frame_id>
  <lidar>
   <scan><horizontal>
    <samples>360</samples><resolution>1</resolution>
    <min_angle>0.0</min_angle>
    <max_angle>6.28318</max_angle>
   </horizontal></scan>
   <range>
    <min>0.12</min><max>3.5</max>
    <resolution>0.015</resolution>
   </range>
   <noise>
    <type>gaussian</type>
    <mean>0.0</mean>
    <stddev>0.05</stddev>
   </noise>
  </lidar>
  <always_on>true</always_on>
  <visualize>false</visualize>
 </sensor>
</link>
```

Шаг 2. Нода-зашумлятор камеры. Если уровень full включает искажение камеры, отдельная ROS 2-нода добавляет гауссов шум к каждому кадру и публикует обработанное изображение в /camera/image_noisy. Реализация на OpenCV 4.x показана в листинге 4.

Листинг 4 – ROS 2-нода: гауссов шум и размытие изображения с камеры

```
import numpy as np
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
```

```
import cv2


class CameraNoiser(Node):
    def __init__(self):
        super().__init__('camera_noiser')
        self.declare_parameter('sigma', 15.0)
        self.declare_parameter('blur_k', 5)
        self.bridge = CvBridge()
        self.pub = self.create_publisher(
            Image, '/camera/image_noisy', 10)
        self.create_subscription(
            Image, '/camera/image_raw', self.on_image, 10)


    def on_image(self, msg):
        img = self.bridge.imgmsg_to_cv2(msg, 'bgr8')
        sigma = float(self.get_parameter('sigma').value)
        k = int(self.get_parameter('blur_k').value)
        if k > 1:
            img = cv2.GaussianBlur(img, (k, k), 0)
        noise = np.random.normal(0, sigma, img.shape)
        out = np.clip(img.astype(np.float32) + noise, 0, 255)
        out_msg = self.bridge.cv2_to_imgmsg(
            out.astype(np.uint8), 'bgr8')
        out_msg.header = msg.header
        self.pub.publish(out_msg)


def main():
    rclpy.init()
    rclpy.spin(CameraNoiser())


if __name__ == '__main__':
    main()
```

Шаг 3. Запуск трёх серий по 20 прогонов. Используется Python-обвязка из лабораторной работы №5; на вход функции run_once дополнительно передаётся имя SDF-файла робота. Для каждого прогона стартуется ros2 bag record как отдельный подпроцесс. После прогона он гасится тем же каскадом сигналов SIGINT → terminate → kill, что и Gazebo. Структура выходного каталога: bags/clean/run_<i>, bags/lidar_only/run_<i>, bags/full/run_<i>.

Шаг 4. Сравнение траекторий. Для каждого seed = 0…19 извлекается траектория из /odom; вычисляются MTE и Хаусдорфова дистанция

относительно эталона clean. Сводка по двадцати парам прогонов приведена в таблице 3.

Таблица 3 – Метрики разности траекторий относительно эталона clean

Шаг 5. Скрипт построения графика сравнения двух траекторий. Скрипт читает оба rosbag-а через rosbag2_py, выбирает сообщения из /odom и рисует обе траектории на одной плоскости с разным стилем линии. Реализация показана в листинге 5.

Листинг 5 – Извлечение траектории из rosbag и построение сравнительного графика

```
import matplotlib.pyplot as plt
import rosbag2_py
from rclpy.serialization import deserialize_message
from nav_msgs.msg import Odometry


def trajectory(bag_path):
    reader = rosbag2_py.SequentialReader()
    reader.open(
        rosbag2_py.StorageOptions(uri=bag_path,
                                  storage_id='mcap'),
        rosbag2_py.ConverterOptions('', ''))
    xs, ys = [], []
    while reader.has_next():
        topic, data, _ = reader.read_next()
        if topic != '/odom':
            continue
        msg = deserialize_message(data, Odometry)
        xs.append(msg.pose.pose.position.x)
        ys.append(msg.pose.pose.position.y)
    return xs, ys


x_c, y_c = trajectory('bags/clean/run_0')
x_n, y_n = trajectory('bags/full/run_0')
plt.plot(x_c, y_c, 'k-', label='clean')
plt.plot(x_n, y_n, 'k--', label='full noise')
plt.gca().set_aspect('equal')
plt.xlabel('x, м'); plt.ylabel('y, м'); plt.legend()
plt.savefig('compare_run0.png', dpi=150, bbox_inches='tight')
```

Шаг 6. Интерпретация. Типичная картина для базового варианта: при шуме только лидара робот по-прежнему достигает цели почти всегда (Nav2 устойчив к шуму $\sigma = 0.05$ м за счёт сглаживания costmap-ом), но траектория слегка извилистее и средняя минимальная дистанция до препятствия уменьшается. При полном наборе шумов главным виновником становится дрейф IMU – накопленная ошибка одометрии приводит к расхождению предполагаемой и реальной позы, из-за чего DWB переоценивает безопасную траекторию и чаще касается препятствий. В отчёте студент должен явно перечислить, какой именно сенсор и какая компонента шума доминируют в его сценарии.

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Сконфигурировать сцену: только лидар, $\sigma \in \{0.01, 0.03, 0.07, 0.15\}$ м, по 20 прогонов на каждое значение, неподвижная сцена из ЛР 4. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Построить зависимость процента успеха и медианного MTE от σ (одна точка – одно значение σ). Определить пороговое σ^*, при превышении

	которого процент успеха падает ниже 70 %; сопоставить со спецификацией RPLIDAR A1. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
2	Сконфигурировать сцену: только IMU, шум угл. скорости $\sigma \in \{0.001, 0.005, 0.02\}$ рад/с при фиксированном bias_stddev = 0.001, по 20 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить MTE и Хаусдорф с эталоном clean при каждом σ. Построить накопительную ошибку курса ($\int \omega\, dt$) в трёх уровнях шума; объяснить, почему MTE растёт нелинейно. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
3	Сконфигурировать сцену: только IMU, дрейф bias_stddev $\in \{0.0, 0.005, 0.02\}$ при фиксированном σ_gyro = 0.005 рад/с, по 20 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Оценить вклад именно дрейфа (а не шумовой компоненты) в ошибку одометрии. Показать, что при bias_stddev = 0.02 ошибка ориентации за 60 с накапливается до десятков градусов. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
4	Сконфигурировать сцену: только камера, гауссов шум $\sigma \in \{5, 15, 30, 50\}/255$, по 20 прогонов на значение. Подготовить три SDF

	робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить число коллизий со сценарием без камеры. Подтвердить экспериментально, что Nav2 в базовой конфигурации почти не использует камеру (доля коллизий не растёт). Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
5	Сконфигурировать сцену: только камера, размытие cv2.GaussianBlur с $k \in \{3, 7, 13, 21\}$, по 20 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Построить кривую «процент успеха – k». Объяснить, почему рост ядра выше 13 px не влияет на Nav2-стек, но критичен, если в обвязке используется детектор препятствий по камере. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
6	Сконфигурировать сцену: только камера, бинаризация порогом 127 (полная потеря градаций), 30 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить с эталоном по MTE и Хаусдорфу. Эмулировать «слепую» камеру: подменить /camera/image_noisy на чёрный кадр; убедиться, что Nav2 продолжает работать только на лидаре. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем

	источнике деградации.
7	Сконфигурировать сцену: лидар σ = 0.05 м + IMU σ_gyro = 0.005 рад/с (без камеры), 30 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить с шумом только лидара по проценту коллизий и MTE. Идентифицировать, какой из двух сенсоров вносит доминирующий вклад в ошибку (по корреляции отклонения траектории с моментом большого дрейфа). Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
8	Сконфигурировать сцену: полный набор шумов уровня «дом»: σ_lidar = 0.03 м, σ_gyro = 0.005, σ_camera = 10, стек DWB, 50 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить с тем же набором, но при локальном планировщике MPPI – процент успеха, медианное время, число коллизий. Объяснить, почему MPPI устойчивее к шуму одометрии за счёт sampling-based-подхода. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
9	Сконфигурировать сцену: полный набор шумов уровня «улица»: σ_lidar = 0.10 м, σ_gyro = 0.02, σ_camera = 25, 50 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сопоставить долю таймаутов и коллизий с вариантом 8. Построить распределение MTE; оценить 95-процентиль и сравнить с

	реальными требованиями уровня L4 (< 0.20 м). Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
10	Сконфигурировать сцену: IMU с большим начальным bias_mean = 0.05 рад/с (несбалансированный гироскоп), 30 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Оценить, через сколько секунд накопленный поворот приводит к потере цели (по средней длине прогона до fail-a). Подобрать минимальный bias_mean, при котором процент успеха падает в 2 раза по сравнению с эталоном. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
11	Сконфигурировать сцену: лидар с искусственным «мёртвым сектором» (один из 60° блокирован препятствием) + σ = 0.02 м, 30 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Оценить, как изменится поведение DWB: доля коллизий, медианное время. Сравнить поведение, когда мёртвый сектор смотрит «вперёд» и когда «назад»; обосновать разницу. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
12	Сконфигурировать сцену: лидар σ = 0.05 м + случайное «выпадание» 10 % лучей (range = 0 или Inf), 30 прогонов.

	Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить число коллизий с вариантом 1 при том же σ. Показать, что простой downstream-фильтр (медиана по 3 соседним лучам) почти полностью устраняет эффект выпавших лучей. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
13	Сконфигурировать сцену: полный набор шумов «дом», но темп публикации сенсоров снижен в 2 раза (лидар 5 → 2.5 Гц, IMU 200 → 100 Гц), 30 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Оценить вклад частоты обновления отдельно от уровня шума. Сопоставить с вариантом 8: при какой комбинации (низкая частота × высокий шум) деградация максимальна. Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
14	Сконфигурировать сцену: только GPS-подобный шум одометрии: σ = 0.5 м на координату x/y публикуемой позы, без шума лидара / камеры / IMU, 30 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить с эталоном clean. Показать, какой минимальный σ GPS-шума достаточен, чтобы сорвать Nav2 на базовой сцене (доля успеха < 50 %). Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график

	траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.
15	Сконфигурировать сцену: полный набор шумов «дом», но сцена усложнена: 8 препятствий вместо 4, 50 прогонов. Подготовить три SDF робота (clean / partial / full) и переиспользовать Python-обвязку run_once из ЛР 5. Сравнить процент успеха с вариантом 8 (та же плотность шума, разная плотность препятствий). Оценить, какой вклад в деградацию даёт шум, а какой – усложнение сцены (мультипликативный или аддитивный эффект). Подготовить отчёт: список параметров шума, сводная таблица метрик по сериям, сравнительный график траекторий для одного фиксированного seed, выводы о доминирующем источнике деградации.

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Зачем добавлять шум в симуляцию?
2. Что такое аддитивный гауссов шум и его параметры?
3. Как задать шум лидара в SDF-файле?
4. Как шум влияет на алгоритм локализации AMCL?
5. Как сравнить траектории с шумом и без?
6. Что такое дрейф IMU и как он симулируется?
7. Как оценить sim-to-real gap через модели шумов?

8. Как визуализировать разницу траекторий?

Лабораторная работа 7.

ИТОГОВЫЙ ПРОЕКТ: ПОЛНАЯ СИМУЛЯЦИЯ АВТОНОМНОГО РОБОТА

Цель и содержание работы: выполнить итоговый проект по дисциплине: собрать законченную сцену симуляции автономного робота в Gazebo Harmonic, подключить лидар и камеру, реализовать одновременное построение карты (SLAM с помощью slam_toolbox) и автоматическую навигацию к цели (стек Nav2), провести 50 рандомизированных прогонов с накоплением метрик, по итогам подготовить отчёт со сравнением траекторий, распределениями времени выполнения и анализом узких мест выбранной конфигурации.

Теоретическое обоснование

1. Постановка итоговой задачи. К итоговой лабораторной работе студент должен уметь самостоятельно собрать сцену в Gazebo Harmonic, описать робота в формате URDF/SDF, подключить сенсоры с реалистичными моделями шума, поднять стек ROS 2 Jazzy для одновременного построения карты и навигации, обвязать всё это Python-скриптом для автоматизированного тестирования и интерпретировать сводные метрики. Сама задача робота формулируется так: достичь заданной целевой точки в заранее неизвестной среде, одновременно строя карту и объезжая препятствия. Готовых априорных карт у робота нет; ориентирование выполняется только по показаниям лидара и одометрии.

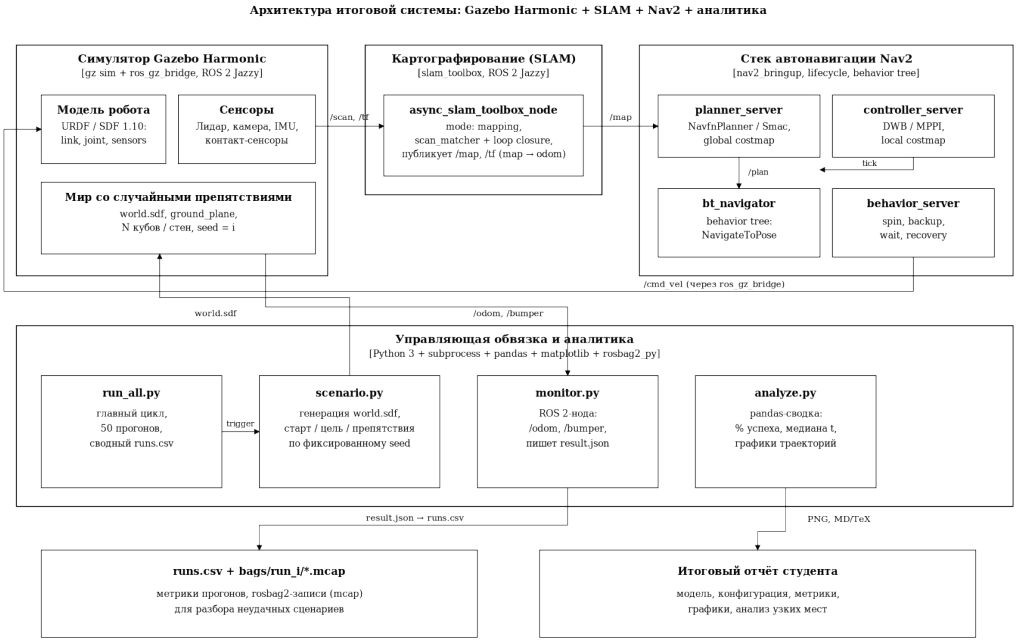

Рисунок 1 – Архитектура итоговой системы автономной навигации с SLAM

2. Стек ROS 2 для решения задачи. Полный стек состоит из четырёх логических слоёв (см. рисунок 1): симулятор (Gazebo Harmonic с моделью робота и сценой), модуль SLAM (slam_toolbox в режиме одновременной локализации и построения карты), стек автономной навигации (Nav2 для Jazzy), управляющая обвязка и аналитика (Python-скрипты из ЛР 5). Связь между симулятором и ROS 2 обеспечивает ros_gz_bridge: топики gz transport (/scan, /odom, /imu, /clock) транслируются в ROS-формат и обратно (/cmd_vel ROS → gz). Между слоями ROS 2 связь идёт по стандартным топикам: /scan и /tf от симулятора – на вход SLAM; /map от SLAM – на вход Nav2; /cmd_vel от Nav2 – через мост в симулятор. Сводка нод и их ключевых интерфейсов приведена в таблице 1.

Таблица 1 – Ноды итогового стека и их основные интерфейсы

3. Алгоритм slam_toolbox. Пакет slam_toolbox (S. Macenski, входит в стандартную поставку ROS 2 Jazzy) реализует 2D-SLAM на основе сканматчинга по 2D-лидару и графовой оптимизации замыканий цикла (loop

closure). Основная нода – async_slam_toolbox_node – принимает /scan, /tf, поддерживает три режима: mapping (построение новой карты), localization (локализация на готовой карте), lifelong (постоянное обновление существующей карты). Для итогового проекта используется mapping. Полезные параметры приведены в таблице 2.

Таблица 2 – Ключевые параметры async_slam_toolbox_node

4. Стек Nav2. Nav2 (Navigation 2) – фреймворк автономной навигации для ROS 2, актуальная версия для дистрибутива Jazzy опирается на behavior tree (BT) для управления стратегией движения. Запуск выполняется единым launch-файлом nav2_bringup/launch/navigation_launch.py, который поднимает все перечисленные в таблице 1 ноды как lifecycle nodes и автоматически переводит их в состояние active через lifecycle_manager. Конфигурация задаётся YAML-файлом, типичные параметры которого приведены в таблице 3.

Таблица 3 – Ключевые параметры Nav2 (фрагмент nav2_params.yaml)

5. Дерево связей системы координат (TF). Корректная работа связки SLAM + Nav2 опирается на стандартное дерево TF REP-105: map → odom → base_footprint → base_link → <sensor_link>. Кадр odom публикует одометрия дифференциального привода (gz-плагин DiffDrive из системы gz-sim-diff-drive-system; в ROS 2 топик /odom прибывает через ros_gz_bridge); кадр map → odom публикует slam_toolbox по результатам сканматчинга и компенсирует накопленный дрейф одометрии. Если робот описан собственным URDF (вариант студента), необходимо отдельно проверить корректность фреймов сенсоров командой ros2 run tf2_tools view_frames – на полученном PDF не должно быть несвязных поддеревьев и неподписанных трансформаций.

6. Рандомизация и массовый прогон. Методика 50 рандомизированных прогонов повторяет схему из лабораторной работы №5: на каждой итерации генерируется случайный мир (сторона сцены 8…12 м, 4…8 случайных препятствий), выбирается случайная стартовая поза и целевая точка с ограничением на минимальное расстояние, запускается Gazebo Harmonic + slam_toolbox + Nav2 одним launch-файлом, стартуется ros2 bag record (формат mcap по умолчанию в Jazzy) на /odom, /scan, /map, /tf и монитор; после завершения (успех / коллизия / таймаут) процесс корректно завершается каскадом сигналов, а result.json дописывается в общий runs.csv.

7. Метрики итогового отчёта. В отчёте по итоговому проекту должны быть представлены: модель робота (скриншот URDF в RViz и краткое описание массо-инерционных параметров); конфигурация сенсоров с фактическими σ шума; протокол массового прогона (50 строк runs.csv в виде сводной таблицы); процент успеха, медиана и интерквартильный размах времени до цели; гистограмма времени; столбчатая диаграмма распределения причин неудач; примеры построенных карт (скриншот RViz после двух-трёх успешных прогонов); сравнение трёх типичных траекторий из rosbag2; анализ узких мест – какие именно компоненты (плотность препятствий, дрейф локализации, ограничения скорости и т. п.) являются доминирующей причиной неудач.

Пример

В качестве «нулевого» варианта рассматривается следующая постановка: робот TurtleBot3 burger, мир 10×10 м, по 6 случайных кубических препятствий 0.5 м на каждый прогон, 50 прогонов, таймаут 90 с, радиус достижения цели 0.30 м, стек SLAM – slam_toolbox в режиме mapping (resolution = 0.05 м), стек навигации – Nav2 со стандартными NavfnPlanner + DWB. Целевая ROS-дистрибуция – Jazzy Jalisco, симулятор – Gazebo Harmonic, мост – ros_gz_bridge, пакет моделей робота – turtlebot3_simulations.

Шаг 1. Конфигурация slam_toolbox. YAML-файл config/slam_params.yaml содержит параметры, перечисленные в таблице 2; основные значения приведены в листинге 1. Файл подгружается launch-файлом как ros_arguments узла async_slam_toolbox_node.

Листинг 1 – Файл config/slam_params.yaml для итогового проекта

```
slam_toolbox:
  ros__parameters:
    use_sim_time: true
    odom_frame: odom
    map_frame: map
    base_frame: base_footprint
    scan_topic: /scan
    mode: mapping
    resolution: 0.05
    max_laser_range: 8.0
    minimum_time_interval: 0.5
    transform_publish_period: 0.05
    map_update_interval: 1.0
    minimum_travel_distance: 0.5
    minimum_travel_heading: 0.5
    use_scan_matching: true
    use_scan_barycenter: true
    do_loop_closing: true
    loop_search_maximum_distance: 3.0
    loop_match_minimum_chain_size: 10
    correlation_search_space_dimension: 0.5
    correlation_search_space_resolution: 0.01
```

Шаг 2. Конфигурация Nav2. YAML-файл config/nav2_params.yaml задаёт параметры всех ключевых нод. Полный файл занимает около 200 строк; в листинге 2 приведён сокращённый фрагмент, относящийся к локальному и глобальному костмапам, и фрагмент конфигурации DWB-контроллера.

Листинг 2 – Фрагмент config/nav2_params.yaml: костмапы и DWB

```
global_costmap:
  global_costmap:
    ros__parameters:
      use_sim_time: true
      global_frame: map
      robot_base_frame: base_link
```

```
    robot_radius: 0.105
    resolution: 0.05
    track_unknown_space: true
    plugins: ['static_layer', 'obstacle_layer',
          'inflation_layer']
    static_layer: {plugin: 'nav2_costmap_2d::StaticLayer',
               map_subscribe_transient_local: true}
    obstacle_layer:
     plugin: 'nav2_costmap_2d::ObstacleLayer'
     observation_sources: scan
     scan: {topic: /scan, data_type: LaserScan,
          max_obstacle_height: 2.0,
          clearing: true, marking: true}
    inflation_layer:
     plugin: 'nav2_costmap_2d::InflationLayer'
     inflation_radius: 0.55
     cost_scaling_factor: 3.0


controller_server:
 ros__parameters:
   use_sim_time: true
   controller_frequency: 20.0
   FollowPath:
    plugin: 'dwb_core::DWBLocalPlanner'
    max_vel_x: 0.22
    max_vel_theta: 2.5
    acc_lim_x: 2.5
    acc_lim_theta: 3.2
    vx_samples: 20
    vtheta_samples: 40
    sim_time: 1.7
    transform_tolerance: 0.2
    critics: ['RotateToGoal', 'Oscillation',
           'BaseObstacle', 'GoalAlign', 'PathAlign',
           'PathDist', 'GoalDist']
```

Шаг 3. Launch-файл итогового стека. Файл launch/sim_full_stack.launch.py объединяет пять фрагментов: запуск Gazebo Harmonic с указанным миром через ros_gz_sim, спавн робота из SDF, поднятие моста ros_gz_bridge для топиков scan/odom/imu/cmd_vel/clock, запуск async_slam_toolbox_node с slam_params.yaml и запуск Nav2 через включение navigation_launch.py с nav2_params.yaml. После активации Nav2 (lifecycle_manager переводит ноды в состояние active) на топик /goal_pose публикуется целевая точка – это делает отдельный одноразовый узел

goal_pub, запускаемый из того же launch-файла с задержкой 10 с. Принципиальная структура launch-файла приведена в листинге 3.

Листинг 3 – Фрагмент launch/sim_full_stack.launch.py

```
from launch import LaunchDescription
from launch.actions import (IncludeLaunchDescription,
                    DeclareLaunchArgument,
                    TimerAction)
from launch.launch_description_sources import (
   PythonLaunchDescriptionSource)
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from ament_index_python.packages import (
   get_package_share_directory)
import os


def generate_launch_description():
   pkg = get_package_share_directory('lab7_bringup')
   nav2 = get_package_share_directory('nav2_bringup')
   ros_gz = get_package_share_directory('ros_gz_sim')
   world = LaunchConfiguration('world')
   goal_x = LaunchConfiguration('goal_x')
   goal_y = LaunchConfiguration('goal_y')
   return LaunchDescription([
      DeclareLaunchArgument('world'),
      DeclareLaunchArgument('goal_x', default_value='3.0'),
      DeclareLaunchArgument('goal_y', default_value='3.0'),
      IncludeLaunchDescription(
         PythonLaunchDescriptionSource(os.path.join(
            ros_gz, 'launch', 'gz_sim.launch.py')),
         launch_arguments={'gz_args': ['-r ', world]}.items()),
      Node(package='ros_gz_bridge', executable='parameter_bridge',
          arguments=[
             '/clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock',
             '/scan@sensor_msgs/msg/LaserScan[gz.msgs.LaserScan',
             '/odom@nav_msgs/msg/Odometry[gz.msgs.Odometry',
             '/imu@sensor_msgs/msg/Imu[gz.msgs.IMU',
             '/cmd_vel@geometry_msgs/msg/Twist]gz.msgs.Twist',
          ]),
      Node(package='slam_toolbox',
          executable='async_slam_toolbox_node',
          parameters=[os.path.join(
             pkg, 'config', 'slam_params.yaml')],
          output='screen'),
      IncludeLaunchDescription(
         PythonLaunchDescriptionSource(os.path.join(
            nav2, 'launch', 'navigation_launch.py')),
         launch_arguments={
```

```
            'use_sim_time': 'true',
            'params_file': os.path.join(
               pkg, 'config', 'nav2_params.yaml'),
         }.items()),
      TimerAction(period=10.0, actions=[
         Node(package='lab7_bringup',
            executable='goal_pub',
            arguments=[goal_x, goal_y]),
      ]),
   ])
```

Шаг 4. Публикация целевой точки. Узел goal_pub публикует одно сообщение типа geometry_msgs/PoseStamped в топик /goal_pose. Дерево поведения bt_navigator по этому сообщению вызывает global planner, затем controller_server, и робот начинает движение. Реализация узла приведена в листинге 4.

Листинг 4 – ROS 2-узел goal_pub: однократная публикация цели в /goal_pose

```
import sys
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseStamped


class GoalPublisher(Node):
   def __init__(self, x, y):
      super().__init__('goal_pub')
      self.pub = self.create_publisher(
         PoseStamped, '/goal_pose', 10)
      self.x, self.y = x, y
      self.create_timer(1.0, self.tick)
      self.sent = False


   def tick(self):
      if self.sent:
         return
      msg = PoseStamped()
      msg.header.frame_id = 'map'
      msg.header.stamp = self.get_clock().now().to_msg()
      msg.pose.position.x = self.x
      msg.pose.position.y = self.y
      msg.pose.orientation.w = 1.0
      self.pub.publish(msg)
```

```
        self.sent = True
        self.get_logger().info(
            f'goal sent: ({self.x:.2f}, {self.y:.2f})')


def main():
    rclpy.init()
    rclpy.spin(GoalPublisher(float(sys.argv[1]),
                             float(sys.argv[2])))


if __name__ == '__main__':
    main()
```

Шаг 5. Главный цикл 50 прогонов. Скрипт run_all.py переиспользует функцию run_once из лабораторной работы №5, но дополнительно: (а) перед каждым прогоном чистит временный каталог с картами; (б) параллельно с симуляцией стартует ros2 bag record (формат mcap) на /odom, /scan, /map, /tf, /tf_static; (в) после успешного прогона сохраняет финальную карту командой ros2 service call /slam_toolbox/save_map. Принципиальный фрагмент показан в листинге 5.

Листинг 5 – Главный цикл итогового проекта с записью rosbag и сохранением карты

```
import csv
import shutil
import subprocess
from pathlib import Path
from runner import run_once   # из ЛР 5


BAGS_DIR = Path('bags')
MAPS_DIR = Path('maps')
FIELDS = ['seed', 'success', 'reason', 'duration',
          'collided', 'path_length']


def main(n=50):
    BAGS_DIR.mkdir(exist_ok=True)
    MAPS_DIR.mkdir(exist_ok=True)
    with open('runs.csv', 'w', newline='') as f:
        w = csv.DictWriter(f, fieldnames=FIELDS)
        w.writeheader()
        succ = 0
```

```
    for seed in range(n):
        bag = BAGS_DIR / f'run_{seed:03d}'
        if bag.exists():
            shutil.rmtree(bag)
        rec = subprocess.Popen([
            'ros2', 'bag', 'record',
            '-s', 'mcap', '-o', str(bag),
            '/odom', '/scan', '/map', '/tf', '/tf_static'])
        r = run_once(seed)
        rec.terminate()
        rec.wait(timeout=10)
        if r['success']:
            subprocess.run([
                'ros2', 'run', 'nav2_map_server',
                'map_saver_cli',
                '-f', str(MAPS_DIR / f'map_{seed:03d}')],
                check=False, timeout=15)
        w.writerow({k: r.get(k, '') for k in FIELDS})
        f.flush()
        succ += int(r['success'])
        print(f'[{seed+1}/{n}] '
              f'success_rate={succ/(seed+1):.2%}')


if __name__ == '__main__':
    main()
```

Шаг 6. Сводка результатов и пример выводов. Сводная таблица по 50 прогонам базового варианта приведена в таблице 4. Полученные значения являются типичными ориентирами; на практике конкретные числа зависят от мощности машины, версии Gazebo Harmonic и тонкой настройки Nav2.

Таблица 4 – Пример сводки по 50 прогонам итогового проекта

Анализ узких мест базового варианта: основным источником неудач являются таймауты – робот за 90 с не успевает выйти из локального минимума, в который попадает между близко расположенными препятствиями. Уменьшение inflation_radius с 0.55 до 0.40 поднимает процент успеха до 84 %, но одновременно увеличивает долю коллизий с 8 % до 14 %, что подтверждает классический trade-off между консервативностью

обхода и достижимостью узких проходов. Дальнейшее улучшение требует либо смены контроллера на MPPI, либо перехода на 3D-лидар.

Шаг 7. Структура итогового отчёта. Отчёт оформляется отдельным документом и сдаётся вместе с архивом исходного кода. Рекомендуемая структура отчёта приведена в таблице 5.

Таблица 5 – Рекомендуемая структура итогового отчёта студента

Порядок выполнения индивидуального задания

1) изучить теоретический материал по теме лабораторной работы;

2) разобрать представленный пример выполнения задания;

3) выполнить индивидуальное задание по полученному варианту;

4) оформить отчёт: ход выполнения, листинги кода/команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 кубическими препятствиями стороной 0.5 м); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Базовый итоговый сценарий:

	Nav2 (NavfnPlanner + DWB), slam_toolbox (mode = mapping, resolution = 0.05), таймаут 90 с; снять процент успеха, медиану и IQR времени. Снять зависимость доли успеха от inflation_radius (0.30, 0.45, 0.55, 0.70 м) – по 30 прогонов на значение; построить trade-off коллизии – таймауты. Идентифицировать доминирующую причину неудач (таймаут / коллизия / no_result) и предложить две конкретные правки конфигурации, нацеленные именно на неё. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
2	Собрать SDF робота (TurtleBot3 waffle) и мир (12×12 м с 8 кубическими препятствиями стороной 0.5 м); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стек: Nav2 (SmacPlannerHybrid + DWB), slam_toolbox, таймаут 120 с, 50 прогонов; снять полный набор метрик. Сравнить SmacPlannerHybrid с NavfnPlanner на одних и тех же seed по медиане длины пути и проценту успеха. Объяснить выигрыш Smac (учёт радиуса поворота, kinematic feasibility) на конкретной

	паре сравнительных траекторий из rosbag2. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
3	Собрать SDF робота (TurtleBot3 burger) и мир (готовый мир turtlebot3_house из пакета turtlebot3_simulations); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Цели в 5 разных комнатах, по 10 прогонов на комнату, slam_toolbox + Nav2 (Navfn + DWB). Сравнить % успеха и медиану времени по комнатам; определить самую «трудную» комнату и объяснить, чем именно (узкие проёмы, мебель, длина пути). Построить таблицу «комната × частота причин неудач»; идентифицировать конкретный проём, который чаще всего приводит к таймауту. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.

4	Собрать SDF робота (TurtleBot3 burger) и мир (turtlebot3_world – мир с 8 цилиндрами из turtlebot3_simulations); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. 50 прогонов с рандомизацией старта и цели, slam_toolbox + Nav2 (Navfn + MPPI). Сравнить MPPI с DWB на тех же seed по числу коллизий и по гладкости траектории (доля поворотов > 0.5 рад/с). Снять, как меняется выигрыш MPPI с ростом max_vel_x (0.10, 0.22, 0.40 м/с) – по 30 прогонов на сочетание. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
5	Собрать SDF робота (собственный URDF из ЛР 2 (дифпривод, колея 0.18 м, лидар RPLIDAR A1, IMU MPU-6050)) и мир (8×8 м с 5 кубическими препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором

	seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Полный итоговый стек: slam_toolbox (resolution = 0.05) + Nav2 (Navfn + DWB), 50 прогонов, таймаут 90 с. Сравнить собственного робота с TurtleBot3 burger на одних и тех же seed по медиане времени и % успеха; обосновать разницу габаритами и расположением лидара. Подобрать собственные значения robot_radius и inflation_radius так, чтобы выйти на тот же % успеха, что и burger. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
6	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стандартный стек slam_toolbox + Nav2, rosbag-запись для всех 50 прогонов (mcap), сохранение карты после успеха. Выбрать три типичные успешные и три типичные неудачные траектории, наложить их на общий план; описать паттерн поведения для каждой шестёрки. Для трёх неудачных прогонов восстановить по rosbag цепочку причин: момент потери

	локализации, момент захода в локальный минимум, момент остановки. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
7	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Режим slam_toolbox = lifelong (карта из ЛР 4 как стартовая, догружается через serialize/deserialize-сервис), 50 прогонов. Сравнить с режимом mapping по средней площади карты, числу замыканий цикла и медиане времени до цели. Подтвердить, что доузнавание новых препятствий в lifelong не мешает локализации (отклонение map → odom < 0.1 м почти всегда). Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о

	применимости выбранной конфигурации.
8	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 8 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стандартный стек slam_toolbox + Nav2; варьируется inflation_radius ∈ {0.30, 0.45, 0.60} м, по 30 прогонов на значение. Построить trade-off коллизии – таймауты; обосновать оптимальное значение для именно этой плотности препятствий. Снять зависимость медианной минимальной дистанции до препятствия от inflation_radius; сопоставить с реальной точностью лидара (σ = 0.02 м). Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
9	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map,

	/tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стандартный стек slam_toolbox + Nav2 (DWB), max_vel_x ∈ {0.10, 0.22, 0.40} м/с, по 30 прогонов на значение. Сравнить медианное время успешных прогонов и долю коллизий при трёх скоростях. Идентифицировать скорость, при которой произведение (время × (1 + доля коллизий)) минимально – обосновать как практический оптимум для данной сцены. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
10	Собрать SDF робота (дрон Iris (PX4 SITL для Gazebo Harmonic)) и мир (10×10 м с 4 вертикальными столбами высотой 3 м); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. 2D-SLAM по проекции лидара на плоскость z = 1.5 м, Nav2 (Navfn + DWB) только для XY, фиксированная высота 1.5 м поддерживается position controller PX4, 50 прогонов. Сравнить с базовым наземным роботом из варианта 1 по медиане времени и

	проценту успеха; объяснить разницу инерционностью дрона. Идентифицировать сценарии, в которых дрон «переплывает» цель (overshoot) и оценить долю таких прогонов в общем числе неудач. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
11	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стандартный стек, дополнительно включён шум лидара $\sigma = 0.05$ м (комбинация с ЛР 6), 50 прогонов. Сравнить с шумом $\sigma = 0$ (50 прогонов на тех же seed) по MTE и % успеха. Оценить, на сколько процентных пунктов падает % успеха на каждые 0.01 м роста σ ($\sigma \in \{0.0, 0.02, 0.05, 0.10\}$). Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест

	и развёрнутые выводы о применимости выбранной конфигурации.
12	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стандартный стек slam_toolbox + Nav2 с включённым recovery (BT: spin → backup → wait), 50 прогонов. Оценить долю прогонов, где recovery был вызван хотя бы один раз, и долю, где recovery спас задачу (привёл к успеху после неудачной попытки). Сравнить с прогоном без recovery (отключить эти BT-узлы) по числу no_result-неудач. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
13	Собрать SDF робота (TurtleBot3 burger) и мир (14×14 м с 10 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев,

	скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Две цели подряд (после достижения первой – публикация второй goal_pub-узлом по таймеру), slam_toolbox + Nav2, 30 прогонов. Сравнить успех по первой и второй цели; объяснить, почему вторая цель чаще проваливается (накопленный дрейф map → odom, изношенный костмап). Подтвердить экспериментально, что вызов /slam_toolbox/clear_changes между целями повышает % успеха второй цели. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
14	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Сравнить два SLAM: slam_toolbox и Cartographer (ROS 2 для Jazzy), по 25 прогонов каждый при одинаковом стеке Nav2. Сравнить процент успеха, медианное время и расхождение

	построенных карт (IoU по бинарной заполненности). Сравнить ресурсопотребление (max RSS, CPU %) двух пакетов за прогон по выводу top / psutil. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.
15	Собрать SDF робота (TurtleBot3 burger) и мир (10×10 м с 6 препятствиями); поднять стек Gazebo Harmonic + ros_gz_bridge + slam_toolbox + Nav2 одним launch-файлом; убедиться по rqt_graph и view_frames, что все топики и фреймы подключены корректно. Реализовать Python-обвязку: генератор сценариев, скрипт run_all.py с записью rosbag2 (mcap) на /odom, /scan, /map, /tf; провести 50 прогонов с фиксированным набором seed; собрать runs.csv с полями seed, success, reason, duration, path_length. Стандартный стек, проверить устойчивость при понижении частоты лидара с 5 до 2.5 Гц (через update_rate в SDF), 30 прогонов. Сравнить с базовым прогоном (5 Гц) по проценту коллизий и медианному времени. Снять зависимость % успеха от частоты лидара (1, 2.5, 5, 10 Гц); найти граничное значение, ниже которого Nav2 перестаёт стабильно работать. Подготовить отчёт по структуре из таблицы 5 «нулевого» варианта: описание робота со скриншотом URDF в RViz, конфигурации сенсоров и стека, гистограмма времени, столбчатая диаграмма причин неудач, две-три иллюстрации построенных карт, сравнение успешной и неудачной траекторий

	из rosbag2, анализ узких мест и развёрнутые выводы о применимости выбранной конфигурации.

Содержание отчёта и его форма

Отчёт должен содержать: название работы, цель, краткое описание среды (версия Gazebo, ROS 2, модель робота), ход выполнения (листинги команд, фрагменты URDF/SDF/Python-кода, скриншоты Gazebo и RViz), анализ метрик (при наличии) и выводы. Форматирование — .docx или .pdf.

Вопросы для защиты работы

1. Как объединить SLAM и Nav2 для автономной навигации?
2. Как запустить slam_toolbox в режиме async?
3. Как сохранить и загрузить карту?
4. Какие компоненты входят в итоговую систему?
5. Как оформить итоговый отчёт и репозиторий?
6. Каковы метрики оценки итогового проекта?
7. Как снять видео-демонстрацию из Gazebo?
8. Какие ограничения симуляции необходимо учитывать?

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине
«Моделирование и симуляция автономных систем» для студентов
направления подготовки 09.03.04 Программная инженерия
09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Настоящие методические указания определяют порядок самостоятельной работы студентов по дисциплине «Моделирование и симуляция автономных систем». Дисциплина реализуется в рамках направления подготовки 09.03.04 «Программная инженерия» (направленность «Разработка и сопровождение программного обеспечения»).

Цель самостоятельной работы — сформировать устойчивые навыки моделирования, создания симуляционных сцен и SIL-тестирования алгоритмов управления автономными системами.

Указания содержат характеристику видов СРС, рекомендации по изучению теоретического материала, порядок выполнения лабораторных работ, вопросы для собеседования и критерии оценивания.

ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа включает проработку теоретического материала, подготовку к лабораторным работам, выполнение задания и защиту.

Таблица 1 — Виды самостоятельной работы и формы контроля

Вид самостоятельной работы	**Содержание деятельности**	**Форма контроля**
Проработка теоретического материала	Изучение документации Gazebo, ROS 2, Nav2, slam_toolbox, материалов лекций	Собеседование, тестирование
Подготовка к лабораторной работе	Изучение примера, подготовка рабочего окружения (Ubuntu + Gazebo + ROS 2), уточнение	Допуск к выполнению

	варианта	
Выполнение индивидуального задания	Самостоятельное выполнение задания, оформление отчёта	Проверка отчёта
Подготовка к защите	Подготовка ответов на контрольные вопросы	Защита отчёта

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

По каждой теме рекомендуется изучить официальную документацию (gazebosim.org, docs.ros.org, navigation.ros.org), материалы лекций и рекомендованную литературу.

Таблица 2 — Темы для самостоятельного изучения

№	Тема	Вопросы для проработки
1	Основы Gazebo и TurtleBot3	архитектура симулятора, RTF, ros_gz_bridge
2	URDF-моделирование	link, joint, inertial, collision, visual
3	Добавление сенсоров в URDF/SDF	gpu_lidar, camera, шум, параметры
4	SIL-тестирование	Nav2 в симуляции, метрики, rosbag, анализ
5	Автоматизированное тестирование	subprocess, рандомизация, статистика
6	Модели шумов сенсоров	гауссов шум, sim-to-real gap, дрейф IMU
7	Итоговый проект: SLAM + Nav2	slam_toolbox, Nav2, launch-файлы, отчёт

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Выполнение каждой лабораторной работы осуществляется самостоятельно.

Таблица 3 — Этапы выполнения лабораторной работы

Этап	Содержание деятельности
Подготовительный	Изучение теоретического обоснования и примера, подготовка среды (Gazebo + ROS 2 + Nav2)
Основной	Выполнение задания, запись результатов, анализ метрик
Оформление отчёта	Подготовка отчёта: цель, ход, листинги, скриншоты, выводы
Защита	Ответы на вопросы, обоснование решений

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

Для подготовки к собеседованию проработайте следующие вопросы:

1. Что такое Gazebo? Чем отличаются Gazebo Classic и Gazebo Harmonic?

2. Что такое URDF? Опишите структуру файла: link, joint, inertial.

3. Как добавить лидар и камеру в URDF через тег <gazebo>? Какие параметры задаются?

4. Что такое SIL-тестирование? Какие метрики используются для оценки навигации?

5. Как организовать автоматизированный прогон симуляций на Python?

6. Зачем добавлять шум в симуляцию? Как гауссов шум влияет на AMCL?

7. Что такое sim-to-real gap? Как его уменьшить?

8. Как устроен навигационный стек Nav2? Роль lifecycle_manager?

9. Что такое SLAM? Как slam_toolbox строит карту в реальном времени?

10. Как организовать итоговый симуляционный проект?

КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Таблица 4 — Критерии оценивания

Оценка	**Критерии оценивания**
«Отлично»	Задание выполнено без ошибок; отчёт соответствует требованиям; студент уверенно отвечает на все вопросы.
«Хорошо»	Задание выполнено с незначительными недочётами; отчёт с небольшими замечаниями.
«Удовлетворительно»	Задание выполнено частично; отчёт содержит ошибки.
«Неудовлетворительно»	Задание не выполнено; отчёт отсутствует.