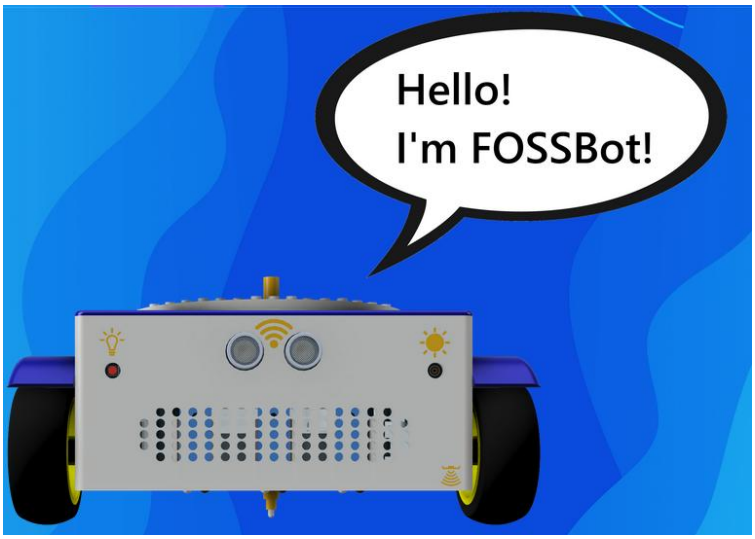
S.T.E.P.S.
Robotics and Education

STEPS for Pre-Service and In-Service trainers Pedagogical Manual



Contents

PART A – Introduction to STEPS

What is the STEPS Project?

STEPS stands for STEM Teaching and Education Platform for Students. It is a European Erasmus+ initiative designed to support teachers in delivering innovative, hands-on STEM education using educational robotics.

At its core, the project promotes engaging, inclusive, and cross-curricular learning experiences by combining:

- The FOSSBot — a low-cost, open-source physical robot, and
- CoppeliaSim — a virtual simulation platform that allows programming a digital version of the robot (a digital twin).

The STEPS project empowers educators to:

- Introduce robotics and programming in the classroom,
- Address curriculum goals through interactive activities,
- Use both physical and virtual tools, ensuring flexibility and equity,
- Design lessons that are age-appropriate, accessible, and interdisciplinary.

Objectives for Teachers:

- Equip students (ages 5–18) with 21st-century STEM skills
- Foster computational thinking, creativity, and collaboration
- Provide step-by-step teaching resources across education levels
- Promote the use of open educational technologies
- Encourage inclusive practices based on Universal Design for Learning (UDL)

Whether you are working with young learners or advanced students, with access to real robots or not, STEPS offers a comprehensive and adaptable platform for bringing robotics into the classroom — not as an add-on, but as a meaningful and motivating way to teach science, mathematics, technology, and more.

What is FOSSBot and why use it in education (Open-source educational robot)?

FOSSBot is an open-source educational robot designed specifically for classroom use. The name “FOSSBot” comes from Free and Open-Source Software + Robot, reflecting its core values: transparency, adaptability, and accessibility.

Built with 3D-printable parts and programmable using free tools, FOSSBot allows teachers and students to explore robotics and programming without expensive

proprietary kits. It supports both block-based and Python programming, making it suitable for different age groups and experience levels.

Key Features:

- Modular, 3D-printed chassis – customizable and repairable
- Affordable hardware – compatible with Raspberry Pi and simple sensors
- Visual and text programming options – adaptable for beginners and advanced learners
- Digital twin in CoppeliaSim – allows simulation of the robot for virtual learning
- Cross-platform software – runs on Linux and open educational environments

FOSSBot is not just a robot—it's a pedagogical tool that helps teachers:

- Teach programming concepts through real-world interaction
- Encourage problem solving and design thinking
- Support collaborative, project-based learning
- Bridge theory and practice in STEM subjects
- Create inclusive and equitable learning environments (even without physical robots, thanks to simulation)

Because it is open-source, educators are free to modify, extend, and share activities and code, fostering a culture of collaboration and innovation. Students learn not only how robots work, but also how to question, improve, and adapt technology — key skills for the future.

Picture 1. Physical FOSSBot

FOSSBot embodies the maker spirit: accessible, hands-on, creative, and empowering. It invites both teachers and students to explore how technology can serve learning, not just as a tool, but as a medium for discovery.

Why combine physical and digital robots?

In the STEPS project, teachers are encouraged to use both the physical FOSSBot and its virtual counterpart in CoppeliaSim. ***They can write and test their programs in CoppeliaSim and then directly port them to FOSSBot for use in the real world, with the actual robot.*** This blended approach combines the best of hands-on experimentation with the flexibility of simulation — making robotics education more inclusive, scalable, and effective.

Physical Robots: Tangible, Engaging, Authentic

Working with a real FOSSBot in the classroom allows students to:

- See immediate real-world effects of their code
- Practice fine motor skills, spatial reasoning, and sensor interaction
- Engage in team-based projects, building and testing together
- Experience the excitement of building something that moves!

However, physical robots may be limited by cost, availability, or time constraints — as well as by technical restrictions such as maintenance needs, lack of 3D printers, or insufficient access to assembly tools and spare parts, especially in large or under-resourced classrooms.

Digital Robots: Accessible, Flexible, Powerful

Using Virtual Environments, students can program a digital version of FOSSBot in a safe, repeatable virtual environment. This allows them to:

- Experiment without hardware damage or time limits
- Test ideas faster and iterate more easily
- Learn from home or in remote/hybrid settings
- Visualize complex sensor data and robot behavior in 3D

Educational Benefits of Blending Both:

- Supports differentiation: students can learn the same concepts in multiple ways
- Bridges abstract coding with concrete experience
- Promotes equity: all learners can participate, regardless of hardware availability
- Builds debugging and design skills in both real and simulated settings

By moving fluidly between real and virtual robots, teachers can offer more inclusive, engaging, and resilient lessons. It's not a question of choosing one or the other — it's about creating a versatile learning ecosystem that adapts to your students' needs.

What is CoppeliaSim? (Virtual robotics simulation environment)

The virtual CoppeliaSim environment used in the STEPS project functions as the digital twin of the physical FOSSBOT. It allows teachers and students to simulate, program, and observe robot behavior in realistic 3D worlds that mirror real physics and sensor responses. Everything that can be done with the physical FOSSBOT—moving, sensing, reacting—can also be performed virtually, offering flexibility, safety, and immediate feedback.

Through the virtual environment, learners and educators can design complete learning spaces (e.g., mazes, tracks, classrooms), use simulated sensors and actuators, and test code in real time. This enables experimentation without the risk of hardware damage, provides equal access when physical robots are limited, and supports learning continuity beyond the classroom. Students can practice, debug, and refine their algorithms independently at school or home, using the same block-based or Python code as on the real robot.

In STEPS, the virtual environment is not an optional supplement but an integral pedagogical tool. It expands access to robotics education, supports step-by-step progression from simulation to physical experimentation, and fosters inclusive participation for all learners. Combined with the physical FOSSBOT, it forms a complete learning ecosystem that bridges theory and practice, allowing students to experiment safely, think critically, and transition smoothly from digital to real-world robotics.

Pedagogical Approach (STEAM – Computational Thinking)

The STEPS project is founded on a pedagogical vision that combines the creativity of STEAM education with the rigor of Computational Thinking (CT). Together, these frameworks provide educators with powerful tools to design engaging, interdisciplinary, and future-ready learning experiences.

What Is STEAM Education?

STEAM stands for Science, Technology, Engineering, the Arts, and Mathematics. It is an evolution of the older STEM framework that intentionally includes the Arts and Design to foster creativity, empathy, and real-world relevance.

STEAM learning:

- Encourages project-based, problem-centered learning

- Promotes cross-disciplinary thinking
- Values design, aesthetics, and storytelling alongside technical skills
- Engages learners emotionally and cognitively

Robotics, and especially educational robots like FOSSBot, are ideal tools for STEAM because they:

- Embody core physics and math principles (motion, angles, sensors)
- Require programming and engineering thinking
- Invite students to design, create, decorate, or narrate robot behaviors (e.g. drawing shapes, dancing, solving missions)
- Enable collaboration and iteration, key skills in both arts and sciences

STEAM is not about teaching five subjects — it's about integrating them in meaningful, student-centered ways.

What Is Computational Thinking?

Computational Thinking (CT) is a set of cognitive strategies for formulating problems and solving them using processes that can be implemented by a computer or automated system.

According to Wing (2006), CT involves:

- Decomposition – breaking down complex problems into smaller parts
- Pattern Recognition – identifying similarities and trends
- Abstraction – focusing on relevant information, ignoring irrelevant details
- Algorithm Design – developing step-by-step solutions

In the classroom, CT is not only about coding — it's about developing habits of mind that apply across all disciplines:

- In science: planning investigations and interpreting data
- In math: generalizing procedures and patterns
- In language arts: structuring narratives and analyzing structure
- In art and music: creating patterns and sequences

Teaching robotics supports CT by offering a visible, testable, and tangible feedback loop between thought, code, and outcome.

How STEAM and Computational Thinking Are Integrated in STEPS

The STEPS project approaches robotics not as an isolated technical skill, but as a vehicle for interdisciplinary learning. It blends the creative ethos of STEAM education with the problem-solving rigor of Computational Thinking (CT), offering

educators a flexible and inclusive framework for designing meaningful classroom experiences.

Robotics as a Natural STEAM Medium

The FOSSBot robot — whether used physically or virtually — naturally activates multiple dimensions of STEAM:

- **Science** is present in the exploration of movement, light, sound, and cause-effect relationships using real-world sensors and actuators.
- **Technology** is applied through the use of programming languages, digital interfaces, and hardware integration.
- **Engineering** emerges as students design, build, and test robotic behaviors, iterating based on trial and error.
- **Arts** are involved when students choreograph movements, design the robot's appearance, or use it to create drawings and visual patterns.
- **Mathematics** is embedded in every task involving angles, distances, repetition, and measurement.

Rather than teaching each subject in isolation, STEPS encourages students to see connections across disciplines as they work toward a common goal — whether it's navigating a maze, drawing a shape, or simulating a real-world task.

Computational Thinking as the Learning Engine

While STEAM provides the what and why, Computational Thinking provides the how of learning in STEPS. Students are constantly engaging in CT processes, often without realizing it:

- They begin by decomposing a large problem (e.g., "Make the robot draw a square") into smaller steps: move forward, turn, repeat.
- They use pattern recognition to reuse and adapt solutions ("This code worked for a triangle — can we modify it for a square?").
- They practice abstraction by focusing on what matters (the algorithm) and ignoring what doesn't (the exact color of the floor).
- They engage in algorithmic design by building logical, sequenced instructions and adjusting them based on feedback.

Importantly, these CT skills are not limited to robotics or coding. They support students in structuring thinking across all subjects — including math problem-solving, writing processes, and scientific experimentation.

Integration Through Meaningful Contexts

Rather than presenting STEAM and CT as content to be delivered, STEPS embeds them in authentic, student-driven challenges. Each activity encourages learners to:

- Design and build something that responds to the real world (e.g., a robot that avoids obstacles or traces a shape)
- Test, evaluate, and refine their ideas based on observation and logic
- Collaborate with peers, assigning roles and communicating clearly
- Explain their thinking, using both code and natural language
- Reflect on failure as a necessary part of iteration and learning

Through this approach, students do not simply learn about robotics — they learn through robotics.

Teacher as Facilitator of Integrated Learning

The role of the teacher in STEPS is not that of an expert delivering knowledge, but of a facilitator and designer of learning environments. With the support of FOSSBot, you can guide your students to:

- Pose problems worth solving
- Choose tools that fit the task
- Engage diverse learners in shared exploration
- Make thinking visible through simulation, code, and collaboration
- Assess not just the final outcome, but the reasoning and process behind it

The emphasis is on growth, curiosity, and connection, not perfection.

In Summary

The integration of STEAM and Computational Thinking in STEPS is not incidental — it is intentional and foundational. It transforms robotics from a niche technical subject into a creative, inclusive, and cross-disciplinary learning experience.

- STEAM provides the context for learning — making it engaging, relevant, and expressive.
- CT provides the method for problem-solving — making it structured, testable, and transferable.
- Robotics serves as the bridge — a dynamic tool that activates both.

As a teacher, this means you are not just teaching code or sensors. You are building thinkers, makers, collaborators, and citizens — one robot at a time.

How this manual is organized

This manual is designed to be a practical, classroom-ready guide that supports teachers at all levels of education in integrating robotics into their teaching. Whether you are new to educational robots or an experienced STEM educator, this resource will help you use FOSSBot and its virtual twin (CoppeliaSim) to create engaging, inclusive, and curriculum-aligned learning experiences.

Target Audience

This manual is written for:

- Primary and secondary school teachers
- Technical education instructors
- Teacher trainers and facilitators
- Educators working in formal and non-formal STEM education

No prior experience in robotics is required. The manual provides step-by-step pedagogical support alongside technical tools.

Structure and Flow

The manual is divided into six main parts:

PART A – Introduction to STEPS

An overview of the STEPS project and its educational vision. It introduces the key tools (FOSSBot and CoppeliaSim) and explains the pedagogical foundations: STEAM education, Computational Thinking, and Universal Design for Learning (UDL).

PART B – Programming the Physical Robot (FOSSBot)

Practical guidance on how to introduce and teach programming using the real robot, including block-based and Python options. It shows how robotics concepts map to school curricula and supports progressive learning.

PART C – Inclusion, Accessibility & UDL

Offers strategies for adapting robotics lessons for diverse learners, including multilingual students, students with disabilities, and those in under-resourced environments. Emphasizes multiple modes of engagement, differentiation, and flexible assessment.

PART D – Working in the Virtual Environment

Covers how to set up and use the robot's digital twin in CoppeliaSim. This includes building virtual environments, running simulations, and connecting them to classroom instruction. A key focus is on simulation as an inclusive and flexible tool.

PART E – Educational Scenarios (Age-Differentiated)

Provides a series of ready-to-use teaching scenarios, grouped into three age/grade bands:

- Ages 5–8 (Kindergarten & Early Primary)
- Ages 9–13 (Upper Primary & Lower Secondary)
- Ages 13–18 (Upper Secondary & Technical Education)

Each scenario includes: learning goals, subject links, teacher instructions, code examples, UDL tips, and extension ideas.

PART F – Appendices and Resources

Includes Glossary of Educational & Technical Terms and Recommendations for Open-Source Resources

How to Use the Manual

You don't need to read this manual cover to cover. Instead, treat it as a toolbox:

- Use PART B when introducing programming or launching a robotics unit.
- Use PART C to design accessible and inclusive activities for all students.
- Use PART D if your students are learning remotely or if you need an equitable entry point for all.
- Use PART E to find full classroom scenarios, differentiated by age and subject.
- Use PART F for Recourses

Each section is modular and self-contained, allowing you to enter at the point that fits your teaching needs, level of experience, and learning objectives.

Note

The STEPS FOSSBot manual has been designed not only as a technical guide, but as a comprehensive educational resource that helps teachers integrate robotics into STEM/STEAM learning in a structured, inclusive, and progressive way. The intention is the manual to stand as more than a collection of activities — it becomes a teaching framework that combines robotics, pedagogy, and inclusivity. Its logic rests on four key pillars:

1. Balanced Structure
    - The manual combines theory, practical guidance, and ready-to-use scenarios.
    - Teachers first gain background knowledge (concepts, sensors, programming logic), and then see how these ideas are applied in practice through classroom-ready examples.
    - This structure ensures that teachers of varying expertise levels can build confidence gradually.
2. Educational Approach

- Rooted in constructivist and inquiry-based learning: students learn best by *doing, testing, reflecting, and iterating*.
- The manual emphasizes scaffolding: starting from simple activities (movement, obstacle detection) and moving towards complex tasks (data collection, problem-solving, edge/fall detection).
- Universal Design for Learning (UDL) principles ensure accessibility for diverse learners.

3. Beyond Scenarios
    - The inclusion of theory sections and educational notes ensures teachers understand *why* each activity matters, not just *how* to run it.
    - This helps teachers adapt and create their own scenarios, rather than relying solely on pre-made ones.
    - Connections to curriculum standards and competencies make the manual relevant across subjects and age groups.
4. Progressive Learning Pathway
    - Sessions are sequenced to build computational thinking, robotics literacy, and programming skills step by step.
    - Each stage (e.g., obstacle avoidance, line following, sensor use) develops both technical skills and critical thinking abilities.
    - The progression prepares students for real-world problem solving, not just robotic exercises.

PART B – Programming the Physical Robot (FOSSBot)

For teachers who need a pedagogical introduction to programming FOSSBot (PR4 already submitted, but here we refocus on how programming supports learning), although at the end of the section we will give a short summary of PR4 and a link to Instruction manual for the assembly and programming of S.T.E.P.S. educational robot)

Overview of FOSSBot Capabilities

The **FOSSBot 2.0** is a low-cost, open-source educational robot specifically designed to promote hands-on, accessible, and inclusive STEM learning across educational levels. Its capabilities span motion control, environmental sensing, audio-visual feedback, and programmable logic — all of which are easily understandable by students and configurable by teachers.

This section provides a teacher-oriented overview of what the robot can do, how it works, and why these features matter for student learning.

1. Programmable Motion and Mechanical Design

FOSSBot's movement system consists of:

- Two DC gear motors with odometers
- A metal caster wheel for balance
- Removable 66 mm rubber wheels protected by spoilers

Students can control:

- Linear motion (forward and backward)
- Rotational turns (left, right, spin-in-place)
- Curved paths (by adjusting left/right motor speed differentially)

Movement can be driven by:

- Time
- Distance (via odometry)
- Real-time sensor input

Pedagogical Implication: This allows for cross-curricular math and physics integration, such as calculating angles, speed, or path planning, while reinforcing algorithmic thinking through trial and error.

2. Sensor Capabilities: Reacting to the Environment

FOSSBot includes a rich array of sensors that support real-time interaction and data-driven behavior:

- Infrared (IR) Floor Sensors
  - Detect lines or contrasting surfaces on the floor.
  - Enable line-following tasks, maze navigation, and floor-based logic.

- Ultrasonic Distance Sensor (HC-SR04)
  - Detects obstacles or walls in front of the robot (up to 2 meters).
  - Used for obstacle avoidance, stopping, or conditional redirection.

- Infrared Proximity Sensors (7 perimeter sensors)
  - Mounted around the robot body.
  - Detect proximity to nearby objects.
  - Enable reactive behavior, such as wall-following or spatial awareness.

- IMU Sensor (MPU6050: Accelerometer + Gyroscope)
  - Detects tilt, vibration, or sudden changes in motion.
  - Useful for balancing tasks, collision detection, or orientation feedback.

- Light Sensor (LDR)
  - Detects ambient light levels.
  - Used in light-seeking or light-avoiding robot behaviors.

Pedagogical Implication: Sensor-based programming teaches students to work with real-world data, develop conditionals, and build models of autonomous decision-making — foundational for AI and robotics literacy.

Visual and Audio Feedback

FOSSBot can communicate with users through:

- A RGB LED, which changes color based on program logic
- A buzzer and speaker, capable of tones and sound feedback
- An amplifier for increased volume output

These allow students to:

- Use lights and sound as outputs for events (e.g. alert, success/failure)
- Design interactive scenarios or storytelling robots
- Link physical robot actions with expressive modalities

Pedagogical Implication: Multisensory output enables UDL-aligned differentiation. It also supports younger learners and non-verbal learners by linking sensor events to sound/light cues.

4. The LEGO-Compatible and Pen-Holder Design

The robot includes:

- A magnetically attached top cover with a LEGO-compatible surface, supporting customization and creativity.
- A removable pen holder, enabling the robot to draw on paper ("plotter mode").

This means students can:

- Create robot artwork (geometry, symmetry, spirals)
- Attach custom physical elements (flags, sensors, LEGO builds)
- Design STEAM-integrated lessons combining programming with art or storytelling

Pedagogical Implication: These features foster creativity, design-thinking, and ownership over the learning process — key attributes of 21st-century education.

5. Logic Structures and Control

Students can program FOSSBot using either Blockly (visual, block-based) or Python. They can:

- Sequence commands
- Use loops, conditionals, and variables
- Create functions and event-driven behavior
- Write reactive logic based on live sensor input

Pedagogical Implication: These structures support development of computational thinking, logical reasoning, and abstraction. Activities progress naturally from simple step-by-step commands to modular and event-based logic.

6. Connectivity and Setup

FOSSBot includes:

- A Raspberry Pi Zero W for control and communication
- A WiFi-configurable system that connects to a custom browser-based Blockly environment
- A microSD-based OS image with preloaded software

Setup is accessible and robust for step by step connection see at https://stepsproject.eu/activating-FOSSBot-2/):

- Robot boots with WiFi and LED status indicators
- Programming occurs in-browser — no installation required
- Students can connect, program, and run their robot using any device with a browser

Pedagogical Implication: The low technical barrier supports classroom deployment at scale, even for non-specialist teachers. Students can work independently, in pairs, or in rotating stations.

Final Insight

FOSSBot's capabilities are carefully designed to align with:

- Developmental readiness (ages 5–18+)
- Curricular goals across STEM and the arts
- Inclusion principles: multimodal access, creativity, tangible feedback
- Scalability for group work, remote use, or simulation-based learning

As a teacher, every movement, sensor, or blink becomes an opportunity to ask — "What caused this?" — and guide students to think like coders, designers, and engineers.

Introduction to Educational Robotics

Why Teach Programming Through Robots?

Teaching programming through educational robotics represents one of the most effective and engaging strategies for developing core computational thinking, problem-solving, and engineering design skills in learners of all ages. In the context of the STEPS project, FOSSBot is not simply a tool — it is a pedagogical medium that makes abstract computing concepts visible, testable, and meaningful through embodied action.

Programming with Purpose: Tangible, Immediate, Engaging

Unlike conventional screen-based coding, programming a robot produces real-world physical outcomes. When students code FOSSBot to move, turn, follow a line, or react to obstacles, they:

- Receive instant sensory feedback (visual, spatial, temporal)
- Witness the consequences of their logic in motion
- Develop a sense of agency and ownership over the machine they control

This embodiment supports the constructionist approach to learning (Papert, 1980), where knowledge is actively built by making and manipulating artifacts. In other words:

"Students don't just learn programming. They learn through programming — by seeing ideas in action."

Development of Key 21st-Century Competencies

Robotics-based programming integrates multiple **cognitive and metacognitive skills**, including:

- Computational Thinking (Wing, 2006): abstraction, decomposition, algorithmic design
- Logical Reasoning: sequencing, branching, conditional execution
- Systems Thinking: understanding interactions between code, sensors, motors, and environment
- Problem-solving: iterative testing, debugging, and refinement
- Collaboration: co-planning, pair programming, sharing strategies

These competencies are not only relevant in computer science but are transferable across all STEM disciplines, and are recognized as foundational for the European Digital Competence Framework for Educators (DigCompEdu) and students (DigComp 2.2).

Multimodal, Inclusive, and Differentiated Learning

Educational robots like FOSSBot support diverse learning styles and modalities:

- Visual learners see robot actions in real space
- Kinesthetic learners engage by physically manipulating the robot
- Verbal/logical learners explore cause-effect through code
- Neurodiverse learners benefit from predictable patterns and visual structure
- ELL students can follow instructions and visual prompts, even before mastering language

By combining block-based and text-based options, FOSSBot programming offers multiple entry points into coding, making it highly adaptable to varied classrooms and inclusive by design (UDL principles).

Motivation and Emotional Engagement

Several studies (e.g., Eguchi, 2014; Bers, 2018) have shown that students demonstrate increased persistence, curiosity, and confidence when engaging with physical robots. The cause is clear:

- Robots are playful and expressive — they can move, beep, light up, draw
- Students are motivated to improve their programs when they see visible impact
- The trial-error cycle becomes natural and non-threatening ("If it didn't work, change the code and try again")

This emotional engagement leads to deeper cognitive processing and longer-lasting learning.

Cross-Disciplinary Applications

Programming FOSSBot supports not just ICT goals, but also:

- Mathematics: angles, measurement, coordinate systems, logic
- Physics: speed, time, motion, force, distance sensors
- Art: robot-generated patterns, drawing, choreography
- Language arts: writing robot instructions, documenting missions, storytelling

In this way, robotics becomes a curricular integrator, not a standalone subject.

Final Thought for Teachers

Teaching programming through robots doesn't replace traditional instruction — it enhances it by giving learners a meaningful, embodied context for abstract ideas. With FOSSBot, programming becomes:

- Collaborative
- Creative
- Conceptually rich
- And grounded in the real world

It is a powerful way to move from coding as syntax to coding as thinking — and from student as user to student as creator.

INFOBOX: Pedagogical Frameworks Behind STEPS Robotics Education

European Frameworks

DigCompEdu (European Framework for the Digital Competence of Educators)

FOSSBot activities contribute to the following educator competences:

- 2.2 Digital Resources: Selecting and adapting open tools like FOSSBot/CoppeliaSim
- 3.3 Collaborative Learning: Students working together to debug and improve code
- 5.3 Facilitating Learners' Digital Competence: Developing students' coding, problem-solving, and critical thinking

DigCompEdu Framework PDF

DigComp 2.2 (Digital Competence Framework for Citizens) FOSSBot supports student development in:

- Area 1: Information & Data Literacy (interpreting sensor data)
- Area 3: Digital Content Creation (programming and simulation)
- Area 5: Problem Solving (debugging and logic refinement)

DigComp 2.2 Website

Universal Design for Learning (UDL)

UDL principles embedded in robotics lessons:

- Multiple Means of Representation: Visual simulation + real robot + diagrams
- Multiple Means of Action and Expression: Students can program with blocks, Python, or design physical environments
- Multiple Means of Engagement: Choice of task, role in group (e.g., planner, coder, tester)

CAST UDL Guidelines

Tip for Teachers: These frameworks can help you align your robotics lessons with national curricula, inclusive education goals, and digital strategy plans — all while keeping learning playful and meaningful.

Programming Languages Used in STEPS

The STEPS approach embraces a dual-language strategy for teaching programming, combining block-based and text-based environments. This ensures that robotics can be introduced across all levels of schooling — from early primary to upper secondary and technical education — while aligning with each learner's cognitive stage, curricular goals, and developmental readiness.

Block-Based Programming: Visual Logic for Conceptual Learning

Block-based programming uses graphical blocks that snap together like puzzle pieces, each representing a programming structure (e.g., "move forward," "repeat," "if"). In STEPS, this is introduced through environments such as Snap4Arduino, Blockly, or Open Roberta, depending on classroom setup.

This approach is ideal for:

- Younger learners (ages 5–12)
- Beginners of any age
- Students with language or cognitive challenges

Block-based coding allows learners to:

- Focus on logical flow without worrying about syntax
- Develop computational thinking early: sequencing, loops, conditionals
- Build confidence and immediate engagement through visual feedback
- Experiment freely without risk of error messages or crashes

It also supports Universal Design for Learning (UDL) by offering accessible entry points through color-coded blocks, minimal text, and drag-and-drop interactivity.

Teachers can introduce algorithms, decision trees, and structured thinking in an intuitive, low-barrier format — with real-world outcomes via the robot.

Python Programming: From Logic to Literacy

As students grow in confidence and capability, the transition to Python introduces them to a widely used, readable, and high-level programming language. In STEPS, Python can be used to program both:

- The physical FOSSBot directly via USB or Bluetooth
- The digital twin of the robot in CoppeliaSim via its remote API

Python is suited to:

- Learners aged 12+, especially in secondary or technical education
- Cross-curricular applications, such as physics (motion, speed), math (angles, variables), or ICT (functions, logic)
- Projects requiring precision, reusability, or data-driven behavior

With Python, students develop:

- Formal programming literacy (syntax, data types, functions)
- Skills in modular design and debugging
- The ability to interact with external systems (sensors, simulation, files)

Python also serves as a stepping stone toward more advanced domains: web development, robotics control frameworks (e.g., ROS), or AI systems.

Although the STEPS project did not employ the Python programming language directly within its core teaching framework, it is included in this manual for reasons of completeness and future scalability. Python is a powerful and widely used language in both education and robotics, and its mention allows teachers and students to understand how FOSSBOT activities can evolve into text-based programming. Moreover, Python programming code is included in the sections below, offering educators the opportunity to explore equivalent examples and gradually transition from block-based to Python programming when appropriate.

Bridging the Two: A Progressive Approach

STEPS supports a pedagogical progression from block-based to text-based programming. This can be designed as:

- Parallel: Students choose their preferred language for the same task
- Sequential: Start with blocks, then reconstruct the same logic in Python

- Collaborative: Team roles — one student writes blocks, another transcribes to text
- Reflective: Compare block logic to equivalent Python code to deepen abstraction

This inclusive and flexible approach:

- Allows differentiation within a mixed-ability classroom
- Reduces intimidation for students transitioning to code
- Encourages metacognition: thinking about how we structure instructions

The goal is not to abandon simplicity, but to build on it — helping students transfer skills from visual to formal language with confidence and creativity.

Pedagogical Tips for Teaching Programming with FOSSBot

Teaching programming with FOSSBot goes beyond delivering content — it involves designing learning experiences that are engaging, inclusive, and cognitively meaningful. This section offers practical strategies, grounded in pedagogy and learning science, to help teachers implement robotics successfully across diverse classroom settings.

1. Start with Meaningful Problems, Not Syntax

Students learn best when coding is embedded in purpose. Begin with authentic challenges:

- “Can you make the robot draw a square?”
- “How can we help the robot find its way through a maze?”
- “What should the robot do when it sees a red object?”

These tasks anchor learning in real-world goals, activating curiosity and problem-solving rather than memorization.

Lead with questions that invite experimentation. Let the syntax serve the purpose, not the reverse.

2. Use Physicality to Reinforce Abstraction

FOSSBot makes invisible computing concepts visible:

- A loop becomes a repeated motion
- A sensor triggers a visible reaction
- A variable changes how far the robot moves

By watching their programs unfold in real space, students connect code to consequence, strengthening their mental models.

Have students predict what the robot will do before running code — then reflect on what actually happened.

3. Normalize Debugging and Productive Struggle

Debugging is not a mistake — it's a skill. Create a classroom culture where:

- Errors are seen as part of the process
- Students document and discuss bugs
- Peer support is encouraged before asking the teacher

You can introduce "debugging sheets" or team roles like:

- Driver (writes the code)
- Navigator (checks logic and conditions)
- Debugger (tracks test results and problems)

Celebrate "productive failure" as much as success. Each iteration deepens understanding.

4. Differentiate through Multiple Programming Modes

Use the dual-language model of STEPS to offer choice and progression:

- Visual (block-based) coding for concept development, younger learners, or those with processing differences
- Python for advanced learners, cross-disciplinary applications, and high schools

You may also assign roles strategically:

- One student designs logic in blocks
- Another translates it into Python
- A third documents or draws the flow

Use flexible grouping to balance challenge and support — everyone contributes uniquely.

5. Integrate Curriculum Naturally

Robotics is not an "extra." It supports:

- Math (angles, units, proportions)
- Science (motion, sensors, feedback)
- Language (instructions, documentation, storytelling)

- Art (drawing paths, choreographed movement, robot costumes)
- Design & Technology (problem framing, iterative prototyping)

Design tasks that embed coding into existing subjects and local learning goals. Co-teach with colleagues where possible.

A robot that draws a shape can address both geometry and aesthetics. Make every lesson count twice.

6. Scaffold Complexity, Gradually

Start small, then extend:

- Begin with a single motion
- Add a sensor trigger
- Introduce loops or conditionals
- Finally, challenge students to design their own tasks

Provide visual aids (flowcharts, pseudocode), reusable templates, and editable starter code — but remove scaffolds progressively.

Use the "I do → We do → You do" model to guide independence.

7. Provide Multiple Modes of Representation (UDL)

To ensure accessibility:

- Use color-coded flowcharts or step-by-step cards
- Offer printed and digital code samples
- Use voice prompts or peer explanations
- Let students demonstrate learning via video, drawing, code, or verbal presentation

Inclusion is not about simplifying — it's about offering multiple pathways to mastery.

Think like a designer. Can every learner see themselves in this task?

8. Reflect, Discuss, and Share

After each task, reserve time for:

- Explaining decisions
- Comparing strategies
- Writing reflections ("What did I change and why?")
- Demonstrating solutions to peers

This builds metacognition, reinforces learning, and fosters community.

Learning happens not just in the coding — but in the telling of what the code means.

Programming with Robots: Ages 5–8 (Kindergarten & Early Primary)

Pedagogical Focus

At this age, the goal is not to "teach code" in the abstract, but to:

- Develop sequencing and cause-effect reasoning
- Introduce basic directional logic (forward, turn, stop)
- Foster language and communication through storytelling
- Encourage creative, multisensory exploration
- Support early math concepts (shapes, numbers, spatial awareness)

All activities are block-based, unplugged-friendly, and designed with UDL principles in mind.

Activity 1 – "Robot Moves Forward"

Objective: Understand simple motion and command execution
Tools: FOSSBot, basic Blockly commands (move forward, wait, stop)
Estimated Time: 30 minutes
Teacher Notes:

- Model the action: Show students what "move forward for 1 second" looks like
- Use floor tiles or masking tape as units of distance
- Let students act out commands before programming the robot

Sample Sequence:

1. Drag block "move forward (1 sec)"
2. Add "wait (1 sec)"
3. Repeat twice

CT Concept: Sequencing – Actions happen in a specific order
STEAM Link: Physical Science (motion) + Math (units of time/distance)
UDL Tip: Pair code with pictures/icons or spoken instructions

Activity 2 – "Robot Dance"

Objective: Use multiple commands to create patterns and explore loops
Tools: Blockly + optional music + colored floor space
Estimated Time: 30–40 minutes
Teacher Notes:

- Encourage creativity — the robot can "dance" by turning left/right or changing light colors
- Ask students to choreograph short routines and name their dance
- Use loops to repeat moves

Sample Commands:

- Move forward
- Turn right
- Turn left
- Change LED color
- Repeat 2 times

CT Concept: **Loops** and **pattern recognition**
STEAM Link: Art + Music (choreography, rhythm)
UDL Tip: Allow physical demonstration before digital coding

Activity 3 – "Robot Follows the Path"

Objective: Use visual input (line-following) and conditional commands
Tools: Line on floor (tape or marker), line sensor, Blockly
Estimated Time: 40 minutes
Teacher Notes:

- Create a simple curved or zig-zag path on the floor
- Introduce the concept of "if robot sees black line → follow**"**
- Let students test and refine speed and timing

CT Concept: Conditionals (if...then)
STEAM Link: Design & Engineering (path-making)
UDL Tip: Use large visual contrasts (black/white) and tactile guidance for accessibility

Activity 4 – "Draw a Square with the Robot"

Objective: Introduce geometry and repetition using the pen-holder
Tools: Robot + marker pen + Blockly + paper mat
Estimated Time: 45–60 minutes
Teacher Notes:

- Attach pen to robot and place on paper
- Introduce concept of a turn angle (90°) and equal side lengths
- Scaffold using real-world comparisons (square = window, tile)

Sample Block Pattern:

- Move forward (2 sec)
- Turn right (90°)

- Repeat 4 times

CT Concept: Loops, functions, decomposition
STEAM Link: Math (geometry), Art (drawing), Language (describing shapes)
UDL Tip: Allow students to color or decorate their square after drawing

Activity 5 – "Tell a Story with Your Robot"

Objective: Combine movement, sound, and color for narrative play
Tools: Blockly (movement + LED + sound blocks), basic props (e.g. paper trees, toy animals)
Estimated Time: 60 minutes
Teacher Notes:

- Students build a mini "world" with objects and design a story
- Robot becomes a character ("FOSSBot goes to the forest," etc.)
- Program robot to move, beep, change color, or stop at key moments

CT Concept: Events + Sequencing + Abstraction
STEAM Link: Language Arts (narrative), Visual Arts (props), ICT (multimedia)
UDL Tip: Provide templates for storyboard, use audio narration, support multimodal expression

Programming with Robots: Ages 9–13 (Upper Primary & Lower Secondary)

Pedagogical Focus

At this developmental stage, students:

- Are ready to move from simple sequences to control structures (loops, conditions, variables)
- Can begin to reason about input/output and design systems with goals
- Should be encouraged to analyze, debug, and explain their code
- Benefit from linking robotics to curriculum concepts in science, math, and technology

The programming language can still be block-based, but students should also begin exploring Python, especially in Grades 6–8.

Activity 1 – "Robot Explorer: Maze Navigation"

Objective: Use conditional logic and loops to navigate a predefined path
Tools: FOSSBot, maze mat or obstacle course, Blockly or Python
Estimated Time: 45–60 minutes

Teacher Notes:

- Create a simple maze using tape or objects
- Encourage testing of turning angles, distances, and detection timing
- Add challenges: "Can your robot reach the goal in the fewest commands?"

CT Concepts: Loops, conditionals, algorithm design
STEAM Links: Geometry (angles), Physics (turn radius), Design (maze optimization)
UDL Tip: Offer multiple maze layouts with varied complexity

Activity 2 – "Obstacle Avoidance with Sensors"

Objective: Use ultrasonic and IR proximity sensors for real-time environmental awareness
Tools: FOSSBot + distance sensors, objects as obstacles, Blockly or Python
Estimated Time: 60–75 minutes
Teacher Notes:

- Introduce sensor values (e.g., 10 cm threshold)
- Let students experiment with what counts as "too close"
- Challenge: "Can the robot move forward as long as the path is clear?"

CT Concepts: Input/output logic, while-loops, sensor calibration
STEAM Links: Physics (distance, reflection), ICT (input data handling)
UDL Tip: Offer printed or visual guides to explain sensor fields and angles

Activity 3 – "Draw Shapes with Functions"

Objective: Use loops and functions to draw polygons with the pen-holder
Tools: Pen + paper + Blockly or Python
Estimated Time: 60 minutes
Teacher Notes:

- Guide students to decompose shapes into angle + length patterns
- Introduce define function() or Python def blocks
- Extend: Have students create and combine shapes into art

CT Concepts: Abstraction, decomposition, modularity
STEAM Links: Math (geometry, perimeter), Art (robot drawing)
UDL Tip: Use drawing templates or shape stencils to support planning

Activity 4 – "Data Collector: Light and Motion Logging"

Objective: Use sensors to collect data and store it in variables or files
Tools: Light sensor or IMU, Blockly (with variables) or Python
Estimated Time: 60–90 minutes
Teacher Notes:

- Students measure brightness or tilt over time
- Log changes in speed or LED response based on sensor input

- Optional: Export values to CSV using Python

CT Concepts: Variables, data storage, input-output modeling
STEAM Links: Physics (light/motion), Data Literacy, Digital Competence
UDL Tip: Use visual dashboards or printed charts for logging observations

Activity 5 – "Design Your Own Mini-Mission"

Objective: Apply all learned structures in a self-defined challenge
Tools: FOSSBot, any available sensors/props, Blockly or Python
Estimated Time: 1–2 lessons
Teacher Notes:

- Let students choose or invent a problem: e.g., "deliver a message," "visit stations," "search and rescue"
- Require basic planning: storyboard or pseudocode
- Encourage creativity, peer review, and sharing

CT Concepts: Planning, system design, debugging
STEAM Links: Cross-disciplinary (science, math, arts)
UDL Tip: Allow video, sketch, or verbal explanation alongside code

Summary for Teachers

At this age, FOSSBot helps shift students from task-based execution to systematic design and logic reasoning. They begin to:

- Understand cause-effect in complex programs
- Apply abstract code to real-world behaviors
- Collaborate using computational vocabulary
- Reflect on how robots interact with dynamic environments

These foundations prepare them for advanced robotics, physical computing, and autonomous systems in later grades.

Curriculum Integration Matrix for ROBOTS Programming

Programming the FOSSBot does not exist in isolation — it naturally links with multiple curriculum areas. The following matrix illustrates how common robot tasks can be mapped to subject goals, computational thinking concepts, and concrete classroom activities.

Robot Task	Curriculum Link	Key Competencies (DigComp / CT)	Example Classroom Activity
Basic Motion (Forward,	Mathematics (measurement, units of time/distance),	Sequencing, estimation, abstraction	Program FOSSBot to move 3 "steps" forward, then turn

Robot Task	Curriculum Link	Key Competencies (DigComp / CT)	Example Classroom Activity
Backward, Turns)	Physical Education (spatial awareness)		right; compare predicted vs. actual distance.
Drawing Shapes (with Pen Holder)	Mathematics (geometry, angles, perimeter), Arts (symmetry, patterns)	Loops, decomposition, abstraction	Students program FOSSBot to draw a square or triangle, then decorate or analyze the figure.
Line Following	Science (systems thinking, control), Technology (automation), Engineering (feedback systems)	Event-driven logic, input/output, algorithmic design	Create a track with tape; program FOSSBot to follow the line and deliver a message to a "station."
Obstacle Avoidance (with Sensors)	Physics (reflection, distance), ICT (conditional logic), Safety Education	Conditional reasoning, debugging, sensor integration	Students design a patrol route where the robot avoids walls or obstacles using ultrasonic and IR sensors.
Light or Sound Detection	Physics (light intensity, acoustics), Environmental Science	Data collection, variable handling, calibration	Program robot to seek the brightest light source; measure results under different classroom conditions.
Speed and Distance Logging (with Odometers)	Mathematics (ratios, speed = distance/time), Physics (motion), Data Literacy	Variables, data structures, analysis	Students log robot speed over time, export values to CSV, and create graphs in Excel or Python.
Storytelling with Robots (Movement + LEDs + Sound)	Language Arts (narrative writing, sequencing), Drama (performance)	Abstraction, multimodal expression	Students program FOSSBot as a "character" in a story, with sound/LEDs marking events.
Maze Navigation	Engineering (design optimization), Mathematics (angles, paths), Problem-Solving	Algorithm design, loops, conditionals	Build a maze with tape or blocks; challenge students to program the most efficient navigation strategy.
Collaborative Mission (Team Challenge)	Citizenship & Social Studies (collaboration, planning), Digital Literacy	Collaboration, communication, planning	Students work in roles (planner, coder, debugger, presenter) to complete a robot delivery mission.

Robot Task	Curriculum Link	Key Competencies (DigComp / CT)	Example Classroom Activity
Simulation in CoppeliaSim	ICT (digital twin, modeling), Design & Technology	System modeling, testing, abstraction	Students program the virtual FOSSBot to perform the same task as the physical one, compare outcomes.

Table 1. Matrix for FOSSBot Programming

Teachers see clear links to national curricula (math, science, ICT, arts, languages).

Competencies are aligned with DigCompEdu / DigComp 2.2 and computational thinking.

Activities scale in difficulty, supporting progressive learning from primary to secondary.

Programming with ROBOTS: Ages 13–18 (Upper Secondary & Technical Education)

Pedagogical Focus

In this phase, students:

- Transition from guided activities to autonomous problem-solving
- Engage with both block programming and/or formal programming in Python or another programming language
- Explore sensor fusion, modular design, and data structures
- Tackle curriculum-linked challenges (physics, AI, control systems)
- Build toward project-based learning, capstone tasks, or competition prep

The focus is on authentic, applied computing — enabling students to think and act like young engineers, developers, or designers.

Activity 1 – "Robot Patrol with Sensor Fusion"

Objective: Combine multiple sensors (ultrasonic + proximity + IMU) for dynamic obstacle avoidance
Tools: FOSSBot App, real robot + sensors
Estimated Time: 90 minutes

Teacher Notes:

- Have students design a patrol loop (e.g., perimeter of a room)
- Add logic for: "if obstacle ahead → turn," "if tilt detected → stop," "if proximity on side → slow down"
- Emphasize timing and priority of sensor inputs

CT Concepts: Sensor fusion, conditionals, interrupt logic

STEAM Links: Physics (acceleration), Computer Science (decision-making systems)
UDL Tip: Allow visual modeling before coding, e.g., flowcharts or pseudo-logic

Activity 2 – "Speed and Distance Logger"

Objective: Use odometry and time to calculate speed, store data, and visualize results
Tools: FOSSBot App, odometers, time() function, file output (CSV or JSON)
Estimated Time: 60–90 minutes

Teacher Notes:

- Students measure how far FOSSBot travels in a fixed time
- Log and calculate average speed
- Bonus: Export results to spreadsheet and plot graph

CT Concepts: Variables, data structures (lists/dictionaries), file I/O
STEAM Links: Math (speed = distance/time), ICT (data visualization), Physics
UDL Tip: Provide scaffolded code templates with placeholders for input/output

Activity 3 – "Virtual Twin Challenge"

Objective: Program FOSSBot's virtual counterpart in the connected virtual environment to solve a mission
Tools: FOSSBot App, simulation scene
Estimated Time: 2–3 lessons

Teacher Notes:

- Assign a challenge (e.g., navigate to 3 points in order, or follow a virtual path)
- Teach API basics: connecting, sending motor commands, reading position
- Compare real and simulated robot behavior (e.g., friction, noise)

CT Concepts: Remote control, simulation accuracy, parameter tuning
STEAM Links: Engineering design, Robotics, Digital Systems
UDL Tip: Offer debugging support videos and command-line cheat sheets

Activity 4 – "Voice-Activated Robot (Optional Extension)"

Objective: Integrate voice recognition with robot actions via external Python library
Tools: FOSSBot App, microphone, robot API
Estimated Time: 2 lessons (optional)
Teacher Notes:

- Requires basic familiarity with Python libraries and threading
- Teach basic structure: listen → parse → match → execute command

- Safety first: limit commands to low-speed, well-spaced actions

CT Concepts: Event-based programming, external libraries, interface design
STEAM Links: AI, HCI, Natural Language Processing
UDL Tip: Offer pre-trained command sets and editable scripts

Activity 5 – "Capstone Project: Real-World Problem Solving"

Objective: Design and implement a real-use case of robotics in society
Tools: All available (robot, sensors, simulation, optional AI/IoT)
Estimated Time: 4–5 lessons or as an extended project
Teacher Notes:

- Students define their own problem (e.g., delivery bot, COVID-safe assistant, smart traffic agent)
- Require: problem analysis, planning doc, system diagram, test phases
- Include presentation or live demonstration

CT Concepts: Project lifecycle, modular architecture, teamwork
STEAM Links: All integrated — ethics, engineering, entrepreneurship
UDL Tip: Allow for technical, narrative, and visual components in assessment

Summary for Teachers

FOSSBot for this age group is a gateway to authentic robotics and engineering. It supports:

- Deeper conceptual learning (feedback, control, algorithms)
- Cross-subject integration (data science, AI, electronics)
- Competency-based approaches aligned with DigCompEdu, STEAM, and career orientation

These activities also lay the foundation for participation in:

- Science fairs and hackathons
- Inter-school robotics competitions
- Vocational training and tech careers

Students are no longer just coding robots. They are designing intelligent systems — and imagining the future.

Common Robot Tasks for Learning

Across all educational levels, robotics education relies on a set of core task types that serve as building blocks for progressively more complex learning. These tasks introduce key computational concepts, activate physical reasoning, and link programming to cross-curricular goals. They are age-scalable, technology-neutral, and ideal for differentiation.

This section presents the four most essential categories of tasks used in STEPS, along with their educational value and classroom application.

1. Basic Movement (Sequencing & Direction)

What students do: Program the robot to move forward/backward, turn, pause, or repeat a pattern using time or distance.

Learning goals:

- Understand sequential commands
- Translate spatial thinking into code
- Estimate and test distances or angles

Typical activities:

- Drive to a marked point on the floor
- Navigate a simple "road" layout
- Dance or draw a pattern using movement

Pedagogical links:

- **Early math** (direction, measurement)
- **Language** (action verbs, instruction writing)
- **CT concept**: Sequencing, iteration

This is the ideal entry point for beginners of all ages.

2. Obstacle Avoidance (Conditional Logic)

What students do: Use sensors (ultrasonic, IR proximity) to detect obstacles and make the robot change direction, stop, or react.

Learning goals:

- Apply real-time input in programming
- Write if...then logic based on sensor values
- Tune parameters like detection distance or reaction speed

Typical activities:

- "Robot patrol" without collisions
- Wall-following or corridor navigation
- Reactive obstacle course

Pedagogical links:

- **Physics** (distance, reflection)

- **CT concept**: Conditionals, input/output logic
- **Data literacy** (interpreting sensor feedback)

A great mid-level task for learning autonomy and responsive systems.

3. Line Following (Input-Driven Navigation)

What students do: Use bottom-facing IR sensors to follow a dark line on a white surface, or vice versa.

Learning goals:

- Combine sensor data with motion commands
- Understand feedback loops and threshold tuning
- Connect logical flow to physical path-following

Typical activities:

- Trace a simple path or shape
- Race along a winding road
- Complete a delivery mission between stations

Pedagogical links:

- **Design** (path creation)
- **Systems thinking** (environment-robot interaction)
- **CT concept**: Event-driven programming

Supports collaborative troubleshooting, visual debugging, and physics-in-action.

4. Drawing Shapes (Loops, Geometry & Functions)

What students do: Attach a pen to FOSSBot and program it to draw regular shapes (e.g., squares, triangles, spirals) on paper.

Learning goals:

- Use **loops** to repeat movements
- Explore **angles**, **turning radius**, and side lengths
- Introduce **functions** to organize code

Typical activities:

- Draw a square, then a triangle, then a house
- Use nested loops to create patterns
- Build creative artwork (e.g., robot-generated mandalas)

Pedagogical links:

- **Mathematics** (geometry, measurement)
- **Art** (form, repetition, symmetry)
- **CT concept**: Loops, functions, decomposition

A powerful integration of logic, creativity, and spatial reasoning.

Final Insight for Teachers

These four core task types:

- Can be adapted for all age groups by adjusting logic depth, speed, or required precision
- Are ideal for assessment, differentiation, and interdisciplinary projects
- Build up toward open-ended missions and real-world simulations (e.g., delivery bots, automated sketching, AI navigation)

They are not just tasks — they are frameworks for thinking like a programmer, designer, and problem-solver.

Differentiating Difficulty per Learner Group

In any robotics classroom, learners will vary widely in age, experience, cognitive style, and confidence with programming. The STEPS pedagogical model encourages teachers to differentiate not by simplifying, but by offering multiple levels of challenge and entry points for each task.

This section outlines strategies for adapting activities to different learner needs using the same tools and concepts — with alignment to Universal Design for Learning (UDL) and competency-based education.

1. Vary the Complexity — Not the Task

Choose one core task (e.g., “Draw a square” or “Avoid obstacles”) and adapt it across three levels:

Example Task: "Draw a Shape" – Differentiated by Level

- **Basic level**: The student manually places individual movement and turn blocks to form a square. Each step is explicitly coded without abstraction.
- **Intermediate level**: The student uses a loop (e.g. repeat 4 times) to simplify the pattern and reduce code duplication, showing an understanding of repetition.
- **Advanced level**: The student defines a reusable function such as draw_shape(sides), using parameters and possibly angle calculations to generate various polygons dynamically.

This keeps students working on a shared goal, fostering inclusion and collaboration — while allowing everyone to stretch at their own level.

2. Scaffold with Support Tools

Offer visual organizers, starter code, and checklists for students who need structure, while allowing others to work from scratch.

Examples:

- Flowchart templates for planning logic
- Block-code cards or printed block libraries
- Python skeletons with placeholders
- Peer role guides (planner, coder, tester)

Use "fadeable scaffolds": tools students can eventually remove as they grow more independent.

3. Build Choice into the Lesson

Instead of assigning the same format to everyone, provide structured options:

- "Choose your input device: button or proximity sensor"
- "Use Blockly or Python — your choice"
- "Demonstrate your project using a chart, video, or a live run"

Choice increases engagement, autonomy, and ownership of learning.

4. Use Roles in Mixed-Ability Groups

Group students intentionally and assign rotating roles:

- **Logic Lead** – plans the robot's behavior
- **Coder** – implements the logic in Blockly or Python
- **Debugger** – tests and adjusts parameters
- **Presenter** – documents and shares the result

This promotes cooperation and ensures that all students contribute meaningfully, even if their skill levels differ.

5. Use the Simulation as an Alternative Path

If a student struggles with the physical robot (e.g. due to motor skills, anxiety, or classroom logistics), they can:

- Program the digital twin in CoppeliaSim
- Test and debug without needing access to hardware
- Upload/share their solution in a safe, flexible environment

Simulation is not a downgrade — it's a pedagogically powerful alternative or extension.

6. Offer Tiered Challenges

Design every robotics challenge with 3 "ladders" of success:

Tiered Challenge Example: Obstacle Avoidance

- **Bronze level**: The student programs the robot to detect and avoid a single obstacle, then stop. This is a basic task demonstrating conditional logic and sensor usage.
- **Silver level**: The student programs the robot to navigate through a zigzag path using multiple sensors (e.g. front and side proximity), requiring more advanced logic and testing.
- **Gold level**: The student designs their own obstacle-avoidance strategy from scratch and explains the logic behind it. This may include adjusting speed, angles, or thresholds dynamically.

Students aim as high as they can — but every tier demonstrates meaningful learning.

Final Insight for Teachers

Differentiation is not about different activities — it's about different access to the same ideas.

Robotics offers a uniquely rich environment to:

- Personalize pathways
- Celebrate diverse strengths
- Maintain high expectations for all learners

In STEPS, every student — regardless of skill or background — has a place in the robotics classroom.

Short Recap of PR4 – Instruction Manual for the FOSSBot

The STEPS PR4 deliverable (see official project's site: https://stepsproject.eu) provides teachers and trainers with the complete *technical foundation* for preparing and running the FOSSBot robot in the classroom. While Part B of this manual focuses on how to teach programming pedagogically, PR4 ensures that educators have the necessary technical setup and operating knowledge.

Key Coverage in PR4

1. **3D Printing & Assembly**

- Step-by-step instructions for downloading STL files and printing the robot's plastic parts.
- Guidance on recommended materials (e.g., PETG) and printer settings.
- Illustrated assembly process including wheels, chassis, spoilers, pen holder, and battery cover.

2. **Electronics Setup & Wiring**
   - Full list of electronic components and wiring diagrams.
   - Instructions for soldering and connecting sensors, motors, and the PCB boards.
   - Practical cabling tips for safe and accessible classroom use.
3. **WiFi Configuration & First Run**
   - How to flash the pre-configured system image onto a microSD card.
   - Setting up WiFi credentials via the configuration file.
   - LED status indicators (red/green/blue) to confirm successful connection.
4. **Blockly Programming Environment**
   - Accessing the browser-based Blockly interface.
   - Kindergarten mode for very young learners (drag-and-drop movement cards).
   - Standard Blockly workspace with categories of blocks for motion, sensors, and logic.
5. **Sample Tasks in PR4**
   - **Obstacle avoidance** using the ultrasonic sensor.
   - **Line following** with IR sensors.
   - **Light detection** using the LDR sensor.
     These illustrate how to move from basic movement to sensor-driven behaviors.

Positioning PR4 in Relation to Part B

- **PR4** = *"Build it and make it run"* → Technical assembly, setup, and introductory programming.
- **Part B** = *"Teach with it"* → Pedagogical frameworks, curriculum integration, age-specific activities, and inclusive strategies.

Together, they form a complete package: **PR4 ensures that the robot is operational**, while **Part B empowers teachers to use it as a meaningful educational tool**.

PART C – Inclusion and Universal Design (UDL) in Robotics

Introductory

Inclusion and Universal Design for Learning (UDL) are essential principles for ensuring that robotics and STEM education are accessible, engaging, and meaningful for all learners. Robotics activities often involve abstract concepts, technical tools, and problem-solving tasks that may present barriers to students with diverse needs and backgrounds. By applying UDL, educators can design learning experiences that provide multiple ways of presenting information (e.g., visuals, audio, simulations), multiple avenues for students to express their understanding (e.g., coding, building, storytelling), and varied opportunities for engagement (e.g., teamwork, self-paced exploration). These strategies not only remove barriers for students with disabilities or limited resources, but also enrich the learning experience for everyone—fostering creativity, collaboration, and equity in STEM classrooms.

What is UDL and why does it matter in STEM and Robotics

Universal Design for Learning (UDL) is a research-based educational framework that aims to make learning accessible, flexible, and engaging for all students. It is grounded in cognitive neuroscience and emphasizes the need to address learner variability rather than treating it as an exception. Instead of designing a "one-size-fits-all" lesson, UDL encourages teachers to provide multiple means of representation (how information is presented), multiple means of action and expression (how students demonstrate their knowledge), and multiple means of engagement (how students are motivated and involved in learning) (CAST, 2018).

In the context of STEM and Robotics, UDL is particularly important because these fields often present challenges that can exclude certain learners if not carefully designed. Robotics combines abstract programming concepts, hands-on construction, problem-solving, and teamwork—all of which can be exciting but also intimidating for students with different abilities, backgrounds, or learning profiles. Applying UDL principles ensures that every student has an entry point:

- Visual learners can engage through diagrams, block-based coding, or simulations.
- Kinesthetic learners can physically manipulate robots or act out algorithms.
- Students who struggle with text or language can rely on visuals, icons, or collaborative discussion.
- Advanced learners can be challenged with open-ended problem-solving, while beginners receive scaffolding and simplified coding environments.

UDL also matters in robotics because it models inclusivity in STEM practice itself. Real-world engineering teams are diverse, requiring members with different strengths. By structuring robotics lessons around UDL, teachers prepare students to value collaboration, adapt to different perspectives, and contribute in unique ways. This has broader social importance: it encourages participation from groups historically underrepresented in STEM—such as girls, neurodiverse learners, English Language Learners (ELL), or students from rural/low-resource schools (Meyer, Rose, & Gordon, 2014).

Finally, UDL strengthens computational thinking and problem-solving by showing that there are multiple valid pathways to a solution. Whether students are coding in blocks, writing Python, simulating in CoppeliaSim, or sketching an algorithm on paper, UDL ensures that robotics activities emphasize conceptual understanding, creativity, and persistence, rather than narrow technical barriers.

Quick Reference: UDL in STEM & Robotics

What is UDL?

- A pedagogical framework for including all learners.
- Three core principles:
    1. **Multiple Means of Representation** → How knowledge is presented.
    2. **Multiple Means of Action/Expression** → How learners show what they know.
    3. **Multiple Means of Engagement** → How learners are motivated and stay involved.

Why does it matter in Robotics?

- Robotics requires coding, mechanics, and teamwork → different students find different entry points.
- Provides access for all: visual learners → diagrams, kinesthetic learners → handling robots, ELL → icons and visuals.
- Offers scaffolding for beginners, challenges for advanced students.

Benefits:

- Inclusion of students with different abilities and needs.
- Promotes equity (girls in STEM, rural schools, neurodiverse learners).
- Strengthens computational thinking: multiple pathways to a solution.
- Reflects real STEM practice: teamwork, diversity, and innovation.

Teacher's Cheat Sheet: UDL in STEM & Robotics

UDL Principle	In Robotics	Classroom Tips
Multiple Means of Representation (how knowledge is presented)	Code shown in both block-based and text-based formats; diagrams of circuits; videos of robot	Provide visual diagrams and icons alongside code; offer step-by-step video demos;

UDL Principle	In Robotics	Classroom Tips
	actions; real objects and simulations.	use simulation when equipment is limited.
Multiple Means of Action & Expression (how students show what they know)	Students build robots, write code, simulate movements, or act out algorithms physically.	Let students choose: code in blocks, sketch flowcharts, or explain verbally; encourage both digital and hands-on outputs.
Multiple Means of Engagement (how students stay motivated)	Group challenges, robotics games, self-paced projects, or open-ended design tasks.	Mix team-based activities with individual exploration; gamify tasks (points, levels, challenges); allow students to personalize robot behavior.
Adapting Difficulty & Scaffolding	Simplified coding environments for beginners; advanced students extend logic with variables, sensors, or functions.	Pair stronger and weaker students (peer mentoring); give optional "challenge blocks" for faster learners.
Supporting Diverse Learners	Visual learners → diagrams; kinesthetic learners → robot handling; ELL → icons + translations; girls in STEM → highlight real role models; rural schools → digital simulation.	Use plain language; encourage teamwork across diverse roles; highlight diverse STEM careers; provide offline worksheets if no robot is available.

Table 5. UDL in STEM & Robotics

Designing Accessible Activities

Designing accessible activities means intentionally creating robotics lessons that reduce barriers and provide multiple entry points for diverse learners. Universal Design for Learning (UDL) emphasizes that not all students process information, express ideas, or engage with tasks in the same way. In robotics, this becomes particularly important because activities often involve abstract programming concepts, physical manipulation of hardware, teamwork, and problem-solving. By offering flexible pathways, teachers make sure every student can participate meaningfully, regardless of their abilities, learning style, or prior knowledge.

Multiple Representations (images, videos, audio)

Not all learners benefit equally from a single mode of instruction. Some need visual supports like diagrams of circuits, flowcharts, or videos of the robot in action. Others may process better with auditory cues such as spoken explanations or sound effects from the robot itself. Combining these modes ensures that key ideas—like how sensors work or how loops control movement—are accessible whether a student learns best by seeing, hearing, or reading.

Multiple Expression Options (draw, code, simulate, act out)

Students should be able to demonstrate their understanding in different ways. For example, one student may write block-based code, another might draw a flowchart of the robot's logic, a third could run a simulation in software, and another might act out the robot's movement sequence physically. All these approaches allow learners to express the same core understanding while aligning with their strengths.

Multiple Engagement Modes (pairs, small teams, self-paced)

Motivation and persistence increase when students have choices in how they engage. Some may thrive in pairs where peer support reduces anxiety; others may prefer small teams that allow role specialization (builder, coder, tester); still others may prefer self-paced exploration where they can experiment independently with the simulator. By offering varied modes of participation, teachers help sustain interest and reduce frustration—crucial for inclusion, especially for students who may feel intimidated by STEM tasks.

In practice, designing accessible activities is about planning lessons that anticipate learner diversity rather than reacting afterward. A robotics task presented with diagrams + code + spoken explanation, allowing output through coding or drawing, and engagement either in groups or individually, ensures that no student is excluded and that all can experience success.

Example Task: Make the Robot Follow a Square Path

Learning Goal: Students understand loops, movement commands, and turning angles by programming a robot to trace a square.

Multiple Representations (how the task is presented)

- **Visual:** Teacher shows a diagram of a square with arrows for each movement and turn.
- **Text/Code:** Teacher provides sample pseudocode (repeat 4 times → move forward, turn right 90°).
- **Audio:** Teacher explains aloud: *"The robot repeats forward and right turns four times to form a square."*
- **Video/Simulation:** Students watch a short clip of the robot completing the square in the simulator.

Multiple Expression Options (how students show understanding)

- **Draw:** Students sketch the square path and label movements.
- **Code:** Students use block-based or text-based code to program the robot.
- **Simulate:** Students run the program in a virtual environment (FOSSBot App).
- **Act Out:** Students physically walk the square, turning their bodies 90° to represent the robot.

Multiple Engagement Modes (how students participate)

- **Pairs:** One student writes the code, the other tests and debugs.
- **Small Teams:** Students divide roles (drawer, coder, tester, presenter).
- **Self-Paced:** Individual students experiment with adjusting side length or turn angle to explore variations (rectangle, triangle).

Why This Works

This task shows how **one learning objective** (programming a square path) can be made accessible to all learners. A student who struggles with abstract code can succeed by **drawing or acting it out**; a student who excels in coding can extend the task by adding loops or conditions; all students remain engaged because they have **choice and flexibility**.

Adapting Difficulty

In a robotics classroom, students enter with very different levels of prior knowledge, confidence, and problem-solving ability. Some may already be comfortable with coding, while others are still learning the basics of loops or conditions. Universal Design for Learning (UDL) encourages teachers to adapt the level of difficulty so that every learner can participate meaningfully while being challenged appropriately. This avoids frustration for beginners and boredom for advanced learners.

Scaffolding Code

Scaffolding means breaking down complex tasks into manageable steps. For example, instead of asking students immediately to program a maze solver, the teacher first introduces a guided version:

- Step 1: Move forward 10 cm.
- Step 2: Turn right 90°.
- Step 3: Repeat steps in a loop. Gradually, scaffolds (e.g., teacher-provided pseudocode, partial solutions, or hint blocks) are removed as students gain independence. Scaffolding helps students build confidence while still progressing toward the same learning goals.

Simplified Environments

Not all students are ready to jump straight into text-based coding or full robotics simulations. Simplified environments such as block-based programming (Blockly, Scratch-like tools, or FOSSBot's drag-and-drop interface) provide an accessible starting point. Visual coding environments reduce cognitive load by focusing on concepts rather than syntax errors. As students gain proficiency, they can transition to text-based Python or more complex simulators. This staged approach ensures progression without exclusion.

Peer Support Models

Students often learn best from one another. Peer support models—such as pair programming, "expert-apprentice" groups, or small teams with rotating roles—allow stronger students to explain concepts while others practice new skills with guidance. This not only supports struggling learners but also reinforces understanding for the more advanced students. For example, one student might debug code while another writes, and a third documents the robot's behavior. Peer support aligns with UDL by valuing collaboration, communication, and shared problem-solving.

In practice, adapting difficulty ensures that all students can work on the *same core task* (e.g., programming a robot to avoid obstacles) but at a level suited to their ability: beginners might follow scaffolded block instructions, while advanced students experiment with additional logic (variables, randomization, optimization).

Example Task: Obstacle Avoidance at Different Levels

Learning Goal: Students program the robot to move safely in a space with obstacles.

Beginner Level – With Scaffolding (Block-based)

- **Environment:** Visual drag-and-drop blocks.
- **Scaffolded Code:** Teacher provides most of the structure:

[repeat while (true)]
→ [if (Obstacle exists)]
→ [Turn right 90°]
→ [else]
→ [Move forward 10 cm]

- **Support:** Teacher explains each block and demonstrates in the simulator. Students drag blocks into place and run the program.
- **Focus:** Understanding *if/else logic* and how sensors affect movement.

Intermediate Level – Simplified Environment with Small Extensions

- **Environment:** FOSSBot's block environment with fewer hints.
- **Activity:** Students create the loop themselves and extend it:
  - Print messages when the robot detects an obstacle.
  - Add a counter to track how many times the robot avoided something.
- **Focus:** Building independence and linking code to **data collection**.

Advanced Level – Peer-Supported Exploration (Python)

- **Environment:** Python coding connected to simulator.
- **Activity:** In teams, students:
  - Code the robot to detect distance values (not just obstacle yes/no).
  - If distance < 30 cm, turn left 45°. If < 15 cm, turn right 90°.
  - Add randomness to avoid repetitive behavior.
- **Peer Support Model:** Teams rotate roles (coder, tester, debugger).
- **Focus:** Applying **thresholds, variables, and multiple conditions** in real-world robotics.

Why This Works

All students work on the *same problem*—**obstacle avoidance**—but difficulty is adapted:

- Beginners succeed with a scaffolded, block-based solution.
- Intermediate students experiment with extending logic.
- Advanced students explore more complex sensor-driven navigation in Python.

This ensures **equity and engagement**: nobody is excluded, and everyone is challenged at their own level.

Example Task: Square Path with Scaffolding

Learning Goal: Students understand loops, movement commands, and turning angles by programming a robot to trace a square.

Beginner Level – Full Scaffolding (Block-based)

- Environment: Block coding (drag-and-drop).
- Scaffolded Code: Teacher provides nearly complete solution:

[Move forward 10 cm]
[Turn right 90°]
[Move forward 10 cm]
[Turn right 90°]
[Move forward 10 cm]
[Turn right 90°]
[Move forward 10 cm]
[Turn right 90°]

- Focus: Students see how movement + turning combine to form a square. No loops yet.
- Support: Teacher walks through each step, shows robot simulation.

Intermediate Level – Partial Scaffolding (Introduce Loops)

- Environment: Block coding, fewer hints.

- Activity: Teacher provides a repeat 4 times loop block and asks students to fill in the inside:

[repeat 4 times]
→ [Move forward 10 cm]
→ [Turn right 90°]

- Focus: Students learn the power of loops—reducing repeated code.
- Support: Teacher explains loop logic but students assemble the body of the loop themselves.

Advanced Level – Independent & Extended (Python + Team Roles)

- Environment: Python coding.
- Activity: In small teams, students:
  - Write a function `draw_square(side_length)` using a loop.
  - Experiment with different side lengths (10, 20, 30 cm).
  - Extend to draw other shapes (triangle with 3 sides, hexagon with 6).
- Peer Support Model: One student codes, another debugs, another tests the robot.
- Focus: Abstraction with functions, loops, and parameters, plus creative exploration.

Why This Works

- Beginners practice concrete movement without abstract logic.
- Intermediate learners see how loops simplify and generalize code.
- Advanced students build reusable code structures and extend concepts.

Thus, the same Square Path task adapts to multiple levels, supporting inclusion while maintaining challenge.

Supporting Diverse Learners

STEM and robotics classrooms are diverse spaces where students bring different strengths, needs, and challenges. A central principle of Universal Design for Learning (UDL) is to anticipate learner variability and design learning activities so that no student is excluded. Supporting diverse learners in robotics is not only about accessibility but also about ensuring equity and broad participation in STEM fields. Below we analyze four key groups and how UDL strategies apply.

Neurodiverse Students

Neurodiversity includes learners with autism, ADHD, dyslexia, or other cognitive differences. Robotics can be highly motivating for these students, but also overwhelming due to complex instructions or sensory overload. UDL strategies include:

- **Clear, predictable structure:** Use step-by-step visual guides, checklists, and consistent routines.
- **Choice and flexibility:** Allow students to express solutions through drawings, simulations, or verbal explanations, not just code.
- **Sensory support:** Provide quiet workspaces or noise-cancelling tools if the robotics environment is overstimulating.
  These approaches help neurodiverse learners use their strengths (e.g., pattern recognition, problem-solving) while minimizing barriers.

English Language Learners (ELL)

Students learning in a second language may struggle with technical terminology or long instructions. In robotics, this barrier can prevent them from fully engaging with coding or teamwork. UDL strategies include:

- **Visual supports:** Use diagrams, icons, and block-based programming to reduce language load.
- **Language scaffolding:** Provide bilingual glossaries, key vocabulary cards, and sentence starters for team collaboration.
- **Peer and group support:** Pair ELL students with supportive peers to promote inclusion and collaborative learning.
  This ensures that language is not a barrier to accessing robotics concepts and allows ELL students to learn both coding and academic language in context.

Girls in STEM

Girls are still underrepresented in robotics and STEM education due to stereotypes, lack of role models, and reduced confidence. UDL-informed practices help counteract this by:

- **Inclusive role models:** Highlight women in robotics and engineering to challenge stereotypes.
- **Collaborative, real-world projects:** Emphasize creativity, problem-solving, and social impact—areas shown to increase girls' interest.
- **Equitable participation structures:** Rotate roles (builder, coder, presenter, tester) so girls are not sidelined into support tasks.
  These strategies build confidence and ensure that girls see themselves as active contributors to robotics.

Rural / Low-Equipment Schools

Schools in rural or under-resourced contexts may lack physical robotics kits, sensors, or high-speed internet. Without adaptation, this can limit access to hands-on learning. UDL strategies include:

- **Simulation environments:** Use digital tools like the FOSSBot simulator to replicate robotics tasks.
- **Low-tech adaptations:** Encourage unplugged activities (e.g., acting out robot algorithms, drawing flowcharts) to practice logic and computational thinking.
- **Community partnerships:** Share resources through mobile labs, regional hubs, or collaborative projects between schools. This ensures that robotics learning is not confined to well-equipped classrooms, but remains inclusive and meaningful everywhere.

In practice, supporting diverse learners means proactively planning robotics lessons that:

- Remove barriers for neurodiverse students.
- Reduce language load for ELL learners.
- Encourage participation and confidence for girls.
- Provide alternatives for resource-limited schools.

Such strategies reflect the spirit of UDL: ensuring equity, access, and engagement for all students in robotics.

Quick Reference: Supporting Diverse Learners in Robotics

Learner Group	Typical Challenges	UDL-Based Strategies in Robotics
Neurodiverse Students (Autism, ADHD, Dyslexia, etc.)	Sensory overload, difficulty with long instructions, need for structure	• Use step-by-step visual guides & checklists • Provide quiet/low-stimulus areas • Allow multiple ways to express solutions (draw, simulate, code)
English Language Learners (ELL)	Struggle with technical vocabulary, instructions in a second language	• Use block coding & icons to reduce language load • Provide bilingual glossaries & key terms • Pair with supportive peers for teamwork
Girls in STEM	Underrepresentation, stereotypes, reduced confidence	• Highlight women role models in robotics/engineering • Rotate roles (coder, builder, presenter) to ensure equity • Frame projects around creativity & real-world impact
Rural / Low-Equipment Schools	Limited robotics kits, low internet access, fewer opportunities	• Use simulators (FOSSBot App) as alternatives • Encourage unplugged activities (acting out algorithms, drawing flowcharts) • Build partnerships with community labs & regional hubs

Table 6. Supporting Diverse Learners in Robotics

Example Activity: Smart Step Counter with Obstacle Detection

Base Task: Students program the robot to:

1. Set steps = 0
2. Repeat while true → move forward one step, increase counter by 1
3. If obstacle is detected → stop and print steps

Adaptations for Diverse Learners

1. Neurodiverse Students

- **Challenge:** May struggle with complex instructions or sensory overload.
- **Adaptation:**
  - Provide a visual flowchart of the step counter loop.
  - Use quiet simulation environment before real robot to reduce noise/motion stress.
  - Break task into mini-goals: (a) move one step, (b) add counter, (c) detect obstacle.

2. English Language Learners (ELL)

- **Challenge:** Technical terms (e.g., *obstacle, counter, repeat*) may be confusing.
- **Adaptation:**
  - Use block-based version with clear icons instead of text-only code.
  - Provide bilingual vocabulary cards (e.g., *obstacle* = *εμπόδιο*).
  - Allow students to act out the program: one student is the robot, another counts steps, another acts as obstacle.

3. Girls in STEM

- **Challenge:** Risk of lower confidence or being sidelined into non-technical roles.
- **Adaptation:**
  - Rotate roles in the group (one codes, one tests, one explains results).
  - Frame the step counter as a real-world problem (e.g., "How many steps until a rescue robot finds a person?").
  - Highlight **female engineers** who design navigation systems in robotics.

4. Rural / Low-Equipment Schools

- **Challenge:** Limited access to physical robots or sensors.
- **Adaptation:**
  - Run the task in a digital simulator (FOSSBot App).
  - Use unplugged activity: students walk across the room counting steps until they meet a desk (obstacle).
  - Record results on a shared class chart, comparing groups' counts.

Why This Works

Every student engages with the *same core concept*—counting steps until obstacle detection—but adaptations ensure equity:

- Neurodiverse learners benefit from structure and predictability.
- ELL learners rely on visuals, icons, and physical acting to reduce language load.
- Girls gain confidence through equitable roles and meaningful context.
- Rural schools stay included through simulations and unplugged activities.

PART D – Virtual Robotics

Introductory

In the STEPS framework, using a virtual robot environment is more than a technical convenience — it is a pedagogical necessity. Simulation supports inclusion, experimentation, scalability, and hybrid learning, fully aligned with digital competence and STEAM education goals. It ensures that every learner, regardless of resources, can access authentic robotics experiences.

One of the greatest advantages of simulation is its power to promote accessibility and equity. Physical robots like FOSSBOT are often limited in number and expensive to expand, leading to unequal participation in class. In a virtual environment, every student can have their own robot on their screen, working at their own pace without waiting for equipment. This transforms "waiting your turn" into "taking ownership of your learning."

Simulation also enables personalized exploration and self-paced learning. Students can code, test, and debug individually, observing real-time robot behavior without fear of damaging hardware. They can rewind, repeat, or modify experiments freely, fostering curiosity and reflective thinking. Such autonomy encourages learners to design their own variations and engage in true inquiry-based learning.

Finally, the virtual environment ensures continuity and inclusiveness. Students can participate remotely, complete robotics lessons from home, and submit digital results for assessment. It supports learners who face physical, logistical, or accessibility challenges and empowers those who might hesitate to handle real hardware. In this way, simulation becomes a powerful equalizer — extending robotics learning to every student, everywhere.

Supporting Inclusion and Universal Design for Learning (UDL)

Beyond physical access, true inclusion in robotics education means ensuring that every student — regardless of ability, background, or learning profile — can meaningfully engage with the content. Simulation environments provide essential affordances for teachers working within the principles of Universal Design for Learning (UDL) and inclusive pedagogy.

Addressing Diverse Physical and Sensory Needs

Not all students can safely or confidently manipulate physical robots. Challenges may include:

- Fine motor limitations (difficulty with plugging wires, positioning robots)
- Mobility constraints (e.g. navigating the floor or robotics area)
- Visual impairments (struggling to observe robot movement from a distance)
- Auditory processing difficulties (missing beeps or sound feedback)

With simulation:

- Students can zoom, rotate, or slow down the environment to suit their needs
- Visual feedback can be amplified (e.g. robot path traces, coordinate readings)
- Controls can be adapted to keyboard-only or accessible interfaces

When physical interaction is a barrier, simulation becomes a doorway.

Multiple Means of Representation and Expression

UDL encourages providing more than one way to receive, process, and express informationVirtual Environments enables:

- Visual learners to observe behavior, paths, angles, and sensors graphically
- Verbal/logical learners to follow structured commands and debug logically
- Kinesthetic learners to design and test movements through trial and adjustment
- Creative learners to build narratives or visual scenes in the simulator

And when it comes to assessment:

- Students can submit their solution as code, a screenshot, a screen recording, or a live demonstration

Inclusion isn't about lowering the bar — it's about changing the approach.

Supporting Executive Function and Cognitive Load

For some learners, especially those with ADHD, dyslexia, or processing differences, programming a physical robot involves a high cognitive load:

- Remembering multiple steps
- Coordinating code with movement
- Recovering from errors in real space

Simulation provides:

- A controlled environment where the robot can be paused, restarted, or reset instantly

- A chance to focus on logic without managing physical setup
- Fewer distractions, more clarity of cause and effect

Teachers can also scaffold the experience with:

- Templates or starter environments
- Simplified sensor feedback
- Pre-built "partial solutions" that students modify

Simulation acts as a cognitive scaffold — focusing student effort on logic and strategy, not logistics.

Reducing Social Barriers and Participation Gaps

In physical robot work, classroom dynamics can unintentionally exclude some learners:

- Confident students dominate the hardware
- Others take on passive or peripheral roles (e.g. note-taking only)
- Girls, multilingual students, or neurodivergent learners may "step back" from hands-on control

Virtual Environments levels these dynamics by:

- Putting every student in equal control of their own simulation
- Allowing them to try, fail, and iterate privately before sharing
- Supporting text-based, visual, or verbal approaches to engagement

Inclusion means every student has a way in — and a reason to stay.

One of the main strengths of the STEPS–FOSSBOT framework is that students can experiment freely, test quickly, and learn from mistakes in a safe, hybrid environment. Using the virtual FOSSBOT simulation, they can take intellectual risks — trying new algorithms, debugging logically, and refining their solutions — without fear of damaging real equipment or spending time on complex setups. This synergy between the physical and virtual FOSSBOTs encourages continuous learning through experimentation rather than repetition.

Instant Iteration: Learn by Trying

In a physical classroom, each test run requires repositioning the FOSSBOT, resetting props, and waiting for the robot to complete its task. In the virtual environment, the same code can be launched instantly, paused, or replayed in seconds, allowing dozens of iterations in one session. This rapid feedback loop motivates students to explore variations ("What happens if I change this angle or speed?"), turning small failures into opportunities for reflection and growth.

Visualizing Invisible Behaviors

FOSSBOT's sensors and motion systems involve complex internal processes — such as distance thresholds, wheel encoders, or turning radius — that are often invisible during real operation. The simulation environment makes these behaviors visible through traces, variable monitors, and real-time graphs. Students can observe and analyze how the FOSSBOT reacts, predict outcomes, and adjust parameters precisely, deepening their understanding of cause and effect in robotic control.

Safe Exploration of Advanced Ideas

Through simulation, students can safely test **advanced FOSSBOT functions** such as increased velocity, multiple sensors, or complex navigation on challenging surfaces. These experiments, which might be risky or impractical in real life, foster deeper understanding of efficiency, optimization, and system reliability — key skills for future engineers and innovators.

Encouraging Ownership and Autonomy

By combining physical and virtual experimentation, STEPS allows learners to take full control of their learning process. Students can debug independently, personalize challenges, and refine solutions at their own pace. This autonomy is ideal for project-based learning, competitions, or capstone work — turning learners into **active creators and FOSSBOT engineers**, not just robot users.

Bridging Theory and Real-World Application

The STEPS–FOSSBOT framework turns abstract ideas into something students can see and interact with. By translating code into the visible behavior of the FOSSBOT robot, students connect classroom theory with real-world systems, strengthening understanding, improving knowledge transfer, and supporting meaningful STEM integration. In subjects such as physics, geometry, and computer science—where core concepts are often invisible—the FOSSBOT makes them tangible: students can observe vectors and trajectories, track acceleration and turning radius, visualize sensor ranges, and monitor variables or collision events. They can explain Newton's laws through robot motion, explore geometry through turning angles, or demonstrate logical reasoning using sensor thresholds. Abstract theory becomes concrete, visual, and embodied through the robot's actions.

Students are encouraged to think like scientists and engineers—forming hypotheses, testing under controlled conditions, analyzing results, and designing systems with feedback and constraints. Using both the physical and virtual FOSSBOT, they cultivate essential skills such as modeling, measurement, optimization, and iteration. These directly align with curriculum goals across STEM subjects, AI and robotics electives, and project-based learning. Through experimentation, students see how small code changes affect larger robotic behaviors, transfer logic between virtual and physical robots, and apply math and physics concepts in authentic tasks. For example, a loop becomes more than

syntax—it represents a motor's motion; a conditional becomes a decision point for obstacle avoidance; a function becomes a reusable building block. These connections strengthen conceptual depth and make learning more durable across contexts.

The FOSSBOT ecosystem also allows students to experiment freely, test quickly, and learn safely. They can take intellectual risks—trying new ideas, debugging logically, and refining solutions—without fear of damaging hardware or losing time to setup. In the virtual environment, tests can be launched instantly, paused, or repeated in seconds, enabling dozens of variations in a single session. This rapid feedback loop encourages experimental thinking, normalizes small failures, and promotes reflective debugging where errors become data for growth.

Students can push their experiments further by adjusting the FOSSBOT's speed, adding or modifying sensors, or testing challenging edge cases. These situations may be too risky or impractical in real life, yet they are valuable for developing deeper understanding of efficiency, reliability, and adaptive design. By removing logistical barriers, the framework supports independent work and creativity. Learners debug autonomously, design their own challenges, and refine solutions at their own pace—transforming from users into engineers who control and direct their tools.

Finally, the FOSSBOT framework aligns with practices found in university research, autonomous systems, and industrial automation. Early exposure helps students grasp advanced ideas such as digital twins, introduces them to professional tools like Python and ROS, and prepares them for future studies in mechatronics, AI, data science, and engineering. In this way, students gain early insight into how robotic systems are conceived, tested, and improved—bridging classroom learning with professional practice and innovation.

Supporting Whole-Class and Group Work

Supporting Collaborative and Scalable Learning

Using a virtual environment allows every student to engage in robotics learning, even when physical FOSSBOTs are limited. Teachers can run simultaneous simulations, enabling individuals or groups to test code, explore scenarios, and iterate ideas without waiting turns or facing hardware constraints. This transforms the classroom from "one-at-a-time" to "everyone-at-once," fostering inclusion, active participation, and meaningful collaboration. Clear team roles — such as programmer, designer, debugger, or documenter — ensure that every learner contributes, while teachers gain more time for reasoning, reflection, and problem-solving rather than setup and troubleshooting.

Bridging Education, Industry, and Future Skills

The virtual environment also acts as a bridge between school-level learning and real-world technology. It mirrors professional tools used in robotics research, engineering, and automation, giving students authentic exposure to industry

practices while they develop computational thinking and digital literacy. Through simulation, learners build both technical (coding, testing, optimization) and soft skills (collaboration, documentation, iterative design). This approach aligns with STEAM education, VET priorities, and digital competence frameworks, preparing students not only to program robots but to think and work like future engineers and innovators.

What Does the Simulator App Do?

The app allows to write the programs in the Blockly environment, exactly as it is in the real FOSSBot.

Instead of manually:

- Opening the virtual environment (CoppeliaSim),
- Loading a scene file,
- Configuring settings...

The app does all of this with one click.

It:

- Starts CoppeliaSim automatically
- Loads the correct .ttt file (FOSSBot model + virtual environment)
- Optionally connects to Python code if needed

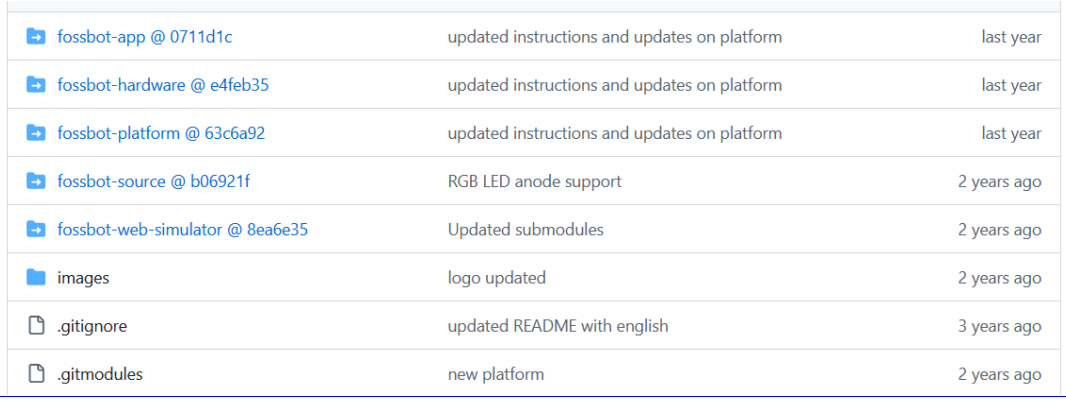
This greatly reduces classroom setup time and avoids user errors.

Step-by-Step Installation Guide

1. Download the App

The latest version of the app is hosted at:

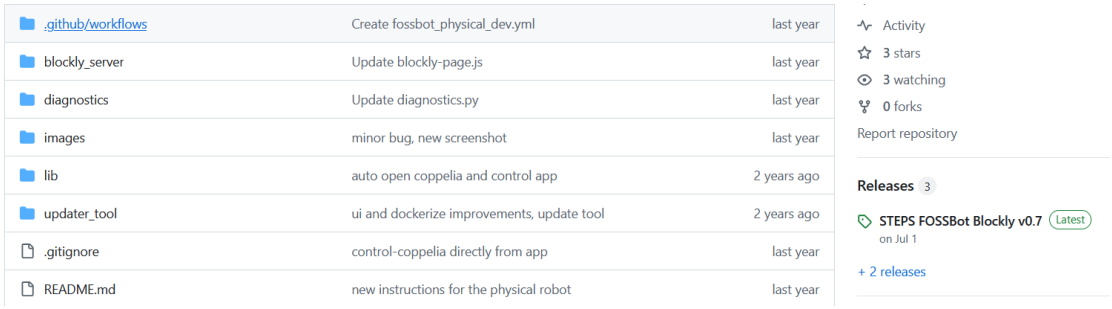
https://github.com/eellak/FOSSBot



fossbot-app @ 0711d1c	updated instructions and updates on platform	last year
fossbot-hardware @ e4feb35	updated instructions and updates on platform	last year
fossbot-platform @ 63c6a92	updated instructions and updates on platform	last year
fossbot-source @ b06921f	RGB LED anode support	2 years ago
fossbot-web-simulator @ 8ea6e35	Updated submodules	2 years ago
images	logo updated	2 years ago
.gitignore	updated README with english	3 years ago
.gitmodules	new platform	2 years ago

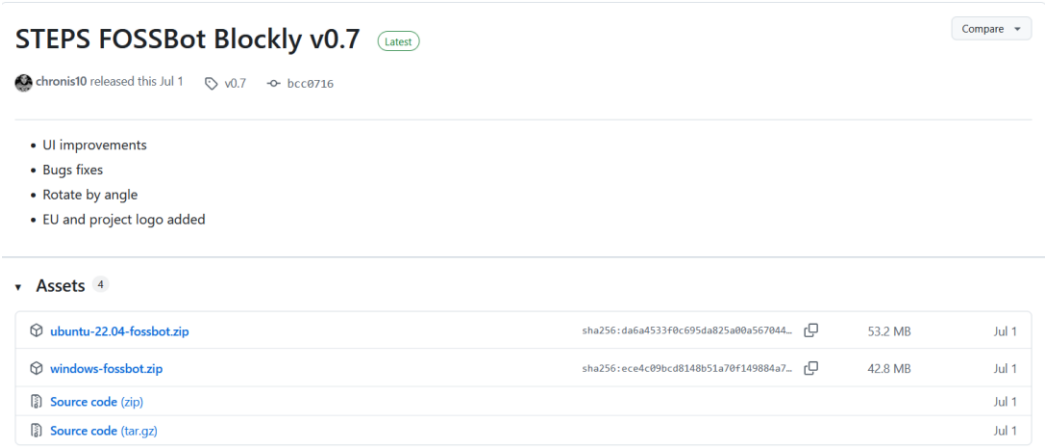
Picture 5, https://github.com/eellak/FOSSBot

Click on the "FOSSBot-app @ 0711d1c"



Picture 6, FOSSBot-app @ 0711d1c

Click on the "STEPS FOSSBot Blockly v0.7, (Latest)



Picture 7, STEPS FOSSBot Blockly v0.7, (Latest)

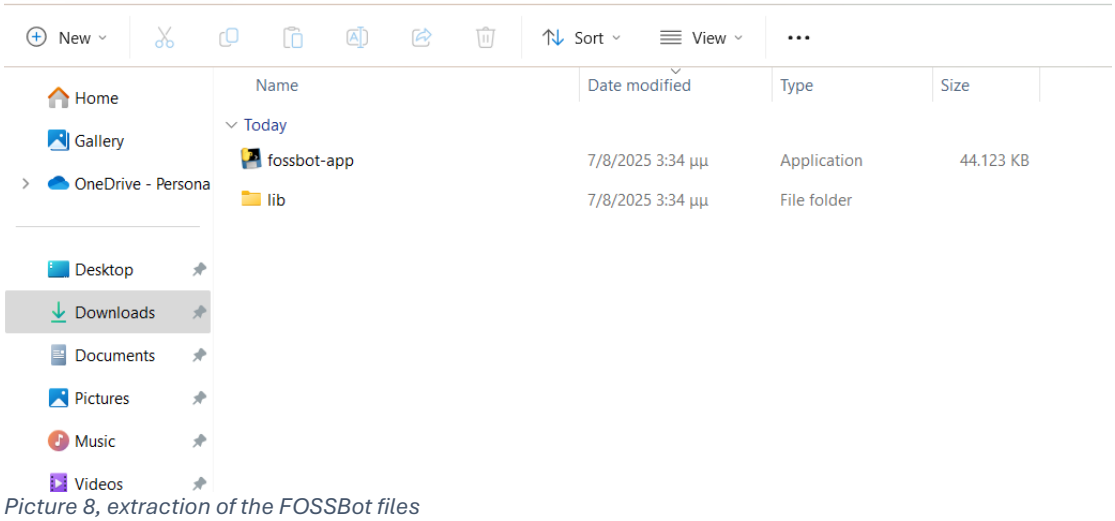
Look for the folder or release named FOSSBot-app (it may also be found under "simulator" or "FOSSBot-sim").

Choose the version for your operating system and download it:

- windows-FOSSBot.zip for Windows
- ubuntu-22.04-FOSSBot.zip (Linux/macOS) — may require permissions

2. Extract the Files

- On Windows: unzip the .zip file
- On Linux/macOS: extract .tar archive or move the file to a desired folder
- Create a desktop shortcut if needed for easier access

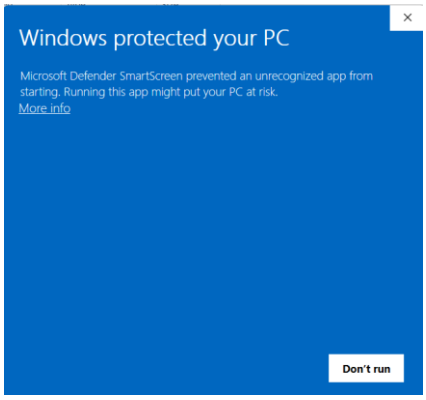


Picture 8, extraction of the FOSSBot files

3. First-Time Setup

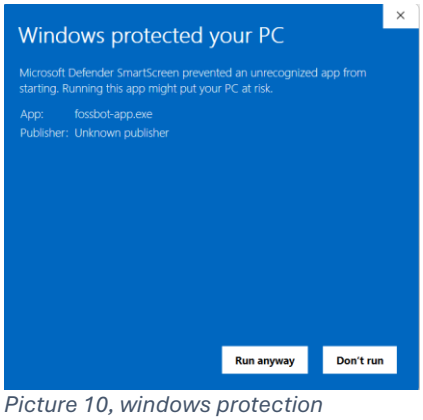
When you open the app (*FOSSBot-app*) for the first time:

You will maybe see the above



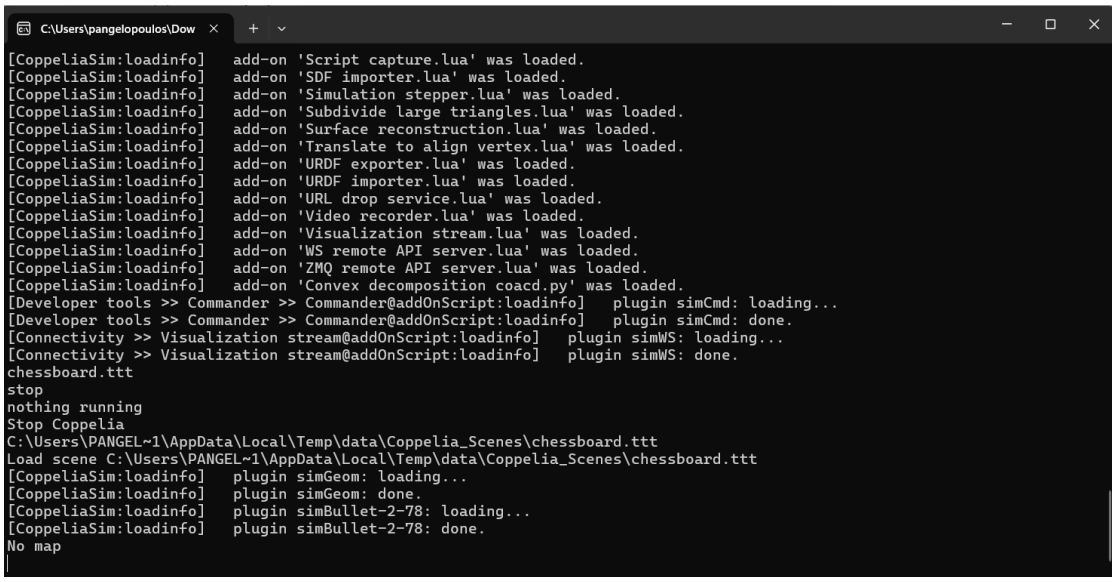
Picture 9, windows protection

Click on “More Info”



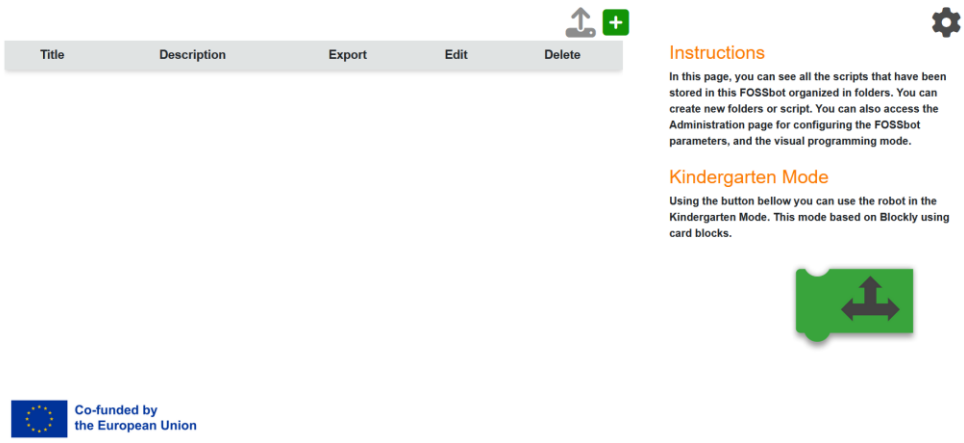
Picture 10, windows protection

Click on “Run anyway” and FOSSBot application will start. As the application runs the command environment remains open



Picture 11, Command environment

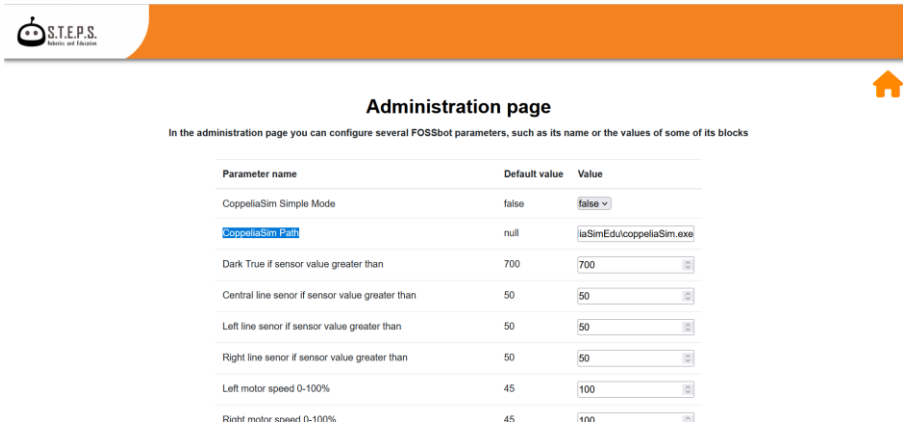
And on your browser opens the virtual FOSSBot environment on the http://localhost:8081/



Picture 12, FOSSBot Environment

Importing the FOSSBot model

You'll be asked to select the path to CoppeliaSim OR click on ⚙ (top right) on the administration page and copy "C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu\coppeliaSim.exe" or the path you have installed coppeliasim at the "CoppeliaSim Path" field (see picture below).

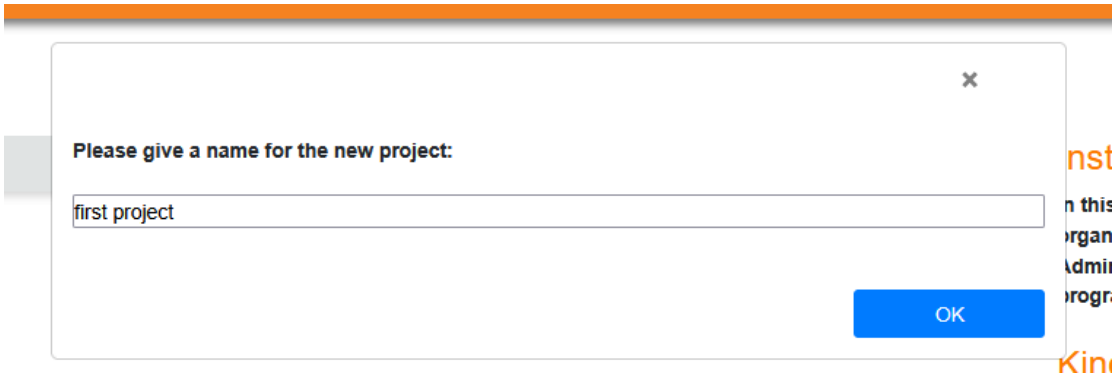


Picture 13. Administration Page

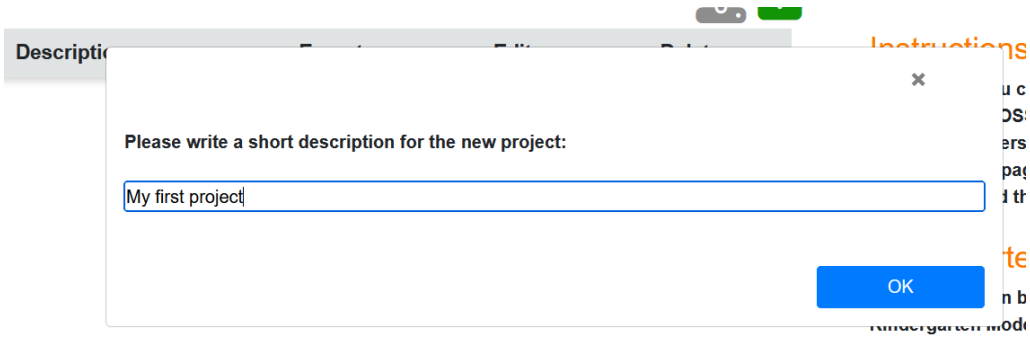
This is the folder where coppeliaSim.exe or coppeliaSim is located
Save the setting. The app will remember it.

4. Launch the Simulation

Click the + buton, give a name and a short description to your project and It will automatically load the FOSSBot model and environment. The app will also launch CoppeliaSim, in both the FOSSBot environment and on the pc separately.

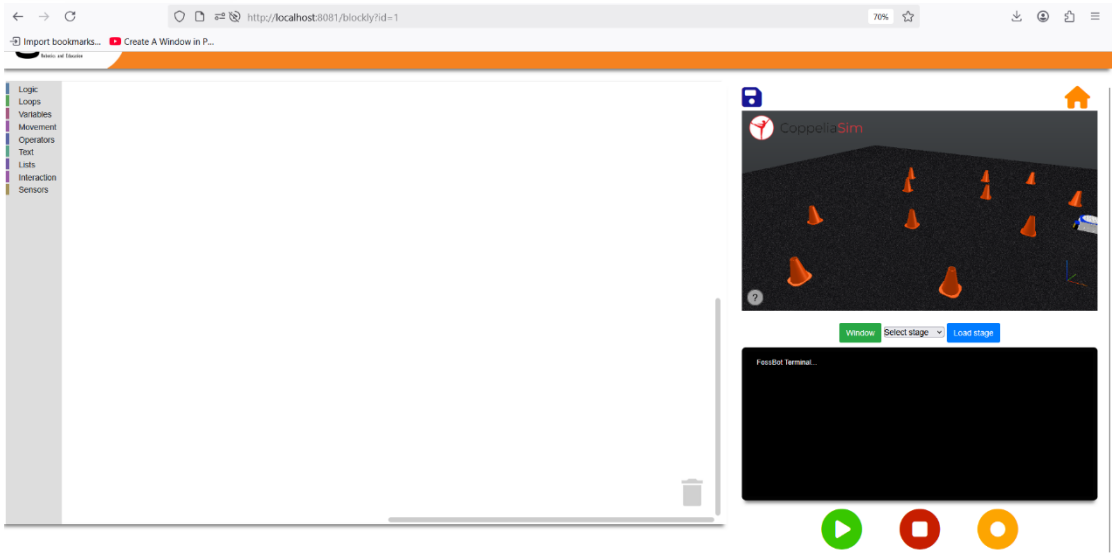


Picture 14, New Project name

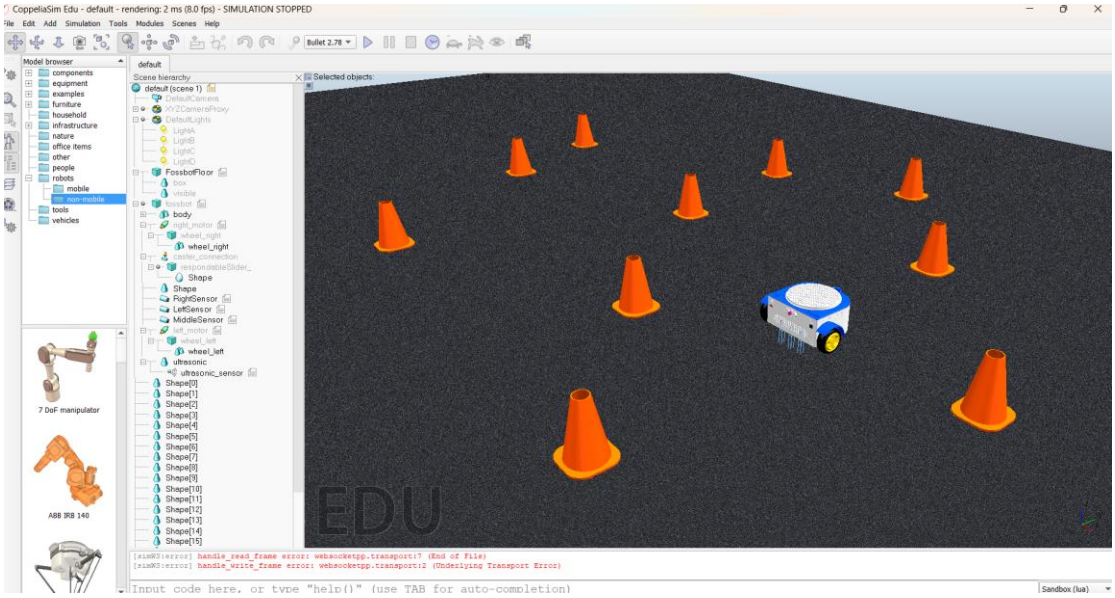


Picture 15, Short description

In a few seconds, you'll see the 3D robot and simulation floor — ready for action!



Picture 16, 3D robot and simulation floor



Picture 17, Coppelia Sim automatically launched

Troubleshooting Tips

Problem	Solution
App won't run (Linux/macOS)	Make executable: chmod +x FOSSBot-app
App can't find CoppeliaSim	Check the correct path to coppeliaSim.exe
Robot doesn't appear	Make sure the .ttt file is in the same folder or bundled with the app
macOS blocks app	Allow it in "Security & Privacy" settings (System Preferences)

Table 4. Tips

Teacher Tips

- Install the app on each lab device where students will use the simulator
- Keep the .ttt robot scene and app in the same folder for easier access
- Show students how to launch the simulator with one button, instead of navigating inside CoppeliaSim
- You can also preload custom environments (e.g. line-following or obstacle courses)

This app is ideal for younger learners, mixed-ability classes, and quick lesson transitions.

Setting up a new World

While the STEPS project is primarily focused on the educational use of the FOSSBot robot and its core simulation environment, the design of custom virtual worlds in CoppeliaSim falls *outside the formal scope* of this manual.

However, for reasons of completeness — and to support teachers or students who wish to experiment with their own virtual environments — this section offers a brief, optional guide on how to add simple elements like floors, walls, and obstacles.

This content is not required for basic STEPS lessons, but may enrich your curriculum if you want to introduce exploration, design thinking, or student-created challenges using the virtual simulator.

Designing a custom simulation space for the virtual FOSSBot, *(Focus: floor layout, obstacles, exploration environments)*

This section is meant to empower teachers — even those with no experience in 3D design — to build or modify simple virtual worlds inside CoppeliaSim that FOSSBot can navigate.

Let's break it down step by step:

Why create a custom world?

- Simulations are most effective when tailored to your lesson goals.
- A basic flat floor can become:
  - A maze
  - A road with intersections
  - A track for line-following
  - A puzzle-solving arena
- Designing environments helps students visualize geometry, physics, and logic in action.

Understanding the Virtual Scene in CoppeliaSim

Every simulation world is made up of **objects**, such as:

- Floor: A base where the robot moves
- Walls: To block or guide movement
- Obstacles: Cubes, cylinders, ramps, etc.
- Lines: Flat visual elements for line-following
- Sensors & goals: For more complex missions

Each object has:

- A position (x, y, z)
- A size (length, width, height)
- A color or texture
- Physical properties (collisions, mass, friction)

Steps to Build a Basic World (Manual)

This approach assumes you're using **CoppeliaSim EDU**, not just the Player.

1. Add a Floor

1. Go to the **Add** menu → **Primitive Shape** → **Plane**
2. Resize the plane to act as a floor:
    - X: 2.0
    - Y: 2.0
    - Z: 0.01
3. Position it at $z = 0$ so it sits flat

Optional: Change its color by selecting the object and going to **Object → Appearance**

2. Add Walls or Obstacles

1. **Add → Primitive Shape → Cuboid**
2. Adjust size for a wall:
    - X: 0.1
    - Y: 1.0
    - Z: 0.3
3. Move it to the edge of your floor
4. Duplicate (Ctrl + D) to create more walls

You can rotate, stack, or curve walls to form:

- Mazes
- Tunnels
- Rooms

3. Add a Line-Following Path (Optional)

1. **Add → Dummy → Path** or use flat **textures**
2. Or, import an image (e.g., black line on white) as a texture on the floor
3. Make sure the path is visible from the robot's top sensors

4. Group and Save Your Environment

1. Select all added objects (Shift + Click)
2. Use Object → Grouping to manage them as one

3. File → Save Scene As... and name your world (maze1.ttt, line_course.ttt, etc.)

You can create multiple scenes for:

- Beginner, Intermediate, and Advanced layouts
- Specific challenges (obstacle avoidance, mapping, etc.)

Teacher Tips for Designing Student-Friendly Worlds

- Start simple: Just a floor and one obstacle
- Use bright colors for visual clarity
- Keep the world small and symmetrical for younger learners
- Label objects (e.g., goal_zone, wall1, start_line) for debugging
- Create a shared library of environments that students can copy and explore

Students can also design their own virtual worlds as part of a project — this builds spatial reasoning and logic planning!

Setting up CoppeliaSim: Installation & requirements

Before students can begin programming and experimenting with virtual robots, they need a properly installed and configured simulation environment. This section provides clear instructions and guidance for teachers and students to install CoppeliaSim, the simulation platform used in the STEPS project.

Software Requirements Overview. For teachers new to virtual environments

To use FOSSBot in a virtual environment, your students will need a few key software components. Don't worry — no programming experience is required to install them. This section explains what each one does, why it's needed, and how to prepare your classroom computers for virtual robotics with **CoppeliaSim**.

What Are the Required Software Components?

Below are the **four essential parts** your students will need installed and ready to use:

1. **CoppeliaSim EDU or CoppeliaSim Player**

The 3D simulation environment where the robot lives and moves

- CoppeliaSim is like a virtual playground where students can see and control a simulated robot. It shows the robot's movement, sensors, and interactions with obstacles, all in 3D.
- The EDU version allows full editing of simulation environments. The Player version is simpler and only allows students to run pre-prepared scenes (useful for younger students).
- Think of it as the "stage" where the robot performs.

Needed for all students, whether they're using block-based tools or Python.

2. FOSSBot Simulator App

An easy-to-use launcher that connects CoppeliaSim and the FOSSBot scene

- The Simulator App is a small tool made specifically for STEPS and FOSSBot.
- It automatically opens CoppeliaSim and loads the robot model and virtual world.
- Students don't need to navigate menus — the app makes everything simple and ready in one click.
- Without this app, teachers would have to load robot files manually and adjust settings by hand.

Highly recommended, especially for schools or teachers with limited experience in simulation tools.

3. Python 3 and Required Libraries

The programming language students will use to control the robot

- Python is a friendly, readable language taught in many schools.
- It allows students to write code that sends commands to the virtual FOSSBot (e.g., "move forward", "turn left", "stop if there's an obstacle").
- Students will also install a small helper library called pyzmq, which helps Python talk to the simulation.

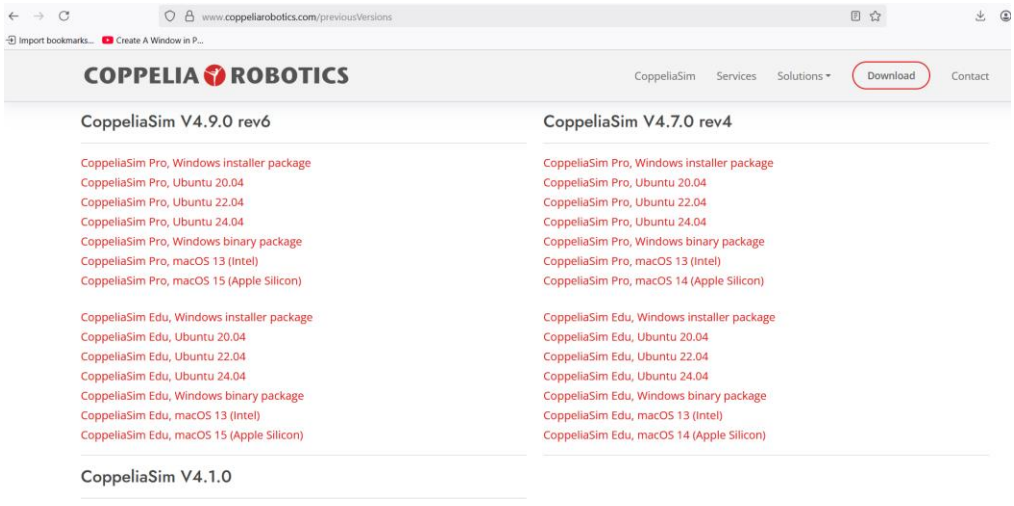
Don't worry — you won't need to install dozens of things. Just Python + 1 or 2 support packages, and you're ready to go. Needed only if you're using Python programming, not block-based tools.

Step-by-Step Installation Guide

1. Visit the official download page

Install the Coppelia Simulator (EDU or Player) https://www.coppeliarobotics.com/previousVersions

** Download version 4.7 or prior, the latest version is not supported yet.**

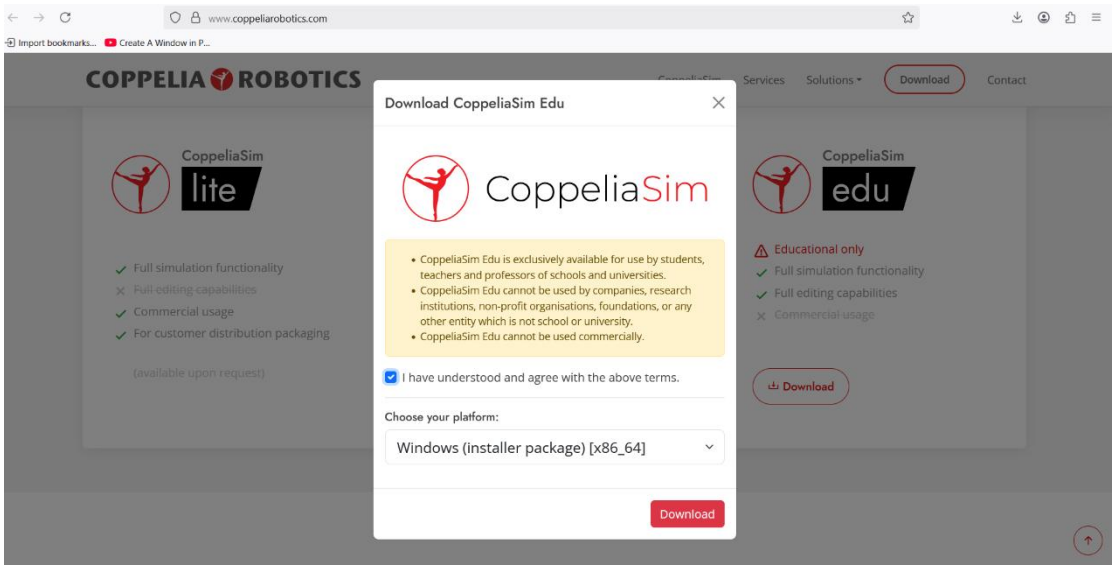


Picture 1, Windows (64-bit): Download

2. Choose the correct version for your operating system

- **Windows** (64-bit): Download CoppeliaSim Edu, Windows installer package *(Choose the "Edu" version for editing features)*
- **Linux (Ubuntu)**: Download CoppeliaSim Edu, Ubuntu 24.04
- **macOS (Intel)**: Download CoppeliaSim Edu, macOS 13 (Intel) or CoppeliaSim Edu, macOS 14 (Apple Silicon)

Note: M1/M2 Macs may have compatibility issues and require workarounds

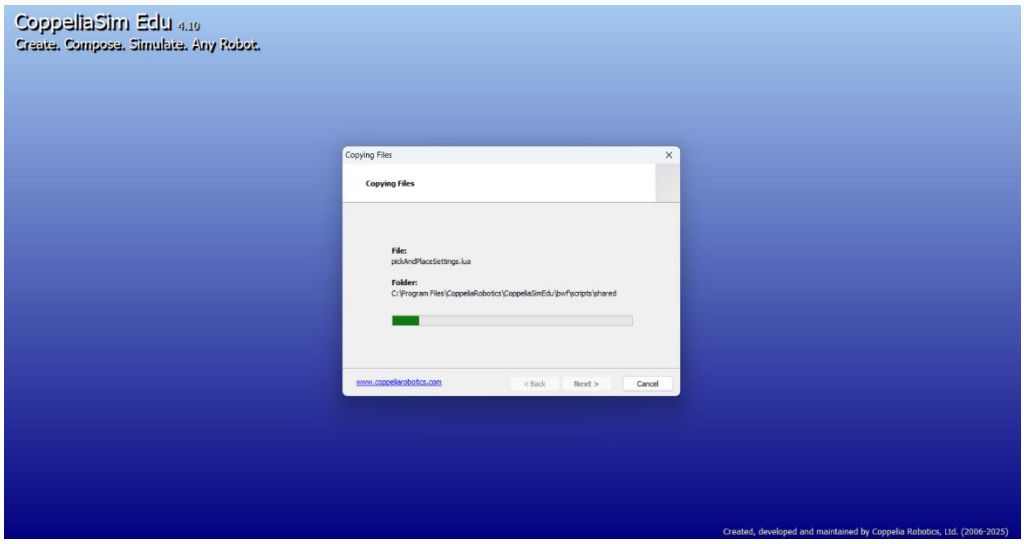


Picture 2, CoppeliaSIm download

Make sure you have at least 1 GB of free disk space before downloading.

3. Extract the archive to a folder

- Dooble-click the downloaded file *CoppeliaSim_Edu_V4_7_0_rev4_Setup*
- Install the application



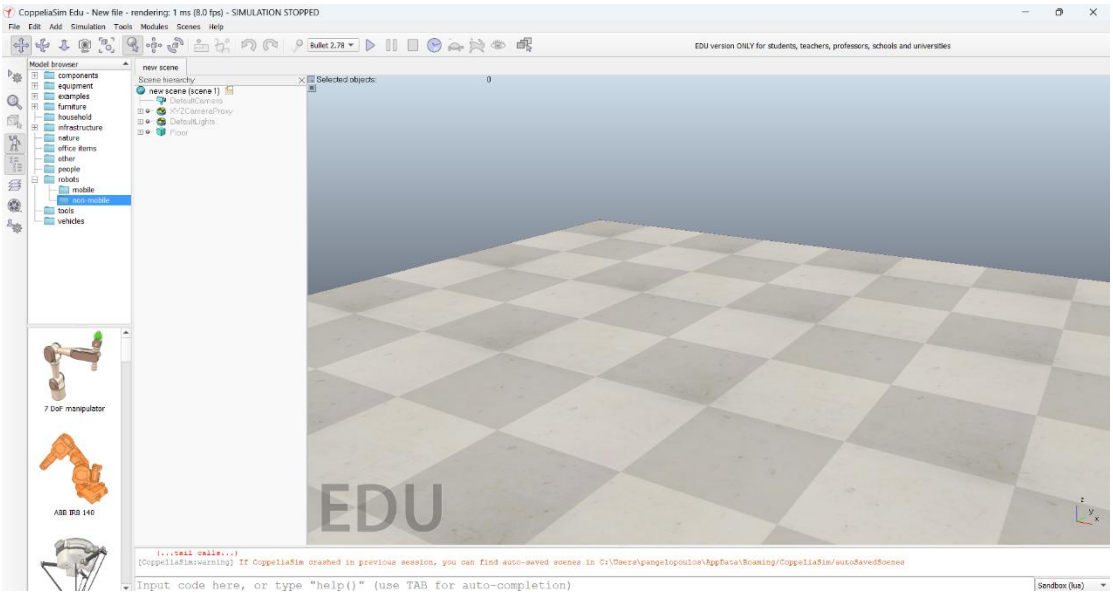
Picture 3. Installing CoppeliaSim

4. Launch CoppeliaSim

- On Windows: double-click coppeliaSim.exe

- On Linux: open a terminal and run ./coppeliaSim.sh
- On macOS: run coppeliaSim from the extracted folder (you may need to allow the app in System Preferences → Security & Privacy)

If everything works correctly, you'll see a 3D workspace with menus and a simulation toolbar.



Picture 4.CoppeliaSIm environment

CoppeliaSim is now ready! You can open the FOSSBot scene (see Section 3) or use the FOSSBot-app to do it automatically.

Teacher Tips

Extract the program to a shared folder if using in a lab environment
Test the launch once per device — no installation means you can copy it to USB or cloud storage
If you're using the FOSSBot-app, you will later link CoppeliaSim to that app's settings

Installing the FOSSBot Simulator App

An easier way to launch and manage your virtual robot

The FOSSBot Simulator App is a lightweight tool developed specifically for the STEPS project. Its purpose is to simplify simulation use in the classroom by automating the launch of CoppeliaSim and loading the virtual FOSSBot model without requiring students or teachers to navigate complex menus.

This app is strongly recommended for all school installations, especially if students are new to robotics or simulation software.

PART E – Pedagogical Scenarios and Real Use Cases

Introduction

All activities, examples, and experiments described in this section can be implemented using the physical FOSSBOT robot. However, for reasons of practicality, accessibility, and classroom efficiency, the virtual environment is primarily used for demonstration and experimentation. The same code, logic structures, and behaviors function identically on the physical robot, provided that the necessary connection, construction, and full assembly of the printed and electronic components have been completed. Teachers and students who wish to work with the real FOSSBOT can recreate the presented examples by constructing the corresponding physical environments or worlds. This dual approach — physical and virtual — ensures that the learning experience remains authentic, flexible, and fully transferable between simulation and real-world robotics practice.

Classroom strategies using Virtual Worlds with or without real robot

1. Pedagogical Rationale

The STEPS learning environments, both virtual and physical, are designed to provide students with a controlled, repeatable, and safe space for robotics exploration. In the virtual worlds, activities are carried out within digital scenarios created by the project team, ensuring uniformity and easy replication across schools. In the physical setting, the same challenges can be reconstructed using real FOSSBOTs, printed components, and classroom setups. Together, these two environments overcome resource limitations while teaching essential robotics, programming, and problem-solving skills through consistent, authentic learning experiences.

2. Teaching Approaches

A. Simulation-based learning (core approach)

- The most of student activities happen inside CoppeliaSim.
- Teachers provide pre-built virtual environments (e.g., mazes, walls, paths) created in STEPS.

B. Scalable classroom model

- Every student (or pair) can have their own “robot” in the simulator.
- No waiting time or hardware bottleneck.

- Teachers can easily reset worlds if errors occur, ensuring smooth flow.

C. Comparative reflection

- Encourage students to discuss how virtual robot behavior might differ if tested in real life (slippage, sensor noise, battery).
- Builds awareness of simulation as a model, not a perfect mirror of reality.

3. Classroom Management Strategies

Classroom Management Strategies – Virtual FOSSBOT

- **Stations not needed**: Since all students can access CoppeliaSim on their laptops, simultaneous practice is possible.
- **Pair programming**: Still effective – driver/navigator roles promote collaboration.
- **Challenge-based learning**: Assign incremental missions (drive forward, turn, follow a line in the world).
- **Debugging practice**: Students run multiple trials without fear of damaging equipment.

Classroom Management Strategies – Physical FOSSBOT

- **Station setup:** Organize small working areas or floor mats where students can safely run their FOSSBOTs. Mark start and finish zones or use tape to define paths.
- **Pair programming**: Maintain clear roles — one student programs while the other handles setup and observation. Switch roles frequently to balance engagement.
- **Challenge-based learning:** Use short, progressive missions such as moving a set distance, turning at specific angles, or following a taped line.
- **Debugging practice**: Encourage students to identify and correct errors directly on the robot. Small, repeated trials help them connect code with real-world motion, building problem-solving confidence and responsibility for the equipment.

4. Curriculum Integration

- **Mathematics**: Coordinate geometry, angles, distances.
- **Computer Science**: Algorithms, conditionals, loops, functions.
- **Physics**: Motion, speed, acceleration (conceptual, as modeled in simulation).
- **STEM Projects**: Navigation challenges, obstacle avoidance in digital mazes, or designing efficient routes.

5. Equity and Access

- **Uniform access**: Every school in STEPS works with the same virtual scenarios.
- **No hardware dependency**: Eliminates inequality between schools with/without robots.
- **Inclusivity**: Students with limited physical access can still fully participate.

6. Teacher Preparation

- Download and test the provided STEPS virtual worlds in advance.
- Ensure all student computers can run CoppeliaSim smoothly.
- Prepare step-by-step coding examples for connecting Python to the simulator.
- Highlight to students that **all activities remain in the virtual world** (no real-world obstacles required).

Preparation Guidelines – Physical FOSSBOT

- Assemble and test each FOSSBot in advance to ensure all printed and electronic components are properly connected and calibrated.
- Check batteries and connections before class to prevent interruptions during activities.
- Prepare sample programs demonstrating basic movement, turning, and sensor interaction, so students can test and adapt them.
- Set up a safe testing area with clear boundaries (e.g., floor mats, tape lines, or obstacle zones) and remind students to handle the robot carefully while experimenting.

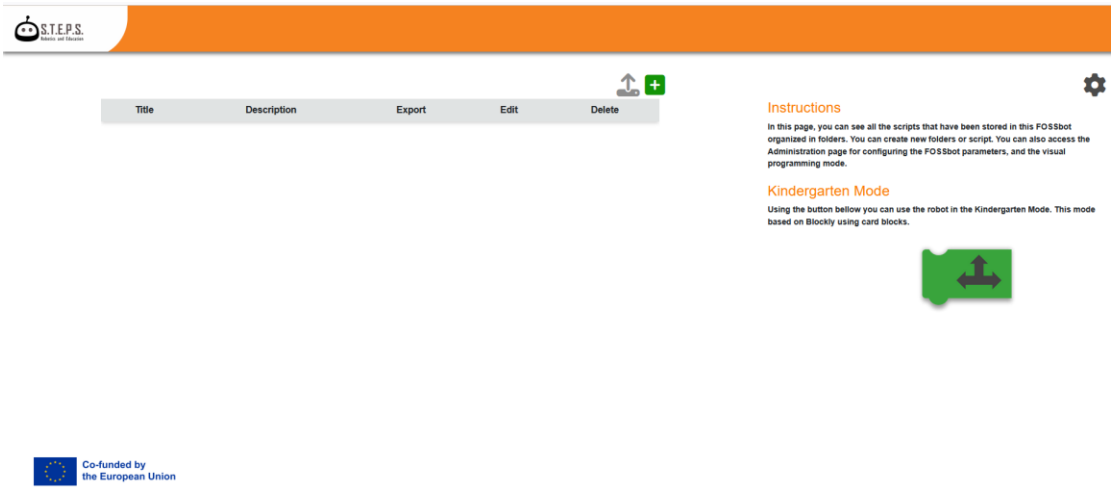
Key Insight: By focusing exclusively on STEPS-designed virtual worlds, teachers guarantee equal opportunities for all learners, scalability for whole classes, and a strong foundation in programming and robotics—without logistical barriers.

Group 1: Ages 5–8 (Kindergarten & Grades 1–2)

As said before the FOSSBot Simulator runs at http://localhost:8081/

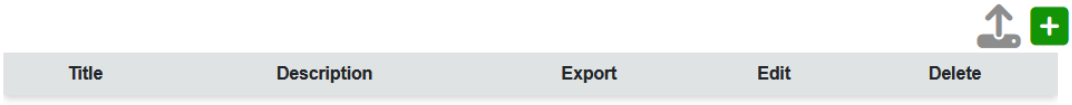
What you're seeing

Empty table (center): your scripts/projects will appear here with Title / Description / Export / Edit / Delete.



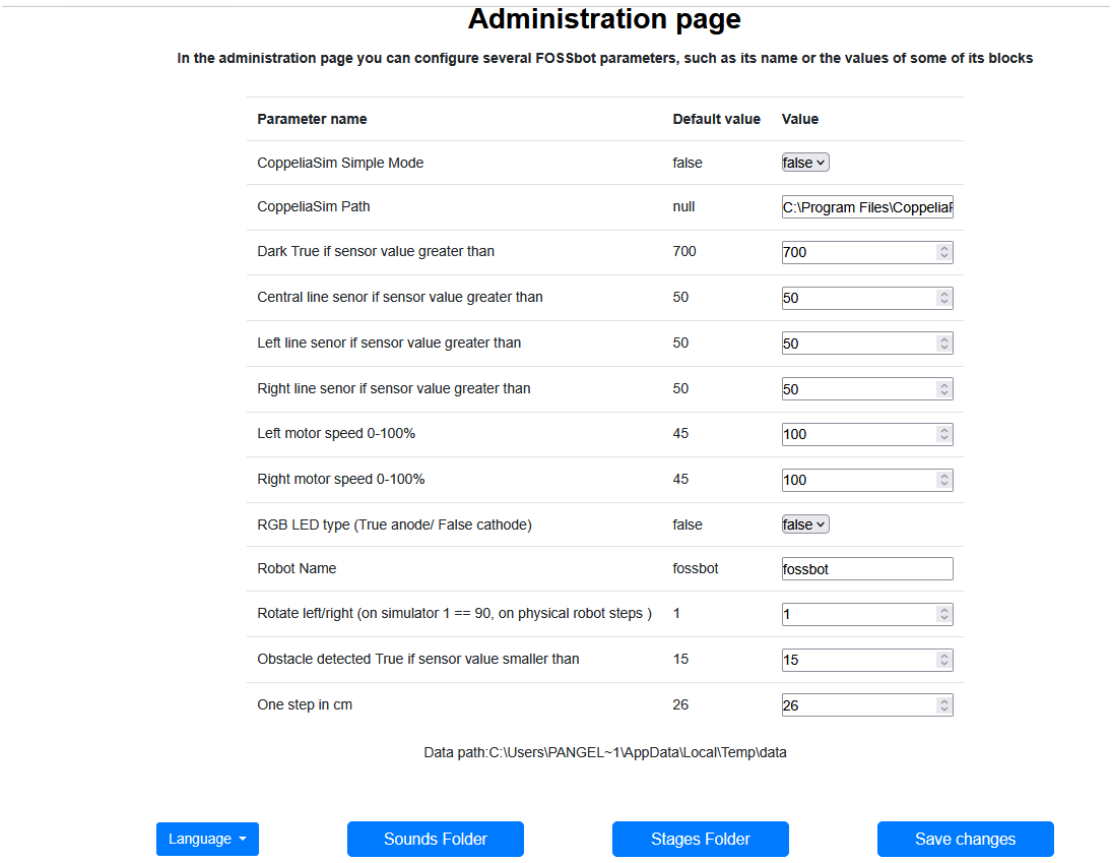
Picture 18. First Screen

Green "+" button (top-right of the table): create a new folder or new script, upload arrow (next to +): import a script/folder you previously exported and a menu including (Title, Descritpion, Export, Edit, and Delete). We will see later what they mean, as they mainly concern classes of students in higher grades (e.g., high school)."



Picture 19

Gear icon (right side): Administration/Settings (robot mode, connection, speeds, etc.).



Picture 20. Gear icon (right side)- Administration/Settings

What each parameter does

CoppeliaSim Simple Mode (true/false)

- *What:* Switches the simulator integration to a very basic control profile (no advanced sensor logic; blocks map to simple time-based moves).
- *When to use:* For quick kindergarten demos, or if your scene has no sensors.
- *Recommendation:* Keep false when you want line/obstacle blocks to use real sensor values from the scene.

CoppeliaSim Path

- *What:* Full path to CoppeliaSim.exe (or app bundle). Lets the FOSSBot app launch the simulator directly.
- *Tip:* You can leave it empty if you start CoppeliaSim yourself.

Dark True if sensor value greater than (e.g., 700)

- *What:* Global rule telling the app which range means “dark” for camera/reflectance readings (often 0–1023).
- *How to set:* Sample the value over black and over white; set this threshold roughly in the middle.

- *Use:* Affects how "line detected?" style blocks interpret the raw value.

Central line sensor if sensor value greater than (0–100)
Left line sensor if sensor value greater than (0–100)
Right line sensor if sensor value greater than (0–100)

- *What:* Per-sensor thresholds (normalized scale) for deciding "on line".
- *How to set:* Place each sensor over the line and the floor, note values, then choose a midpoint (e.g., 50).
- *Tip:* If the robot "hunts" left/right, increase or decrease these by ~5 until stable.

Left motor speed 0–100%
Right motor speed 0–100%

- *What:* Default duty/speed for forward moves in Blockly/Kindergarten Mode.
- *How to set:* Start around 40–60% for pre-primary scenes. If the robot veers, lower the faster side by a few points so both track straight.

RGB LED type (True anode / False cathode)

- *What:* Hardware wiring option for the physical robot's RGB LED.
- *Simulator:* Irrelevant—leave as default.
- *Physical:* Set according to your robot build so LED color blocks behave correctly.

Robot Name

- *What:* The object/model name the app expects in the scene tree.
- *How to set:* Must match your robot in CoppeliaSim (e.g., /FOSSBot). If you renamed the model, update this to keep handle lookups working.

Rotate left/right (on simulator: 1 == 90°, on physical: steps)

- *What:* Defines what **one "rotate step"** means. In the simulator, **1 = 90°** turn.
- *Examples:*
  - 0.5 → 45° turn; 2 → 180° turn.
  - For kindergarten blocks like "Turn Right (1)", you get a quarter-turn.
- *Physical robot:* The value is interpreted as **motor steps**; tune empirically.

Obstacle detected True if sensor value smaller than (e.g., 15)

- *What:* Threshold for proximity/distance readings to count as obstacle present.

- *Units:* Often centimeters (or a normalized “distance”); test once in your scene.
- *How to set:* Put an object at the desired stop distance; read the live value; set slightly **above** that.

One step in cm (e.g., 26)

- *What:* Distance used by “Step Forward/Backward (1)” in Blockly.
- *How to set (quick calibration):* Run Step Forward (1), measure the travel in cm, then change this value to

Buttons & paths

Language

- Switches UI language only.

Sounds Folder

- Opens/sets the folder with audio files you can trigger from sound blocks (use short .wav/.mp3).

Stages Folder

- Folder where **stage backgrounds / layouts** for Kindergarten Mode are kept (e.g., mats, islands). Add your own PNG/JPGs here to appear in the stage picker.

Save changes

- Writes all the above settings to the app’s config.

Data path (read-only line at bottom)

- Where the app stores its **scripts, exports, and config** on your machine. Useful for backups or moving content to another PC.

Quick recommended starting values (virtual / kindergarten)

- **Simple Mode:** false
- **Motor speeds:** Left 50, Right 50 (adjust for straightness)
- **Rotate step:** 1 (quarter-turn = 90°)
- **One step in cm:** 25–30 (scene-dependent)
- **Line sensor thresholds:** 50 each (tune ±5)

- **Obstacle threshold:** 15 (tune to your world scale)

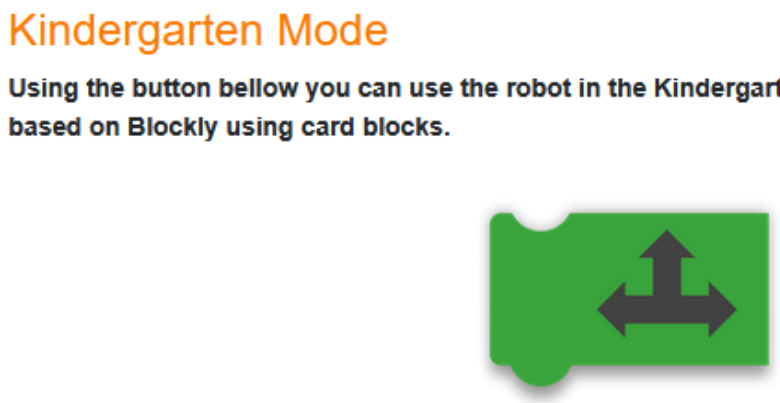
Fast calibration routine (2 minutes)

1. Set Left/Right speed 50 → run Forward (2 s); if it drifts, drop the faster side to 47–48.
2. Run Rotate Right (1); if it's not ~90°, change Rotate… to 0.9 or 1.1.
3. Run Step Forward (1); measure distance; set One step in cm to the measurement.
4. Test line detection on black/white; adjust line thresholds so it triggers on the line but not off it.

Kindergarten Mode area (right, green puzzle-card): opens the Blockly editor for pre-primary use.

Kindergarten Mode is a playful, visual programming space designed for pre-primary learners. Click the green puzzle-card to open the Blockly editor. Children build a program by drag-and-dropping large, color-coded blocks—Forward, Left/Right, Stop (plus optional Light and Sound)—and snapping them into a simple left-to-right sequence. Each block uses friendly units (seconds, steps, quarter-turns) so educators can fine-tune movement without typing.

Press Run to watch the virtual FOSSBot in CoppeliaSim follow the plan; use Reset to return to the start and try again. Save stores the activity in your script list; Export lets you share it with colleagues. The mode emphasizes sequencing, prediction, and gentle debugging with minimal text, making it ideal for non-readers and inclusive classrooms. For best results, keep programs short, change one thing at a time, and celebrate small wins.



Picture 21. Kindergarten Mode Button

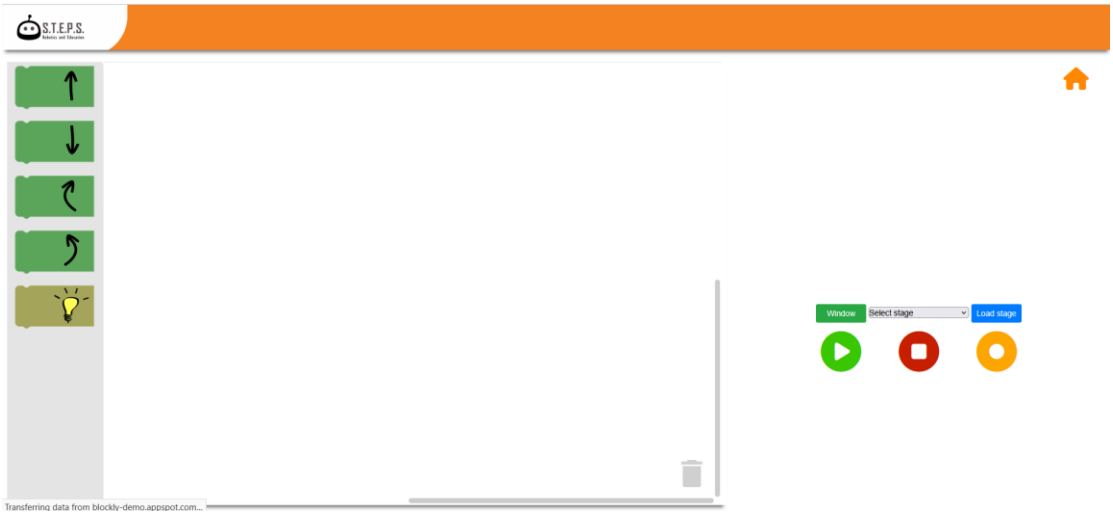
What's on the screen

1) Left toolbox (green blocks) – "actions"

- **Forward** (↑): move the robot forward.

- **Backward** (↓): move the robot backward.
- **Turn Left** (curved ↶): rotate left.
- **Turn Right** (curved ↷): rotate right.
- **Light** (bulb, yellow block): toggle a light/feedback action (useful for cues).

When you drop a block into the program, most blocks let you set a simple value (e.g., *steps*, *seconds*, or *turn units*).



Picture 22. On the screen

2) Center workspace (white area) – "your program"

- Drag blocks from the left and snap them top-to-bottom to form a sequence.
- Reorder by dragging; delete by dropping on the trash can (bottom-right).
- The scrollbar lets you navigate long sequences.

3) Right control panel

- **Stage controls:**
  - **Window**: opens/closes the stage preview in a separate window.
  - **Select stage → Load stage**: choose a background/mat (islands, grid, etc.). Stages are read from the *Stages Folder* (see gear ▶ Administration).
- **Run controls:**
  - **Green Play**: execute your block sequence in CoppeliaSim.
  - **Red Stop**: halt immediately.
  - **Orange Reset**: return the robot/stage to the start state (blocks stay).

4) Home (top-right house)

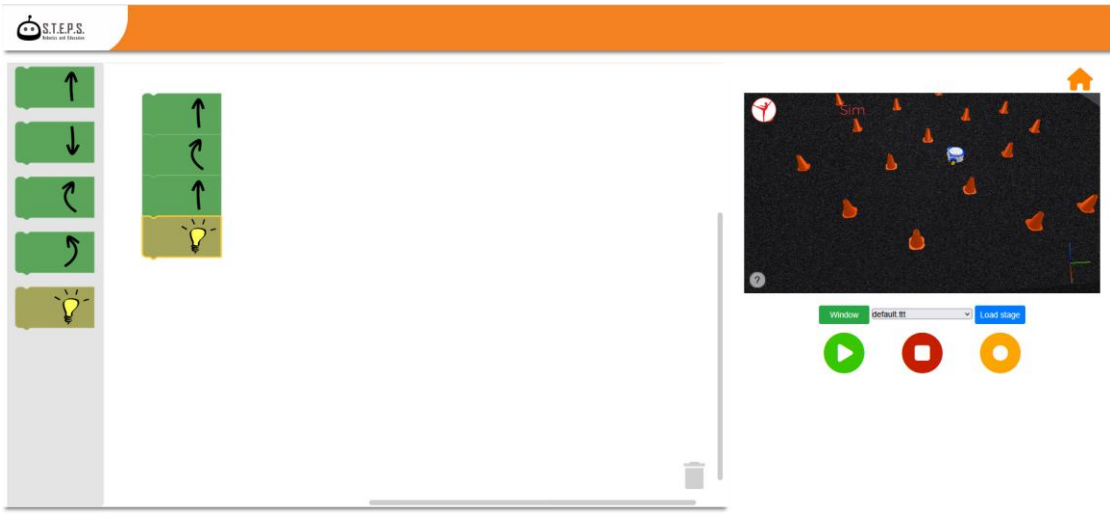
- Go back to the app's start page (script list). Use this when you finish.

How blocks map to movement

- **Forward/Backward:** distance is in steps or seconds (depending on the block). The length of one step uses the admin setting **"One step in cm"**.
- **Turn Left/Right:** rotation uses the admin setting **"Rotate left/right (1 == 90° on simulator)"**.
- **Speed:** forward speed uses **Left/Right motor speed (0–100%)** from Administration.
- **Sensors/lights (if used):** thresholds come from the line/obstacle settings in Administration.

First run (recommended flow)

1. Open **CoppeliaSim**, load your STEPS scene with FOSSBot, press **Play**.
2. In Kindergarten Mode, **Select stage → Load stage** (e.g., default.ttt).
3. Build: **Forward (2)** → **Turn Left (1)** → **Forward (1)** → **Light**.
4. Press **Play**. If the turn is too wide/narrow, change the **Turn** amount (0.8–1.2). If distance is off, tweak **steps** or adjust **One step in cm** in Administration.



Picture 23. First run

Right-Click Menu on Blocks: What Each Option Does

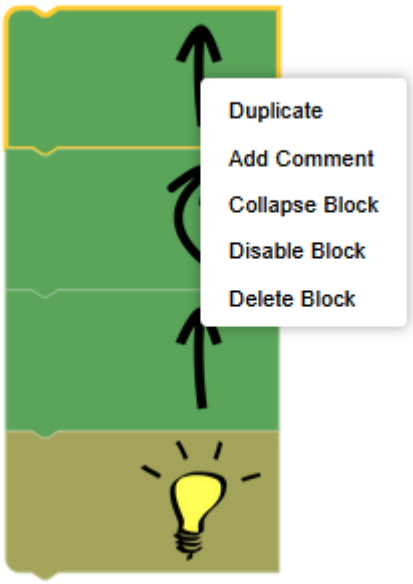
Here's what each **right-click menu** item on a Blockly block does in Kindergarten Mode:

- **Duplicate.** Makes an exact copy of the selected block **including any blocks attached beneath it**. The copy appears next to the original so you

can drag it into place. Great for repeating similar steps (e.g., two Forward blocks).

- **Add Comment.** Attaches a small note to the block. A comment bubble appears; click it to show/hide your note (e.g., “short turn”). Comments are saved with the program and don’t affect execution.
- **Collapse Block.** Folds the block (and its children) into a compact, single-line preview to reduce clutter. When a block is collapsed, the menu option changes to **Expand Block** to show it again. Collapsing doesn’t change behavior.
- **Disable Block.** Greys out the block so it **doesn’t run**, but stays in the workspace. Use this to test alternatives without deleting your original idea. Re-enable via the same menu item (it will read **Enable Block**).
- **Delete Block.** Removes the selected block **and any blocks attached below it**. Use with care. If you delete by mistake, you can typically undo (e.g., with the Undo button or keyboard shortcut, depending on your setup).

Tip: Prefer **Disable** for quick experiments, **Collapse** to tidy long programs, and **Duplicate** to build sequences faster.



Picture 24. Right-Click Menu on Blocks

Quick troubleshooting

- **Nothing moves:** CoppeliaSim must be running & playing; check Administration → connection/mode.
- **Robot veers:** reduce the faster motor speed by 1–3 points in Administration.
- **Turns not 90°:** adjust the **Rotate…** value (e.g., 0.95 or 1.05).
- **Stage not visible:** click **Window** or re-**Load stage**.

Teaching tips

- Keep programs **short (3–6 blocks)**; change **one thing at a time**.
- Use verbal cues ("forward… now turn left") and celebrate quick fixes.
- Pair blocks with printed **command cards** so non-readers can plan first.

Controlling the Stage with the Mouse (Kindergarten Mode)

Use these gestures in the **Stage window** (opened via **Window → Load stage**). They work on a mouse or trackpad (touch equivalents in italics).

Left-click + drag — Pan. Move the mat/background in any direction. *On a trackpad: two-finger drag.*

Mouse wheel / two-finger pinch — Zoom. Scroll the wheel to zoom **in/out**. Zoom is centered on the pointer, so point where you want to zoom. *On a trackpad: pinch to zoom; two-finger up/down often zooms too, depending on OS.*

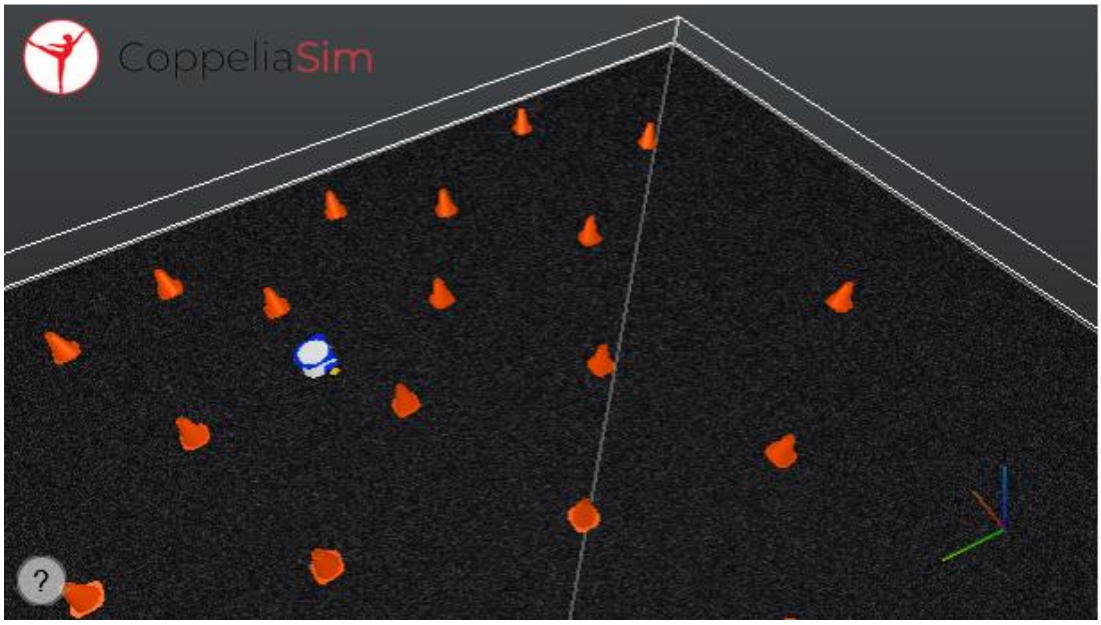
Right-click — Context actions. Opens the stage's quick menu (when available in your build). Typical items include **Reset view**, **Center on robot/start**, **Show/Hide grid**. If your build has no menu, use the **orange Reset** button to restore the start state.

Middle-click (or Shift + left-drag) — Alternate pan. Another way to pan if your wheel is busy with zoom or on compact mice.

Double-click — Center view. Double-click any point on the stage to quickly center the camera there (handy after zooming).

Home / Reset buttons

- **Orange Reset**: returns robot and stage to the initial state (your blocks stay).
- **House icon (top-right)**: leaves Kindergarten Mode and returns to the main screen (does not close CoppeliaSim).



Picture 25. Controlling the Stage with the Mouse

Tips

- For precise navigation: zoom in near the goal, pan a little, then zoom out to check the route.
- If zoom feels inverted, change the scroll/zoom direction in your OS settings.
- On touchpads, if pinch doesn't zoom, enable "pinch-to-zoom" in system preferences.

Kindergarten Scenario — "Chessboard Treasure Hunt" (FOSSBot in CoppeliaSim)

A single, classroom-ready activity using the **Chessboard stage** in **Kindergarten Mode (Blockly)**. Ideal for first contact with sequencing and spatial reasoning.

Age group & ISCED level

4–8 years, **ISCED 0 (pre-primary, two first grades of primary)**

Learning goals

- Build **sequencing** with 3–6 blocks (forward/turn/stop).
- Connect **cause → effect**: predict → run → observe → fix one thing.
- Use **positional language** (forward, left, right, square, turn).
- Practice **estimation** (1 step ≈ 1 square) and **counting**.
- Collaborate (roles, turn-taking, explaining choices).

Subject links

- **Math:** counting, measurement (1 step per square), 90° turns, shapes (L, square).
- **ICT/CS:** visual programming (Blockly), debugging, algorithms as steps.
- **Language:** action verbs; explaining a plan; reflection.
- **Art/Design (light touch):** place a "treasure" icon on a square; celebrate with Light/Sound blocks.

Required tools

- **Simulation only:** CoppeliaSim + FOSSBot model loaded and **playing**.
- **FOSSBot Simulator App → Kindergarten Mode (Blockly)**.
- **Stage: Chessboard** (from Stages folder).
- Optional but helpful: printed **command cards** (Forward / Left / Right / Stop) and a **6-slot sequencing mat**.

Programming (Kindergarten Mode settings & blocks)

- **Blocks used:** Forward, Backward (optional), Turn Left, Turn Right, Stop, (optional) Light/Sound.
- **Admin (gear) starting values:**
  - **Rotate step = 1** (≈ 90°).
  - **One step in cm = chessboard square size** (e.g., if a square ≈ 25–30 cm in your scene, enter that).
  - **Left/Right motor speed = 50 / 50** (adjust ±2 if it drifts).

Step-by-step instructions (35–40 minutes)

1) Launch & load (2')

- Open **Kindergarten Mode**, **Select stage → Chessboard → Load stage**.
- CoppeliaSim will be loaded automatically

2) Show the toolbox (2')

- Point to the **green blocks** (Forward / Left / Right / Stop).
- Explain: "Each block is one action. We stack actions top-to-bottom."

3) Calibrate once (5')

- Build Forward(1) → **Run**. The robot should land near the center of the next square.
  - If not, change **One step in cm** slightly and retry.
- Build Turn Right(1) → check quarter turn; if off, set **Rotate step** to **1 (=90°)**

4) Plan with cards (4')

- On the table, place cards to go from **Start** to a marked **Treasure square** (e.g., 2 squares forward, turn right, 1 square forward).

5) Translate to blocks & run (8')

- Drag blocks to match the plan; **Run**.
- If it misses, apply the **"change one thing"** rule (usually steps).

6) New mission (8')

- From start: **visit the treasure, then return** to start.
- Encourage counting backwards or using a 180° turn (Right(2) or Left(2)).

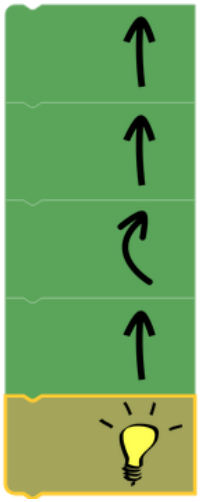
7) Share & reflect (5')

- Ask two teams to show their program and name the **one change** that fixed it.

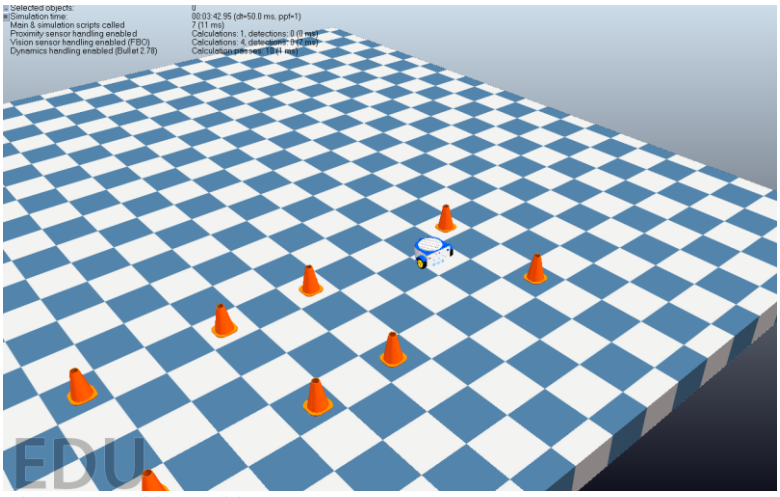
Block examples (ready to build)

A. L-path to a nearby treasure:

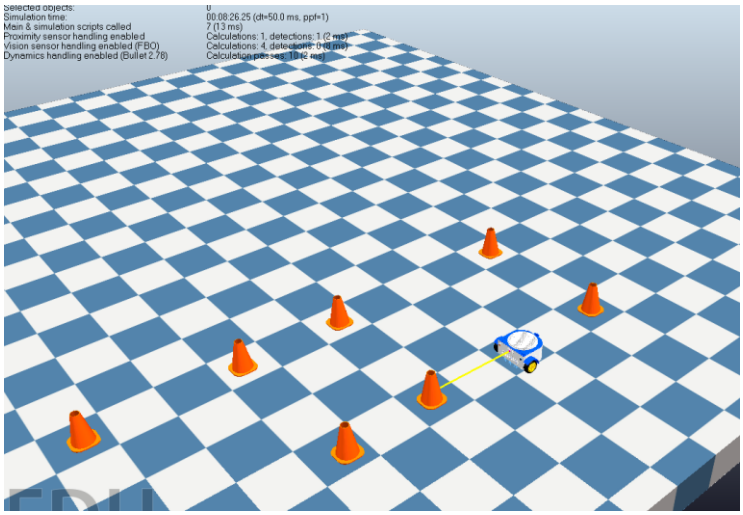
(Forward → Forward → Turn Right → Forward → Light)



Picture 26.



Picture 27. Initial Position



Picture 28. Position After running the blocks

Activity Sequencing with the Physical FOSSBOT

Activity Overview

This hands-on activity introduces young learners (ages 4–8) to sequencing and spatial reasoning using the physical FOSSBOT. It mirrors the *Chessboard Stage in Kindergarten Mode (Blockly)*, allowing students to build, test, and debug short movement sequences on a real grid floor.

Preparation and Setup

1. **Stage Design**
    - Create a large chessboard-style grid on the floor using **colored tape**. Each square should measure about **25–30 cm**, matching one robot step.
    - Mark a **Start** square and one or more **Treasure squares** using stickers or icons.

2. **Robot Preparation**
    - Ensure each FOSSBOT is **assembled, charged, and calibrated**.
    - Connect via **Wi-Fi or USB** to the **FOSSBOT App** and open **Kindergarten Mode (Blockly)**.
    - In the Admin (gear) settings:
        - *Rotate step* = 1 (≈ 90°)
        - *One step in cm* = floor square length (e.g., 25 cm)
        - *Motor speed* = 50 / 50 (adjust slightly if it drifts)

Classroom Flow (≈ 35–40 minutes)

1. **Introduction (2 min)** – Show students how FOSSBOT follows block commands (Forward, Left, Right, Stop, Light).
2. **Calibration (5 min)** – Run **Forward (1)** to verify the robot moves one square; adjust settings if needed.
3. **Planning (5 min)** – Use printed **command cards** or a 6-slot **sequencing mat** to plan a route to the treasure.
4. **Programming (10 min)** – Build the sequence in Blockly and run it; apply the "change one thing" rule for debugging.
5. **Challenge (8 min)** – Add a *return to start* or *L-path* variation.
6. **Sharing & Reflection (5 min)** – Teams present their program and discuss one improvement they made.

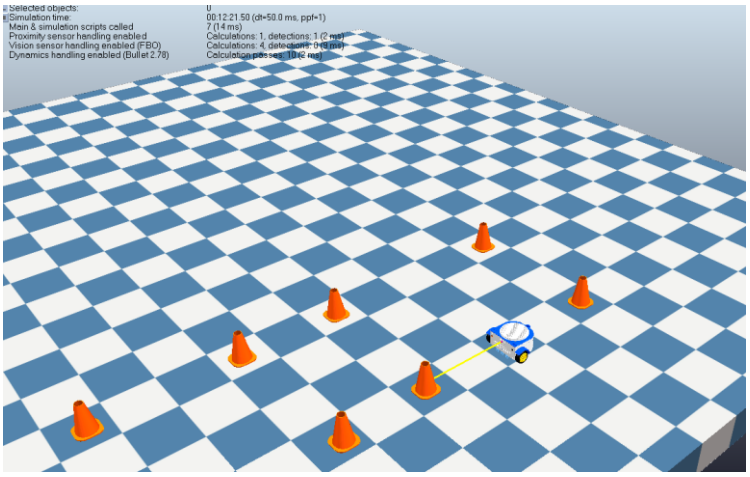
Example Program (ready to build)

Forward → Forward → Turn Right → Forward → Light
Goal: Move two squares forward, turn right, reach the treasure, and celebrate by turning on the light.

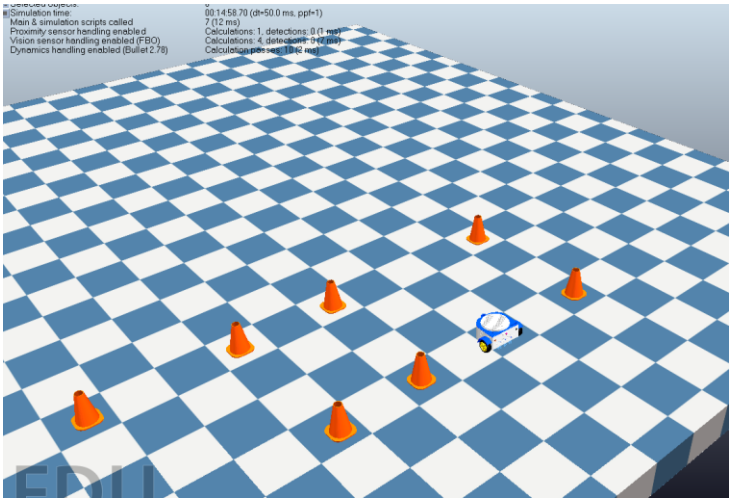
B. Square walk (shape practice):

Forward(1) → Right(1) → Forward(1) → Right(1) → Forward(1) → Right(1) → Forward(1) → Light

Picture 29



Picture 30. Initial Position



Picture 31. Position After running the blocks

Activity Sequencing Square Walk" – Shape Practice with the Physical FOSSBot

Activity Overview

This activity helps young learners (ages 4–8) recognize and create geometric shapes using the **physical FOSSBOT**. Students will program the robot to trace a **square path**, connecting movement and direction with early geometry concepts. It builds on the same sequencing principles introduced in the *Chessboard Stage (Kindergarten Mode)* and adds repetition, shape recognition, and estimation practice.

Preparation and Setup

1. **Stage Design**
   - Use the same **chessboard-style floor grid** (25–30 cm squares).
   - Mark a **Start** position with colored tape. The robot will return close to this square after completing the square path.
2. **Robot Preparation**
   - Check that each FOSSBOT is **charged, connected**, and **calibrated** in *Kindergarten Mode (Blockly)*.
   - Admin settings:
     - *Rotate step* = 1 (≈ 90°)
     - *One step in cm* = size of one floor square
     - *Motor speed* = 50 / 50

Classroom Flow (≈ 35 minutes)

1. **Warm-up (5 min)** – Review previous commands: *Forward*, *Turn Right*, *Light*.
2. **Planning (5 min)** – Ask: "What shape has four equal sides and four turns?"
   Have students predict how many *Forward* and *Turn* blocks are needed.
3. **Programming (10 min)** – Students build and run the sequence:
   Forward → Right → Forward → Right → Forward → Right → Forward → Light.
   Encourage them to count turns aloud and check alignment after each move.
4. **Debugging (8 min)** – If the robot drifts, adjust motor speed slightly or correct rotation steps.
5. **Extension (7 min)** – Challenge groups to draw a different shape (e.g., rectangle, triangle) by changing the number of steps or turns.

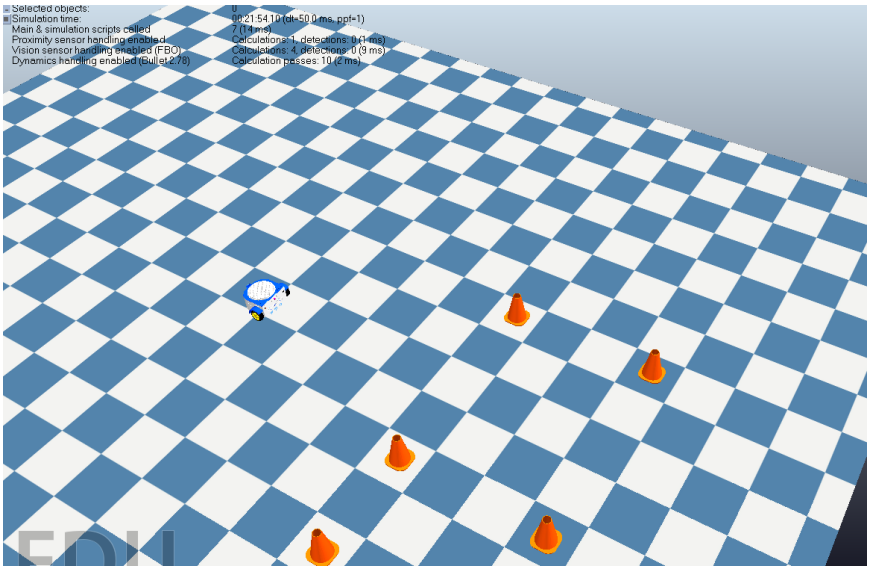
Learning Goals

- Understand that four 90° turns create a closed square path.
- Strengthen sequencing, repetition, and spatial awareness.
- Apply counting and estimation (1 step ≈ 1 square).
- Develop basic debugging and collaboration skills.

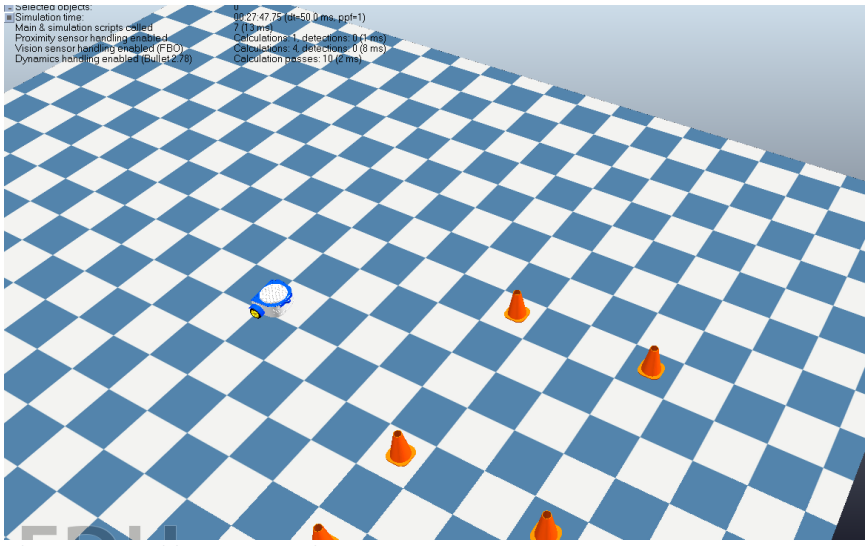
C. Go & Return:

Forward(3) → Left(1) → Forward(1) → Right(2) → Forward(1) → Right(1) → Forward(3) → Light

Picture 32



Picture 33. Initial Position



Picture 34. Position After running the blocks

Troubleshooting (fast fixes)

- **Nothing moves:** CoppeliaSim must be **playing**; click **Play** (green) in Blockly; reload stage.
- **Turn not 90°:** tweak **Rotate step** (0.9–1.1).
- **Distance off:** adjust the block's step number or refine **One step in cm**.
- **Veering:** nudge motor speeds (e.g., Left 48 / Right 50).

Why this works (trainer note)

The chessboard gives a clean 1:1 mapping between steps and squares and a stable 90° turn unit. That lets teachers model computational thinking with minimal cognitive load: plan → run → compare → adjust one variable. It scales effortlessly to shapes, paths, and simple optimization ("fewest blocks").

Activity Title: "Go & Return" – Planning and Reversal with the Physical FOSSBot

Activity Overview

In this activity, students guide the **physical FOSSBOT** on a journey to a target square and back to the starting point, reinforcing the concepts of **path reversal**, **directional logic**, and **sequential reasoning**. It combines counting, orientation, and simple geometry, helping learners understand that a path can be retraced by reversing steps and turns.

Preparation and Setup

1. **Stage Design**
   - Use the same **chessboard floor grid** (25–30 cm per square).
   - Mark a **Start** square and a **Target square** positioned about three squares ahead and one square to the left.
   - Place a "Treasure" or "Goal" icon on the target square to motivate students.
2. **Robot Preparation**
   - Ensure FOSSBOTs are **assembled, charged**, and **connected** to the **FOSSBOT App (Kindergarten Mode – Blockly)**.
   - In Admin (gear) settings:
     - *Rotate step* = 1 (≈ 90°)
     - *One step in cm* = grid square size (e.g., 25–30 cm)
     - *Motor speed* = 50 / 50

Classroom Flow (≈ 40 minutes)

1. **Introduction (5 min)** – Remind students how to use *Forward*, *Turn Left*, *Turn Right*, and *Light* blocks. Explain that the robot will "go visit the treasure and then return home."
2. **Planning (6 min)** – Use printed **command cards** to plan the route: move forward 3 squares, turn left, move 1 square, then turn around and retrace.

3. **Programming (10 min)** – Build and test the following sequence:
   🧩 **Forward(3) → Left(1) → Forward(1) → Right(2) → Forward(1) → Right(1) → Forward(3) → Light**
   Students observe how two *Right(1)* turns equal a 180° turn (facing back).
4. **Debugging (8 min)** – If alignment drifts, adjust *motor speed* slightly or recheck the number of squares. Encourage "change one thing" corrections.
5. **Extension (8 min)** – Challenge groups to create a similar "go & return" route but to a new direction (e.g., right first instead of left).
6. **Reflection (3 min)** – Ask: "What did we change to make the robot come back to start?"

Learning Goals

- Strengthen sequencing and orientation (left vs. right, forward vs. reverse).
- Understand how to reverse a path using turns and step counting.
- Reinforce concepts of distance, direction, and logical reasoning.
- Foster collaboration and communication during problem-solving.

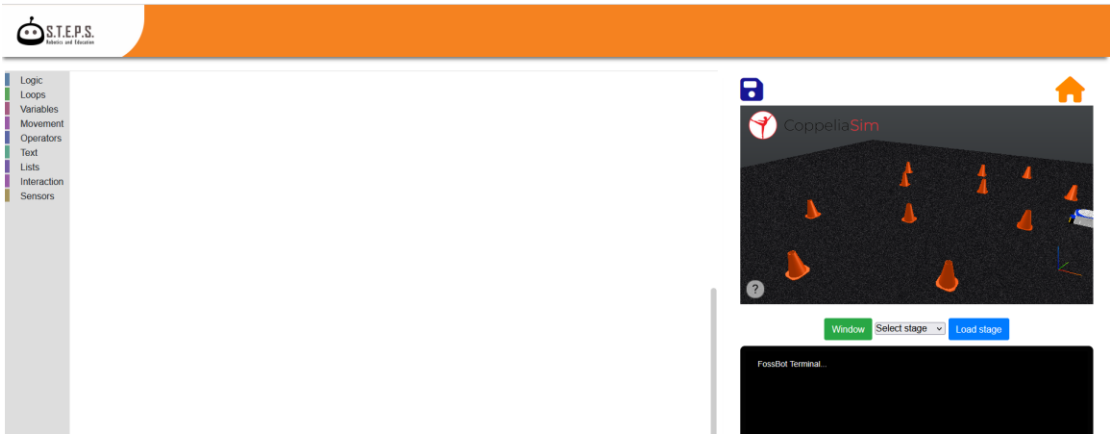
Load the programming stage for older student ages

In the context of the STEPS project, several simulation stages have already been created and developed to support the scenarios presented above. Each stage corresponds to a specific type of sensor use or robotic behavior. Depending on the chosen scenario, the appropriate stage should be loaded so that students can experiment directly with the intended functionality. At the same time, teachers may adapt or slightly modify these ready-made stages to design alternative scenarios that better match their pedagogical goals. The available stages include: sens1_obstacle_avoid,

- sens2_follow_line,
- sens3_light_sensor,
- sens4_velocity_calculation,
- sens5_noise_detection,
- sens6_edge_detection, sens7_fall_detection,
- sens8_brightest_light,
- sens9_changing_rgb,
- final_project,
- Template

How to Load the FOSSBot Application with the Programming Stage

1. **Locate the FOSSBot application file** on your computer.
   - On **Windows**: usually a .exe file.
   - On **Linux/Mac**: usually a script (.sh) or an app bundle.
2. **Double-click the file to run it.**
   - The **FOSSBot Block Programming environment** will open (the one you showed in your screenshot).
3. **CoppeliaSim will start automatically in the background.**
   - You don't need to open or load a stage manually in CoppeliaSim.
   - The correct virtual environment (stage) is launched together with the FOSSBot app.
4. You will now see:
   - On the **left side**: block categories (Logic, Loops, Movement, etc.).
   - On the **right side**: the CoppeliaSim window with the loaded stage (cones, maze, etc.).
5. From here you can simply **drag blocks** into the workspace and press **Run** ▶.
   The robot will move inside the loaded stage automatically.



Picture 35. Blocs programming stage

In the FOSSBot block programming environment, commands are organized into categories. Each category groups blocks that share a common purpose, making it easier for students to design programs. This visual approach allows learners to build programs step by step, by dragging and connecting blocks like puzzle pieces. By combining blocks from different categories, students can create logical sequences, control the robot's movement, repeat actions, react to sensor input, and interact with their environment.

This structured design not only simplifies programming but also introduces the fundamental concepts of computer science in a playful and accessible way. Each category serves as a building block of computational thinking: **Logic for decisions, Loops for repetition, Variables for memory, Movement for actions, Operators for calculations, Text for messages, Lists for collections, Interaction for communication, and Sensors for awareness of the world around the robot.**

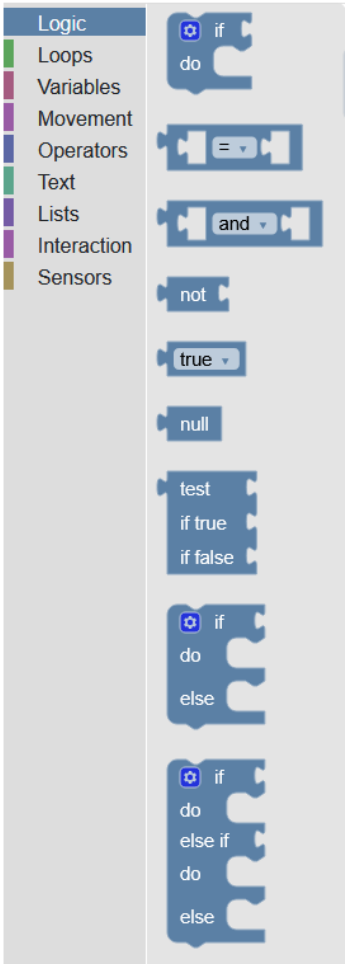
Block Categories Overview (FOSSBot – Block Programming Environment)

1. **Logic.** Blocks for conditions, comparisons, and Boolean values (true/false).
   Used when you want the robot to decide something (e.g., *if sensor detects obstacle then stop*).
2. **Loops.** Blocks that repeat actions a certain number of times or until a condition is met.
   Used for repeating movements (e.g., *repeat forward 4 times to make a square*).
3. **Variables.** Blocks to create, set, and change values stored in variables.
   Useful for counting steps, storing sensor readings, or tracking scores.
4. **Movement.** Blocks to move the robot forward, backward, turn left/right, stop, control speed, etc.
   These are the main action commands for navigation.
5. **Operators.** Blocks that handle numbers and strings (add, subtract, compare, join text, etc.). Often used inside conditions or variable assignments.
6. **Text.** Blocks that work with text (strings).
   For example, displaying or combining messages.
7. **Lists.** Blocks for handling collections of data.
   You can store multiple values (e.g., a sequence of directions) in one variable.
8. **Interaction.** Blocks for communicating with the user (ask, answer, print).
   Useful in Python mode, but sometimes also connected with GUI feedback.
9. **Sensors.** Blocks that read from FOSSBot's sensors (distance, line, etc.).
   Allow the robot to react to the environment.

Logic Blocks

Logic blocks are used to make decisions in a program. They allow the robot to "think" by checking conditions and acting differently depending on the results. For example, the robot can check if a path is free, if a sensor detects an obstacle, or if a value is equal to another. Logic is at the core of programming because it brings **decision-making** into action: instead of executing steps blindly, the robot adapts to situations. These blocks introduce learners to fundamental computer science concepts such as **if statements, comparisons, and Boolean logic (true/false values)**, which are present in every programming language.

Logic Blocks Explanation



Picture 36. Logic Blocks

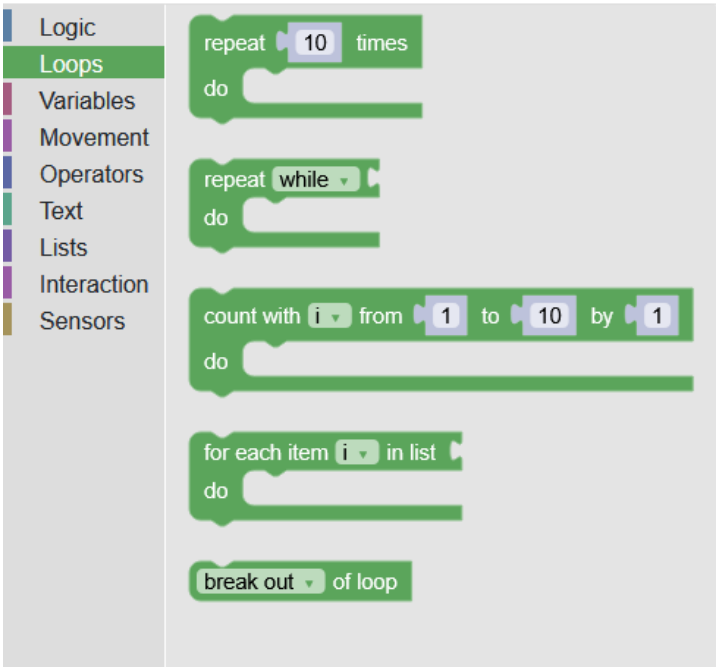
1. **if – do**
    - Executes a block of code only if a condition is true.
    - Example:
      If the distance sensor detects an obstacle, stop moving.
2. **Comparison (=, ≠, <, >, ≤, ≥)**
    - Compares two values and returns true or false.
    - Example:
      Check if steps = 5.
3. **and / or**
    - Combines conditions.
    - and: both conditions must be true.
    - or: at least one condition must be true.
    - Example:
      *If sensor detects obstacle **and** steps > 3, then stop.*
4. **not**
    - Reverses the truth of a condition.
    - Example:
      If not obstacle detected → move forward.
5. **true / false**
    - Boolean values, used as conditions.

- Example:
  Repeat while true → creates an infinite loop (use with caution).

6. **null**
   - Represents an empty value.
   - Example:
     Set variable = null when you want to clear it.
7. **test (if true / if false)**
   - A conditional block that chooses between two paths depending on the test result.
   - Example:
     If sensor < 10 → go backward, else → go forward.
8. **if – do – else**
   - Executes one block if condition is true, another if false.
   - Example:
     If button pressed → beep, else → move forward.
9. **if – do – else if – do – else**
   - Multiple branching decisions.
   - Example:
     If distance < 5 → stop. Else if distance < 10 → slow down. Else → move normally.

Text – Loops

Loops allow the robot to **repeat actions** without writing the same code multiple times. They are one of the most important tools in programming because they save time, reduce errors, and make programs shorter. With loops, you can instruct the robot to move forward several times, repeat until a condition is met, or go through a list of values. This teaches learners the concept of **iteration**—doing something again and again until a goal is achieved.



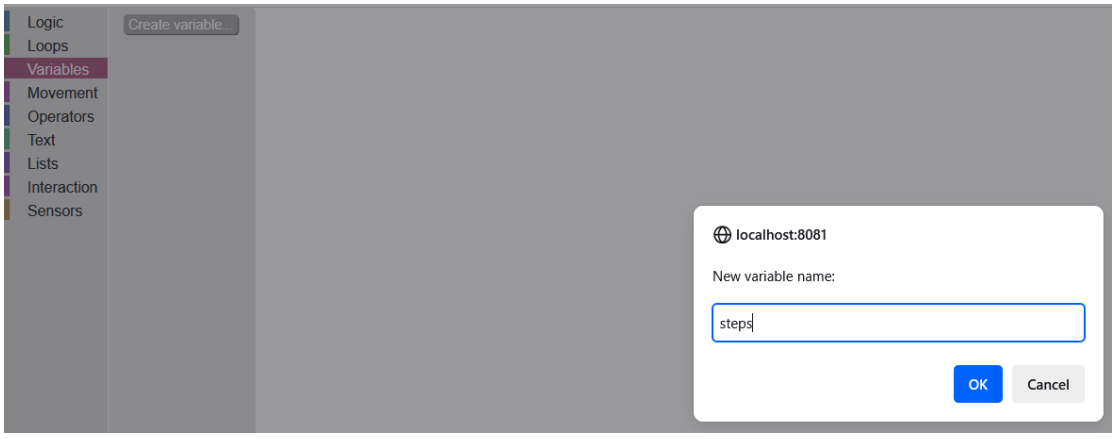
Picture 37. Loops Blocks

Loops Blocks Explanation

1. **repeat [n] times – do.** Repeats the enclosed actions exactly *n* times. This block is used when you know in advance how many times an action should be repeated.
2. **repeat while [condition] – do.** Keeps repeating the actions as long as a condition is true. It is useful when you don't know beforehand how many repetitions will be needed.
3. **count with [i] from [1] to [10] by [1] – do.** Uses a counter variable that increases or decreases each time. This block is helpful for running actions a fixed number of times with a changing value.
4. **for each item [i] in list – do.** Goes through every element in a list one by one. This block is used to process or act on each item of a collection.
5. **break out of loop.** Immediately stops the current loop, even if it has not finished. It is useful when a condition is met that makes further repetition unnecessary.

Variables

Variables are like **named containers** that store information for later use. They can hold numbers, text, or other values that may change while the program is running. By using variables, the robot can **remember things**, such as how many steps it has taken, whether a button was pressed, or the result of a sensor reading. Variables make programs flexible, because instead of working only with fixed numbers, they allow the robot to work with changing values.



Picture 38. Create a variable called steps

Variables Blocks Explanation

1. **Create variable**
   - Opens a dialog box where you give a new variable a name (e.g., steps, count, distance).

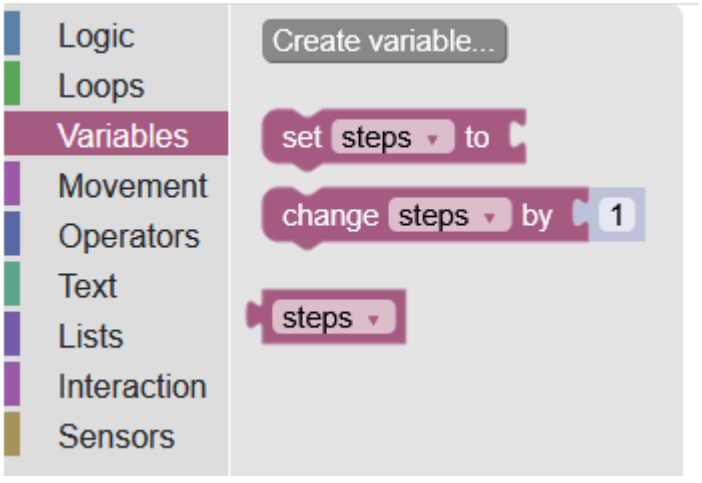
- Once created, the variable appears as a block that can be used in other categories (for setting, changing, or checking its value).

2. **Set [variable] to [value]**
   - Assigns a new value to the variable.
   - Example: *Set steps to 0 at the start of a program.*
3. **Change [variable] by [value]**
   - Increases or decreases the value stored in the variable.
   - Example: *Change steps by 1 each time the robot moves forward.*
4. **[variable] (get value)**
   - Returns the current value stored in the variable.
   - Example: *Check if steps = 10 to decide when to stop.*
5. **Delete variable** (optional, depending on environment)
   - Removes a variable if it is no longer needed.

Blocks that appear after creating a variable

1. **set [steps] to [ ]**
   - Sets the variable to a specific value.
   - Example: *Set steps to 0 at the beginning of the program.*
2. **change [steps] by [1]**
   - Increases (or decreases if you use a negative number) the value of the variable.
   - Example: *Change steps by 1 each time the robot moves forward.*
3. **[steps]**
   - This is the variable itself (a "get" block).
   - It returns the current value stored inside steps.
   - Example: *Use steps in a comparison (if steps = 10 then stop).*

Every time you create a **new variable** (e.g., distance, score, count), these **same three blocks** will appear but with the new variable's name.



Picture 39. After we create a variable, the system automatically generates the corresponding blocks for using it.

Once you **create a variable**, new blocks automatically appear in the **Variables category** for that specific variable.

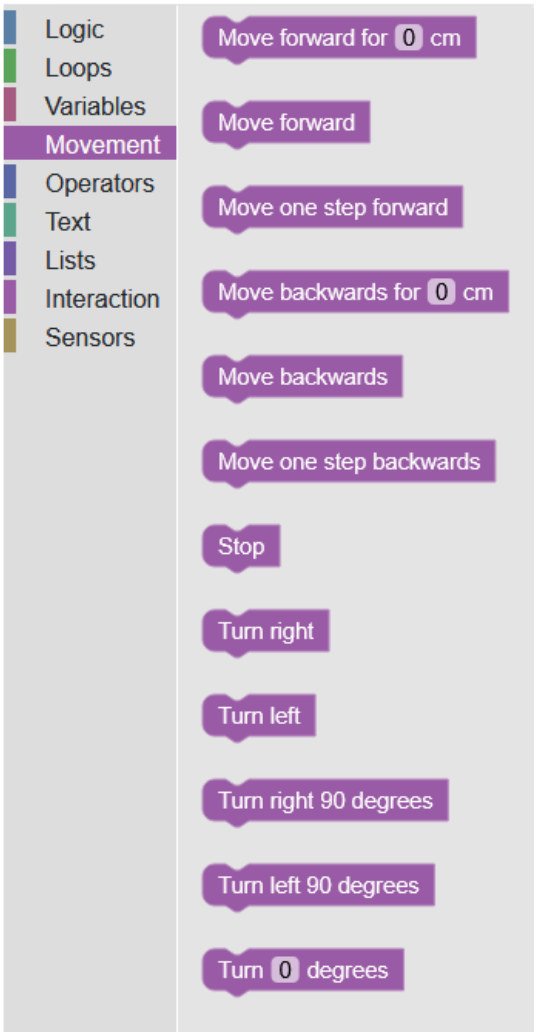
Movement Blocks

Movement blocks control the **physical actions** of the robot. They allow it to move forward, backward, or turn in different ways. These blocks are the foundation of any program, since they transform instructions into visible robot behavior. Learners can use precise measurements (in centimeters or degrees) for accuracy, or simple step commands for easier control. With Movement blocks, students begin to connect programming with the robot's real-world actions.

Movement Blocks Explanation

1. **Move forward for [n] cm.** Moves the robot forward by an exact distance in centimeters. Useful for precise paths.
2. **Move forward.** Moves the robot forward continuously until another command (like stop) is given.
3. **Move one step forward.** Moves the robot forward by one predefined unit (step). Simple for beginners.
4. **Move backwards for [n] cm.** Moves the robot backward by an exact distance in centimeters.
5. **Move backwards.** Moves the robot backward continuously until stopped.
6. **Move one step backwards.** Moves the robot backward by one predefined step.
7. **Stop.** Stops all movement immediately. Essential to end programs safely.
8. **Turn right.** Turns the robot right continuously until another command interrupts.
9. **Turn left.** Turns the robot left continuously until interrupted.
10. **Turn right 90 degrees.** Turns the robot exactly 90° to the right.
11. **Turn left 90 degrees.** Turns the robot exactly 90° to the left.
12. **Turn [n] degrees.** Turns the robot by a chosen angle (positive = right, negative = left).

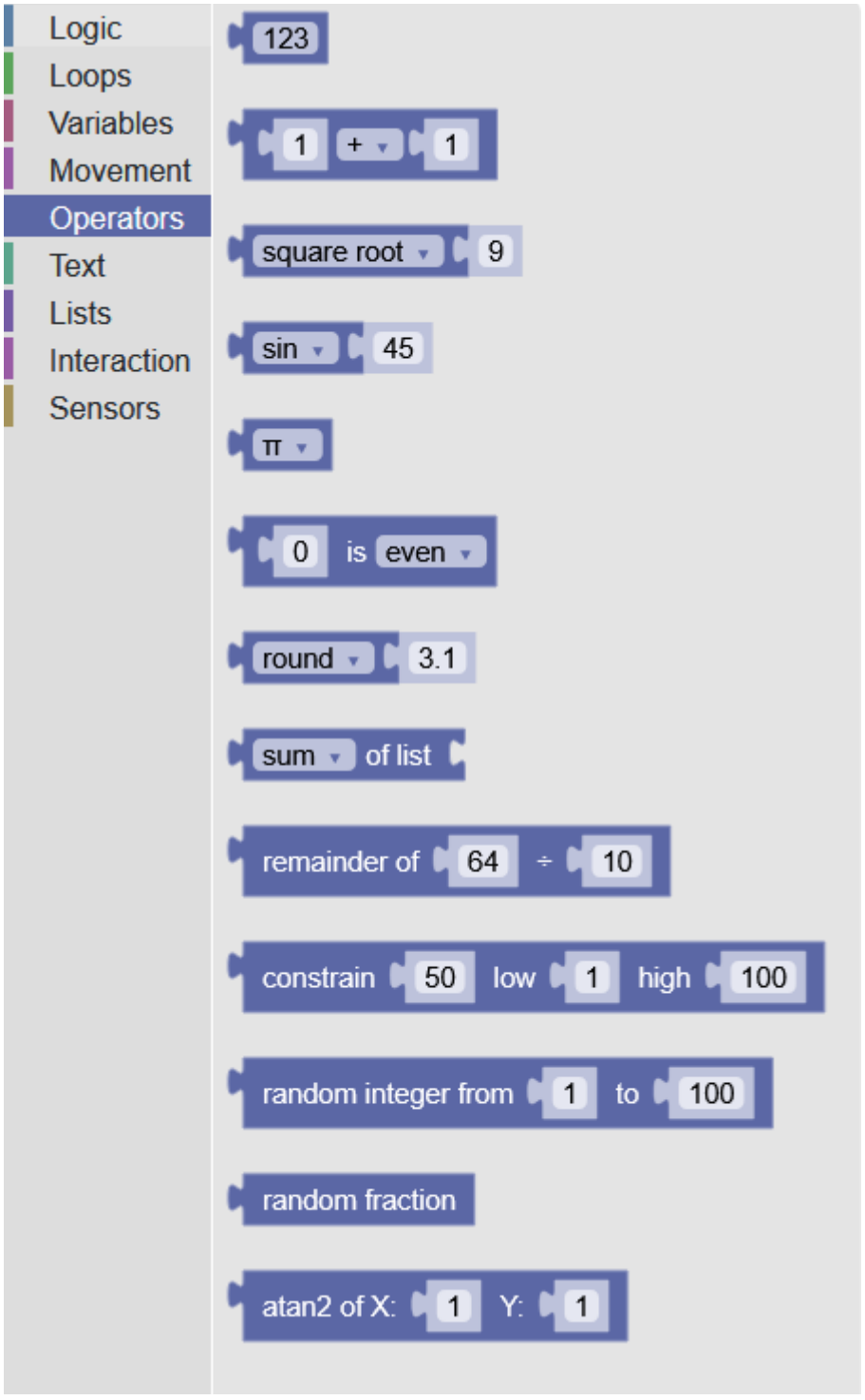
These blocks give learners both **simple controls** (steps, 90° turns) and **advanced precision** (centimeters, degrees).



Picture 40. Movement Blocks

Operators Blocks

Operator blocks perform **mathematical operations**, generate random values, or test numeric conditions. They give programs the ability to calculate, compare, and adapt using numbers instead of only fixed actions. With operators, learners can go beyond simple movements and introduce precision, randomness, or conditional checks (like even/odd). These blocks help students understand how programming connects with mathematics in real-world robotics.



Picture 41. Operators Blocks

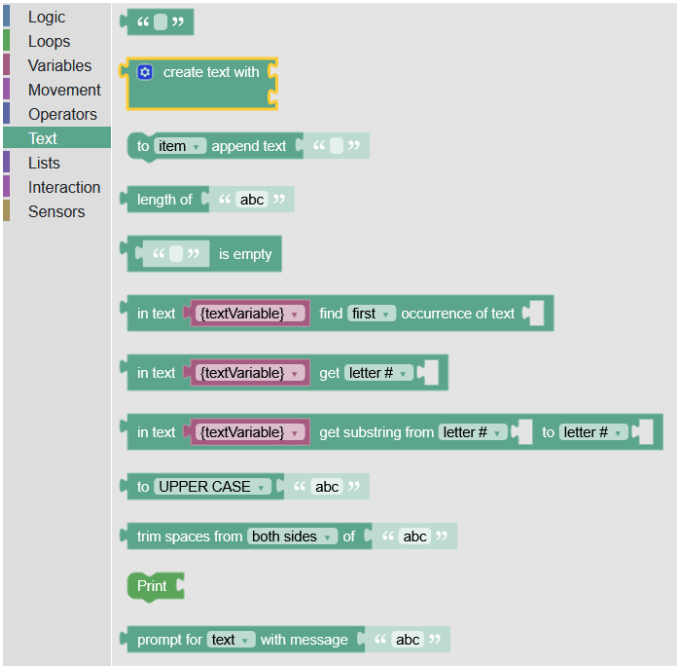
Operators Blocks Explanation

1. **Number (123).** A number block used as a constant value.
2. **Arithmetic (+, –, ×, ÷).** Performs basic calculations between two numbers.
3. **square root [n].** Returns the square root of a number.
4. **sin, cos, tan [angle].** Returns the trigonometric value of an angle (in degrees).
5. **π (pi).** Provides the mathematical constant π (≈ 3.14159).
6. **[n] is even / odd.** Checks if a number is even or odd. Returns true or false.
7. **round [n].** Rounds a decimal number to the nearest whole number.
8. **sum of list.** Adds all the numbers in a list together.
9. **remainder of [a ÷ b].** Returns the remainder after dividing two numbers.
10. **constrain [value] low [min] high [max].** Limits a number so it stays between the minimum and maximum values.
11. **random integer from [min] to [max].** Produces a random whole number between the given range.
12. **random fraction.** Produces a random decimal number between 0 and 1.
13. **atan2 of X: [x] Y: [y].** Returns the angle (in degrees) of a point with coordinates (x, y). Useful in geometry and movement calculations.

These blocks extend the robot's "thinking" with math — from simple addition to randomness and geometry.

Text Blocks

Text blocks are used to create and manipulate words, sentences, and characters. They allow programs to handle information in the form of text instead of numbers. With these blocks, learners can build messages, measure text length, search inside strings, and transform text (for example, converting it to uppercase). This category introduces the concept of **string handling**, which is essential in all programming languages when working with names, instructions, or user input.



Picture 42. Text Blocks

Text Blocks Explanation

1. **Empty text ("").** Represents a blank text value.
2. **create text with [ ].** Combines items into a single piece of text.
3. **to [item] append text ["abc"].** Adds new text to the end of an existing text value or variable.
4. **length of ["abc"].**Returns the number of characters in the text.
5. **["abc"] is empty.** Checks whether the text has no characters.
6. **in text [variable] find first occurrence of text ["abc"].** Finds where a piece of text first appears inside another.
7. **in text [variable] get letter # [ ].** Returns the letter at a specific position in the text.
8. **in text [variable] get substring from letter # [ ] to letter # [ ].** Returns part of a text (substring) between two positions.
9. **to UPPER CASE ["abc"].** Converts text to uppercase letters.
10. **trim spaces from both sides of ["abc"].** Removes extra spaces from the beginning and end of text.
11. **Print.** Displays the given text output.
12. **prompt for text with message ["abc"].** Asks the user to type input text, storing it for use in the program.

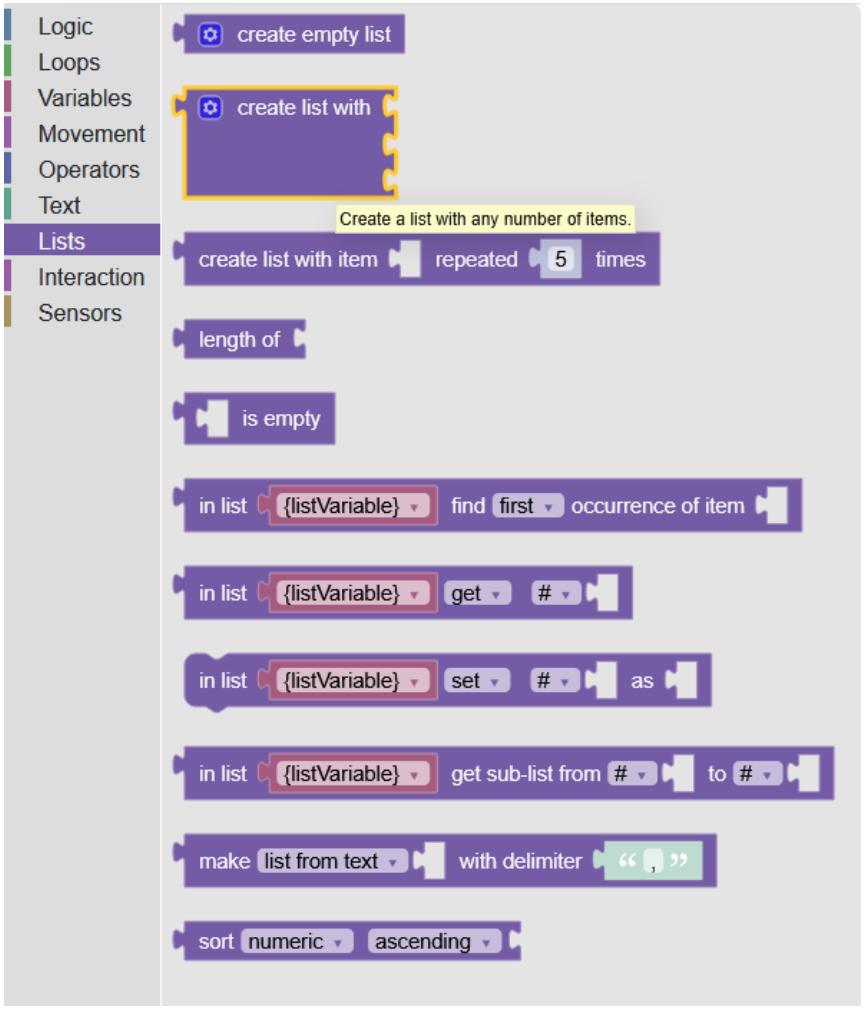
Lists Blocks

Lists allow programs to work with **collections of items** instead of just single values. A list is like a box with many compartments, where each compartment stores an item (numbers, text, or other data). Using lists, learners can organize multiple values in a structured way and then search, change, or sort them. Lists are essential in programming because they make it possible to handle sequences of steps, sets of data, or collections of commands for the robot.

Lists Blocks Explanation

1. **create empty list.** Makes a new list with no items inside.
2. **create list with [ ].** Creates a list with one or more items entered manually.
3. **create list with item [ ] repeated [n] times** Creates a list where the same item is repeated multiple times.
4. **length of [list].** Returns the number of items in the list.
5. **[list] is empty.** Checks if the list has no items.
6. **in list [listVariable] find first occurrence of item** Searches for an item in the list and gives the position of its first appearance.
7. **in list [listVariable] get # [n].** Returns the item at position number *n*.
8. **in list [listVariable] set # [n] as [value].** Replaces the item at position *n* with a new value.
9. **in list [listVariable] get sub-list from # [ ] to # [ ].** Returns part of the list between two positions.
10. **make list from text [ ] with delimiter [,].** Splits text into a list, using a delimiter (like a comma or space).
11. **sort [list] numeric / text, ascending / descending.** Organizes the list in order, either by numbers or alphabetically.

With these blocks, students can move from handling single values to managing structured sets of data, opening the way to more complex and realistic programs.



Picture 43. Lists Blocks

What is a List?

A list is like a box with many compartments. Each compartment can store one item (a number, a word, or something else). Instead of creating a separate variable for each item, you can put them all together in a list.

Why use a List in Programming?

- To **store many values** in one place.
- To **organize data** (e.g., steps, names, sensor readings).
- To **work with sequences** (e.g., a list of moves for a robot).

Simple Example

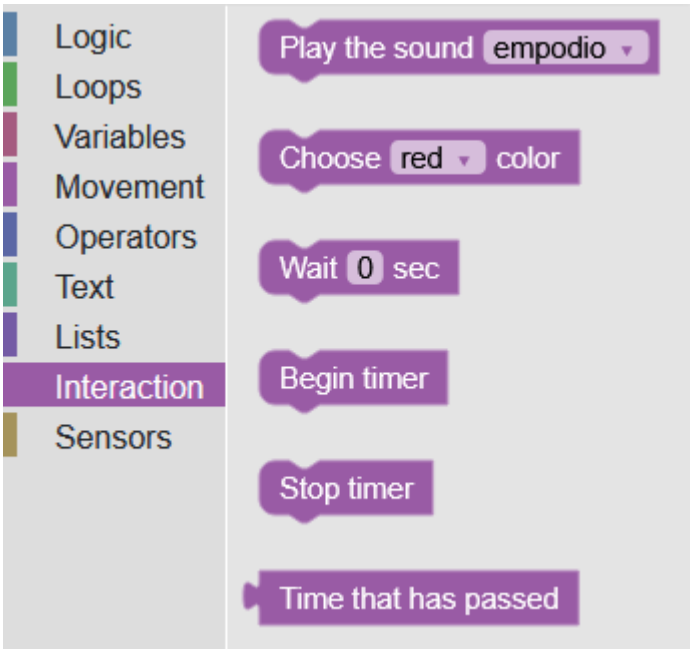
Imagine you want the robot to follow three moves:

- Forward
- Left turn
- Forward

Instead of writing three separate commands, you can store them in a **list**:

Interaction Blocks

Interaction blocks allow the robot to communicate with the environment and the user. They give feedback through sounds, colors, and timed actions. By using these blocks, learners can make the robot respond in more human-like ways: playing sounds, showing lights, or waiting for a specific amount of time. Timers also allow programs to measure how long something takes, making robot behavior more dynamic and responsive.



Picture 44. Interaction Blocks

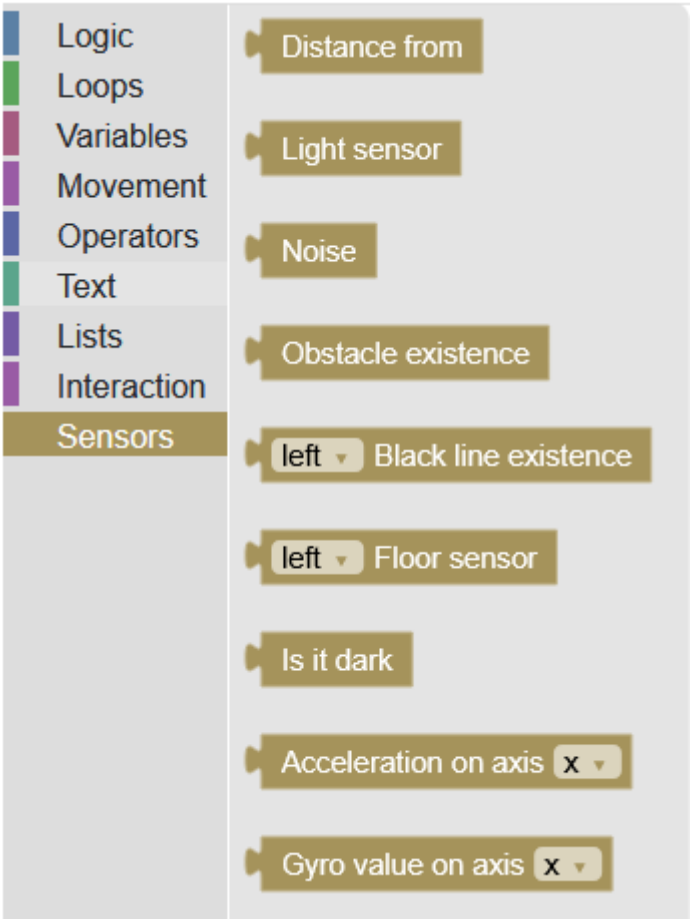
Interaction Blocks Explanation

1. **Play the sound [ ].**Plays a selected sound (e.g., beep, melody). Useful for feedback or alerts.
2. **Choose [color] color.** Changes the robot's light to the chosen color (e.g., red, green, blue).
3. **Wait [n] sec.** Pauses the program for the given number of seconds before continuing.
4. **Begin timer.** Starts a timer that measures elapsed time.
5. **Stop timer.** Stops the timer that was running.
6. **Time that has passed.** Returns the time measured between starting and stopping the timer.

With these blocks, students learn how to add **feedback and timing** to their programs, making the robot more interactive and engaging.

Sensors Blocks

Sensor blocks allow the robot to perceive its environment. Instead of following fixed instructions, the robot can make decisions based on what its sensors detect — such as distance to objects, light intensity, noise, or floor patterns. Using sensors introduces learners to the idea of autonomous behavior, where the robot adapts its actions depending on real-world inputs. This makes programming more interactive and closer to real robotics applications.



Picture 45. Sensors Blocks

Sensors Blocks Explanation

1. **Distance from.** Measures the distance between the robot and an object in front of it. Useful for avoiding collisions.
2. **Light sensor.** Reads the amount of light detected by the robot.
3. **Noise.** Detects sound levels in the environment.
4. **Obstacle existence.** Checks if there is an obstacle nearby (true/false).
5. **[left/right] Black line existence.** Detects if a black line is present under the left or right sensor. Used for line-following activities.

6. **[left/right] Floor sensor.** Reads the color or brightness of the floor under the sensor.
7. **Is it dark.** Returns true if the environment is dark.
8. **Acceleration on axis [x/y/z].** Measures acceleration along the chosen axis.
9. **Gyro value on axis [x/y/z].** Returns the rotation (angular velocity) on the chosen axis.

With these blocks, learners can build programs where the robot reacts to light, sound, distance, lines, and even motion — moving from simple sequences to **true interaction with the environment**.

Age Group: 8–12 years

At this level, the goal is to **combine blocks across categories** (Movement + Logic + Sensors + Interaction) in a way that is easy to follow and fun.

Scenario 1: Stop at an Obstacle

Concepts: Movement + Sensors + Logic

Blocks used:

- Move forward one step
- If do
- Obstacle existence
- Stop

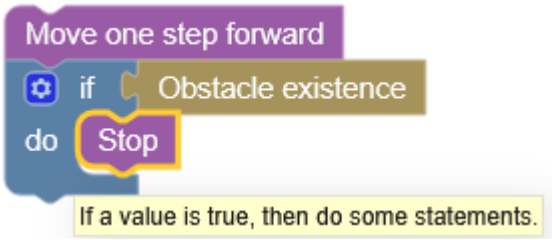
Program idea:

- The robot moves forward.
- If it detects an obstacle, it stops.

Why? Kids learn how a robot can **react to its environment** instead of moving blindly.

Block Code:

1. *[Move one step forward] // Movement block – tells the robot to move once.*
2. *[if (Obstacle existence) do] // Logic + Sensor – checks if there is an obstacle.*
3. *→ [Stop] // Movement block – stops the robot immediately.*



Picture 46. Stop at an Obstacle

Why? The program shows the robot can move, sense, and stop automatically when something is in front.

Activity Title: "Stop at an Obstacle" – Introducing Sensors and Logic with the Physical FOSSBOT

Activity Overview

This foundational activity introduces students (ages 8–12) to the concept of **conditional logic** using the **physical FOSSBOT's obstacle detection sensors**. Students learn how the robot can *"see"* its surroundings and make decisions instead of moving blindly. This is their first step into **sensor-based programming** and **if–then reasoning**.

Preparation and Setup

1. **Stage Design**
   - Use a **flat classroom surface** or taped **mini track** (about 1 meter long).
   - Place one or more **lightweight obstacles** (e.g., small boxes, paper cups) directly in the robot's path.
2. **Robot Preparation**
   - Ensure each FOSSBOT is **assembled, powered**, and **connected** to the **FOSSBOT App (Blockly Mode)**.
   - Open the **Obstacle Sensor** template or ensure the *Obstacle existence* block is available in the toolbox.
   - Verify that obstacle sensors respond (LED indicator or console feedback) when an object is placed close to the front of the robot.

Classroom Flow (≈ 30–35 minutes)

1. **Introduction (5 min)** – Ask: "How can a robot know when to stop?" Show the obstacle sensor and explain that it works like the robot's "eyes."
2. **Planning (5 min)** – Demonstrate the idea of conditions: "If something is in front, then stop."
3. **Programming (10 min)** – Students build the sequence:
   🧩 **Move Forward → If (Obstacle existence) do → Stop**
   (Optional: add a *Light* block to flash when stopping.)

4. **Testing (8 min)** – Students place an object on the path and observe whether the FOSSBOT stops automatically.
5. **Debugging & Discussion (5 min)** – If the robot fails to stop, check the distance to the obstacle or sensor alignment. Discuss how sensors help robots make smart choices.

Learning Goals

- Understand how sensors allow robots to perceive their environment.
- Apply **if–then logic** to control robotic behavior.
- Strengthen observation, prediction, and debugging skills.
- Recognize that programming includes both movement and decision-making.

Scenario 2: Line Follower (Basic)

Concepts: Movement + Sensors

Blocks used:

- Repeat while
- Black line existence (left/right)
- Move forward
- Turn left / Turn right

Program idea:

- While the black line exists, move forward.
- If the line is under the left sensor, turn slightly left.
- If under the right sensor, turn slightly right.

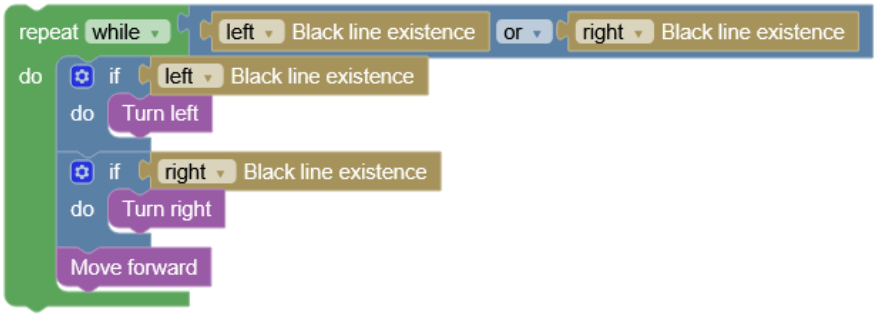
Why? Introduces the idea of **loops and conditional reactions**.

Block Code:

1. *[repeat while (left Black line existence OR right Black line existence)]*
2. *// Loop + Sensor – the program keeps running as long as one of the sensors sees the line.*
3. |
4. → *[if (left Black line existence) do]*
    a. *// Logic + Sensor – check if the left sensor detects the black line.*
    b. → *[Turn left]*
    c. *// Movement – correct direction by turning slightly left.*
5. |
6. → *[if (right Black line existence) do]*
    a. *// Logic + Sensor – check if the right sensor detects the black line.*
    b. → *[Turn right]*

c. *// Movement – correct direction by turning slightly right.*

7. |
8. → *[Move forward]*
    a. *// Movement – continue forward when aligned on the line.*



Picture 47. Line Follower

"Follow the Line" – Basic Line Tracking with the Physical FOSSBOT

Activity Overview

In this activity, students explore how the FOSSBOT can use its line sensors to follow a black path on the floor. They learn about loops and conditional logic, discovering how the robot continuously checks its sensors and corrects its direction. This scenario introduces early ideas of autonomous navigation and reactive behavior.

Preparation and Setup

1. **Stage Design**
    - Use **black electrical tape** or **printed black strips** on a light-colored surface to form a simple **loop or curved line** (width: ~2 cm).
    - Start with gentle curves or an oval shape to make tracking easier.
    - Mark the **Start point** where students will place the FOSSBOT.
2. **Robot Preparation**
    - Ensure FOSSBOT is **assembled, powered**, and **connected** to the **FOSSBOT App (Blockly Mode)**.
    - Check that **line sensors** are clean and properly aligned near the surface.
    - In Admin (gear) settings:
        - *Motor speed* = 40 / 40 (slow speed helps accuracy).
        - *Rotate step* = 1 (small correction turns).

Classroom Flow (≈ 40 minutes)

1. **Introduction (5 min)** – Show students the black line on the floor. Ask: "How can the robot stay on the path by itself?" Explain that FOSSBOT's bottom sensors can "see" the black line.
2. **Planning (5 min)** – Discuss how the robot should behave:

- If it sees the line under the **left sensor**, it turns left.
- If under the **right sensor**, it turns right.
- Otherwise, it moves straight.

3. **Programming (10 min)** – Build this sequence in Blockly:
   🧩
   Repeat while (left OR right black line exists)
   → **If (left black line exists) → Turn left**
   → **If (right black line exists) → Turn right**
   → **Move forward**
4. **Testing (10 min)** – Place the robot on the starting point and observe how it follows the line. Adjust motor speed or lighting conditions if tracking fails.
5. **Debugging & Discussion (10 min)** – Ask: "Why does the robot keep checking the sensors?" Let students identify how the *Repeat while* loop helps the robot react continuously.

Learning Goals

- Understand how sensors guide robotic movement.
- Apply **loops** and **if–then logic** to real-world robot behavior.
- Strengthen skills in observation, debugging, and logical reasoning.
- Experience autonomous decision-making in a simple, visual way.

Scenario 3: Timer Challenge

Concepts: Movement + Interaction

Blocks used:

- Begin timer
- Move forward
- Stop timer
- Time that has passed
- Print

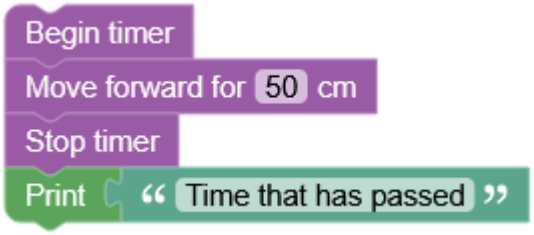
Program idea:

- Start a timer.
- Move forward 50 cm.
- Stop timer.
- Print how much time the movement took.

Why? Introduces **measurement and simple data collection**.

Block Code:

[Begin timer] // Interaction block – starts measuring time.
|
[Move forward for 50 cm] // Movement block – robot moves a fixed distance.
|
[Stop timer] // Interaction block – stops the timer.
|
[Print (Time that has passed)] // Text + Interaction – shows how much time passed.



Picture 48. Timer Challenge

Picture 49. FOSSBot Terminal

These are **age-appropriate, simple, and hands-on** examples. Children can immediately see how their blocks affect the robot's behavior.

Timer Challenge – Measuring Movement with the Physical FOSSBOT

Activity Overview

In this activity, students learn how to use the **FOSSBOT's timing and motion blocks** to measure how long it takes the robot to move a set distance. This introduces the concept of **measurement, time, and data collection**, helping learners connect programming with real-world motion and simple scientific investigation.

Preparation and Setup

1. **Stage Design**
   - Use a **straight 50 cm track** on the classroom floor or table, marked with **Start** and **Finish lines** using colored tape.
   - Ensure the track is flat and free of obstacles for consistent timing.
2. **Robot Preparation**
   - Make sure each FOSSBOT is **assembled, charged**, and **connected** to the **FOSSBOT App (Blockly Mode)**.
   - In Admin (gear) settings:
     - *One step in cm* ≈ 25–30 cm (for calibration).
     - *Motor speed* = 50 / 50 (adjust slightly if drifting).

Classroom Flow (≈ 35 minutes)

1. **Introduction (5 min)** – Ask students: "How long does it take our robot to move 50 cm?" Explain that they can use a timer in the program to measure and print the result.
2. **Planning (5 min)** – Demonstrate the logic: Start timer → Move forward → Stop timer → Print time. Discuss what "measurement" means in robotics.
3. **Programming (10 min)** – Students build the following sequence:
   🧩 **Begin timer → Move forward for 50 cm → Stop timer → Print (Time that has passed)**
   (Optional) Add a **Light** block to flash when finished.
4. **Testing (8 min)** – Place FOSSBOT at the start line, run the code, and observe the printed output. Encourage students to note results.
5. **Extension (7 min)** – Have groups test different speeds (e.g., 40 vs. 60) and compare times to see the relationship between **speed and duration**.

Learning Goals

- Understand how robots can measure and record time.
- Relate motion, speed, and time through coding.
- Practice data collection and simple analysis.
- Connect programming with scientific observation and reasoning.

Scenario 4: Step Counter with Variable

Age group / ISCED level: 10–12 years old (Upper Primary / Lower Secondary, ISCED 2)

Learning goals:

- Understand how loops repeat instructions a fixed number of times.
- Learn how variables store and update values during program execution.
- Practice connecting robot movement with numerical data collection.
- Explore the use of printing results for feedback and debugging.

Subject links:

- **Mathematics:** Counting, iteration, and simple arithmetic.
- **ICT / Computer Science:** Loops, variables, and output.
- **Physics:** Relating steps to distance traveled.

Required tools:

- FOSSBot robot or simulator
- Open space for movement (or simulated stage)
- Console/Screen for printed messages

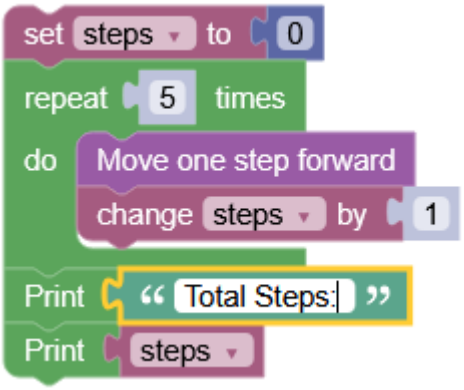
Program Explanation (Step by Step)

1. **Initialization**
   - A variable steps is created and set to **0**.
2. **Repetition with loop**
   - The program uses a **repeat 5 times** loop.
   - In each iteration, the robot moves one step forward.
   - After moving, the variable steps increases by 1.
3. **Output**
   - After completing the loop, the program prints the message *"Total Steps:"*.
   - Then it prints the value of the variable steps to show how many steps were taken.

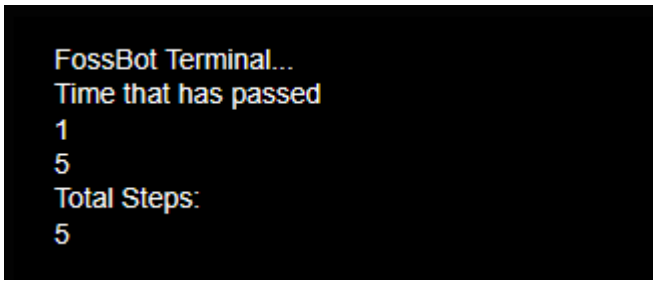
Block Code:

```
[ set steps to 0 ]              // Variables – start at 0
[ repeat 5 times ]              // Loop – run actions 5 times
    |
    → [ move one step forward ]   // Movement – perform one step
    → [ change steps by 1 ]       // Variables – count the step
[ print steps ]                 // Text – display total steps
```

print("Total steps:", steps) # Output - show how many steps were taken



Picture 50. Step Counter with Variable



Picture 51. FOSSBot Terminal

Step Counter with Variable – Counting Robot Steps with Loops (Physical FOSSBot)

Activity Overview

In this activity, students (ages 10–12) explore how the FOSSBOT can use loops and variables to record how many steps it takes. They learn that a variable can store and update information as the program runs, and that loops allow actions to repeat automatically. This scenario bridges programming logic with mathematical counting and introduces data feedback through printed results.

Preparation and Setup

1. **Stage Design**
   - Use a straight track or taped path about 1 meter long on the classroom floor.
   - Mark a clear Start position; no obstacles are required for this exercise.
2. Robot Preparation
   - Ensure the FOSSBOT is assembled, charged, and connected to the FOSSBOT App (Blockly Mode).
   - In Admin (gear) settings:
     - *One step in cm* ≈ 25–30 cm (depending on your track).
     - *Motor speed* = 50 / 50.

- Make sure the Print block is visible so that results can be displayed on the console or screen.

Classroom Flow (≈ 40 minutes)

1. **Introduction (5 min)** – Ask: "How can we make the robot count how many steps it takes?" Explain that we can store the number of steps in a *variable* that increases every time the robot moves.
2. **Planning (5 min)** – Show the blocks on the screen and discuss how loops allow repetition without copying commands.
3. **Programming (10 min)** – Students build the following sequence:
   
   Set steps to 0
   Repeat 5 times:
   → **Move one step forward**
   → **Change steps by 1**
   Print steps
   (Optional) Add a text block before printing: "Total steps:".
4. **Testing (10 min)** – Place the FOSSBOT on the Start line and run the program. Observe both the robot's motion and the printed message.
5. **Extension (10 min)** – Modify the loop to repeat 3, 6, or 10 times. Ask: "How far did the robot travel?" Encourage students to calculate distance = steps × step length.

Learning Goals

- Understand that **loops** repeat instructions automatically.
- Use **variables** to store and update values dynamically.
- Relate numeric data to real-world robot motion (steps = distance).
- Practice debugging and interpreting printed output.

Age Group: 12 + years

Scenario1. Obstacle Avoider

Age group / ISCED level: 12–15 years old (Lower Secondary, ISCED 2)

Learning goals:

- Understand how robots use sensors to detect obstacles.
- Learn to apply conditional logic (if / else) in loops.
- Explore continuous execution with a repeat while (true) loop.
- Connect sensor readings to safe navigation strategies.

Subject links:

- **Physics:** Motion and collision avoidance.
- **ICT / Computer Science:** Loops, conditions, and sensor-based decisions.
- **Engineering:** Design of autonomous navigation systems.

Required tools:

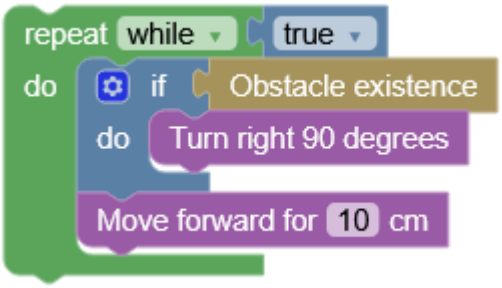
- FOSSBot robot or simulator
- A space with small obstacles (boxes, books, or walls in CoppeliaSim)

Program Explanation (Step by Step)

1. **Continuous loop:** The program runs indefinitely using repeat while (true).
2. **Obstacle check:**
   - If the obstacle sensor detects something in front: → The robot **turns right 90°** to avoid it.
   - Else (no obstacle detected): → The robot **moves forward 10 cm** safely.
3. **Autonomous behavior:** The robot keeps repeating this process, allowing it to wander around while avoiding collisions.

Block Code:

[repeat while (true)] // Loop – program runs continuously
|
→ [if (Obstacle existence) do] // Logic + Sensor – check for obstacle
→ [Turn right 90 degrees] // Movement – avoid obstacle
else
→ [Move forward for 10 cm] // Movement – continue path



Picture 52. Block Code. Stops clicking the stop button

Why? Demonstrates decision-making in real time. Connects to Physics (motion, reaction, direction).

Obstacle Avoider – Autonomous Navigation with the Physical FOSSBOT

Activity Overview

In this scenario, students (ages 12–15) program the physical FOSSBOT to move autonomously and avoid obstacles using its front distance sensor. The activity introduces the concept of continuous loops**,** conditional logic, and reactive behavior, showing how robots make real-time decisions to navigate safely without collisions.

Preparation and Setup

1. Stage Design
    - Use a flat classroom surface or floor area (at least 1.5 × 1.5 m).
    - Place small, lightweight obstacles (e.g., boxes, plastic cups, books) randomly around the space.
    - Make sure the testing area is clear of fragile or heavy objects.
2. Robot Preparation
    - Ensure the FOSSBOT is assembled, charged, and connected to the FOSSBOT App (Blockly Mode).
    - Check that the front obstacle sensor is clean and properly aligned.
    - In Admin (gear) settings:
        - *Motor speed* = 50 / 50 (adjust slightly if drifting).
        - *Rotate step* = 1 (for 90° turns).
    - Optional: Add a Light or Sound block to indicate obstacle detection.

Classroom Flow (≈ 40 minutes)

1. Introduction (5 min) – Discuss how self-driving cars or robots avoid collisions. Ask: “How can our FOSSBOT decide when to turn?”
2. Planning (5 min) – Explain the logic:
    - The robot keeps moving.

- If it detects an obstacle, it turns right 90°.
- If not, it moves forward safely.

3. Programming (10 min) – Students build this sequence in Blockly:

Repeat while (true)
→ If (Obstacle existence) d
→ → Turn right 90°
Else
→ Move forward for 10 cm

4. Testing (10 min) – Place the robot at the edge of the course and run the program. Observe how it navigates and avoids obstacles.
5. Debugging (5 min) – Adjust the obstacle distance threshold if needed, or slow down motor speed for more accurate turns.
6. Reflection (5 min) – Discuss: "What would happen if we changed the turn direction?" or "How could we make the robot choose between left or right turns?"

Learning Goals

- Understand how sensors provide environmental awareness.
- Apply if / else logic for reactive decision-making.
- Explore continuous execution using *repeat while (true)* loops.
- Relate programming to real-world autonomous navigation.

Scenario 2: Smart Step Counter with Obstacle Detection

Age group / ISCED level: 11–14 years old (Upper Primary / Lower Secondary, ISCED 2)

Learning goals:

- Learn how to use variables as counters.
- Understand how sensors stop a program when a condition is met.
- Practice combining movement, logic, and counting in one loop.
- Connect robotic navigation with data collection.

Subject links:

- **Mathematics:** Counting steps, relating steps to distance.
- **ICT / Computer Science:** Loops, conditions, variables, stopping execution.
- **Physics:** Motion and measurement of distance through repeated steps.

Required tools:

- FOSSBot robot or simulator
- Open space with obstacles (books, boxes, or simulated walls)
- Console/Screen to display total steps taken

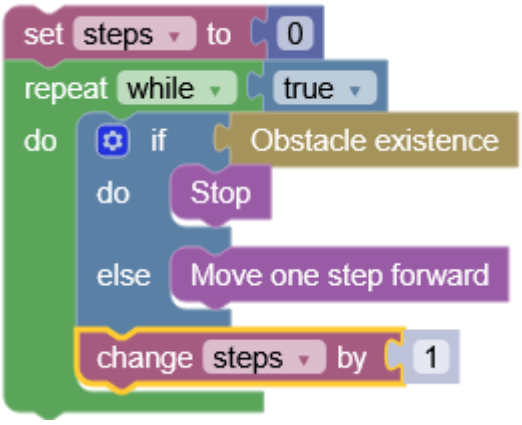
Program Explanation (Step by Step)

1. **Initialization**
   - A variable steps is set to **0** to track the number of movements.
2. **Continuous loop**
   - The program repeats forever using repeat while (true).
3. **Obstacle detection**
   - If an obstacle is detected:
     → The robot **stops** and prints the total steps taken.
   - Else:
     → The robot moves one step forward and increases the steps counter by 1.
4. **End condition**
   - The program effectively halts once an obstacle is encountered, leaving the final step count as the result.

Block Code:

[set steps to 0] // Variables – initialize counter

[repeat while (true)] // Loop – keep running
|
→ *[if (Obstacle existence) do] // Logic + Sensor – check for obstacle*
→ *[Stop] // Movement – stop robot*
→ *[Print steps] // Text – show steps before stopping*
else
→ *[Move one step forward] // Movement – keep going*
→ *[change steps by 1] // Variables – count the step*



Picture 53. Smart Step Counter with Obstacle Detection

Why? Introduces a combination of counting and environment awareness. Connects to Math (counters, conditions) and Physics (motion until blocked). It is appropriate for introducing problem-solving with variables: the robot measures *how far it went before being blocked*. Students see how a counter variable works together with **sensors** to create meaningful feedback.

Smart Step Counter with Obstacle Detection – Counting and Sensing with the Physical FOSSBOT

Activity Overview

In this activity, students (ages 11–14) combine movement, logic, and variables to create a simple autonomous counting robot. Using the FOSSBOT's obstacle sensor, they program the robot to count its steps while moving forward and stop automatically when it detects an obstacle. This introduces the concept of data collection through iteration and shows how sensors can control when a program stops.

Preparation and Setup

1. Stage Design
    - Use an open floor space (at least 1 meter long) with lightweight obstacles such as boxes, books, or plastic containers.
    - Place one obstacle in the robot's path about 60–80 cm ahead of the starting position.
    - Ensure good lighting for accurate sensor detection.
2. Robot Preparation
    - Confirm that the FOSSBOT is assembled, charged, and connected to the FOSSBOT App (Blockly Mode).
    - Check the front obstacle sensor by moving an object near the sensor — the robot should detect it.
    - In Admin (gear) settings:
        - *Motor speed* = 50 / 50
        - *One step in cm* ≈ 25–30 cm (for distance calibration)
        - *Rotate step* = 1 (optional for future navigation challenges)
    - Make sure the Print block is available so students can see the result on the screen.

Classroom Flow (≈ 40 minutes)

1. **Introduction (5 min)** – Ask: "Can a robot know how far it has moved?" Explain that each movement can be counted, and sensors can tell the robot when to stop.
2. **Planning (5 min)** – Discuss the structure:
    - Start counting at 0.

- Keep moving while no obstacle exists.
- Stop and print the number of steps when an obstacle is detected.

3. **Programming (10 min)** – Students build this sequence in Blockly:

Set steps to 0
Repeat while (true
→ If (Obstacle existence) do
→ → Stop
→ → Print steps
Else
→ Move one step forward
→ Change steps by 1

4. **Testing (10 min)** – Place the FOSSBOT at the start of the path and run the program. Observe how it counts and stops automatically.
5. **Debugging (5 min)** – If the robot doesn't stop correctly, adjust the obstacle's distance or sensor sensitivity.
6. **Extension (5 min)** – Have students add a *Light* or *Sound* block after stopping to signal task completion.

Learning Goals

- Understand how **variables** store and update data.
- Apply **loops** and **if / else** logic to manage continuous behavior.
- Connect **sensor input** to program control and stopping conditions.
- Practice debugging and interpreting printed results.

Scenario 3: Obstacle Detection and Right Turn

Age group / ISCED level: 11–14 years old (Upper Primary / Lower Secondary, ISCED 2)

Learning goals:

- Understand how robots use sensors to detect obstacles.
- Apply conditional logic (repeat until) in a real robotics task.
- Explore timing as a control mechanism for movement.
- Strengthen computational thinking through decomposition and debugging.

Subject links:

- Mathematics: Measuring distance thresholds, angles of turning.
- Physics: Motion, distance, and time relation.

- ICT / Computer Science: Loops, conditions, timers.

Required tools:

- FOSSBot robot or digital FOSSBot simulation
- Distance sensor enabled

Program Explanation (Step by Step)

- Repeat until obstacle existence: The robot keeps moving until an obstacle is detected.
- Print “Moving forward!”: A message shows on screen.
- Repeat until distance < 50: The robot moves forward until it is closer than 50 cm to the obstacle.
- Print “Obstacle detected!”: The robot notifies that an obstacle is in range.
- Begin timer: A timer starts to measure turning time.
- Print “Turning right!”: The robot announces its action.
- Repeat while time < 1: The robot turns right for 1 second.
- Stop timer: The turning ends.

Scenario Narrative

Students imagine the robot is exploring a corridor. It moves safely forward, constantly checking for obstacles. When something is detected within 50 cm, the robot stops, announces the obstacle, and decides to turn right for exactly one second, avoiding the obstacle before continuing.

Extensions / Variations

- Mathematics link: Measure the exact turning angle for 1 second of rotation and calculate the robot’s orientation.
- Physics link: Explore the relation between speed, distance, and reaction time.
- Programming link: Replace the fixed timer with a condition, e.g., “turn until no obstacle is in front.”
- Advanced task: Add a counter variable to track how many obstacles the robot avoids.

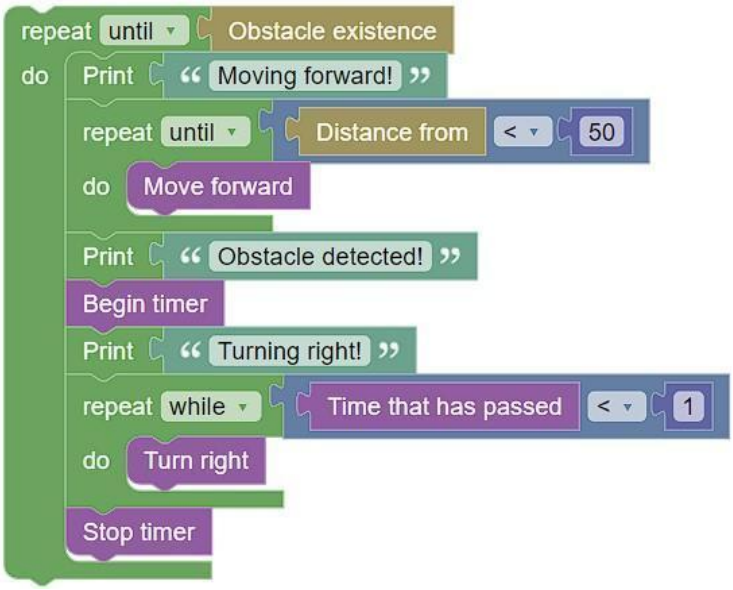
Pseudocode

[repeat until (Obstacle existence)] // Loop – continue until an obstacle is detected
|
→ [Print "Moving forward!"] // Text – announce movement
→ [repeat until (Distance from < 50)] // Loop – keep moving until closer than 50 cm
|
→ [Move forward] // Movement – advance step by step

[Print "Obstacle detected!"] // Text – notify obstacle found

[Begin timer] *// Timer – start counting time*
[Print "Turning right!"] *// Text – announce turning action*
[repeat while (Time that has passed < 1)] *// Loop – keep turning until 1 second passes*
|
→ *[Turn right]* *// Movement – rotate to the right*
[Stop timer] *// Timer – stop counting*

Block Code

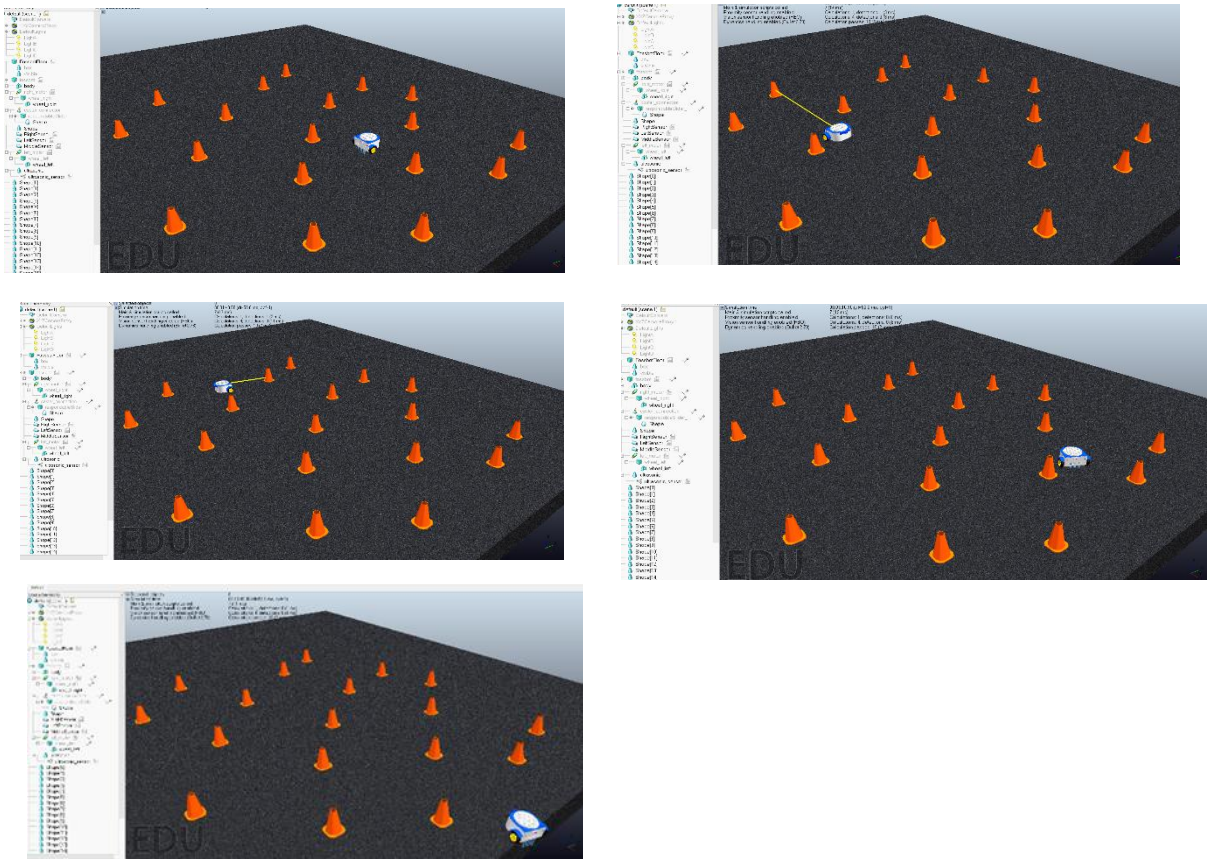


Picture 54. Block Code

Attention Points for Teachers / Students

- The first loop (repeat until obstacle existence) ensures the robot is in motion only when no obstacle is directly in front.
- The distance condition (< 50) gives the robot a safety buffer to stop before crashing. This can be adjusted depending on classroom space.
- The timer method is not precise for angles, since turning speed may vary. It introduces the idea of time-based control vs sensor-based control.
- Students should experiment: how long does "1 second turn" equal in degrees (e.g., ~90°)?
- Encourage debugging: If the robot doesn't turn correctly, adjust either speed settings or time threshold.

Scenes of FOSSBot Movement



Picture 55. Scenes of the scenario

FOSSBot Terminal during the execution of the code

Picture 56. FOSSBot Terminal during the scenario

Experimenting with Python

For those who want to experiment with Python, here is the equivalent code for the block program you just saw. The code uses a loop, a distance sensor, and a timer to reproduce the same behavior: move forward until an obstacle is detected, stop before hitting it, then turn right for one second.

This translation helps students see how visual blocks connect to real programming syntax, and it also shows the importance of comments for understanding.

import time

```
# --- Variables and setup ---
distance_threshold = 50      # Safety distance in cm before obstacle
turning_time = 1             # Time to turn right (in seconds)
# Functions below simulate FOSSBot hardware commands
def obstacle_exists():
    # Returns True if an obstacle is detected by front sensor
    # (Replace with actual FOSSBot API call in simulation or robot)
    pass
def distance_from_obstacle():
    # Returns the current distance to the nearest obstacle in cm
    pass
def move_forward():
    # Moves the robot one step or continuously forward
    pass
def turn_right():
    # Turns the robot to the right
    pass
def stop():
    # Stops the robot
    pass
# --- Main program ---
while not obstacle_exists():                # Keep running until obstacle detected
    print("Moving forward!")                # Inform about robot action
    while distance_from_obstacle() > distance_threshold:
        move_forward()                      # Keep moving closer until safe limit reached
# At this point, obstacle detected within 50 cm
print("Obstacle detected!")
stop()                                      # Stop before turning
# Start timing the turn
print("Turning right!")
start_time = time.time()                    # Record current time
while (time.time() - start_time) < turning_time: # Loop while less than 1 second passed
    turn_right()                            # Execute right turn
stop()                                      # Stop after turn is complete
```

Obstacle Detection and Right Turn – Sensor-Based Navigation with the Physical FOSSBOT

Activity Overview

In this activity, students (ages 11–14) program the physical FOSSBOT to detect obstacles using its distance sensor and respond automatically by turning right for a set duration. This introduces the use of conditional loops, timers, and sensor thresholds to control behavior dynamically. The activity helps learners understand how robots use sensory input and timing to make safe navigation decisions.

Preparation and Setup

1. Stage Design
    - Use a flat classroom area or taped corridor-style path (around 1.5 m long).
    - Place one or two light obstacles (boxes, plastic cups, or books) about 50–70 cm from the starting point.
    - Mark a Start zone where each group positions the robot before running the code.
2. Robot Preparation
    - Ensure the FOSSBOT is assembled, charged, and connected to the FOSSBOT App (Blockly Mode).
    - Check that the front distance sensor is clean and functioning (it should detect nearby objects visually or via indicator).
    - In Admin (gear) settings:
        - *Motor speed* = 50 / 50
        - *Rotate step* = 1 (≈ 90° turn)
    - Optional: Add Print, Light, or Sound blocks for feedback during testing.

Classroom Flow (≈ 40–45 minutes)

1. Introduction (5 min) – Ask: "How can a robot know when to stop before hitting something?" Demonstrate the distance sensor by placing an object in front of FOSSBOT.
2. Planning (5 min) – Discuss the logic:
    - Keep moving until an obstacle is detected.
    - When detected, print a message, turn right for one second, and continue.
3. Programming (10 min) – Students build the program:

Repeat until (Obstacle existence)
→ Print "Moving forward!"
→ Move forward
Repeat until (distance < 50)
→ Print "Obstacle detected!"
→ Stop
→ Begin timer
→ Print "Turning right!"
Repeat while (time < 1)
→ Turn right
→ Stop timer

4. Testing (10 min) – Place FOSSBOT in the start zone and observe its movement and turning behavior when it detects an obstacle.

5. Debugging (8 min) – Adjust detection distance (e.g., 40–60 cm) or turning time (1–1.5 seconds) for accuracy.
6. Reflection (5 min) – Discuss: “How does timing affect turning angle?” or “What happens if we replace the timer with a new condition?”

Learning Goals

- Use sensors to trigger program behavior.
- Apply conditional logic with *repeat until* and *repeat while* loops.
- Understand how timing controls movement and direction.
- Strengthen computational thinking through testing, measuring, and refining.

Scenario Narrative

Students imagine that the FOSSBOT is exploring a corridor. It moves safely forward, constantly checking for obstacles. When one appears within 50 cm, the robot announces “Obstacle detected!”, stops, and performs a one-second right turn to continue its path safely — just like a real autonomous vehicle.

Extensions / Variations

- Mathematics: Measure the robot’s turning angle for one second of rotation; calculate orientation change.
- Physics: Explore the relationship between speed, distance, and reaction time.
- Programming: Replace the timer with a condition, e.g., *turn until no obstacle exists*.
- Advanced: Add a counter variable to track how many obstacles the robot detects or avoids.

Scenario 4: Line Following with Sensors

Age group / ISCED level: 13–16 years old (Lower Secondary to Upper Secondary, ISCED 2–3)

Learning goals:

- Understand how robots use multiple sensors to follow a path.
- Explore conditional logic with if, else if, and else.
- Connect real-world robotics with algorithms for decision-making.
- Experiment with debugging and calibration.

Subject links:

1. Mathematics: Geometry of paths and curves.
2. Physics: Reflection, light intensity, sensor behavior.

3. ICT / Computer Science: Conditionals, loops, logical operators.

Required tools:

- FOSSBot robot or digital simulator
- A printed or taped black line path on a white surface

Program Explanation (Step by Step)

1. Repeat until obstacle existence: The robot keeps working until it meets an obstacle.
2. If both sensors detect the line: Move forward 1 cm.
3. Else if only the left sensor detects the line: Announce detection, then turn right to re-align.
4. Else if only the right sensor detects the line: Announce detection, then turn left.
5. Else (no line under sensors): Print the sensor states and move forward 1 cm.

Scenario Narrative

Students imagine the robot is a train following a track. The black tape is the track. The robot constantly checks its sensors:

- If it sees the line under both sensors, it moves straight.
- If only the left or right sensor sees the line, it adjusts direction (right or left).
- If no sensor detects the line, it logs all sensor values to help debugging and moves forward slightly.

This activity teaches **feedback control**, which is a key principle in robotics.

Extensions / Variations

- Math link: Ask students to measure how accurately the robot follows curves.
- Programming link: Add a counter for the number of turns taken.
- Challenge: Make the robot stop if it completely loses the line for more than 5 seconds.
- Physics link: Experiment with different colored lines (blue, red) and discuss why sensors may fail.

Pseudocode

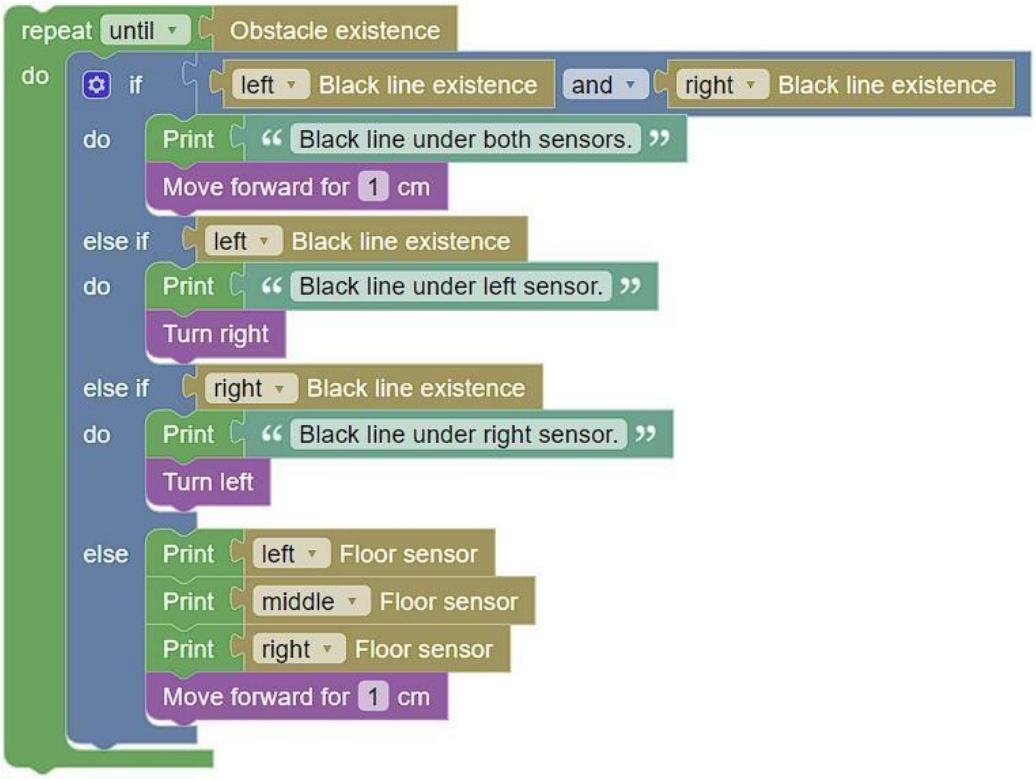
[repeat until (Obstacle existence)] // Loop – continue until obstacle detected
|
→ [if (Left sensor detects black line AND Right sensor detects black line)]
→ [Print "Black line under both sensors."] // Text – both aligned
→ [Move forward 1 cm] // Movement – continue straight

→ *[else if (Left sensor detects black line)]*
→ *[Print "Black line under left sensor."] // Text – left only*
→ *[Turn right] // Movement – correction*

→ *[else if (Right sensor detects black line)]*
→ *[Print "Black line under right sensor."] // Text – right only*
→ *[Turn left] // Movement – correction*

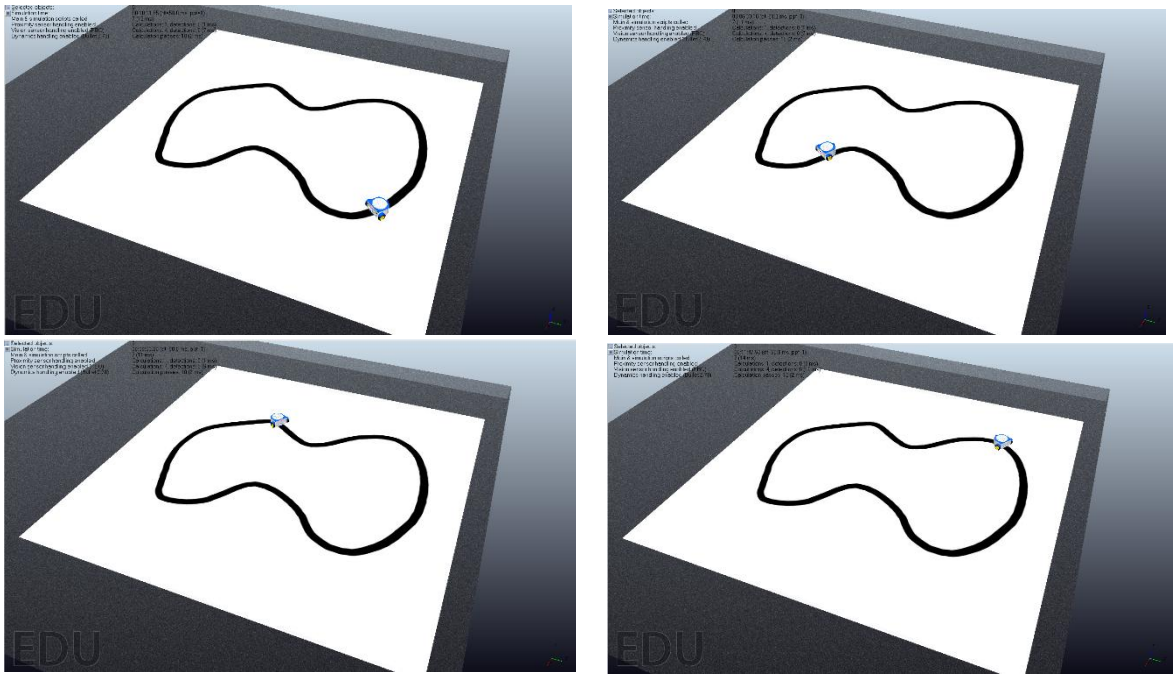
→ *[else]*
→ *[Print left/middle/right floor sensor values] // Debugging info*
→ *[Move forward 1 cm] // Try to relocate line*

Block Code



Picture 57. Block Code

Scenes of FOSSBot Movement



Picture 58. Scenes of the scenario

Picture 59. FOSSBot Terminal during the scenario

Experimenting with Python

```
import time

# --- Parameters ---
step_distance = 1      # Distance in cm for each forward move
turn_delay = 0.2       # Time in seconds for a small corrective turn
# --- Functions simulating FOSSBot API ---
def obstacle_exists():
    # Returns True if an obstacle is detected by front sensor
    pass
def left_sensor():
    # Returns True if the left floor sensor sees a black line
    pass
def right_sensor():
    # Returns True if the right floor sensor sees a black line
    pass

def floor_values():
```

```
    # Returns readings from all sensors (left, middle, right)
    return (0, 0, 0)  # placeholder values
def move_forward(distance):
    # Moves robot forward by given distance (in cm)
    pass
def turn_left():
    # Turns the robot slightly left
    time.sleep(turn_delay)
def turn_right():
    # Turns the robot slightly right
    time.sleep(turn_delay)
def stop():
    # Stops robot movement
    pass
# --- Main program ---
while not obstacle_exists():                    # Loop – continue until obstacle detected
    if left_sensor() and right_sensor():          # Case 1: Both sensors on line
        print("Black line under both sensors.")
        move_forward(step_distance)
    elif left_sensor():                         # Case 2: Left sensor on line
        print("Black line under left sensor.")
        turn_right()                            # Correct towards the line
    elif right_sensor():                          # Case 3: Right sensor on line
        print("Black line under right sensor.")
        turn_left()                             # Correct towards the line
    else:                                       # Case 4: No line detected
        l, m, r = floor_values()
        print(f"No line. Sensor values -> L:{l}, M:{m}, R:{r}")
        move_forward(step_distance)               # Try moving forward to find the line
stop()                                          # Stop when obstacle is reached
```

Teacher & Student Notes

- The sensor functions (left_sensor(), right_sensor(), floor_values()) are placeholders. Replace them with real FOSSBot API calls.
- The step size (1 cm) and turn delay (0.2 seconds) must be adjusted depending on robot calibration.
    - Encourage students to experiment with parameters:
    - What happens if turn_delay is too long or too short?
- Does increasing step_distance make the robot more efficient or less accurate?
- This example introduces feedback control: the robot constantly checks its sensors and adapts its path.
- Teachers can highlight how this resembles real-world systems (e.g., autopilot, self-driving cars).

Line Following with Sensors – Feedback and Decision-Making with the Physical FOSSBOT

Activity Overview

In this advanced scenario, students (ages 13–16) program the physical FOSSBOT to follow a black line path on a white surface using its two bottom sensors. The robot continuously checks the sensors and makes small corrections to stay on the path, introducing the concept of feedback control — a key principle in robotics and automation. Through conditional statements (*if, else if, else*), learners experience how robots make real-time decisions based on sensory input.

Preparation and Setup

1. Stage Design
    - Create a black line track using electrical tape (2 cm wide) on a white or light-colored floor or cardboard sheet.
    - Start with a straight path, then add gentle curves or intersections for extension activities.
    - Ensure good lighting conditions to improve sensor detection.
2. Robot Preparation
    - Verify that the FOSSBOT is assembled, charged, and connected to the FOSSBOT App (Blockly Mode).
    - Check that both line sensors underneath the robot are clean and aligned.
    - In Admin (gear) settings:
        - *Motor speed* = 40 / 40 (slower speeds give more precise results).
        - *Rotate step* = 1 (≈ 90° turn)
    - Optional: Add Print or Sound blocks to signal detection or debugging messages.

Classroom Flow (≈ 45 minutes)

1. Introduction (5 min) – Ask: “How does a robot know where to go?” Explain that the two bottom sensors detect light reflection — dark areas (the line) reflect less light, while bright areas reflect more.
2. Planning (5 min) – Describe the logic:
    - If both sensors see the black line → Move forward.
    - If only one sensor detects the line → Turn slightly to re-align.
    - If neither sensor detects the line → Print readings and move forward slowly.
3. Programming (10 min) – Students build this sequence in Blockly:

Repeat until (Obstacle existence)

→ If (left sensor detects black AND right sensor detects black) → Move forward 1 cm
→ Else if (left sensor detects black) → Print “Left line detected!” → Turn right
→ Else if (right sensor detects black) → Print “Right line detected!” → Turn left
→ Else → Print “No line detected!” → Move forward 1 cm

4. Testing (10 min) – Place the FOSSBOT at the start of the taped line and run the program. Encourage students to observe its corrections and note moments of misalignment.
5. Debugging (10 min) – If the robot fails to follow the line, adjust *motor speed*, *sensor height*, or ensure proper contrast between the line and background.
6. Reflection (5 min) – Discuss: “How does the robot ‘know’ when to turn?” and connect the concept to feedback loops in real-world systems (e.g., car lane assist, drones, industrial robots).

Learning Goals

- Understand how multiple sensors guide robotic navigation.
- Apply conditional logic (if / else if / else) to control direction.
- Learn about feedback control and real-time decision-making.
- Strengthen debugging, observation, and calibration skills.

Scenario Narrative

Students imagine the FOSSBOT as a train following its track. The black line represents the rail. When the sensors detect both tracks, the robot moves straight; when only one track is detected, it steers slightly to re-center. If no line is detected, it continues cautiously and logs sensor readings for analysis.

Extensions / Variations

- Mathematics: Measure how accurately the FOSSBOT follows curved paths.
- Programming: Add a counter variable to track the number of turns.
- Challenge: Make the robot stop if the line is lost for more than 5 seconds.
- Physics**:** Experiment with different colored lines (blue, red) to analyze how light intensity affects sensor accuracy.

Implementing the Remaining Scenarios with the Physical FOSSBOT

All remaining STEPS scenarios can also be carried out using the physical FOSSBOT, following the same logic and programming principles shown in the simulator. To prepare, teachers should first ensure that each robot is fully assembled, charged, and calibrated, with all printed and electronic components correctly connected. The real-world stages can be recreated easily using simple classroom materials — for example, black tape for line

paths, boxes for obstacles, printed grids for movement tasks, and lamps or flashlights for light sensor experiments. It is important to provide a clear, safe testing area where robots can move freely, and to check that sensor readings (distance, light, or line) are accurate before each session. In group settings, students can take turns programming, observing, and adjusting the robot's behavior, mirroring the experimentation process from the virtual environment. This blended approach connects theory and practice, allowing learners to experience authentic, hands-on robotics while reinforcing coding, logic, and problem-solving skills.

Teacher Checklist: Preparing and Running Physical FOSSBOT Scenarios

Before the Lesson

- Ensure each FOSSBOT is fully assembled, charged, and connected to the FOSSBOT App.
- Verify sensor functionality (distance, light, or line) through quick test runs.
- Prepare or print physical stages (e.g., taped lines, obstacle courses, grids).
- Mark Start and Finish points clearly on the floor or working surface.
- Check Wi-Fi connectivity or USB communication with each student device.
- Have spare batteries, cables, and printed command cards available.

During the Lesson

- Begin with a short demonstration run to show the scenario logic.
- Organize students into small groups (2–3 learners per robot).
- Assign rotating roles — programmer, observer, and recorder.
- Encourage "change one thing" debugging (adjust one block or value at a time).
- Use light or sound signals to indicate successful program completion.

After the Lesson

- Ask students to reflect or record observations (e.g., number of steps, timing, or obstacles detected).
- Discuss what could be improved in sensor calibration or movement precision.
- Safely turn off and store robots, ensuring sensors remain clean and undamaged.
- Encourage students to compare physical results with their virtual simulations.

Scenario 5: Light Sensor Example

(This scenario tests FOSSBot's light sensor)

Age group / ISCED level: 12–15 years old (Lower Secondary, ISCED 2)

Learning goals:

- Understand how light sensors detect environmental changes.
- Practice loops (repeat while, repeat until, repeat times).
- Use robot color feedback to signal environmental states.
- Explore the relationship between light conditions and robot behavior.

Subject links:

- Physics: Light, darkness, reflection.
- ICT / Computer Science: Conditional logic, loops.
- Art/Design: Using light signals (red/green) as communication.

Required tools:

- FOSSBot robot or simulation
- A room with controlled lighting (dark → light → dark transitions)

Program Explanation (Step by Step)

1. Print *"It's dark"* and switch FOSSBot color to red.
2. While it is dark, the robot keeps turning right.
3. Once light is detected, print *"Light detected"*, switch color to green, and turn right until darkness returns.
4. Print *"Darkness again"*, set color back to red, and turn right 10 times.
5. While still dark, turn left.
6. Once light is detected again, print *"Light detected again"*, set color to green, and turn left until darkness returns.

Scenario Narrative

In this scenario, students imagine FOSSBot as a night watch robot that reacts differently to light and dark conditions. It changes its LED color (red for dark, green for light), moves differently depending on the sensor's reading, and continuously reports its status. This helps students see how sensors can drive decisions and how robots can "adapt" to the environment.

Extensions / Variations

- Physics link: Experiment with different light intensities (torch, lamp, sunlight).

- Math link: Measure how long the robot turns before switching from light to dark, then calculate average time per loop.
- Programming link: Replace fixed "10 turns" with a variable counter.
- Creative link: Use LED colors as a "signal code" (e.g., green = safe, red = danger).

Pseudocode

```
[ Print "It's dark." ]              // Text – announce darkness
[ Set LED color to red ]            // Output – red means dark
[ repeat while (Is it dark) ]       // Loop – while darkness continues
      → [ Turn right ]              // Movement – rotate until light appears

[ Print "Light detected." ]         // Text – announce light
[ Set LED color to green ]          // Output – green means light
[ repeat until (Is it dark) ]       // Loop – until darkness returns
      → [ Turn right ]

[ Print "Darkness again." ]         // Text – announce darkness
[ Set LED color to red ]            // Output – back to red
[ repeat 10 times ]                 // Fixed repetition loop
      → [ Turn right ]

[ repeat while (Is it dark) ]       // While still dark
      → [ Turn left ]               // Turn left instead

[ Print "Light detected again." ]   // Text – announce new light detection
[ Set LED color to green ]          // Output – signal safe/light
[ repeat until (Is it dark) ]       // Until dark again
      → [ Turn left ]
```

Attention Points for Teachers / Students

- The light sensor may need calibration: strong ambient light could cause "false detections."
- LED colors are useful for debugging: students can easily see robot status.
- Encourage experimenting with different light sources (flashlight vs phone torch).
- Highlight how robots in the real world (e.g., solar trackers, automatic lights) use similar logic.

Block Code

Picture 60. Block Code

What the Program Does

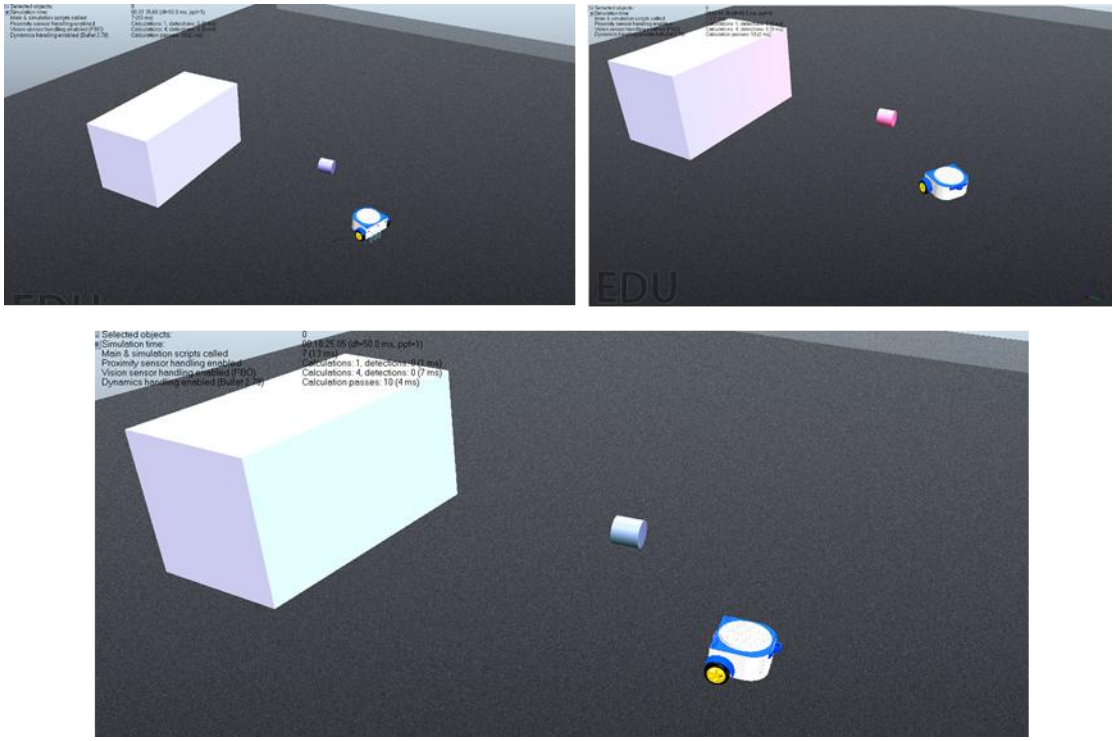
1. The robot starts in the dark. It says “It’s dark” and turns its LED red.
2. While it stays dark, the robot keeps turning right.
3. When it finally sees light, it says “Light detected”, changes its LED to green, and keeps turning right until it’s dark again.
4. Back in darkness, it says “Darkness again”, turns its LED red, and makes 10 right turns.
5. If it is still dark, the robot turns left instead of right.
6. When light is detected once more, it says “Light detected again”, changes LED to green, and turns left until it becomes dark again.

In Short

The robot is like a “light watcher”:

- Red = dark → turn around until you find light.
- Green = light → turn until it’s dark again.
- Sometimes it makes a fixed number of turns (10 times), just as a test.

Scenes of FOSSBot Movement



Picture 61. Scenes of the scenario

Picture 62. The FOSSBot Terminal

Experimenting with Python

```
import time
# --- Parameters ---
turn_delay = 0.2   # time (seconds) for each small turn step
turn_repeats = 10  # fixed number of turns in one part of the program
# --- Functions simulating FOSSBot API ---
def is_dark():
    # Returns True if the light sensor detects darkness
    pass
def set_color(color):
    # Changes robot LED color (e.g., "red", "green")
    pass
def turn_right():
```

```
    # Turns robot slightly to the right
    time.sleep(turn_delay)
def turn_left():
    # Turns robot slightly to the left
    time.sleep(turn_delay)
def print_message(msg):
    # Displays or logs a message
    print(msg)
# --- Main program ---
# Start in the dark
print_message("It's dark.")
set_color("red")
# While it is dark, keep turning right
while is_dark():
    turn_right()
# Light detected
print_message("Light detected.")
set_color("green")
# Keep turning right until it becomes dark again
while not is_dark():
    turn_right()
# Darkness returns
print_message("Darkness again.")
set_color("red")
# Turn right 10 times as a test
for _ in range(turn_repeats):
    turn_right()
# While it remains dark, turn left
while is_dark():
    turn_left()
# Light detected again
print_message("Light detected again.")
set_color("green")
# Keep turning left until it is dark again
while not is_dark():
    turn_left()
```

Notes for Teachers / Students

- The functions (`is_dark()`, `set_color()`) are placeholders — in a real robot or simulator they call the FOSSBot's API.
- The turns are based on *time*, not angle. This means the robot may not always turn the same exact angle (depends on speed, battery).
- The LED colors act as a status signal (red = dark, green = light). This makes debugging easier.
- Students can experiment with:

- Changing turn_repeats (more or fewer fixed turns).
- Adjusting turn_delay (shorter = smoother movement, longer = bigger steps).
- Testing in different light conditions.

Scenario 6: Measuring Velocity

Age group / ISCED level: 13–16 years old (Lower Secondary, ISCED 2–3)

Learning goals:

- Connect robotics with physics concepts: distance, time, velocity.
- Learn how to use timers in programming.
- Apply a mathematical formula inside a program: velocity = distance / time.
- Develop skills in debugging and unit conversion.

Subject links:

- Physics: Motion, velocity, speed formula.
- Mathematics: Division, units (cm, s).
- ICT / Computer Science: Variables, timers, text output.

Required tools:

- FOSSBot robot or simulator
- A flat surface for movement (30 cm marked distance)

Program Explanation (Step by Step)

1. Print the robot's initial velocity as *0 cm/s*.
2. Start the timer.
3. Move forward for 30 cm.
4. When finished, calculate velocity = 30 ÷ (time that has passed)
5. Print the robot's measured velocity in cm/s.

Scenario Narrative

Students imagine FOSSBot as a physics lab assistant. It moves exactly 30 cm while measuring how long it took. Then it applies the physics formula *velocity = distance / time* and prints the result. This activity demonstrates how math and science equations can be embedded in code and how robots can act as experimental tools.

Extensions / Variations

- Physics link: Change the distance (e.g., 50 cm, 100 cm) and compare velocities.
- Math link: Convert results from cm/s to m/s.
- Programming link: Add variables for distance and velocity to make the code more flexible.
- Challenge: Ask students to repeat the experiment multiple times and calculate the average velocity.

Pseudocode

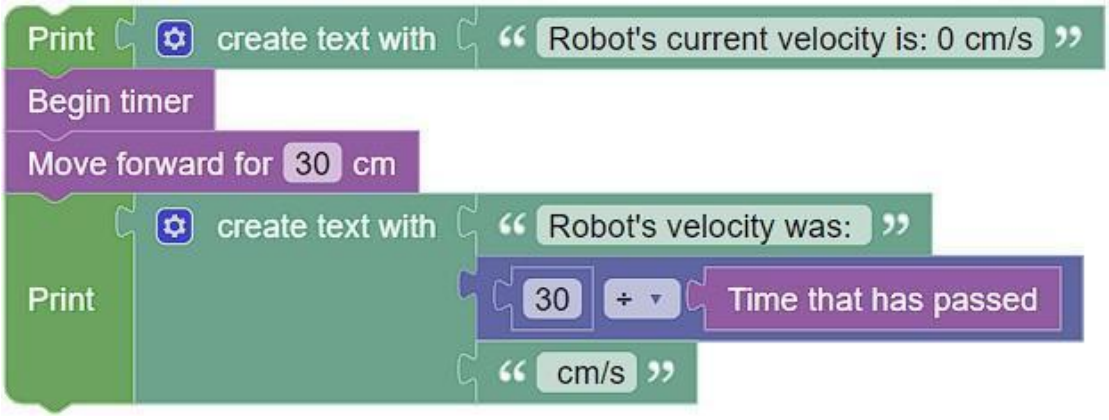
```
[ Print "Robot's current velocity is: 0 cm/s" ]     // Start state
[ Begin timer ]                                     // Timer – start counting
[ Move forward for 30 cm ]                          // Movement – fixed distance

[ Print "Robot's velocity was: " ]                  // Output
    + [ 30 ÷ Time that has passed ]                 // Calculation – distance / time
    + [ " cm/s" ]                                   // Unit – velocity result
```

Attention Points for Teachers / Students

- This experiment depends on accurate timers and consistent robot speed (battery level matters).
- Students should repeat trials and compare results — perfect introduction to experimental error.
- The program demonstrates how robots can be used not just for movement, but for data collection and science experiments.

Blocks Code



Picture 63. Blocks Code

Experimenting with Python

```
import time
# --- Parameters ---
distance = 30  # distance robot will move in cm
# --- Functions simulating FOSSBot API ---
```

```
def move_forward_cm(cm):
    # Moves the robot forward by a specific distance in centimeters
    pass
def print_message(msg):
    # Prints a message to console/log
    print(msg)
# --- Main program ---
# Step 1: Print initial velocity
print_message("Robot's current velocity is: 0 cm/s")
# Step 2: Start timer before movement
start_time = time.time()
# Step 3: Move forward for the given distance
move_forward_cm(distance)
# Step 4: Measure time taken
elapsed_time = time.time() - start_time
# Step 5: Calculate velocity = distance / time
if elapsed_time > 0:
    velocity = distance / elapsed_time
else:
    velocity = 0  # Safety fallback in case of error
# Step 6: Print result
print_message(f"Robot's velocity was: {velocity:.2f} cm/s")
```

Notes for Teachers / Students

- Timer function (time.time()) measures how long the robot needed to travel 30 cm.
- The formula velocity = distance / time is directly implemented — this shows how math can become code.
- Encourage students to:
  - Try different distances (50 cm, 100 cm) and compare results.
  - Convert cm/s to m/s by dividing by 100.
  - Repeat the experiment and compute an average velocity.
- Small errors will appear due to motor power, battery level, and surface friction. This is a good discussion point about real-world data and experimental error.

Scenario 7: Responding to Noise

Age group / ISCED level: 12–15 years old (Lower Secondary, ISCED 2)

Learning goals:

- Learn how robots can **react to sound events**.
- Combine different loop types (repeat until, repeat while).
- Develop problem-solving strategies based on external stimuli.

- Explore how robots can change their path after an interruption.

Subject links:

- **Physics**: Sound and noise as vibrations.
- **ICT / Computer Science**: Sensor input, loops, event-driven behavior.
- **Language Arts**: Using robot "speech" as storytelling or feedback.

Required tools:

- FOSSBot robot or simulator
- Noise source (e.g., clapping, tapping, loud voice)
- Obstacle for the final step (if real robot)

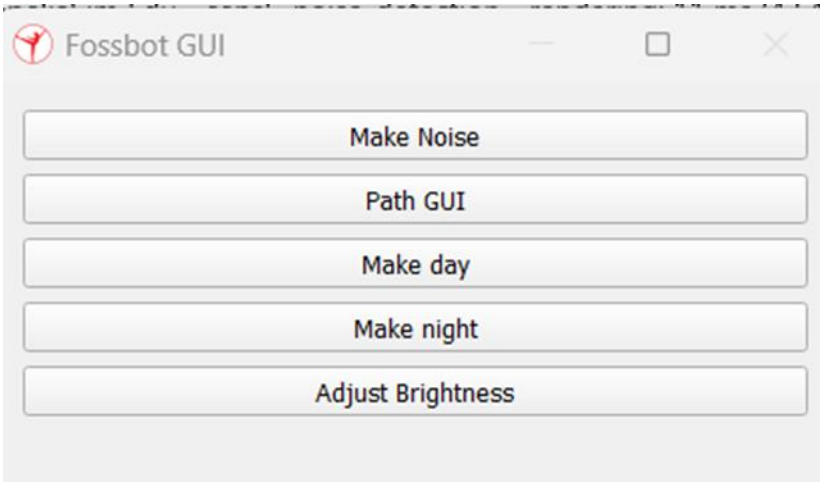
Program Explanation (Step by Step)

1. Launch the FOSSBot application from its installation folder, the program is designed to automatically start CoppeliaSim in the background.
2. Load the *sens5_noise_detection.ttt* stage
3. Print *"Just moving forward..."* and start moving forward until a **noise** is detected.
4. When noise appears, print *"Whoa! What was that noise?!"* and **turn right** as long as the noise continues.
5. Once noise stops, print *"Noise stopped! I'm good!"* and move forward again until the next noise.
6. Finally, print *"Okay, I'm out!"* and move **backwards** until an obstacle is detected.

Scenario Narrative

Students imagine FOSSBot as an **explorer in a noisy cave**. The robot moves forward until it hears a noise. If the noise continues, it turns right as if "searching" for the sound. When the noise stops, it continues forward confidently. At the end, it backs out until it meets a wall, simulating a **safe escape**.

To create and stop a noise click on "Make Noise" or "Stop Noise" respectively to see FOSSBot reactions



Picture 64. FOSSBot GUI (at coppelia stage)

What each button does (conceptually)

1) Make Noise

- **What it triggers:** Briefly sets a *noise event* (e.g., sets a boolean/int signal like `noise=1` for ~0.5–1 s, then resets to `0`).
- **Effect on your robot programs:** Any running behavior that watches the noise sensor (e.g., **Scenario 5**) will react immediately—turning, pausing, or changing course depending on how you coded it.

2) Path GUI

- **What it opens:** A helper mini-tool to create or edit a **path/line** on the floor (color, width, shape).
- **Effect on your robot programs:** Enables **line-following** exercises (e.g., **Scenario 2**). It gives you a clean track to test sensor logic (left/right floor sensors).

3) Make day

- **What it changes:** Sets **bright ambient lighting** in the scene (e.g., raises an ambient-light value or toggles a "day" preset).
- **Effect on your robot programs:** Light sensor reads **bright** → conditions like `is_dark()` become **False**. Your robot may switch to "light" behaviors (LED green, different turning, etc.; see **Scenario 3**).

4) Make night

- **What it changes:** Sets **low ambient lighting** (dark preset).
- **Effect on your robot programs:** Light sensor reads **dark** → `is_dark()` becomes **True**. Your robot will follow its "dark" behaviors (LED red, search/turn routines, etc.).

5) Adjust Brightness

- **What it provides:** A slider (or prompt) to set **continuous brightness** (e.g., a float 0.0–1.0).
- **Effect on your robot programs:** Lets you **calibrate** the threshold where the light sensor flips between "dark" and "light." Perfect for testing edge cases and tuning your code.

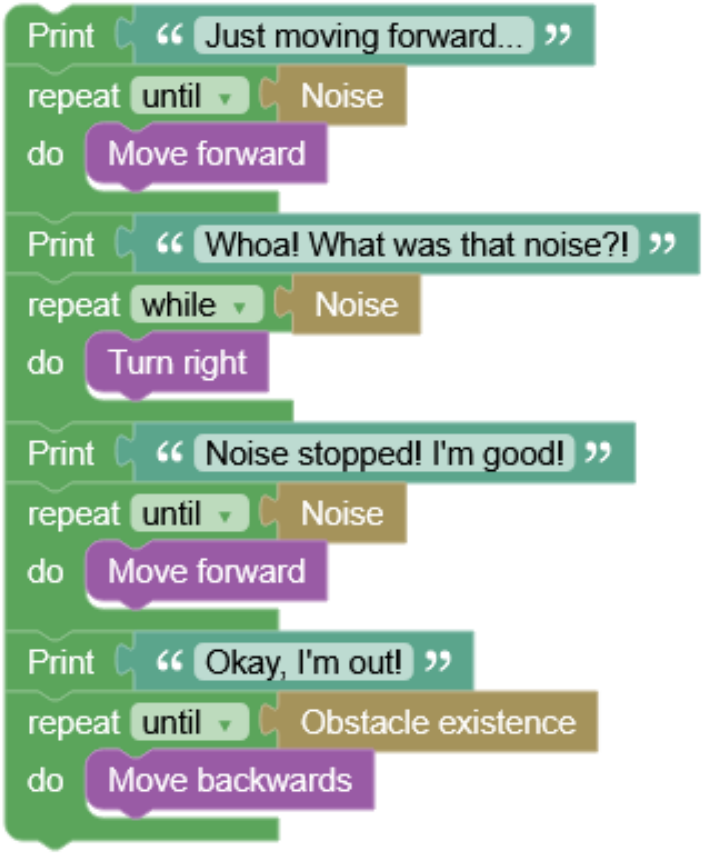
Extensions / Variations

- **Physics link**: Measure how different sounds (clap, whistle, shout) trigger the robot.
- **Math link**: Count how many times the robot reacts to noises in a session.
- **Programming link**: Add a variable noise_count to keep track of total sound events.
- **Creative link**: Turn the scenario into a **storytelling game** — each sound event represents a different danger.

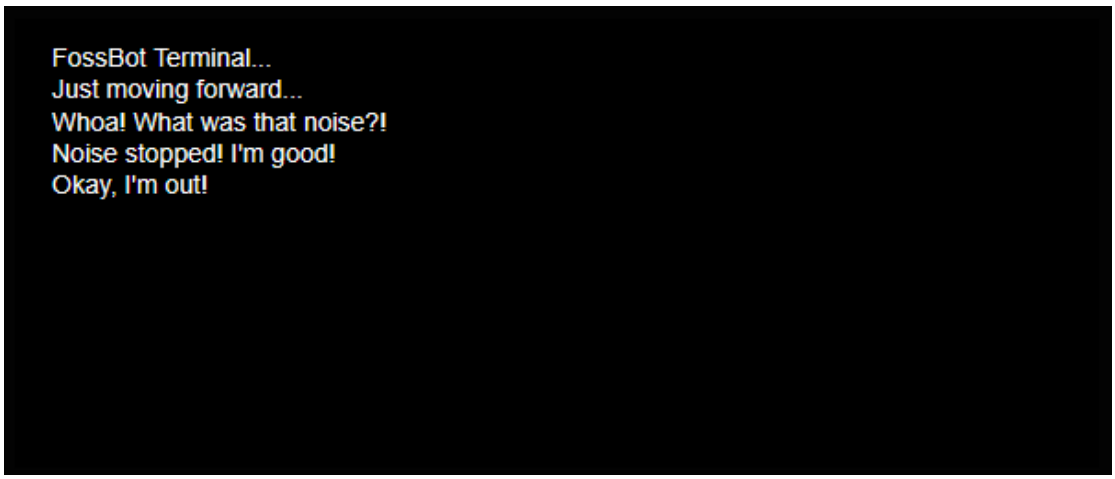
Pseudocode

[Print "Just moving forward..."] // Initial state
[repeat until (Noise)] // Loop – move until noise is detected
→ [Move forward]
[Print "Whoa! What was that noise?!"] // Announce reaction
[repeat while (Noise)] // Loop – while noise continues
→ [Turn right] // Keep turning until quiet
[Print "Noise stopped! I'm good!"] // Announce return to normal
[repeat until (Noise)] // Loop – move until another noise
→ [Move forward]
[Print "Okay, I'm out!"] // Announce final stage
[repeat until (Obstacle existence)] // Loop – move backwards until obstacle
→ [Move backwards]

Blocks Code



Picture 65. Blocks Code



Picture 66. FOSSBot Terminal

Experimenting with Python

```
import time
# --- Functions simulating FOSSBot API ---
def noise_detected():
    # Returns True if the noise sensor detects sound
    pass
```

```
def obstacle_exists():
    # Returns True if an obstacle is detected by front sensor
    pass

def move_forward():
    # Moves the robot forward continuously
    pass
def move_backwards():
    # Moves the robot backwards
    pass
def turn_right():
    # Turns the robot slightly right
    time.sleep(0.2)
def stop():
    # Stops robot movement
    pass
def print_message(msg):
    # Print or log messages for debugging
    print(msg)

# --- Main program ---
# Step 1: Move forward until noise is heard
print_message("Just moving forward...")
while not noise_detected():
    move_forward()
# Step 2: While noise is present, keep turning right
print_message("Whoa! What was that noise?!")
while noise_detected():
    turn_right()
# Step 3: When noise stops, move forward again until noise reappears
print_message("Noise stopped! I'm good!")
while not noise_detected():
    move_forward()
# Step 4: Back out until obstacle is reached
print_message("Okay, I'm out!")
while not obstacle_exists():
    move_backwards()
stop()  # Final stop
```

Notes for Teachers / Students

- Noise sensitivity can be adjusted — background classroom chatter may trigger false detections.
- This scenario introduces the idea of **event-driven behavior**: robot actions depend on external stimuli.

- Backwards motion until obstacle shows a new concept: **reverse navigation**.
- Encourage students to test with different **sound levels and sources** and discuss **sensor reliability**.

Teacher Guidance: Using the FOSSBot GUI

The **FOSSBot GUI** allows you to simulate noise, light, and path conditions directly in CoppeliaSim. Teachers can use it as an **interactive playground** to guide students through experiments in robotics, physics, and computational thinking. The proposed activities represent potential scenarios that teachers and students can explore once they feel more comfortable with the FOSSBot platform. They are not required first steps, but rather opportunities to extend learning and experiment with different sensors and conditions. After becoming familiar with the basic functions of FOSSBot and running simple movement programs, teachers can gradually introduce these exploratory tasks. Each one is designed to highlight how the GUI buttons (noise, light, path, brightness) connect directly with the robot's behavior, encouraging curiosity, experimentation, and deeper understanding of robotics concepts.

Activity 1: Noise Reaction

Steps

1. Run a noise-sensitive program (e.g., Scenario 5).
2. Press *Make Noise* briefly → observe robot response.
3. Hold *Make Noise* longer → observe differences.

Observe

- Does the robot behave differently with short vs. long noise?
- How fast does it react?

Reflect

- Why does continuous noise produce continuous turning?
- How is this similar to real-world event-driven systems (e.g., alarms)?

Activity 2: Light and Darkness

Steps

1. Run a light-sensor program (e.g., Scenario 3).
2. Switch between *Make day* and *Make night*.
3. Use *Adjust Brightness* to fine-tune light levels.

Observe

- At what brightness does the robot change behavior?

- Do LEDs (red/green) always switch at the right time?

Reflect

- Why is calibration important for sensors?
- How does this relate to real robots in outdoor vs. indoor environments?

Activity 3: Path Following

Steps

1. Open *Path GUI* and draw a straight black line.
2. Run a line-following program (Scenario 2).
3. Add curves or zig-zags.

Observe

- Does the robot stay on track?
- What happens when it reaches a sharp turn?

Reflect

- Why does the robot "zig-zag"?
- What strategies could we add in code to make it smoother?

Activity 4: Combining Events

Steps

1. Run a program that uses both light and noise.
2. While following a path, press *Make night* or *Make noise*.
3. Observe how the robot reacts to multiple events.

Observe

- Which event does the robot respond to first?
- Does it ignore one of them?

Reflect

- Why is prioritization important in robotics?
- Can you think of real-world robots that face multiple inputs at once?

Activity 5: Storytelling with Robots

Steps

1. Students imagine the robot as a character (e.g., explorer).
2. Teacher triggers GUI events (night, noise, day).
3. Robot reacts with movement, LEDs, and printed text.

Observe

- How do the robot's behaviors match the story?
- Do students understand cause → effect?

Reflect

- How can we use robots to make abstract concepts (like sensors) more fun and relatable?

Activity 6: Data Collection

Steps

1. Run the velocity program (Scenario 4).
2. Change conditions (light/noise) with the GUI.
3. Record times and velocities.

Observe

- Do times differ across trials?
- What's the average velocity?

Reflect

- Why do results vary?
- How does this show the difference between theory and experiment?

Scenario 8: Edge Detection and Safe Navigation

Age group / ISCED level: 13–16 years old (Lower Secondary, ISCED 2–3)

Learning goals:

- Understand how floor sensors can detect edges or drops.
- Learn to combine multiple conditions with if and else if.
- Practice safety-oriented programming: robot avoids "falling."
- Explore how movement, reversal, and turning ensure safe navigation.

Subject links:

- **Physics**: Stability, balance, and motion.
- **ICT / Computer Science**: Conditional logic, sensor-based decisions.
- **Engineering**: Safety features in robotics design.

Required tools:

- FOSSBot robot or simulator
- A flat surface with visible edges (or simulated table in CoppeliaSim)

Program Explanation (Step by Step)

1. Launch the FOSSBot application from its installation folder, the program is designed to automatically start CoppeliaSim in the background.
2. Load the *sens6_edge_detection.ttt* stage
3. **If both floor sensors detect ground (≠ 0):** The robot moves forward safely.
4. **Else if left ≠ 0 and right = 0:** Edge on the right detected → Robot moves backwards 15 cm, turns left 90°, then continues turning left.
5. **Else if left = 0 and right ≠ 0:** Edge on the left detected → Robot moves backwards 15 cm, turns right 90°.
6. **Else if left = 0 and right = 0:** Edge directly ahead → Robot moves backwards 15 cm, then turns left 90°.

Scenario Narrative

Students imagine FOSSBot exploring the **edge of a table or stage**. The robot continuously checks if the floor is present under both sensors. If one side detects "nothing" (edge), it quickly backs away and turns to safety. If both sensors detect an edge, the robot retreats fully and changes direction. This activity shows how robots can be programmed to **avoid danger automatically**.

Extensions / Variations

- **Math link**: Measure how far the robot moves backwards (15 cm) and calculate time taken.

- **Engineering link**: Discuss how similar edge detection is used in real robots (vacuum cleaners, drones).
- **Programming link**: Add a counter for number of "edge saves" and display it.
- **Challenge**: Change the turn angles (e.g., 45° instead of 90°) and compare safety results.

Pseudocode

[repeat until (Obstacle existence)] // Loop – continue until obstacle stops robot

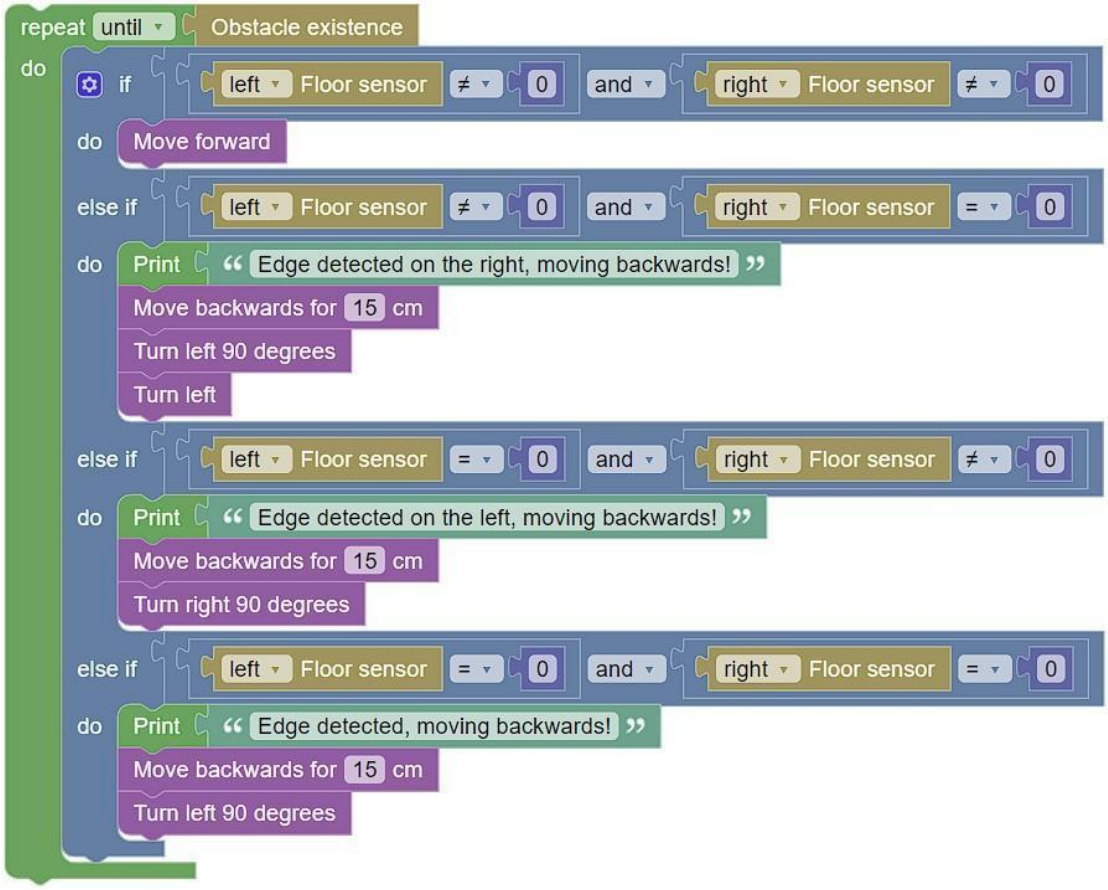
→ *[if (Left sensor ≠ 0 AND Right sensor ≠ 0)]*
→ *[Move forward] // Safe – floor under both sensors*

→ *[else if (Left sensor ≠ 0 AND Right sensor = 0)]*
→ *[Print "Edge detected on the right, moving backwards!"]*
→ *[Move backwards 15 cm]*
→ *[Turn left 90°]*
→ *[Turn left] // Extra left turn*

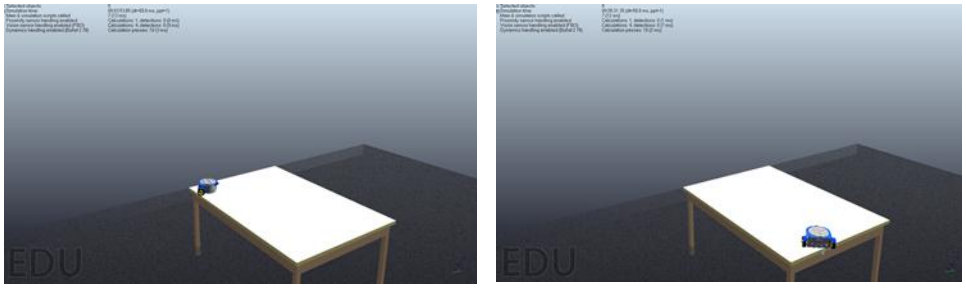
→ *[else if (Left sensor = 0 AND Right sensor ≠ 0)]*
→ *[Print "Edge detected on the left, moving backwards!"]*
→ *[Move backwards 15 cm]*
→ *[Turn right 90°]*

→ *[else if (Left sensor = 0 AND Right sensor = 0)]*
→ *[Print "Edge detected, moving backwards!"]*
→ *[Move backwards 15 cm]*
→ *[Turn left 90°]*

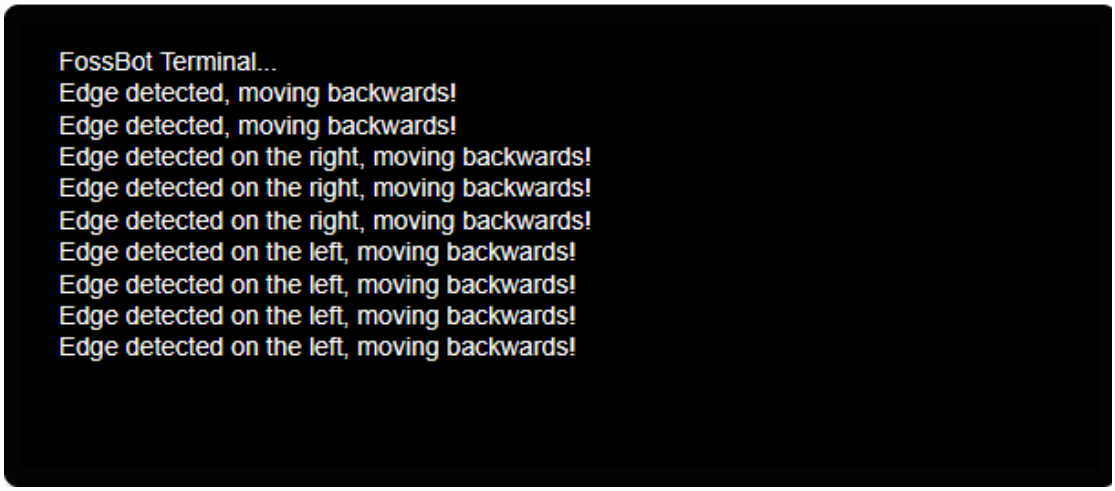
Block Code



Picture 67. Block Code



Picture 68. Scenes from the scenario



Picture 69.FoosBot Terminal

Experimenting with Python

```
import time

# --- Functions simulating FOSSBot API ---
def obstacle_exists():
    # Returns True if an obstacle is detected in front
    pass
def left_floor():
    # Returns 0 if no floor detected (edge), non-zero if safe
    pass
def right_floor():
    # Same as left_floor, but for right sensor
    pass
def move_forward():
    # Moves robot forward continuously
    pass
def move_backwards_cm(cm):
    # Moves robot backwards for a given distance
    pass
def turn_left_90():
    # Turns robot left by 90 degrees
    pass
def turn_right_90():
    # Turns robot right by 90 degrees
    pass
def turn_left():
    # Continuous left turning (for adjustments)
    time.sleep(0.5)
def print_message(msg):
    print(msg)
```

```
# --- Main program ---
while not obstacle_exists():
    if left_floor() != 0 and right_floor() != 0:
        move_forward()  # Safe, keep going
    elif left_floor() != 0 and right_floor() == 0:
        print_message("Edge detected on the right, moving backwards!")
        move_backwards_cm(15)
        turn_left_90()
        turn_left()
    elif left_floor() == 0 and right_floor() != 0:
        print_message("Edge detected on the left, moving backwards!")
        move_backwards_cm(15)
        turn_right_90()
    elif left_floor() == 0 and right_floor() == 0:
        print_message("Edge detected, moving backwards!")
        move_backwards_cm(15)
        turn_left_90()
```

Notes for Teachers / Students

- Floor sensor values may vary — calibration is important.
- The **backward distance (15 cm)** prevents the robot from falling but can be adjusted.
- Students should test **different angles** to see how the robot navigates edges.
- This scenario introduces **safety logic** — real robots must avoid dangerous conditions.

Scenario 9: Light Source Detection and Data Collection

Age group / ISCED level: 13–16 years old (Lower Secondary, ISCED 2–3)

Learning goals:

- Understand how light sensors measure brightness in the environment.
- Learn to combine loops (while/until) with conditional checks.
- Practice using variables and counters to record sensor readings.
- Explore how robots can scan surroundings to locate light sources.

Subject links:

- **Physics:** Light intensity, reflection, and detection.
- **ICT / Computer Science:** Loops, conditional logic, and variable usage.
- **Engineering:** Sensors in robotics for environmental mapping.

Required tools:

- FOSSBot robot or simulator
- Dark room or simulated environment with one or more light sources (lamps, LEDs, or spotlights)

Program Explanation (Step by Step)

1. **Initialization:** Variables for light sources (light_source_1, light_source_2) and counters are set to 0.
2. **Scanning begins:** Robot starts turning right **while it is dark** to locate light.
3. **Rotation until next dark spot:** Robot keeps turning until darkness returns (completing a scan).
4. **Light detection:** Each time the light sensor detects brightness (>0), it:
    - Adds the value to light_source_1.
    - Increments the counter (counter1).
5. **Output:** The program prints *"Light Source 1: [value]"* showing recorded measurements.

Scenario Narrative

Students imagine FOSSBot standing in the middle of a room, slowly turning to scan for lamps or windows. Each time it senses light, it stores the intensity value and counts the measurement. After a full scan, the robot outputs results, giving an impression of where the strongest light comes from. This simulates how robots "map" environments to find beacons or navigate toward a light source.

This program simulates a **robot scanning for a light source** in a dark environment (like a lighthouse or lamp).

- The robot rotates and measures light intensity around it.
- It collects data (total intensity + number of readings).
- It outputs results, which could be used later to **estimate where the strongest light source is located**.

Extensions / Variations

- **Math link:** Calculate the *average* light intensity (total ÷ counter).
- **Physics link:** Test different lamp distances and angles, compare intensity readings.
- **Programming link:** Add a second light source variable (light_source_2) and compare values.
- **Challenge:** Program the robot to automatically move toward the brightest light detected.
- **Data extension:** Graph the results of different light positions using spreadsheet software.

Teachers can frame this scenario as an experiment in robotics + physics**:**

- How does a robot “see” light?
- Why do we use counters? (to know how many readings were taken)
- What patterns emerge if you place different lamps around the stage?

Students can:

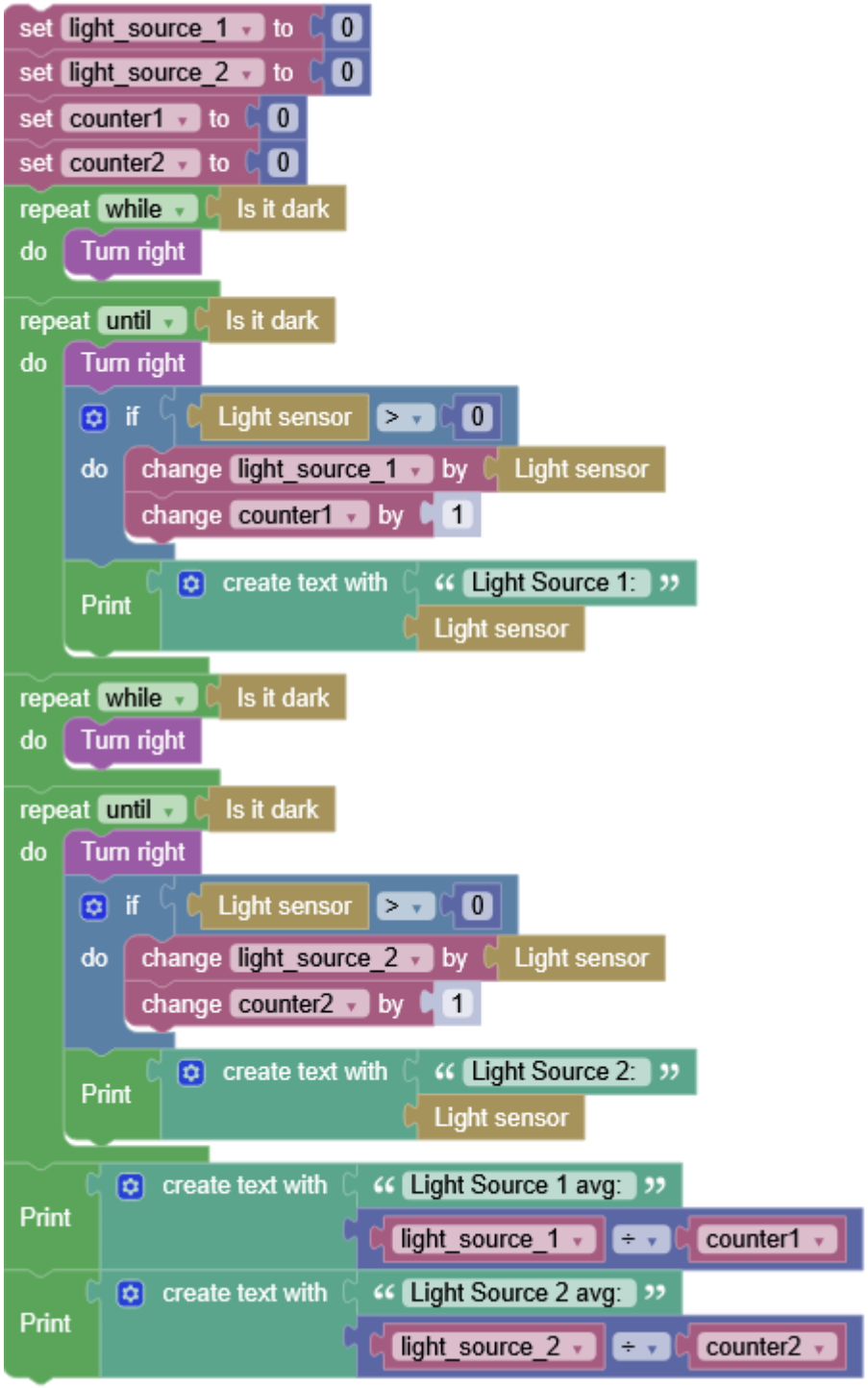
- Adjust the program to measure two light sources instead of one.
- Compare which side of the room has the strongest light.
- Graph the results for a STEM extension.

Pseudocode

```
[ set light_source_1 to 0 ]          // Variable – store total light readings
[ set counter1 to 0 ]                // Variable – count number of readings
[ repeat while (Is it dark) ]
    → [ Turn right ]                 // Rotate until first light is found
[ repeat until (Is it dark) ]
    → [ Turn right ]                 // Continue scanning until dark again
    → [ if (Light sensor > 0) ]
        → [ change light_source_1 by Light sensor ]   // Add intensity
        → [ change counter1 by 1 ]                    // Count reading

[ Print "Light Source 1: " + Light sensor ]   // Output result
```

Block Code



Picture 70. Block Code

Experimenting with Python

```
# ----------------------------------------
# FOSSBot – Light Source Detection Scenario
# ----------------------------------------
# This program makes the robot scan around itself,
# detect light intensity values, and collect them
# into a variable with a counter.
# At the end, it prints the collected data.
# ----------------------------------------

# STEP 1: Initialize variables
light_source_1 = 0   # Total sum of light values detected
counter1 = 0         # Number of readings taken

# Functions below assume access to FOSSBot API
# (or simulated API in CoppeliaSim).
# For this explanation, they are placeholders.

def is_dark():
    """
    Returns True if the environment is dark (sensor ~ 0).
    Returns False if there is some light detected.
    """
    # Example (replace with real sensor read code):
    sensor_value = read_light_sensor()
    return sensor_value == 0

def read_light_sensor():
    """
    Reads the current value from the light sensor.
    Larger values = brighter light.
    """
    # Placeholder: in real code, connect to hardware or simulator.
    return get_sensor_value("light")

def turn_right():
    """
    Turns the robot slightly to the right.
    Repeated calls simulate a rotation.
    """
    rotate_robot("right", speed=30, duration=0.2)

def print_message(text):
    """
    Utility function to print messages to console.
    """
```

```
    print(text)

# STEP 2: Rotate while still dark
while is_dark():
    turn_right()  # Keep turning until some light appears

# STEP 3: Rotate until it becomes dark again
# (This means a full scan of the surroundings)
while not is_dark():
    turn_right()  # Keep rotating during the scan

    # STEP 4: Read light sensor value
    light_value = read_light_sensor()

    # If sensor detects light (value > 0) → store it
    if light_value > 0:
        light_source_1 += light_value   # Add to total brightness
        counter1 += 1
```

Notes for Teachers / Students

- light_source_1 stores the **sum** of all light values measured.
- counter1 counts **how many times** the robot saw light.
- The optional calculation at the end shows the **average brightness**, which connects to mathematics (division, averages).
- In a **real CoppeliaSim / FOSSBot setup**, the placeholder functions (read_light_sensor, rotate_robot) would be replaced with the actual API commands.

Scenario 10: Comparing Two Light Sources

Age group / ISCED level: 13–16 years old (Lower Secondary, ISCED 2–3)

Learning goals:

- Learn how to compare readings from two separate light sources.
- Practice using counters and averages in programming.
- Understand how robots can collect and analyze sensor data systematically.
- Explore how data comparison helps decision-making.

Subject links:

- **Physics:** Light intensity measurement, comparison of sources.
- **Mathematics:** Use of averages and ratios.

- **ICT / Computer Science:** Loops, conditions, variable storage.
- **Engineering:** Multi-sensor environmental scanning.

Required tools:

- FOSSBot robot or simulator
- Two distinct light sources (lamps, LEDs) placed in different positions in the environment.

Program Explanation (Step by Step)

1. **Rotation in the dark:** Robot begins by turning until it detects light.
2. **Scanning phase:** As it rotates, it records brightness values.
3. **Light Source 2:** For each detection, it:
    - Adds sensor reading to light_source_2.
    - Increases counter2 by 1.
4. **Printing results:**
    - Displays each detected value.
    - At the end, prints the **average intensity** for both Light Source 1 and Light Source 2 (total ÷ counter).

Scenario Narrative

Students imagine the robot scanning a room with **two lamps placed in different directions**. While turning, the robot collects data for each source, counts the number of readings, and calculates averages. This allows it to determine which light source is stronger or more consistent. The scenario illustrates how robots can gather **comparative data** from the environment to make intelligent choices.

Extensions / Variations

- **Math link:** Plot average intensities of both sources on a bar chart.
- **Physics link:** Change lamp distance and see how intensity changes with distance.
- **Programming link:** Add logic to automatically move toward the source with the higher average.
- **Challenge:** Include a third light source and extend the code to compare all three.

Pseudocode (block-style format)

```
[ set light_source_2 to 0 ]        // Variable – store total readings for source 2
[ set counter2 to 0 ]              // Variable – count number of readings

[ repeat while (Is it dark) ]
    → [ Turn right ]

[ repeat until (Is it dark) ]
```

→ *[Turn right]*
→ *[if (Light sensor > 0)]*
→ *[change light_source_2 by Light sensor]*
→ *[change counter2 by 1]*

[Print "Light Source 2: " + Light sensor]
[Print "Light Source 1 avg: " + (light_source_1 ÷ counter1)]
[Print "Light Source 2 avg: " + (light_source_2 ÷ counter2)]

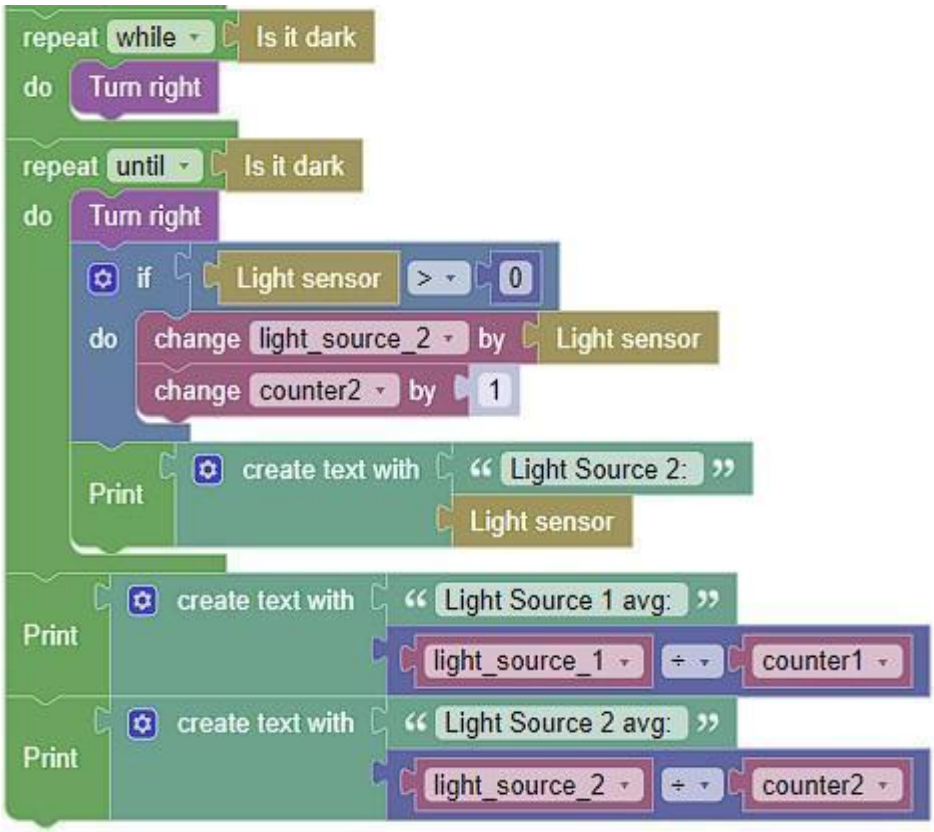
Difference from the Previous Scenario

The previous scenario focused on **detecting and measuring a single light source** using one set of variables (light_source_1, counter1). It showed how the robot can scan, collect values, and print results.

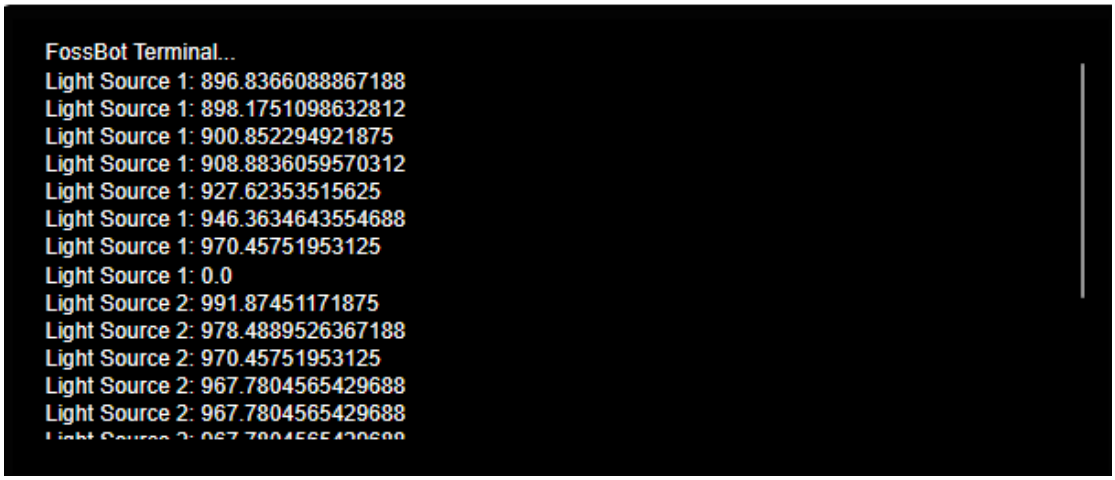
This new scenario **adds comparison**:

- Introduces a second set of variables (light_source_2, counter2).
- Calculates and prints **averages for both sources**, enabling direct comparison.
- Shifts focus from *basic detection* to *data analysis and decision-making*.

Block Code



Picture 71. Block Code



Picture 72. FOSSBot Terminal

Experimenting with Python

```
# ----------------------------------------
# FOSSBot – Light Source Comparison Scenario
# ----------------------------------------
# This program scans the environment for light,
# detects and records sensor values for TWO sources,
# counts the number of readings, and calculates averages.
# ----------------------------------------

# STEP 1: Initialize variables
light_source_1 = 0   # Total brightness for source 1
counter1 = 0         # Count of readings for source 1

light_source_2 = 0   # Total brightness for source 2
counter2 = 0         # Count of readings for source 2

# --- Utility / placeholder functions ---
# (Replace these with the real FOSSBot or CoppeliaSim API calls)

def is_dark():
    """
    Returns True if environment is dark (no light detected).
    """
    sensor_value = read_light_sensor()
    return sensor_value == 0

def read_light_sensor():
    """
    Reads and returns the current light sensor value.
    """
    return get_sensor_value("light")

def turn_right():
    """
```

```
    Makes the robot turn slightly to the right.
    """
    rotate_robot("right", speed=30, duration=0.2)

def print_message(text):
    """
    Prints text to the console.
    """
    print(text)

# STEP 2: Rotate until first light is detected
while is_dark():
    turn_right()

# STEP 3: Scan first light source (Light Source 1)
while not is_dark():
    turn_right()
    light_value = read_light_sensor()
    if light_value > 0:
        light_source_1 += light_value
        counter1 += 1

# STEP 4: Continue scanning to detect second light source
while is_dark():
    turn_right()

while not is_dark():
    turn_right()
    light_value = read_light_sensor()
    if light_value > 0:
        light_source_2 += light_value
        counter2 += 1

# STEP 5: Print averages for both sources
print_message("----- Light Source Comparison -----")

if counter1 > 0:
    avg1 = light_source_1 / counter1
    print_message(f"Light Source 1 average: {avg1:.2f}")
else:
    print_message("Light Source 1 not detected.")

if counter2 > 0:
    avg2 = light_source_2 / counter2
    print_message(f"Light Source 2 average: {avg2:.2f}")
else:
    print_message("Light Source 2 not detected.")

# STEP 6: (Optional) Decide which source is brighter
if counter1 > 0 and counter2 > 0:
    if avg1 > avg2:
```

```
    print_message("Result: Light Source 1 is stronger.")
elif avg2 > avg1:
    print_message("Result: Light Source 2 is stronger.")
else:
    print_message("Result: Both sources have equal brightness.")
```

Teacher Notes

- This program builds directly on the previous scenario by adding a second loop for another light source.
- Students see how to duplicate variable structures (source, counter) and extend logic.
- The new decision block (if avg1 > avg2) introduces comparative reasoning in programming.
- Teachers can highlight how this connects to real-world robotics: a robot choosing which light beacon to follow.

Scenario 11: Timed Movement with Colors

Age group / ISCED level: 12–15 years old (Lower Secondary, ISCED 2)

Learning goals:

- Understand how timers can control robot movement.
- Learn to use sequential instructions with different durations.
- Explore the link between movement, direction, and elapsed time.
- Connect programming logic with color feedback for visualization.

Subject links:

- **Physics:** Motion, distance, and time relationships.
- **Mathematics:** Measuring durations and comparing intervals.
- **ICT / Computer Science:** Loops with conditions, sequential execution.
- **Art / Creativity:** Associating movements with colors.

Required tools:

- FOSSBot robot or simulator
- Open space or simulated environment
- Visual feedback on screen (colors + text)

Program Explanation (Step by Step)

1. **Timer starts** to measure how long the robot has been moving.
2. **Move forward** for 5 time units → color set to red.
3. **Turn right** for 10 time units → color set to green.

4. **Turn left** for 15 time units → color set to blue.
5. **Move backwards** for 20 time units → color set to white.
6. Each action is **printed to the console** so students can follow the sequence.

Scenario Narrative

Students imagine the robot performing a **choreographed movement routine**, where each direction is linked to a specific time interval and color. The robot first advances (red), then turns right (green), then left (blue), and finally moves backward (white). This gives a clear demonstration of how **time-based control** can be used instead of distance or sensors, while colors act as **visual cues** to help track its actions.

Extensions / Variations

- **Math link:** Measure the actual distance covered for each time and calculate speed.
- **Physics link:** Compare how turning right vs. left for different durations changes orientation.
- **Programming link:** Add sound effects (beeps) in sync with colors.
- **Challenge:** Rearrange the sequence or adjust times to create a "robot dance."
- **Advanced:** Add a loop to repeat the routine multiple times automatically.

Pseudocode

[Begin timer] // Start counting elapsed time

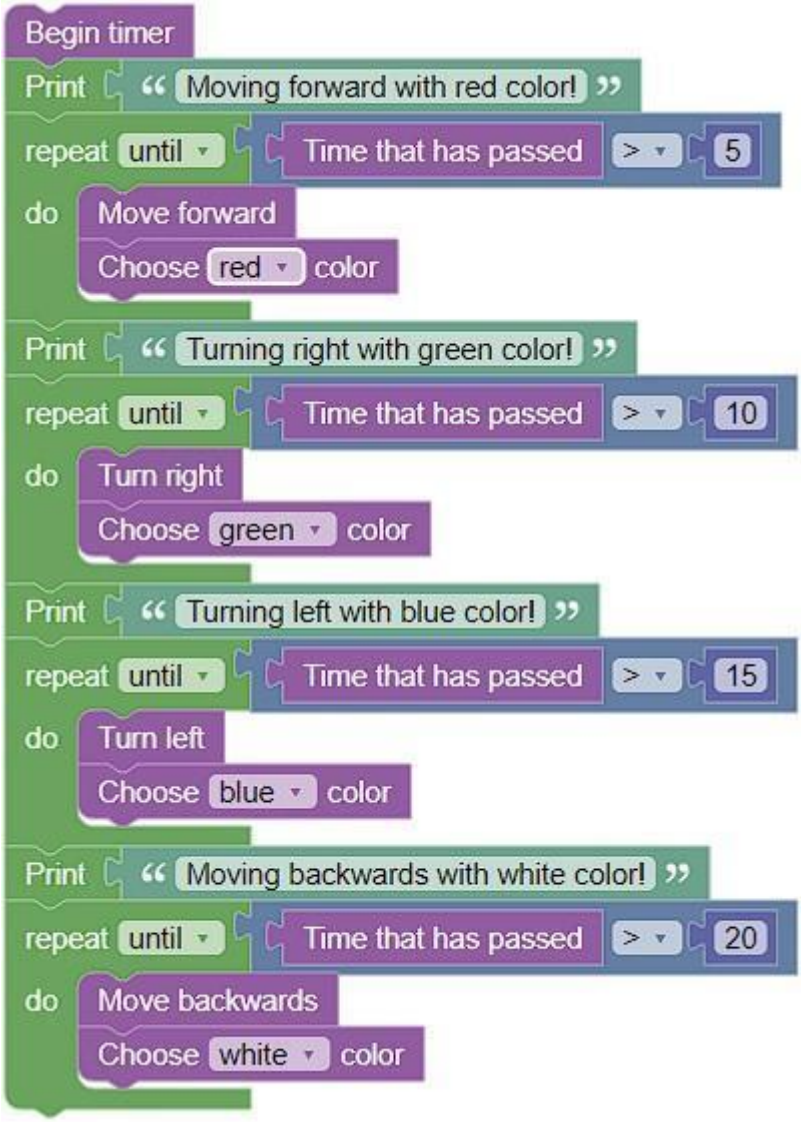
[Print "Moving forward with red color!"]
[repeat until (Time passed > 5)]
    → [Move forward]
    → [Choose red color]

[Print "Turning right with green color!"]
[repeat until (Time passed > 10)]
    → [Turn right]
    → [Choose green color]

[Print "Turning left with blue color!"]
[repeat until (Time passed > 15)]
    → [Turn left]
    → [Choose blue color]

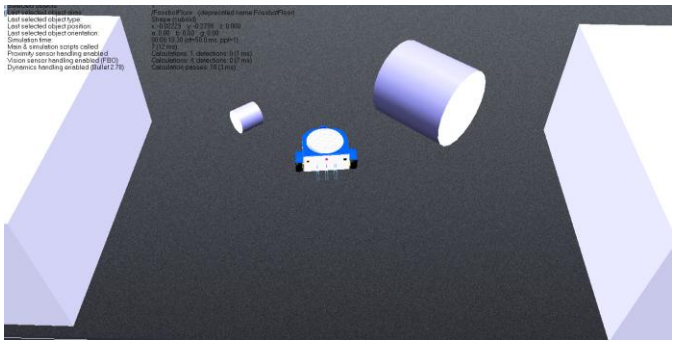
[Print "Moving backwards with white color!"]
[repeat until (Time passed > 20)]
    → [Move backwards]
    → [Choose white color]

Block Code

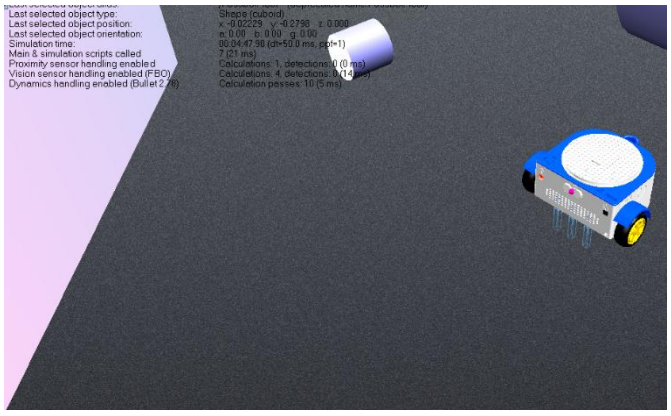


Picture 73. Block Code

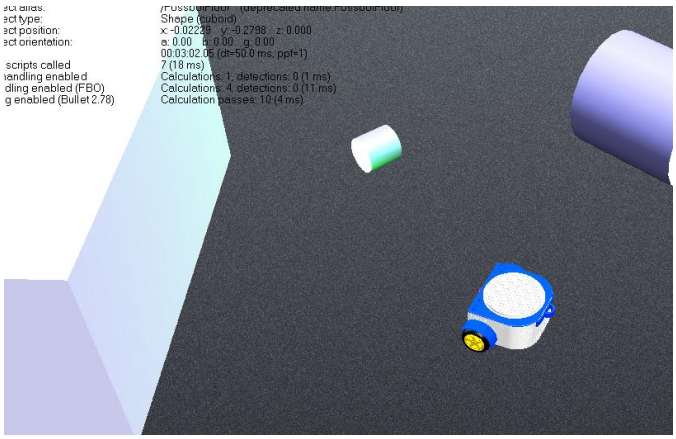
Picture 74. FoosBot Terminal



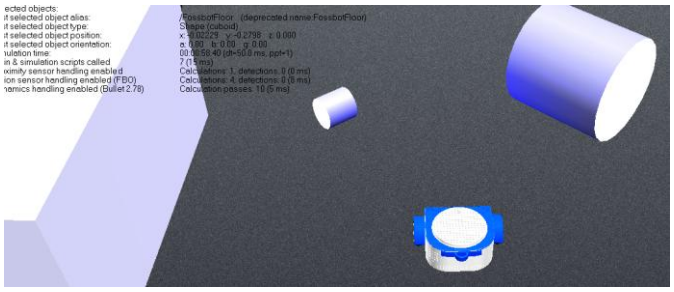
Picture 75. Initial condition



Picture 76. Red Color



Picture 77. Green Color



Picture 78. Blue Color

Scenario 12: Smart Lighting for Environment and Safety

Age group / ISCED level: 2–15 years old (Lower Secondary, ISCED 2)

Students use the **digital FOSSBot application** to create a **smart lighting system**. By programming light and ultrasonic sensors, they design a project where the robot turns its LED on at night and only when movement is detected. This introduces the concepts of **sensors, logic, and automation** in a meaningful real-world context.

Learning Objectives/Outcomes

By the end of this activity, students will be able to:

- Select and connect sensor-equipped devices and/or robotic systems to computers in order to control them or collect data.
- Program an application that controls a pre-assembled robotic or automation system using simple sensors and actuators, within the framework of a learning project.
- Design and program educational robotic and automation systems using physical computing to conduct experiments or creative learning projects involving design and construction.
- Explain how basic sensors work, attempt to construct them, calibrate them, and control them using code on a computer.
- Operate the FOSSBot interface environment (connection, project input, and editing)
- Identify the basic functions and use of sensors (photoresistor and ultrasonic)
- Program automation systems using logical structures (if, else, and)
- Record and interpret sensor values in real environmental conditions
- Create and improve a “smart lighting” system based on environmental data
- Collaborate in groups, making decisions about robot function design and optimization
- Recognize the role of technology in energy saving and safety enhancement

By the end of the scenario, students should have:

a) become familiar with using FOSSBot sensors
b) program sensors in combination with other FOSSBot component
c) developed problem-solving skills, adaptability, and the ability to use new programming and graphical environments through teamwork, collaboration, and creativity

Activity Steps

LESSON PLAN DEVELOPMENT

Description of Teaching and Learning Activities

This scenario employs experiential and inquiry-based learning using the FOSSBot and its visual programming environment.

The learning process focuses on exploration, encouraging students to experiment with commands and conditions by creating an interactive project. No ready-made solutions are provided, promoting knowledge discovery through observation and trial.

Collaborative learning is enhanced through teamwork, as students work in groups of 3–5 to develop their projects, strengthening social interaction and collective knowledge construction.

The scenario integrates gamification elements, adding motivation and engagement to the learning experience. Students create interactive projects using sensors, LEDs, and the computer screen, evoking gameplay experiences.

A worksheet is used for instruction, guiding students through five (5) consecutive activities:

Activity 1: Introduction to the FOSSBot interface and programming environment. Using an image with interactive hotspots, students identify the main functions of the environment.
Activity 2: Understanding basic commands and timing parameters. Students examine the operation of a pre-built project ("black box"), explain the FOSSBot's behavior, and modify the code to change the LED activation/deactivation frequency.
Activity 3: Connecting a sensor to an LED. Students collect data through the FOSSBot's photoresistor, evaluate it, and adjust the code to display messages on the screen and activate the LED.
Activity 4: Adding a second sensor. Students combine photoresistor and ultrasonic sensor data, applying complex logic conditions to activate the LED—thus achieving the main scenario goal of creating a smart light that detects motion.
Activity 5: Understanding check and enhancement of metacognitive awareness via a structured questionnaire (multiple choice and true/false questions) covering concepts from previous activities.

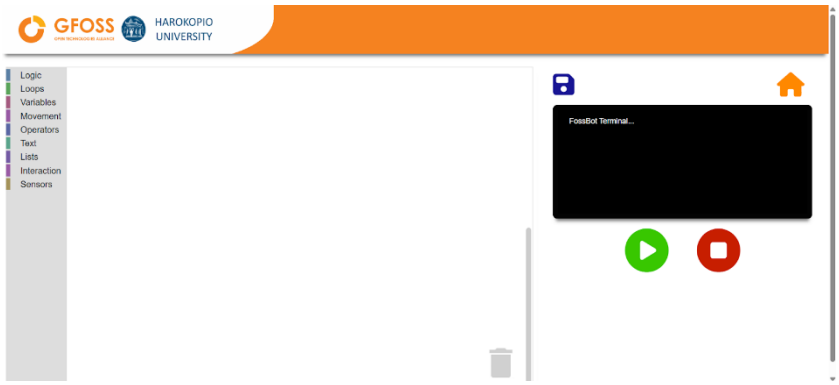
To implement the activities, the FOSSBot must be pre-synchronized with the local Wi-Fi network.

Activity 1

Familiarization with the FOSSBot Programming Environment

Using an Image Hotspots tool (such as https://h5p.org/image-hotspots), interactive points will be created on the buttons of the image below to describe the function of each one:

. Save Code
a. FOSSBot Interface Homepage
b. Run Program
c. Stop Program Execution
d. Trash Bin
e. Blocks Palette
f. Tile Drop Area



Picture 79. FOSSBot Programming Environment

Activity 2

1. Open the FOSSBot. Wait a few seconds for it to connect to your local network.

2. On a computer or laptop connected to the same network as the robot, open Chrome or Firefox and go to the URL: http://FOSSBot-000.local:8081 to access the FOSSBot interface.

3. Upload the project **"black box lights.xml"**.

4. Click **Edit**.

5. Run the project.

6. What do you observe?

.....
.....
.....
.....

0. Make the necessary changes so that the light toggles more quickly. What did you change?

.....
.....

Activity 3

1. Go to the homepage of the FOSSBot interface.
2. Create a new project.
3. Name it **“Smart Lighting”**.
4. In the “short description for the new project,” you may write: *“Creating Smart Lighting for the Environment and Safety.”*
5. First, we will need a sensor to detect light levels. This will be done using a **light sensor (photoresistor)**.
6. Find the range of values from your photoresistor that indicates darkness, using the blocks shown below and running the program (click on):

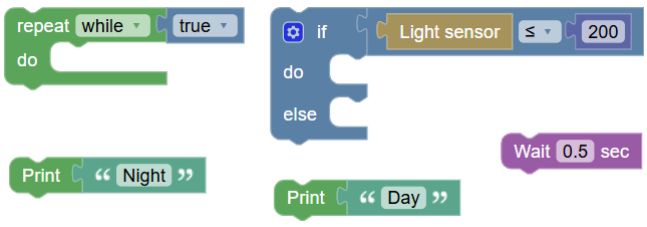
0. Record the value range:

   Daylight: .....

   Darkness: .....

0. Stop the program execution (click on).

0. Modify your program using the blocks below to display the messages **“Day”** and **“Night”** (the threshold values for the photoresistor can be based on your findings in step 7).



Picture 80

0. Run your program. The values will be displayed on your screen.

Picture 81

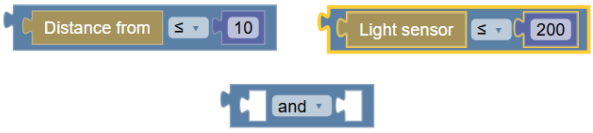
0. Modify your program so that the **FOSSBot's light turns on at night** and **turns off during the day**. You will also need the additional blocks shown below:

Picture 82

Activity 4

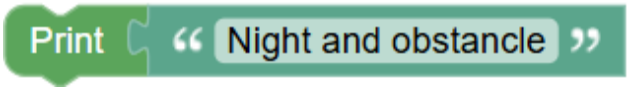
In this activity, we aim to turn on the lights in the dark only when an object (pedestrian or vehicle) passes by. To detect when an object passes, we will need to use an ultrasonic sensor.

1. We will enhance the code from Activity 3. To create a copy of it, go to the homepage of the FOSSBot interface. Select **Export** and then **Import** . Then, enter **edit mode** .
2. Modify the condition inside the **if structure** using the blocks below, in order to achieve greater energy efficiency (i.e., the lights should only turn on when there is an actual need).



Picture 83. 2.Modify the conditions

0. To be more accurate, modify the night-time display text as follows:



Picture 84. modify the night-time display

0. Save your changes.

Pedagogical Value:

This activity combines **STEM learning** with **real-world relevance**. Students link theory with practice, gain experience in automation, and understand how robotics can contribute to **energy efficiency** and **public safety**. The project

encourages **exploratory learning**, **experimentation**, and **group collaboration**, aligning with computational thinking practices.

Pseudocode

[Begin program]
→ [Set LED OFF]
→ [Start sensors]

[repeat while (True)] // Continuous monitoring loop
→ [if (Light sensor > threshold)]
→ [Print "Daytime"]
→ [Set LED OFF]

→ [else] // It is dark
→ [if (Ultrasonic sensor < 50 cm)]
→ [Print "Night – Motion detected → LED ON"]
→ [Set LED ON]
→ [else]
→ [Print "Night – No motion → LED OFF"]
→ [Set LED OFF]

Experimenting with Python

```
# ------------------------------------------------------
# Scenario: Smart Lighting for Environment and Safety
# ------------------------------------------------------
# This program uses a light sensor (photoresistor)
# and an ultrasonic sensor to create an energy-efficient
# smart lighting system with FOSSBot.
# ------------------------------------------------------

# STEP 1: Initialize environment
LED = False                    # LED state (ON/OFF)
LIGHT_THRESHOLD = 300          # Example threshold for day/night
MOTION_DISTANCE = 50           # Distance in cm for detecting motion

def read_light_sensor():
    """Reads the current brightness level."""
    return get_sensor_value("light")  # Replace with FOSSBot API

def read_ultrasonic_sensor():
```

```
    """Reads the distance from ultrasonic sensor."""
    return get_sensor_value("ultrasonic")  # Replace with FOSSBot API

def set_LED(state):
    """Turns the LED ON or OFF."""
    if state:
        turn_on_led()
    else:
        turn_off_led()

def print_message(text):
    """Utility function to show system messages."""
    print(text)

# STEP 2: Continuous monitoring loop
while True:
    light_value = read_light_sensor()        # Check brightness
    distance = read_ultrasonic_sensor()      # Check motion distance

    if light_value > LIGHT_THRESHOLD:
        # Case 1: Daytime
        set_LED(False)
        print_message("Daytime – LED OFF")

    else:
        # Case 2: Nighttime
        if distance < MOTION_DISTANCE:
            # Motion detected at night → LED ON
            set_LED(True)
            print_message("Night – Motion detected → LED ON")
        else:
            # No motion → LED remains OFF
            set_LED(False)
            print_message("Night – No motion → LED OFF")
```

Appendix 1. Glossary of Educational & Technical Terms

21st-Century Competencies – A set of key skills and attitudes (critical thinking, collaboration, creativity, communication, digital literacy) that learners need to succeed in modern societies and workplaces.

Block-Based Programming – A beginner-friendly coding approach using visual blocks instead of text, allowing learners to build programs by stacking commands like puzzle pieces.

Computational Thinking – A problem-solving process involving decomposition, pattern recognition, abstraction, and algorithm design, used both in programming and across subjects.

CoppeliaSim – A robotics simulation software used to model, program, and test robots in virtual 3D environments before real-world deployment.

Cross-Disciplinary Applications – Learning activities or projects that connect concepts across multiple school subjects (e.g., math + art + technology).

Curriculum – The structured set of subjects, learning goals, and competencies designed to guide teaching and student progress.

Debugging – The process of finding and fixing errors (bugs) in code to ensure a program runs correctly.

Differentiated Learning – An instructional approach that adapts teaching methods, materials, and tasks to meet diverse student needs.

DigCompEdu Framework – The European framework for educators' digital competence, outlining skills for teaching, learning, and professional development in the digital age.

Digital Robots – Robots that exist in software environments (like simulations) rather than as physical machines, used for virtual testing and practice.

Educational Scenarios – Structured learning activities or stories that guide students through using technology or robotics to achieve educational objectives.

FOSSBot – An open-source educational robot designed for learning programming, robotics, and STEM concepts, both physically and virtually.

IMU Sensor – Inertial Measurement Unit; a sensor that measures acceleration, orientation, and angular velocity, helping robots understand movement.

Inclusive – An approach to education that ensures all learners, regardless of ability, gender, language, or background, have equitable access to learning.

Infrared (IR) Floor Sensors – Sensors that detect surface patterns, colors, or edges, helping robots follow lines or avoid falling off tables.

Infrared Proximity Sensors – Sensors that measure reflected infrared light to detect nearby obstacles and distances.

Learning Environment – The physical or digital space where learning takes place, including tools, resources, and interactions.

LEGO – Widely used educational kits (e.g., LEGO Mindstorms, LEGO Spike)

that combine building blocks with robotics and programming.

Logic Structures – Programming constructs (if/else, loops, conditions) that control how a program makes decisions and repeats tasks.

Mechanical Design – The physical construction of a robot, including structure, motors, and moving parts.

Multimodal – Using multiple modes of representation (e.g., text, visuals, audio, simulations) to enhance learning and engagement.

Open Educational Technologies – Free and open-source digital tools and resources designed for teaching and learning, ensuring accessibility and adaptability.

Pedagogical Implication – The impact that a teaching strategy or tool has on learning processes and student outcomes.

Physical Robots – Real, tangible robots that can be programmed and interact with their environment.

Programmable Motion – Robot movements controlled by code, such as moving forward, turning, or stopping based on programmed instructions.

Programming Modes – Different ways to program a robot, such as block-based coding, Python, or real-time control.

Python Programming – A widely used text-based programming language, popular in education due to its simple syntax and versatility.

Real-time Programming – Programming where commands are executed instantly, often used for live robot control.

Sensor – A device that detects changes in the environment (light, distance, motion, etc.) and provides data to the robot.

STEAM – An educational approach that integrates Science, Technology, Engineering, Arts, and Mathematics.

STEAM Education – Teaching and learning that combines science, technology, engineering, arts, and mathematics in creative, problem-solving contexts.

STEM Skills – Core competencies in science, technology, engineering, and mathematics, essential for future careers and innovation.

STEPS Project – A European initiative promoting open-source educational robotics (FOSSBot) and simulations for inclusive STEM/STEAM learning.

Ultrasonic Distance Sensor (HC-SR04) – A sensor that measures distance by sending and receiving ultrasonic sound waves, commonly used for obstacle detection.

Universal Design for Learning (UDL) – A framework that provides multiple means of engagement, representation, and expression to support diverse learners.

Appendix 2 – Recommended Open-Source Resources

Purpose

This appendix lists carefully selected open-source tools, platforms, and communities that educators can use alongside FOSSBot. These resources support programming, simulation, content creation, and community exchange — all without licensing barriers.

1. Programming Environments

- **Blockly / Scratch** – Visual block-based programming environments ideal for beginners.
- **Thonny** – Lightweight Python IDE designed for education.
- **Mu Editor** – Beginner-friendly Python editor with built-in support for microcontrollers.
- **VS Code** – Highly extensible editor with Python extensions (open-source core).

2. Robotics & Simulation

- **CoppeliaSim (EDU version)** – Open-source friendly robotics simulator used for FOSSBot's virtual environment.
- **Webots** – Professional open-source robotics simulator.
- **ROS (Robot Operating System)** – Middleware framework for advanced robotics projects.

3. Educational Platforms & Resources

- **Moodle** – Open-source learning management system (LMS).
- **H5P** – Interactive content creation (quizzes, simulations, multimedia).
- **Jupyter Notebooks** – Interactive Python notebooks for coding, math, and data science.
- **Open Roberta Lab** – Browser-based block programming for robots.

4. Content & Media Creation

- **Inkscape** – Vector graphics editor for diagrams, infographics.
- **GIMP** – Open-source image manipulation program.
- **Audacity** – Audio recording and editing tool (for podcasts or narrations).
- **OBS Studio** – Screen recording and live streaming tool.

5. Communities & Knowledge Hubs

- **GFOSS – Open Technologies Alliance** (https://gfoss.eu) – Greek network for open technologies in education.

- **ScratchEd / Scratch Community** – Sharing programming projects for beginners.
- **Stack Overflow** – Developer Q&A (open access).
- **GitHub / GitLab** – Collaborative code sharing, version control.

6. Teacher Notes

- All resources listed are open-source or have strong open-source editions.
- Teachers are encouraged to verify compatibility with local school IT policies.
- A curated set of direct **download links and tutorials** can be provided as a supplementary file to this manual.