Robotics 25-26

Shared document for all participants

Group	Time (day-time)	Names (3 pax)	GitHub Repository	CPU/mem
G1	Miércoles (15:30-17:30) y Martes (15:30-17:30)	Juan Antonio Álvarez Redondo (Jalvarezbhc@alumnos.unex.es) Daniel Jiménez Fernández (djimenezf@alumnos.unex.es) Miguel Felipe Díaz Simón (mdiazsim@alumnos.unex.es)	https://github.com/JuananAlvarez/ROBOTICA25_26	xqqq12th Gen Intel(R) Core(TM) i9-12900KF Maximum Capacity: 128 GB Number Of Devices: 4 Corsair 16GBx12
G2	Miércoles (15:30-17:30) Martes (15:30-17:30)	Samuel Mena Fernández (smenafer@alumnos.unex.es) Pedro José Casado Morcillo(pcasadom@alumnos.unex.es) Francisco Arroyo Blázquez(farroyob@alumnos.unex.es)	https://github.com/XemnasDestroyer/robotica.git	Portatil: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz 16GB de RAM PC (casa): AMD Ryzen 7 5700G 4.6GHz 32GB de RAM PC Sala beta: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz Kingston 16GB x2 Portatil: Intel core i5 10210U 1.60GHz Samsung 16GB ram x1

G3	Martes (15:30 -17:30) y Jueves (17:30 - 19:30)	José Antonio Pavo Romero (jbravoj@alumnos.unex.es) Guadalupe González Santos (ggonzalees@alumnos.unex.es) Máximo Luis Bueno Martínez (mbuenoma@alumnos.unex.es)	https://github.com/Superpor22/Robotica_g3_25_26.git	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz Size: 16GB Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz Size: 32GB

G4	Miércoles (15:30 - 17:30) Y Martes (15:30 - 17:30)	Alejandro Orozco Santano (aorozcos@alumnos.unex.es)
		
		Álvaro Tardío Fernández (atardiof@alumnos.unex.es)
		
		Fernando Pérez De Vega (fperezc@alumnos.unex.es)
			https://github.com/2Fernando2/robomk.git	`13th Gen Intel(R) Core(TM) i9-13900H @ 4.10GHz / 32GB`
				
				`11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz / 24GB`
				
				`11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz / 16GB`

G5	Miércoles (17:30 -19:30) y Martes (15:30 -17:30)	Miguel Iglesias Estévez (miglesiaq@alumnos.unex.es) Darío Mateos Martín (dmateosa@alumnos.unex.es) Fernando Calle Calle (fcalleca@alumnos.unex.es)	https://github.com/XeihtT/Robotica-25-26.git lastest_branch: multiroom_helios2	Procesador: - AMD Ryzen 7 5800Hº Memoria: - Manufacturer: Samsung - Size: 16 GB - Form Factor: SODIMM - Speed: 3200 MT/s Procesador: - Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz Memoria: - Manufacturer: Toshiba Corporation - Size: 8064MiB

				- width: 64 bits - 33 Mhz
G6	Miércoles(15:30 - 17:30) y Jueves (15:30 - 17:30)	Gonzalo Megías Hernández (gmegiash@alumnos.unex.es) Roberto García García (rgarciaee@alumnos.unex.es) Ismael Ramos Márquez (iramosq@alumnos.unex.es)	https://github.com/gmegiash/Proyecto_Robotica_25-26.git	Procesador: -12th Gen Intel(R) Core(TM) i9-12900KF Memoria: -32 Gb
G7	Martes (15:30 - 17:30) y Miércoles(17:30 - 19:30)	Roberto Vázquez Maestre (rvazquezqj@alumnos.unex.es) Adrián Caballero Ramos (acaballefb@alumnos.unex.es) Sergio Sánchez Amado (ssanchezto@alumnos.unex .es)	https://github.com/robertovazquezmaestre/Robotica_g7_25-26.git	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz/ 16GB Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz/ 16GB Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz Memoria: 32 GB
G8	Martes (15:30 -17:30) y Jueves (17:30 -19:30)	Francisco M. Nieto Mendo (fnietome@alumnos.unex.es) Natalia Carrasco de Miguel (ncarrascoy@alumnos.unex.es) Guillermo Caso Vaca (gcasovac@alumnos.unex.es)	https://github.com/FranMUex/Proyecto-Robotica-25-26-NFG	12th Gen Intel(R) Core(TM) i7-12650H 16GB de RAM Version: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz

G9	Martes (15:30 -17:30) y Miércoles (15:30 -17:30)	César Vázquez Lázaro (cvazquezqn@alumnos.unex.es) Angel Luis Suarez Pitel (asuarezq@alumnos.unex.es) Sergio Jiménez Arroyo (sjimenezt@alumnos.unex.es)	https://github.com/C0deC/ROBOTICA_G9_25_26	13th Gen Intel(R) Core(TM) i7-13700H / 2x16 GB RAM -Pc Sala beta -AMD Ryzen 7 2700X 2x16 GB RAM Portatil: 13th Gen Intel(R) Core(TM) i5-13500H @ 2.60GHz
G10	Miércoles (17:30 -19:30) y Jueves (15:30 -17:30)	Sandra Maldonado Marín (samaldona@alumnos.unex.es) Pedro Gutiérrez Buenestado (pgutierrm@alumnos.unex.es) Pablo Polo Alcántara (ppoloalc@alumnos.unex.es)	https://github.com/ppoloalc/RoboticaG10	- PC Sala Beta - PC casa: 10th Gen Intel(R) Core(™) i5-10400F / 16GB RAM
G11	Miércoles 17:30 - 19:30 Jueves 15:30 - 17:30	Samuel Corrionero Fernández (scorrion@alumnos.unex.es) José Pulido Delgado (jopulidod@alumnos.unex.es) Ismael González Loro (igonzaleoa@alumnos.unex.es)	https://github.com/CoferSamuel/ODFE_robotica	Portatil: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz

G12	Miércoles 17:30 - 19:30 y	Antonio Rodríguez Iglesias (arodriguglw@alumnos.unex.es) Víctor Arrocha Sanchez	https://github.com/JairoFarfanC/ROBOTICAG12	`Version: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz`
				
				`2 x 16 GB RAM`

	Jueves	(vidanss21@gmail.com) Jairo Pabel Farfán Callaú (jafarfanc@alumnos.unex.es)		

Grupos por sesión actual

Martes 15:30	Miércoles 15:30	Miércoles 17:30	Jueves 15:30
G1 G2 G3 G4 G5 G7 G8 G9	G1 G2 G4 G6 G9	G5 G7 G10 G11 G12	G3 G6 G8, G10 G11 G12

Configuración en la que todos los grupos empiezan el martes o el miércoles a las 17:30

Martes 15:30	Miércoles 15:30	Miércoles 17:30	Jueves 15:30
G1 G2 G4 G5 G9 G7	G1 G2 G4 G9 G5 G7	G10 G11 G12 G8 G6 G3	G3 G6 G8, G10 G11 G12

INDEX

Task 0: C++23 and development utils upgrade

Tests, quizs and tutorials

- .0https://www.w3schools.com/quiztest/quiztest.asp?qtest=CPP
- Online C++ Programming Test - Online tests for interview, competitive and entrance examinations.
- https://www.cppbuzz.com/c++-interview-questions-and-answers
- https://www.thegeekstuff.com/2016/02/c-plus-plus-11/
- https://thispointer.com/c11-tutorial/

Code snippets in C++

https://alandefreitas.github.io/moderncpp/

Containers

https://dev.to/pratikparvati/c-stl-containers-choose-your-containers-wisely-4lc4

To complete this task you need to:

1. Download, install, compile, link and execute the test code provided in the Campus
2. Modify the user interface in creative ways to acquire fluency with Qt widgets, including:
    a. make the counter count with a QTimer
    b. add a button to reset the count
    c. add a slider to change the period
    d. build a countdown timer or add a reset button
3. Create a new C++ project using the “New project” option in CLion. Study the files created and use the main.cpp to practice the examples in the following sections.
4. Understand the new **auto** keyword in C++:
    a. auto i = 5 vs auto i = 5.2
5. Study the new **std::tuple**< > type, how to create and access them.

    Basic creation and access:

    ```
    std::tuple<int, std::string, double> t1{42, "hello", 3.14};
    auto t2 = std::make_tuple(100, "world", 2.71);
    ```

```
// Access elements
int x = std::get<0>(t1);        // 42
std::string s = std::get<1>(t1); // "hello"
double d = std::get<2>(t1);     // 3.14
```

Function returning multiple values:

```
std::tuple<bool, int, std::string> processData() {
    // ... some processing
    return {true, 200, "success"};
}

auto [success, code, message] = processData();
```

Storing heterogeneous data:

```
std::vector<std::tuple<int, std::string, double>> records;
records.emplace_back(1, "Alice", 85.5);
records.emplace_back(2, "Bob", 92.3);

for (auto [id, name, score] : records) {
    std::cout << id << ": " << name << " scored " << score << "\n";
}
```

Tuple comparison (lexicographical):

```
std::tuple<int, int> a{1, 2};
std::tuple<int, int> b{1, 3};
bool less = a < b;  // true (compares element by element)
```

Empty and single-element tuples:

```
std::tuple<> empty{};          // 0 elements
std::tuple<int> single{42};    // 1 element
```

Tuples are useful when you need to group different types together temporarily without defining a custom struct.

6. Study the **structured binding** access to tuples, pairs, etc:
   Here are simple examples of structured bindings:

   With tuples:

   ```
   auto tuple = std::make_tuple(42, "hello", 3.14);
   auto [num, text, pi] = tuple;
   // num = 42, text = "hello", pi = 3.14
   ```

With pairs:

```
std::pair<int, std::string> p{10, "world"};
auto [id, name] = p;
// id = 10, name = "world"
```

With arrays:

```
int arr[3] = {1, 2, 3};
auto [x, y, z] = arr;
// x = 1, y = 2, z = 3
```

With structs:

```
struct Point { int x, y; };
Point pt{5, 10};
auto [a, b] = pt;
// a = 5, b = 10
```

Common use with functions returning multiple values:

```
std::map<int, std::string> m{{1, "one"}};
auto [iterator, inserted] = m.insert({2, "two"});
// iterator points to element, inserted is bool success flag
```

Structured bindings let you unpack multiple values from containers, tuples, or structs in one clean line instead of accessing elements individually.

7. Study how **lambdas** work in C++. A lambda is a function without name. It always has this structure: auto mylambda = [](){}. The brackets hold the variables that are visible to the lambda. The parenthesis hold the parameters passed to the lambda. The braces hold the code.

   Sort a std::vector<std::tuple<std::string, int>> using a lambda function to specify

   a. the sorting criterium, choosing between the first or the second field and the sorting order.

b. Create a lambda function to transform from degrees to radians and pass it as a parameter in a call to another method.

8. C++ has five value categories that describe how expressions can be used:

   lvalue - Has a name and persistent location in memory. You can take its address.

   ```
   int x = 5;  // x is an lvalue

   int& ref = x;  // can bind to lvalue reference
   ```

   rvalue - Temporary value without a persistent location. Cannot take its address.

   ```
   int x = 5 + 3;  // "5 + 3" is an rvalue (temporary result)
   ```

   xvalue (expiring value) - An lvalue that's about to be moved/destroyed. Created by `std::move()` or casting to rvalue reference.

   ```
   std::move(x)  // x becomes an xvalue
   ```

   prvalue (pure rvalue) - Temporary values like literals or function return values.

   ```
   getString()  // prvalue (if returns by value)
   ```

   glvalue (generalized lvalue) - Either lvalue or xvalue (anything with identity).

   The key distinction: **lvalues** persist and can be assigned to, **rvalues** are temporary. **xvalues** are the bridge - they have identity like lvalues but can be moved from like rvalues.

   This matters for move semantics: rvalue references (`T&&`) can bind to rvalues and xvalues, enabling efficient moves instead of copies.

9. **Move semantics** in C++ allows objects to transfer their resources (like memory) instead of copying them, which is much more efficient. When you use `emplace_back()` with very large objects in a container like `std::vector`, it constructs the object directly in the container's memory and can move it into place rather than creating a temporary copy first. This avoids expensive copying operations.

   For example:

   ```
   std::vector<LargeObject> vec;
   vec.emplace_back(constructor_args); // Constructs directly, no copying
   ```

Instead of: create temporary → copy into vector → destroy temporary, you get: construct directly in final location. This saves both time and memory, especially with large objects.

The emplace_back(auto &&elem) method will receive its argument with the && modifier to indicate that the variable will be “moved” inside and take its property.

10. Study practical **move semantics** and understand how we can avoid the copy of the 10GB chunk.
    There are many examples of how move semantics work. This is one of them:
    https://embeddeduse.com/2016/05/25/performance-gains-through-cpp11-move-semantics/

11. Study the std::chronos library and learn how to measure the elapsed time between two lines of code:

```
#include <chrono>
auto start = std::chrono::high_resolution_clock::now();
 some_expensive_operation(); //The line of code you want to measure
auto end = std::chrono::high_resolution_clock::now();
const auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
```

12. **THREADS**

```
// Create and start a thread
    std::thread t1(simple_function);
// Create and start a lambda thread with parameter n
    std::thread t3([](int n) {std::cout << n << std::endl;},3);
```

13. Replace the QTimer with a custom-made timer using std::threads. Use this Timer class and call the connect method using std::bind(&ejemplo1::doCount, this). Read about the std::thread library and the std::chrono library.

```
#ifndef TIMER_H
#define TIMER_H
#include <thread>
#include <chrono>
#include <functional>
#include <future>
#include <cstdio>
#include <iostream>

//Concept to ensure the parameter is callable with no arguments
template<typename T>
concept Callable = std::invocable<T>;

class Timer
{
   public:
       Timer(){};
       template <class T>
```

```
        void connect(T f)
        {
            std::thread([this, f = std::move(f)]()
            {
                while(true)
                {
                      if(go.load())
                          std::invoke(f);

std::this_thread::sleep_for(std::chrono::milliseconds(period.load()));
                }
            }).detach();
        };
        void start(int p)
        {
        // COMPLETAR
        };
         void stop()
        {
        // COMPLETAR
        };
        void setInterval(int p)
       {
        // COMPLETAR
       }
   private:
        std::atomic_bool go = false;
        std::atomic_int period = 0;
};
#endif // TIMER_H
```

Explain each line of the code:

- What is a **concept** in C++. Which is its role in our code?
- What is a templated method like **connect**?
- How is the lambda thread capturing variables? Which variables?
- Which restrictions has the **f** variable to be **invoked**?
- What does the **detach** method do?
- What are the std::atomic_bool and std::atomic_int types. Why do they not need a mutex?std::invoke

Once the program is working again with the new timer, do these modifications:

- Study the std::bind() function to wrap parameters

```
// ===== 1. BINDING WITH FIXED PARAMETERS =====
std::cout << "=== Binding with Fixed Parameters ===\n";

// Bind all parameters - creates a callable with no parameters
auto bound_print = std::bind(print_values, 10, 3.14, "Hello");
bound_print(); // Call with no arguments

// ===== 2. USING PLACEHOLDERS =====
std::cout << "\n=== Using Placeholders ===\n";

// _1, _2, _3 represent the 1st, 2nd, 3rd arguments when called
```

```
using namespace std::placeholders;

// Bind first parameter, leave others as placeholders
auto partial_print = std::bind(print_values, 42, _1, _2);
   partial_print(2.71, "World"); // Provides the missing arguments
```

- Add a parameter to the method doCount and pass it in the std::bind call
- How could you modify the code to connect the same timer to several methods?

14. Use of the std::random library

```
#include <iostream>
#include <random>
int main()
{
  // 1. Create a random device (seed source) std::random_device rd;
  // 2. Create a random number generator std::mt19937 gen(rd());
  // 3. Create distributions std::uniform_int_distribution<int> dice(1, 6);
 // Dice roll 1-6 std::uniform_real_distribution<double> percentage(0.0, 100.0);
 // Percentage 0-100 std::normal_distribution<double> height(170.0, 10.0);
// Height: mean=170cm, std=10cm
  std::cout << "=== Random Examples ===\n";
// Generate some dice rolls
   std::cout << "Dice rolls: ";
   for (int i = 0; i < 5; ++i)
     { std::cout <<  dice(gen) << " ";
       std::cout << percentage(gen);
       std::cout << height(gen);
     }
   std::cout << "\n";
```

15. Code some examples in using containers and algorithms from the std library https://en.cppreference.com/w/cpp/header. For example:

    a. Create a 10GB large std::vector<int> and pass it by value, reference and using move semantics. Measure the time spent using the chronos library

    b. Create a 10GB large std::vector<int> and sort it, find the smallest element, find elements in a range, shuffle it, or copy. Use the std::chrono library to measure the time spent by the algorithms

    c. Try the parallel version of sort, min, etc and check the speedups. Check with the top command how the cores of the computer are used.

d. Change the container to a large std::map (dictionary) and measure the accessing times using the key.

16. In C++20, you can initialize a struct using the names of the fields,
    struct A

    { float init = 0, end = 0, step = 0; };

    auto B = A{.init=1, .end=2, .step=3}

17. Create a randomly connected graph using std::map

    a. declare a struct Node{ int id; std::vector<int> links;} ;
    b. declare the map: std::map<int, Node> graph;
    c. initialize it:
    ```
    for(int i=0; i<100; i++)
      graph.emplace(std::make_pair(i, Node{i, std::vector<int>()}));
    ```
    d. connect it
    ```
    for(auto &[key, value] : graph)
     {
        vecinos = get_random_number_between 0 and 5
        for(int j=0; j<vecinos; j++)
          value.links.emplace_back( random_number_between 0 and 99);
     }
    ```

18. Study the main algorithms in std that operate on generic containers: https://en.cppreference.com/w/cpp/algorithm.
    For example, use std::transform to a std::vector<int>from an existing std::vector<std::string> by applying a lambda function to each element: tha lambda could compute the sum of the ASCII value of each character in the string. https://en.cppreference.com/w/cpp/algorithm

19. Given this declaration of a map:

    ```
    struct Object
    {
            int id;
            uint timestamp;
    };
    std::map<int, Object> semantic_map;
    ```

    explain what this C++20 snippet do:

    ```
    semantic_map.insert(std::make_pair(0,Object{0, 45});
    semantic_map.insert(std::make_pair(1,Object{1, 3});
    ```

```
auto removable = semantic_map | std::views::values | std::views::filter([p = params](auto
o) { return o.timestamp > 5; });

for (const auto r: removable)
   semantic_map.erase(r.id);
```

ADVANCED TOPIC: Variadic templates

Variadic templates are a C++ feature that allows functions or classes to accept a variable number of template arguments of different types. Basic syntax:

```
template<typename... Args> void myFunction(Args... args) { // Implementation }
```

The `...` (ellipsis) creates a "parameter pack" that can hold zero or more template arguments.

You can call this function with any number of arguments of any types:

```
myFunction(1, "hello", 3.14, 'c');   // 4 arguments of different types

myFunction(42);   // 1 argument

myFunction();   // 0 arguments
```

Common use case:

- **Fold expressions** (C++17+): Applying operations across all arguments.

  Example 1, print function with variable number of arguments

```
#include <tuple>
#include <iostream>

template<typename... Args>

    void print(Args... args)
        { ( (std::cout << args << " "),  ...);     // C++17 fold expression }
    print(1, "hello", 3.14); // Outputs: 1 hello 3.14
```

  Example 2, print the contents of a tuple

```
#include <tuple>
#include <iostream>

template<typename... Args>
void print_tuple(const std::tuple<Args...>& t)
{
  if constexpr (sizeof...(Args) > 0)
```

```
    { std::apply([](const auto&... args)
        { std::size_t i = 0;
         ((std::cout << args << (++i  != sizeof...(Args) ? ", " : "")), ...);
         std::cout << std::endl;
        }, t);
}
else { std::cout << "(empty tuple)" << std::endl; }
}
int main()
{ auto t1 = std::make_tuple(1, "hello", 3.14, 'c');
  auto t2 = std::make_tuple(42, 99.9);
  auto t3 = std::make_tuple(); // empty tuple
  print_tuple(t1);
print_tuple(t2);
print_tuple(t3);
}
```

Study and explain all steps in the examples.

MATHEMATICS - The Eigen Library

20. We will also use the Eigen math library: https://eigen.tuxfamily.org/index.php?title=Main_Page
    You can see some examples at https://programmerclick.com/article/81002002898/

    a. An example of how to create a Mx2 matrix to hold M 2D vector of double type and initialize to random values.

```
#include <Eigen/Dense>
#include <iostream>
#include <random>

int main()

{ const int M = 5; // Number of 2D vectors // Create M×2 matrix of doubles

  Eigen::MatrixXd matrix(M, 2);

 // Method 1: Using Eigen's built-in random initialization

  matrix = Eigen::MatrixXd::Random(M, 2);

  std::cout << "Method 1 - Eigen::Random() (values between -1 and 1):\n";

  std::cout << matrix << "\n\n";

// Method 2: Custom random initialization with specific range
```

```
std::random_device rd;std::mt19937 gen(rd());

std::uniform_real_distribution<double> dis(0.0, 10.0); // Range [0, 10]

matrix = Eigen::MatrixXd::NullaryExpr(M, 2, [&]()

        { return dis(gen); }); std::cout << "Method 2 - Custom range [0, 10]:\n";
std::cout << matrix << "\n\n";

// Method 3: Element-wise initialization

Eigen::MatrixXd matrix2(M, 2);

for (int i = 0; i < M; ++i)

{ matrix2(i, 0) = dis(gen); // x coordinate

 matrix2(i, 1) = dis(gen); // y coordinate

}

std::cout << "Method 3 - Element-wise initialization:\n"; std::cout << matrix2 <<
"\n\n";

// Accessing individual 2D vectors (rows)

std::cout << "First 2D vector: (" << matrix(0, 0) << ", " << matrix(0, 1) << ")\n";
std::cout << "Second 2D vector: " << matrix.row(1).transpose() << std::endl;
return 0; }
```

b. Use the Eigen functions to find the vector (x,y) with minimum norm: sqrt($x^2 + y^2$)

Task 1: Components, Webots and a simple robot

Introductory reading (first part, up to slide 28)
http://www.vernon.eu/cognitive_robotics/CR03-01.pdf

[To record the state of your Yakuake tabs you can run: .local/bin/yaku -s]
cd ..

THINGS TO DO BEFORE

- remove xone:
  sudo dkms remove xone/v0.3-57-g29ec357 --all
  sudo apt remove xone-dkms
  sudo apt upgrade
- Add a new component to the set of elements that we need to start our project: Lidar3D
  En un terminal nuevo:
  - cd /robocomp/components/robocomp-robolab/components/hardware/laser/lidar3D
  - edit the src/CMakeLists.txt and remove the Open3D string.
  - open a new terminal and cd to "software"
  - git clone https://github.com/RoboSense-LiDAR/rs_driver
  - hacer cmake .; make; sudo make install
  - go back to the terminal with Lidar3D
  - cd ...; make -j21
  - bin/Lidar3D etc/config_helios_webots
  - sudo locale-gen en_US.UTF-8
  - sudo update-locale

- Modify the chochoca.cdsl file:

```
import "Lidar3D.idsl";
import "OmniRobot.idsl";
cd
Component chocachoca
{
  Communications
```

```
        {
            requires Lidar3D, OmniRobot;
        };
        language Cpp11;
        gui Qt(QWidget);

    };
```

- Regenerate with robocompdsl chocachoca.cdsl . (close the emergent windows)
- Change the lines in compute to:

  ```
  auto data = lidar3d_proxy->getLidarDataWithThreshold2d("helios", 5000, 1);
  qInfo() << data.points.size();
  ```

- cmake . and make

In this task will start to use RoboComp to build software components.
Go to https:/we / github.com/robocomp/robocomp and read the tutorials that explain how components work and how they communicate through interfaces using the Ice middleware. After installing it, we will:

- review some of the existing interfaces
- understand the code generating process
- create a new component that “requires” the interfaces:
  - OmniRobot.idsl
  - Lidar3D.idsl

  and draws the robot in a Qt window, as it moves in the simulator controlled with the joystick

Initial steps:

1. Install RoboComp following the instructions here:
2. https://github.com/robocomp/robocomp. It is already installed in the Beta’s machines.
3. You also need to clone to repository under ~/robocomp/components
   a. `https://github.com/robocomp/webots-bridge.git`
   b. `https://github.com/robocomp/webots-shadow.git`
4. Install Webots. If not installed, download their .deb package from cyberbotics.com and install with sudo apt install ./webots_2023b_amd64.deb. It is already installed in the Beta’s machines.

5. Run webots and in the File menu select “Open world” and point to ~/robocomp/components/webots-shadow/worlds/Shadow.wbt
6. In another terminal, go to ~/robocomp/components/webots-bridge and build the component with cmake . y make. The run it with bin/Webots2Robocomp etc/config.
7. In another terminal, goto to ~/robocomp/components
8. Clone your personal robotics repository there with a name according to your group
9. Go inside and create a folder “chocachoca”
10. Go inside and create a component there executing: “robocompdsl chocachoca.cdsl”
11. Edit the newly created file: chocachoca.cdsl

```
import "Lidar3D.idsl";
import "OmniRobot.idsl";

Component chocachoca
{
   Communications
   {
       requires Lidar3D, OmniRobot;
   };
   language Cpp11;
   gui Qt(QWidget);

};
```

12. Add the interfaces OmniRobot.idsl and Lidar3D.idsl to the “requires” line in the Communications sections and delete the rest. Choose C++11 and QtWidget
13. Run “robocompdsl chocachoca.cdsl .” Note the dot at the end of the line.
14. If created without errors, move one level up (cd ..) and add the folder to git BEFORE doing cmake. You don’t have to do that again.
15. cmake . and make. Run the executable just created. You will see a periodic message originated in the “compute()” method.
16. Open the component in the IDE and start coding: use the proxies to call the robot in the simulator and obtain the robot position, laser data
17. Start the joytick by going to:
    a. ~/robocomp/components/robocomp-robolab/components/hardware/external-control/joystickpublish
    b. do cmake . make.
    c. En etc/config, insert the id of your robot.
    d. run the component: bin/JoystickPublish etc/config_shadow to run it.
18. Move the joystick and check that YOUR robot moves around.

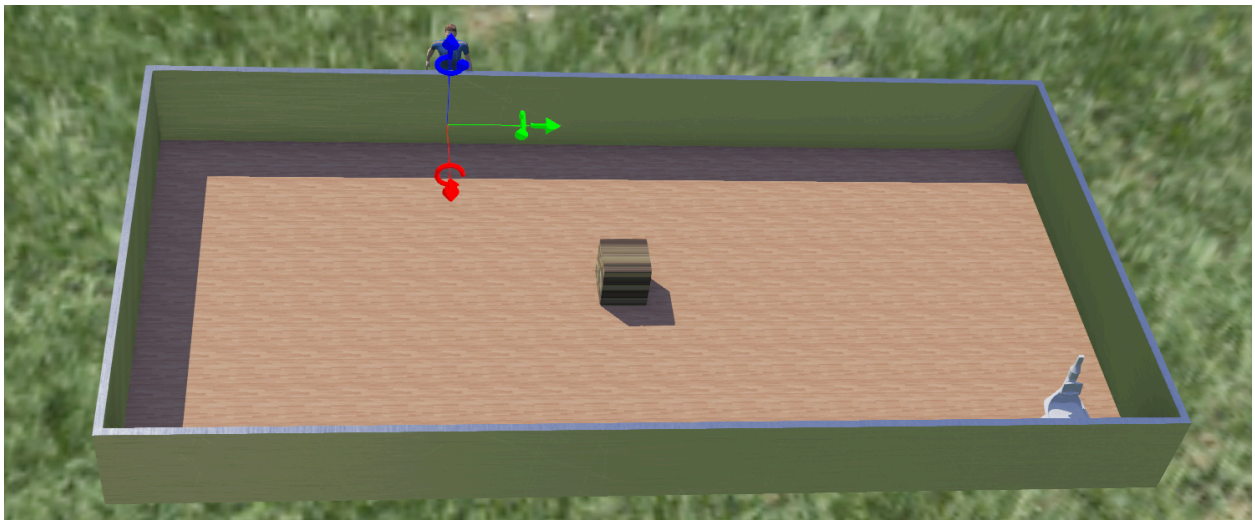


Fig 1. View of the room to be swept

To design a good controller for your robot you need to learn a few things about the robot:

Basic geometry

The figure shows the basic geometry of the robot used in the Webots simulator. In red, the world axis and direction of rotation of the robot with respect to the (third) axis perpendicular to the screen. In green, the Lidar data, is provided as a struct composed f three floats: x, y, z, being the point in the world where the Lidar ray has impacted. The Lidar data is provided in a std::vector as defined in the interface file (Lidar3D.idsl). The vector is sorted so is starts with the points triggered at -PI radians (the back-left) and proceed to the PI radians (the back -right). The central point of the vector is occupied by points directed at 0 radians (in the heading direction of the robot). The axis dis 400mm wide and long.

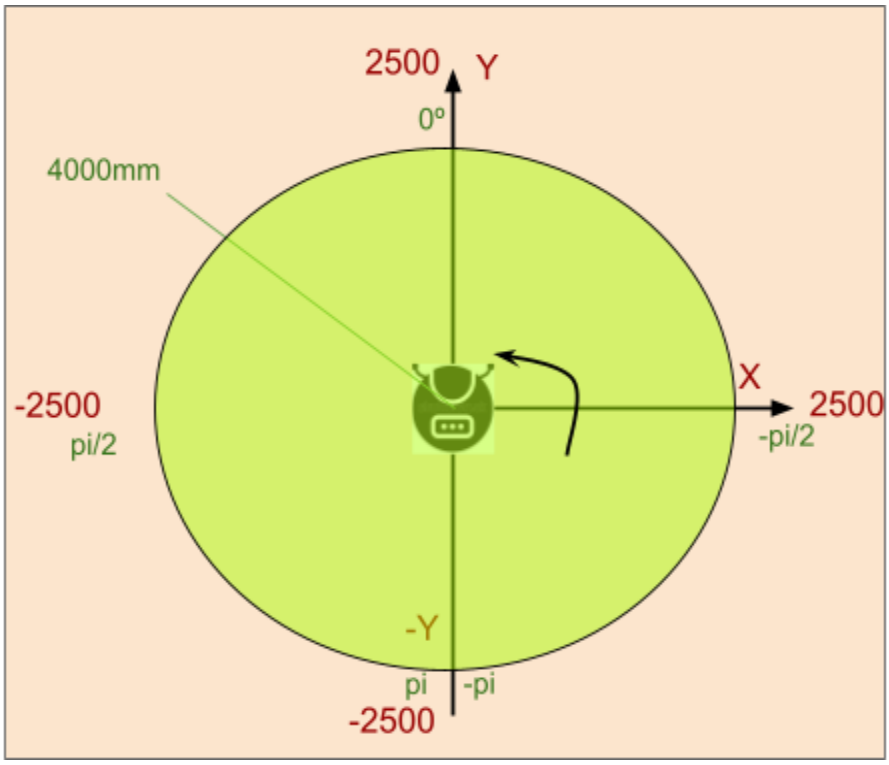


Fig 2. 3D LiDAR projection on floor

Interfaces

The information about the interfaces that you are using to communicate with the simulator Webots is in the folder ~/robocomp/interfaces/IDSLs/(*OmniRobot.idsl, Laser3D.idsl, etc.)* You can open those files and look at the data structures used and the methods provided by the remote component implementing them,

The *OmniRobot.idsl* interface provides two important methods:

- *void setSpeedRobot(advance_speed, rotation_speed);* This method sets the current advance and rotation speed of the robot. Note that the simulator will maintain it even if you close your program. The robot is speed controlled, meaning that you cannot command it to turn 1 radian, for example. You can only tell it to spin at 1 rad/second and it will continue to do that until you send it a different command.
- *void getBaseState(x, z, angle);* This method provides the current pose of the robot as three numbers: x, z in millimetres and angle in radians, as depicted in the previous figure.

The *Lidar3D.idsl* interface provides one important method:

- TData getLidarData(string name, float start, float len, int decimationDegreeFactor)
  name,is the name of the Lidar device (“helios” of “bpearl”)
  start,is the angle in radians to start the reading (0)
  len, is the angle in radians added to start to complete the reading (2*pi)
  decimationDegreeFactor, is 1 for the complete reading, 2 to obtain one in two, etc.

Both calls are synchronous, meaning that they will block until the action is finished.

PART I CHOCACHOCA

Add code to the compute method in SpecificWorker to:

- read 3D lidar data
- filter 3D lidar data to obtain a 2D cloud
- draw lidar in a Qt window
- check distance to closest obstacles
- decide whether to continue or to stop and turn
-

To draw the LiDAR points in the QWidget that comes with the component:

- In SpecificWorker.h:
  add #include <abstract_graphic_viewer/abstract_graphic_viewer.h>
  add these variables as private,

```
// graphics
QRectF dimensions;
AbstractGraphicViewer *viewer;
const int ROBOT_LENGTH = 400;
QGraphicsPolygonItem *robot_polygon;
```

  add this slot,

```
void new_target_slot(QPointF);
```

- In SpecificWorker.cpp,
  in initialize() add

```
this->dimensions = QRectF(-6000, -3000, 12000, 6000);
viewer = new AbstractGraphicViewer(this->frame, this->dimensions);
this->resize(900,450);
viewer->show();
const auto rob = viewer->add_robot(ROBOT_LENGTH, ROBOT_LENGTH, 0, 190, QColor("Blue"));
robot_polygon = std::get<0>(rob);

connect(viewer, &AbstractGraphicViewer::new_mouse_coordinates, this, &SpecificWorker::new_target_slot);
```

in compute add,

```
draw_lidar(filter_data.value(), &viewer->scene);
```

- Add a QFrame to the component's widget. Open the designer application and open the mainUI.ui file and add a QFrame. Resize it to the main widget size. Make sure the name is "frame"

- Add this new method:

```
void SpecificWorker::draw_lidar(const auto &points, QGraphicsScene* scene)
{
  static std::vector<QGraphicsItem*> draw_points;
  for (const auto &p : draw_points)
  {
    scene->removeItem(p);
    delete p;
  }
  draw_points.clear();

  const QColor color("LightGreen");
  const QPen pen(color, 10);
  //const QBrush brush(color, Qt::SolidPattern);
  for (const auto &p : points)
  {
    const auto dp = scene->addRect(-25, -25, 50, 50, pen);
    dp->setPos(p.x, p.y);
    draw_points.push_back(dp);   // add to the list of points to be deleted next time
  }
}
```

The QWidget shows a robot-centred view of the lidar points. Move the robot around with the joystick and check that the drawing is correct.

PART II SWEEPING strategies to control the robot

Since we want the robot to cover the floor as soon as possible without colliding with walls or other robots, the basic bounce and turn strategy will not be good enough. You need to think of different ways (modes) to move the robot and a strategy to change among modes.

Some suggestions are:

- Define a state-machine in “compute”. Declare a *enum class* variable[1] for the current mode and introduce a *switch* statement in “compute”. Depending on the value of the current mode, call one method or another.
- Mode “FORWARD”: advance until close to an obstacle. Then turn.
- Mode “WALL”: after getting close to an obstacle, turn to position the robot parallel to the wall and advance keeping it that way. You just need to control the distance to the wall. If too high turn towards the wall, if to short turn away from the wall. When close to an obstacle, change mode
- Mode “SPIRAL”: when the laser field is mostly full (no obstacles around) start movement where the advance speed is steadily increased and the rotation speed steadily decreased. When close to an obstacle change mode.

Basic pattern for the “compute” method

This snippet shows the basic structure of the “compute” method in your control component. Remember that each mode/behaviour called from the “case” statement must be simple, fast and well-coded. They encode short-time states that the robot engages in and constitute the building blocks of its intelligence. The core part is the logic to switch among behaviours.

```
void compute()

        try-catch block to read the laser data

        std::tuple<State, float, float> result;        //State -> enum class
        switch(state)
        {
                case: IDLE
                case: FORWARD
```

[1] https://stackoverflow.com/questions/18335861/why-is-enum-class-preferred-over-plain-enum

```
                result = FORWARD_method(ldata);
            case: TURN
            case: FOLLOW_WALL
            case: SPIRAL
}
state = std::get<State>(result);
try-catch block to send velocities to the robot
```

Snippet 1

With this schema, each mode/behaviour decides which will be the next state so that the switching logic will be distributed among them. This solution has pros and cons. **Think and write about this in your group .** Being distributed, there is not a single place where we can tune the switching logic and improve the overall behaviour of the robot. For example, when should we switch from FORWARD to SPIRAL? Who is checking the switching condition of having a large amount of free space around?

AUTOEVALUATION with the Aspirador program

apt install libboost-dev
cd ~/robocomp/core
git pull

Clone this repo in ~/robocomp/components:

https://github.com/pbustos/beta-robotica-class.git

`Compile it and execute.`

Cambiad al lidar inferior del robot, el que tiene debajo de la bandeja

`iniciar el lidar3d con: bin/Lidar3D etc/config_pearl_webots`
cambiar el puerto en chocachoca a: Lidar3DProxy = lidar3d:tcp -h localhost -p 11989

New parallel filter to remove unwanted points in the lidar cloud

```
// Filter isolated points: keep only points with at least one neighbour within distance d (mm)

#ifdef emit
#undef emit
#endif
#include <algorithm>
#include <execution>

RoboCompLidar3D::TPoints SpecificWorker::filter_isolated_points(const RoboCompLidar3D::TPoints &points, float d) // set to 200mm
{
    if (points.empty()) return {};

    const float d_squared = d * d;  // Avoid sqrt by comparing squared distances
    std::vector<bool> hasNeighbor(points.size(), false);

    // Create index vector for parallel iteration
    std::vector<size_t> indices(points.size());
    std::iota(indices.begin(), indices.end(), size_t{0});

    // Parallelize outer loop - each thread checks one point
    std::for_each(std::execution::par, indices.begin(), indices.end(), [&](size_t i)
        {
            const auto& p1 = points[i];
            // Sequential inner loop (avoid nested parallelism)
            for (auto &&[j,p2] : iter::enumerate(points))
                {
                    if (i == j) continue;
                    const float dx = p1.x - p2.x;
                    const float dy = p1.y - p2.y;
                    if (dx * dx + dy * dy <= d_squared)
                    {
                        hasNeighbor[i] = true;
                        break;
                    }
                }
        });

    // Collect results
    std::vector<RoboCompLidar3D::TPoint> result;
    result.reserve(points.size());
    for (auto &&[i, p] : iter::enumerate(points))
```

```
        if (hasNeighbor[i])
            result.push_back(points[i]);
    return result;
}
```

Nota: add
#include<cppitertools/enumerate.hpp>
add this library to the line src/CMakeLists.txt

```
SET (LIBS ${LIBS}  tbb)
```

draw_lidar method that draws lines from the robot to the closest frontal point and the closest lateral point. Helpful for debugging. Replace *params.LIDAR_OFFSET* and similar with your own choices.

```
#include <expected>
```

en initialize(), debajo de viewer = …. añadir

```
auto [r, e] = viewer->add_robot(params.ROBOT_WIDTH,
params.ROBOT_LENGTH, 0, 100, QColor("Blue"));
robot_polygon = r;   // declarar en .h
```

```
void SpecificWorker::draw_lidar(auto &filtered_points, QQGraphicsScene *scene)
{
    static std::vector<QGraphicsItem*> items;   // store items so they can be shown between iterations

    // remove all items drawn in the previous iteration
    for(auto i: items)
    {
        scene->removeItem(i);
        delete i;
    }
    items.clear();

    auto color = QColor(Qt::green);
    auto brush = QBrush(QColor(Qt::green));
    for(const auto &p : filtered_points)
    {
        auto item = scene->addRect(-50, -50, 100, 100, color, brush);
```

```
        item->setPos(p.x, p.y);
        items.push_back(item);
    }

    // compute and draw minimum distance point in frontal range
                    auto offset_begin = closest_lidar_index_to_given_angle(filtered_points,
-params.LIDAR_FRONT_SECTION);
                     auto offset_end = closest_lidar_index_to_given_angle(filtered_points,
params.LIDAR_FRONT_SECTION);
    if(not offset_begin or not offset_end)
      { std::cout << offset_begin.error() << " " << offset_end.error() << std::endl; return ;}     //
abandon the ship
         auto min_point = std::min_element(std::begin(filtered_points) + offset_begin.value(),
std::begin(filtered_points) + offset_end.value(), [](auto &a, auto &b)
    { return a.distance2d < b.distance2d; });
    QColor dcolor;
    if(min_point->distance2d < params.STOP_THRESHOLD)
        dcolor = QColor(Qt::red);
    else
        dcolor = QColor(Qt::magenta);
    auto ditem = scene->addRect(-100, -100, 200, 200, dcolor, QBrush(dcolor));
    ditem->setPos(min_point->x, min_point->y);
    items.push_back(ditem);

    // compute and draw minimum distance point to wall
                  auto wall_res_right = closest_lidar_index_to_given_angle(filtered_points,
params.LIDAR_RIGHT_SIDE_SECTION);
                    auto wall_res_left = closest_lidar_index_to_given_angle(filtered_points,
params.LIDAR_LEFT_SIDE_SECTION);
    if(not wall_res_right or not wall_res_left)   // abandon the ship
    {
         qWarning() << "No valid lateral readings" << QString::fromStdString(wall_res_right.error())
<< QString::fromStdString(wall_res_left.error());
        return;
    }
    auto right_point = filtered_points[wall_res_right.value()];
    auto left_point = filtered_points[wall_res_left.value()];
    // compare both to get the one with minimum distance
    auto min_obj = (right_point.distance2d < left_point.distance2d) ? right_point : left_point;
      auto item = scene->addRect(-100, -100, 200, 200, QColor(QColorConstants::Svg::orange),
QBrush(QColor(QColorConstants::Svg::orange)));
    item->setPos(min_obj.x, min_obj.y);
    items.push_back(item);
    // draw a line from the robot to the minimum distance point
```

```
    auto item_line = scene->addLine(QLineF(QPointF(0.f, 0.f), QPointF(min_obj.x, min_obj.y)),
QPen(QColorConstants::Svg::orange, 10));
   items.push_back(item_line);

   // Draw two lines coming out from the robot at angles given by params.LIDAR_OFFSET
   // Calculate the end points of the lines
auto res_right = closest_lidar_index_to_given_angle(filtered_points,
params.LIDAR_FRONT_SECTION);
auto res_left = closest_lidar_index_to_given_angle(filtered_points,
-params.LIDAR_FRONT_SECTION);
   if(not res_right or not res_left)
   { std::cout << res_right.error() << " " << res_left.error() << std::endl; return ;}
   // draw two lines at the edges of the range
   float right_line_length = filtered_points[res_right.value()].distance2d;
   float left_line_length = filtered_points[res_left.value()].distance2d;
   float angle1 = filtered_points[res_left.value()].phi;
   float angle2 = filtered_points[res_right.value()].phi;
   QLineF line_left{QPointF(0.f, 0.f),
                    robot_dpolygon->mapToScene(left_line_length * sin(angle1), left_line_length *
cos(angle1))};
   QLineF line_right{QPointF(0.f, 0.f),
                     robot_polygon->mapToScene(right_line_length * sin(angle2), right_line_length *
cos(angle2))};
   QPen left_pen(Qt::blue, 10); // Blue color pen with thickness 3
   QPen right_pen(Qt::red, 10); // Blue color pen with thickness 3
   auto line1 = scene->addLine(line_left, left_pen);
   auto line2 = scene->addLine(line_right, right_pen);
   items.push_back(line1);
   items.push_back(line2);
}
```

Auxiliary method to compute the index of the closest lidar point to a given angle

```
/**
 * @brief Calculates the index of the closest lidar point to the given angle.
 *
 * This method searches through the provided std::list of lidar points and finds the point
 * whose angle (phi value) is closest to the specified angle. If a matching point is found,
 * the index of the point in the std::list is returned. If no point is found that matches the condition,
 * an error message is returned.
 *
 * @param points The collection of lidar points to search through.
 * @param angle The target angle to find the closest matching point.
 * @return std::expected<int, std::string> containing the index of the closest lidar point if found,
```

```
 * or an error message if no such point exists.
 */
std::expected<int, std::string> SpecificWorker::closest_lidar_index_to_given_angle(const auto &points, float angle)
{
    // search for the point in points whose phi value is closest to angle
    auto res = std::ranges::find_if(points, [angle](auto &a){ return a.phi > angle;});
    if(res != std::end(points))
        return std::distance(std::begin(points), res);
    else
                                return std::unexpected("No closest value found in method <closest_lidar_index_to_given_angle>");
}
```

¡Recordad que sólo podéis incluir estas mejoras si entendéis perfectamente lo que hacen! Se os puede preguntar por cualquier parte del código que presentéis.

Mejora del UI

Podéis añadir un panel lateral al UI para mostrar valores de distancia, ajustar umbrales, etc. Esto sería un ejemplo de mainUI.ui modificado. Podéis abrirlo en el *designer* una vez guardado en un fichero.

robEjemplo de compute()

```
void SpecificWorker::compute()
{
    RoboCompLidar3D::TData ldata;
    try{ ldata =  lidar3d_proxy->getLidarData("bpearl", 0, 2*M_PI, 1);}
    catch(const Ice::Exception &e){std::cout << e << std::endl;}

   // el filtro que hicimos en clase o este más simple que no acumula por altura
    RoboCompLidar3D::TPoints  p_filter;
    std::ranges::copy_if(ldata.points, std::back_inserter(p_filter),
                                    [](auto  &a){ return a.z < 500 and a.distance2d > 200;});

    // opcional para quitar puntos liDAR aislados del resto
    p_filter = filter_isolated_points(p_filter, 200);

    draw_lidar(p_filter, &viewer->scene);

    //Add State machine with your sweeping logic
    RetVal ret_val;
```

```
    switch(state)
    {
        case STATE::FORWARD:
        {
            ret_val = forward(p_filter);
            label_state->setText("FORWARD");
            break;
        }
        case STATE::TURN:
        {
            ret_val = turn(p_filter);
            label_state->setText("TURN");
            break;
        }
        case STATE::WALL:
        {
            ret_val = wall(p_filter);
            label_state->setText("FOLLOW WALL");
            break;
        }
    }
    /// unpack  the std::tuple
    auto [st, adv, rot] = ret_val;
    state = st;

    /// Send movements commands to the robot
    try{ omnirobot_proxy->setSpeedBase(0, adv, rot);}
    catch(const Ice::Exception &e){std::cout << e << std::endl;}
}
```

Ejemplo de un estado (Forward)

```
SpecificWorker::RetVal SpecificWorker::forward(auto &points)
{
   // Enter conditions
   if(points.empty()) {qWarning() << “Empty container at forward(); return;}

    // Exit condition
                        auto offset_begin = closest_lidar_index_to_given_angle(points,
-params.LIDAR_FRONT_SECTION);
                         auto offset_end = closest_lidar_index_to_given_angle(points,
params.LIDAR_FRONT_SECTION);
```

```
 auto min_point = std::min_element(std::begin(points) + offset_begin.value(), std::begin(points)
+  offset_end.value(), [](auto &a, auto &b)   { return a.distance2d < b.distance2d; });
   if (min_point != points.end() and min_point->r < params.STOP_THRESHOLD)
        return RetVal(STATE::TURN, 0.f, 0.f);  // stop and change state if obstacle detected

  // Forward logic
      return RetVal(STATE::FORWARD, params.MAX_ADV_SPEED, 0.f);
}

////////////////////////////////////////////////////////////
```

- Check first if the parameters are OK
- The method's exit condition always comes next.
- Finally, the method's specific logic

Recommendations for the paper

- Add a Section between the Introduction and Activity 1: the sweeping robot, called "Materials and methods" where you describe component-oriented programming and the working environment: Linux, C++23, CMake, RoboComp, CLion, Github, Overleaf and the computers in Beta. Then, start with the description of the problem approached in Activity 1.
- If there is any mathematics in the solution, express it in the paper. The equations for the brakes must be written in mathematical form: y = mx+n. They must also include a drawing showing how the line equation and the std::clamp hard limits combine to form an S-shaped or Z-shaped function.
- The variables in equations are letters or, at most, letters with sub or upper indices, but under no circumstances can a complete word be used as a variable.
- Include the Github URL in the paper below the names (at the spot for the address and affiliation)
- Include at the end (after the Bibliography section) a last section with the authors' information: The names and a **picture** of each one.

TASK 2 Robot self-localisation in the room

We will implement the solution explained in this report

To complete the activity, we have to follow these steps

1. Create a new folder for Activity 2
2. Make sure Activity 1 folder is clean of all cache and IDE temp files
3. Copy recursively Activity 1 contents to the Activity 2 folder
4. cd .. ; git add *activity_2* (just once)
5. git commit activity_2 (add proper comment)
6. Now we have our chocachoca copied to a new folder
7. You can optionally change the name to something like **localiser** by modifying it in chocachocha.cdsl, and all CMakeList.txt files, and regenerating with robocompdsl
8. Make sure the statemachine is out of the compute() method.
9. Comment the state machine call and the code to set new speeds until the localisation is working.
10. In compute(), we leave only the read_data() method.
11. Replace your mainUi.ui file with this one UI, to add a second frame where we will show the room layout and the robot moving inside. Add #define USE_QTGUI to the begining of this file, generated/genericwork.h. Do cmake and make.
12. Update with git pull the repository beta-robotica-class. Find a new folder localiser and copy all the .cpp and .h files there to the src folder of your new component. Add them to git.
13. There is a common_types.h file with the data type we will be using. Include it in specificworker.h. Study the **using** C++ new keyword.
14. To integrate a .cpp/.h class in your project,
    a. add the .h to specificworker.h
    b. add the .cpp file to the src/CMaleList.txt along with the other existing ones.
    c. if a class includes other classes, as in room_detector, repeat this process for that class.cd
    d. declare the variables to instantiate the object. See the next point.
15. You also need to add the files included in room_detector: ransac_line_detector.h/cpp

16. Add, INCLUDE($ENV{ROBOCOMP}/cmake/modules/opencv4.cmake), to the src/CMakeLists.txt to tell the compiler where to find OPenCV
17. Add the following private variables in specificworker.h. These objects provide the algorithms needed to extract corners and match them.
    a. AbstractGraphicViewer *viewer_room; // new frame to show the room
    b. Eigen::Affine2d robot_pose; // Eigen type to represent a rotation+translation
    c. rc::Room_Detector room_detector; // object to compute the corners
    d. rc::Hungarian hungarian; // object to match the two sets of corners
    e. QGraphicsPolygonItem *room_draw_robot; // to draw the robot inside the room
18. Create a struct in specificworker.h to represent your nominal room. Something like this:

```
struct NominalRoom
{
  float width; //  mm
  float length;
  Corners corners;
    explicit NominalRoom(const float width_=10000.f, const float length_=5000.f, Corners  corners_ = {}) : width(width_), length(length_), corners(std::move(corners_)){};
     Corners transform_corners_to(const Eigen::Affine2d &transform) const  // for room to robot pass the inverse of robot_pose
  {
     Corners transformed_corners;
     for(const auto &[p, _, __] : corners)
     {
        auto ep = Eigen::Vector2d{p.x(), p.y()};
        Eigen::Vector2d tp = transform * ep;
        transformed_corners.emplace_back(QPointF{static_cast<float>(tp.x()), static_cast<float>(tp.y())}, 0.f, 0.f);
     }
     return transformed_corners;
  }
};
NominalRoom room{10000.f, 5000.f,
        {{QPointF{-5000.f, -2500.f}, 0.f, 0.f},
            {QPointF{5000.f, -2500.f}, 0.f, 0.f},
            {QPointF{5000.f, 2500.f}, 0.f, 0.f},
            {QPointF{-5000.f, 2500.f}, 0.f, 0.f}}};
```

NOTE: Since the variables in this struct are initialised in three places —default values in the parameter declaration, the member initialiser list, and the variable declaration —study and explain in the paper the initialisation order followed by the compiler.

19. Study the geometry of a change of coordinate frame: 2D transform. Do the transformation on paper to make sure you understand the process.
20. Inside the initialize() method, initialise the variables: (replace params with your struct for constants)

```
viewer_room = new AbstractGraphicViewer(this->frame_room, params.GRID_MAX_DIM);
auto [rr, re] = viewer_room->add_robot(params.ROBOT_WIDTH, params.ROBOT_LENGTH, 0, 100, QColor("Blue"));S
```

```
robot_room_draw = rr;

// draw room in viewer_room
viewer_room->scene.addRect(params.GRID_MAX_DIM, QPen(Qt::black, 30));
viewer_room->show();

// initialise robot pose
 robot_pose.setIdentity();
 robot_pose.translate(Eigen::Vector2d(0.0,0.0));
```

21. Inside the compute() do:
    a. Call the method compute_corners in room_detector to extract corners from the LiDAR point cloud. Examine the room_detector.h file to select a compute_corners() method that suits your data types. Study the code to understand the steps: line extraction with RANSAC, compare two by two the extracted lines to check if they meet at 90º approx, and create the corners. You'll need to document these techniques in the paper.
    b. Call a method to match the measured corners with the nominal corners. Examine the Hungarian class to set the call. You need to study the hungarian method to know what it does and to explain it in the paper.
    c. Before calling the hungarian, both sets of corners must be in the same frame. To transform from the robot to the room, use Y = Rx + T, where R is the current robot rotation matrix and T the current robot translation, both wrt the room frame. To transform from the room to the robot use $x = R^T(y-T)$. Check 2D transform and make some drawings on paper to understand it. The method in the NominalRoom does exactly this and the direction of the transform depends on whether you pass the direct robot_pose or the inverted one, robot_pose.inverse(). Remember the rule: from robot to room -> direct; from room to robot -> inverse.
    d. Set up the matrices W and b as defined in the report

```
Eigen::MatrixXd W(corners.size() * 2, 3);
```

```
i.   Eigen::VectorXd b(corners.size() * 2);
ii.  for (auto &&[i, m]: match | iter::enumerate)
iii. {
iv.     auto &[meas_c, nom_c, _] = m;
v.      auto &[p_meas, __, ___] = meas_c;
vi.     auto &[p_nom, ____, _____] = nom_c;
vii.    b(2 * i)     = p_nom.x() - p_meas.x();
viii.   b(2 * i + 1) = p_nom.y() - p_meas.y();
```

```
ix.       W.block<1, 3>(2 * i, 0)     << 1.0, 0.0, -p_meas.y();
x.        W.block<1, 3>(2 * i + 1, 0) << 0.0, 1.0, p_meas.x();
xi.    }
xii.   // estimate new pose with pseudoinverse
xiii.  const  Eigen::Vector3d  r  =  (W.transpose()  *  W).inverse()  *
       W.transpose() * b;
xiv.   std::cout << r << std::endl;
xv.    qInfo() << "--------------------";
xvi.
xvii.  if (r.array().isNaN().any())
xviii.    return;
xix.
```

e. Solve the linear equation using the pseudoinverse to obtain a 3D vector r[x, y, phi] with the new pose of the robot.

f. Update the Eigen object, robot_pose, to the new values:

```
robot_pose.translate(Eigen::Vector2d(r(0), r(1)));
robot_pose.rotate(r[2]);
```

g. Update the position of the robot by calling the setPos(x,y) method on the **robot_draw** object.

```
robot_room_draw->setPos(robot_pose.translation().x(), robot_pose.translation().y());
double angle = std::atan2(robot_pose.rotation()(1, 0), robot_pose.rotation()(0, 0));
robot_room_draw->setRotation(qRadiansToDegrees(angle));
```

Estimated time for completion: 2 weeks

NEW TOOL!

To start your components, you can use this Python script, subcognitive.py w
ith this configuration example file (modify to your paths) sub.toml

TIPS FOR THE PAPER

- Fix all comments in the reviewed paper.
- Extend the Introduction with types of robots (mobile, aerial, legged, aquatic), and include a representative example for each, with an image. Make this part personal.
- Add the Activity II section. Describe the goal of the activity
- Describe the proposed solution and how it will be measured
- Describe the techniques used with mathematical language
- Describe an experiment with a capture of the UI. No mobile phones allowed.

- Record a video of the robot moving around using either the joystick or your state machine.
- Add the URL of the video to the paper.

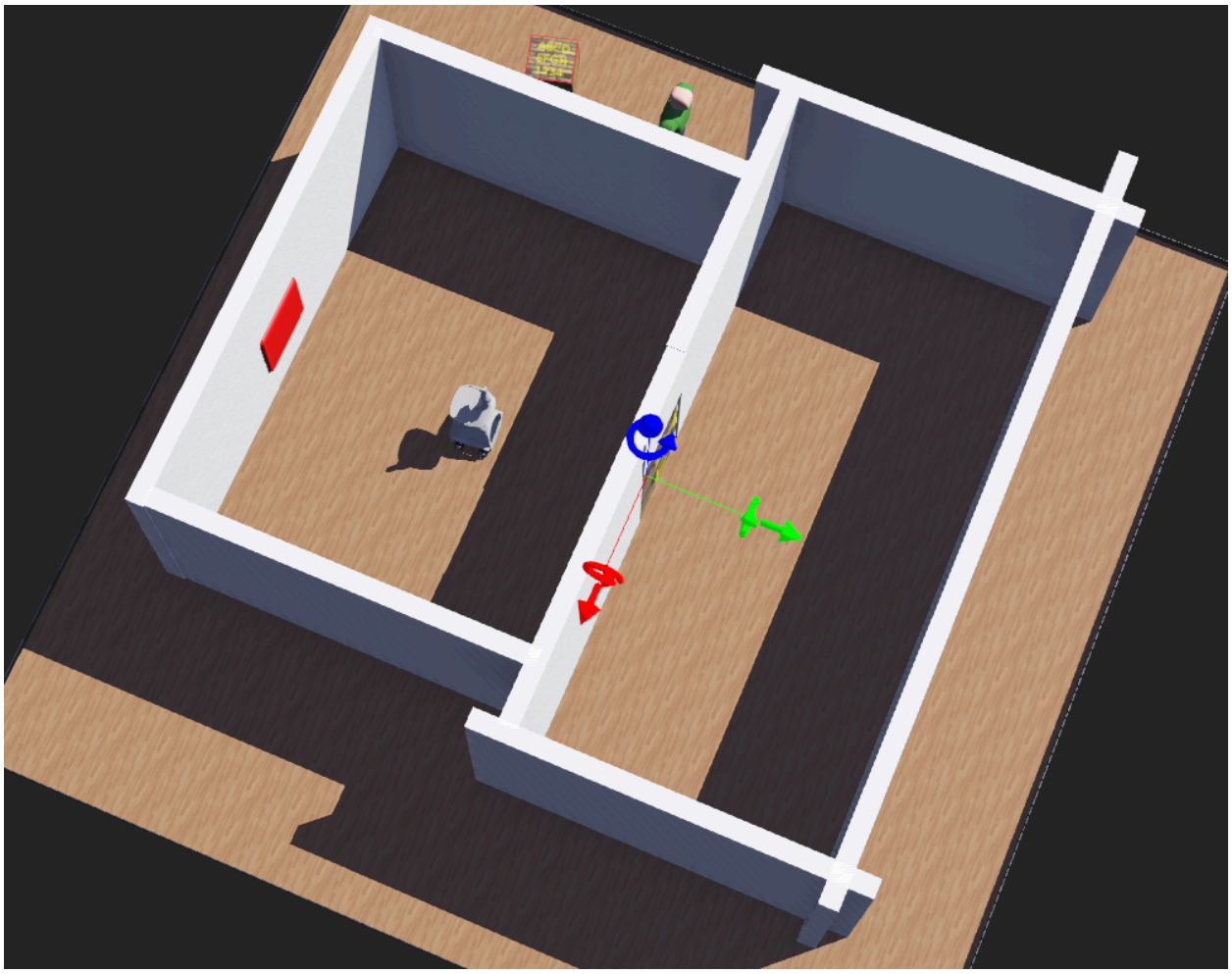
TASK 3. MULTI-ROOM NAVIGATION

In this task the robot will detect a door, cross it and restart the localization process in the new room.

Initial problems to solve:

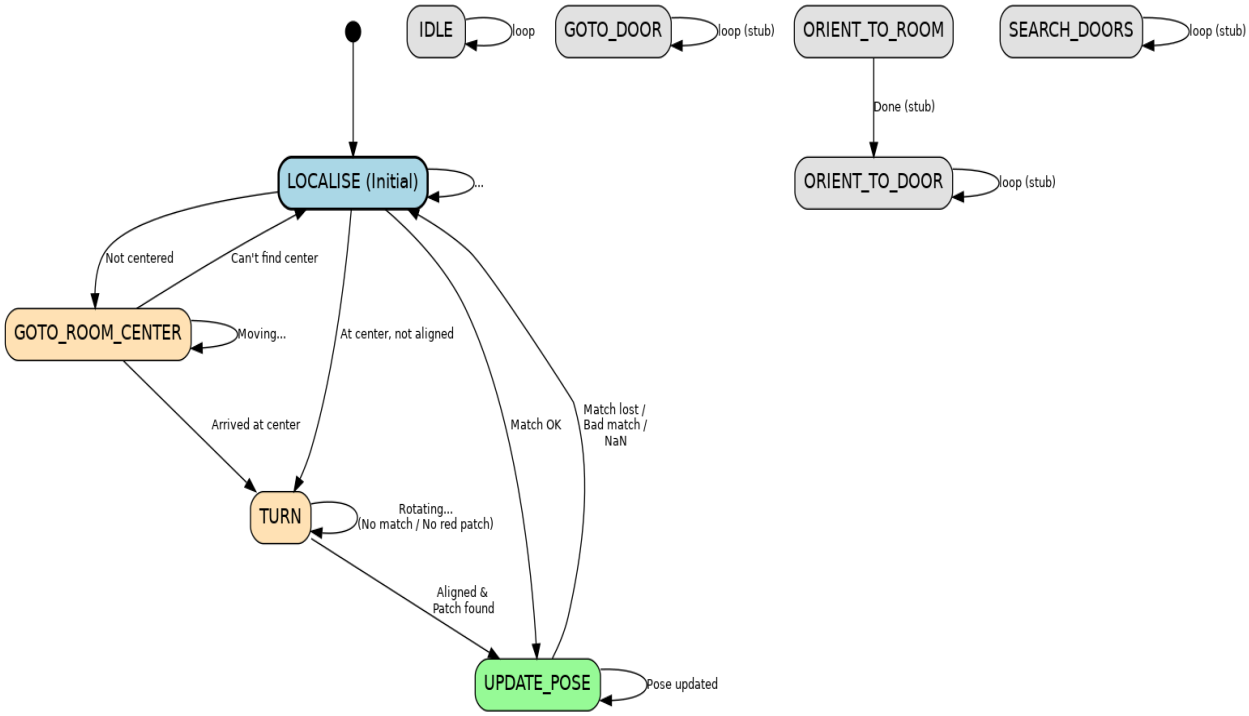
1. Relocalization when the robot gets lost
2. Localization of a (closed) door
3. Approach the door to a 1 meter ahead of it
4. Wait until the door is open (now manually but a command will be available in webots-bridge)
5. Cross the door blindly
6. Wait for the door to close
7. Restart the localization process in the new room

New two-room world



You need to pull on the webots-shadow repository and load the TwoRoomsSimple scenario.

Propuesta inicial de máquina de estados



The states in gray are not connected yet. For the first points of the Task, 1 and 2, we will use the coloured states. The key idea here is to move the robot to the center of the room when it is lost. Then, turn it around until it finds a red colored patch on one of the walls. Then stop. That is the correct orientation.

PHASE I re-localization

- Refresh the beta-robotica-class repo and copy the files to your src directory.
- Note that there is a new UI file is there. This new UI will replace the old one.
- Remember to add `#define USE_QTGUI` to generated/genericworker.h
- You can use this specificworker.h file as a guide to organise your new component.
- This is an initialize() method that shows how to initialise the new elements:

```
void SpecificWorker::initialize()
{
   std::cout << "Initialize worker" << std::endl;
   if(this->startup_check_flag)
   {
      this->startup_check();
   }
```

```
   else
   {
       ////////////// Your code ////////
       // Viewer
       viewer = new AbstractGraphicViewer(this->frame, params.GRID_MAX_DIM);
        auto [r, e] = viewer->add_robot(params.ROBOT_WIDTH, params.ROBOT_LENGTH,
0, 100, QColor("Blue"));
       robot_draw = r;
       //viewer->show();

                  viewer_room   =   new   AbstractGraphicViewer(this->frame_room,
params.GRID_MAX_DIM);
                auto   [rr,   re]   =   viewer_room->add_robot(params.ROBOT_WIDTH,
params.ROBOT_LENGTH, 0, 100, QColor("Blue"));
       robot_room_draw = rr;
       // draw room in viewer_room
            viewer_room->scene.addRect(nominal_rooms[0].rect(),  QPen(Qt::black,
30));
       //viewer_room->show();
       show();

       // initialise robot pose
       robot_pose.setIdentity();
       robot_pose.translate(Eigen::Vector2d(0.0,0.0));

       // time series plotter for match error
       TimeSeriesPlotter::Con fig plotConfig;
       plotConfig.title = "Maximum Match Error Over Time";
       plotConfig.yAxisLabel = "Error (mm)";
       plotConfig.timeWindowSeconds = 15.0; // Show a 15-second window
       plotConfig.autoScaleY = false;       // We will set a fixed range
       plotConfig.yMin = 0;
       plotConfig.yMax = 1000;
                                                    time_series_plotter       =
std::make_unique<TimeSeriesPlotter>(frame_plot_error, plotConfig);
       match_error_graph = time_series_plotter->addGraph("", Qt::blue);

       // stop robot
       move_robot(0, 0, 0);
   }
}
```

- Add Qt6PrintSupport to src/CMakeLists.txt:

```
SET(LIBS ${LIBS} tbb Qt6PrintSupport)
```

- Add to src/CMakeLists.txt to link with OpenCV libraries needed for ImageProcessor
- `INCLUDE($ENV{ROBOCOMP}/cmake/modules/opencv4.cmake)`
- The door-based filter, as suggested by one of the groups, works like charm. The DoorDetector class misses the detect method, which **has to be coded by you**. The filter_points() method MUST be used just after your other filters to remove the points outside the room. This method calls detect() which has to be completed.
- The detect() method has four parts:
    a. use iter::sliding_window(2) to traverse the points vector in pairs: (1,2) (2,3) (3,4) … and subtract the distance2d field from each pair. If the difference is greater than 1000 add to the output Peaks container the shortest one.
    b. draw the peaks using the same code as in draw_lidar() to check that they are being computed ok
    c. Filter the peaks by removing one of those pairs that are very close. This is called Non-maximum-suppresion filter

```
// non-maximum suppression of peaks: remove peaks closer than 500mm
Peaks nmmas_peaks;
for (const auto &[p, a] : peaks)
                if     (const     bool     too_close     =
std::ranges::any_of(nms_peaks,   [&p](const   auto   &p2)   {
return   (p   -   std::get<0>(p2)).norm()   <   500.f;   });   not
too_close)
       nms_peaks.emplace_back(p, a);
peaks = nms_peaks;
```

    d. Go through all pairs of peaks using iter::combinations(2). Check if the two points are within a distance of 1200 and 800 mm. Add the pair to the output Doors container.

Once the filtered points show that the points outside the room have been removed, move to the next step.

- Add the corners and match computation to the compute() method. Add a method to compute the maximum error of the four matches.
- Call now the state-machine method with just one state: GOTO_ROOM_CENTER.
- In goto_room_center, compute the center of the room using the corresponding method in room_detector
- Code a robot_controller method to drive the robot towards a target point (the center of the room). The maths for the controller are HERE. Study the report and code a method to be called with a target parameter and that will return the advance and rotation speeds of the robot.

- Add a call to an update_pose() method in compute(), including all the math of the previous activity.

TIPS

- Do **not** use a UPDATE_POSE state or similar to compute the equations of Activity 2. Put the code in a separate method in specificworker.cpp and call it after the matching as shown in this compute() example:

```
void SpecificWorker::compute()
{
   RoboCompLidar3D::TPoints data = read_data();
   data= door_detector.filter_points(data, &viewer->scene);

   // compute corners
     const auto &[corners, lines] = room_detector.compute_corners(data,
&viewer->scene);
                            const         auto         center_opt         =
room_detector.estimate_center_from_walls(lines);
   draw_lidar(data, center_opt, &viewer->scene);
   // match corners  transforming first nominal corners to robot's frame
   const auto match = hungarian.match(corners,

nominal_rooms[0].transform_corners_to(robot_pose.inverse()));

   // compute max of  match error
   float max_match_error = 99999.f;
   if (not match.empty())
   {
            const auto max_error_iter = std::ranges::max_element(match,
[](const auto &a, const auto &b)
          { return std::get<2>(a) < std::get<2>(b); });
                                                  max_match_error       =
static_cast<float>(std::get<2>(*max_error_iter));
       time_series_plotter->addDataPoint(0, max_match_error);
       //print_match(match, max_match_error); //debugging
   }

   // update robot pose
   if (localised)
       update_robot_pose(corners, match);

   // Process state machine
   RetVal ret_val = process_state(data, corners, match, viewer);
   auto [st, adv, rot] = ret_val;
   state = st;

   // Send movements commands to the robot constrained by the match_error
   //qInfo() << __FUNCTION__ << "Adv: " << adv << " Rot: " << rot;
```

```
    move_robot(adv, rot, max_match_error);

    // draw robot in viewer
                        robot_room_draw->setPos(robot_pose.translation().x(),
robot_pose.translation().y());
                                const          double          angle          =
qRadiansToDegrees(std::atan2(robot_pose.rotation()(1,                    0),
robot_pose.rotation()(0, 0)));
    robot_room_draw->setRotation(angle);

    // update GUI
    time_series_plotter->update();
    lcdNumber_adv->display(adv);
    lcdNumber_rot->display(rot);
    lcdNumber_x->display(robot_pose.translation().x());
    lcdNumber_y->display(robot_pose.translation().y());
    lcdNumber_angle->display(angle);
    last_time = std::chrono::high_resolution_clock::now();;
}
```

- Note how we use a global boolean variable, *localised*, to signal if the robot is localised. Only update the robot's pose if it is localised. Change the value of *localised* in the statemachine when the maximum matching error is too high or the number of matches is fewer than 3.

NEXT STEPS

- Add the following states after GOTO_ROOM_CENTER
    - TURN
    - GOTO_DOOR
    - ORIENT_TO_DOOR
    - CROSS_DOOR

  and from here to GOTO_ROOM_CENTER again to relocate in new room.

- Remember the pattern for each new state:
    - first the exit conditions and then what the method has to do

- IN GOTO_DOOR you can use the before_centre() method coded in the Door type (common_types.h). Send the robot to that point using the robot_controler method. You will have to explain graphically how that point close to the door is computed in the paper.
- In ORIENT_TO_DOOR you need to compute the angle between two vectors: the vector going from the robot to the door's middle point and the vector that joins the two door points. Turn the robot until that angle is close to pi/2

- g

NOTE: You can change the joystickpublish component by this one in the sub.toml file:

```
[[components]]
name = "joy"
cwd                                                                    =
"~/robocomp/components/robocomp-robolab/components/hardware/external_
control/python_xbox_controller"
cmd = "bin/python_xbox_controller etc/config_shadow"
ice_name = "python_xbox_controller:tcp -h localhost -p 11230"
```

TASK IV: Full-apartment discovery and navigation

In the final task the robot will discover all rooms in this apartment creating an “actionable” graph of the rooms and connecting doors. The graph can be used to navigate from one room to another one.

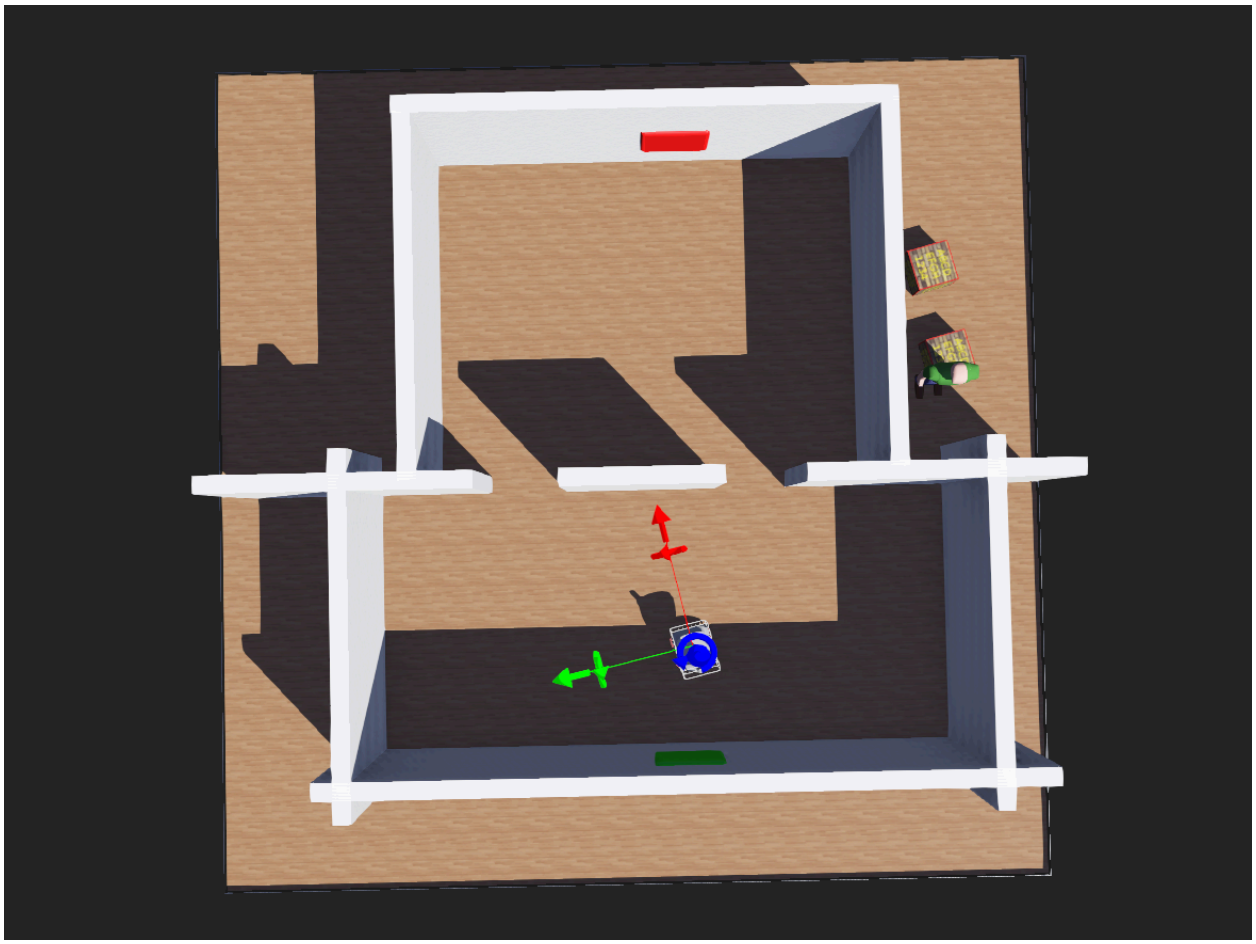
Apartment for final activity

Phase 1: The two doors problem

We will start by solving the the two-rooms, two-doors problem shown below. It is a simplified version of the full apartment version. The robot has to detect and cross both doors selecting randomly one of them first, and the one not recently used, after.

The steps to complete this phase are:

1. Modify the door filter by inverting the loops: first the points, then the doors. This way the filtered set will not include duplicated points.



Two rooms, two doors problem

2. Once the robot is localised in the red room, draw the doors on the room-frame (on the right frame in the UI). Only do this once everytime the robot is re-localised (at the end of TURN). You need to:
    a. Make the doors vector an attribute of the room class, so each room holds its vector of detected rooms
    b. in common_types.h add the following types:
        using Wall = std::tuple<Eigen::ParametrizedLine<float, 2>, int, Corner, Corner>;
        using Walls = std::vector<Wall>;

    c. in NominalRoom, add the following methods:
        i. **Walls get_walls()**; Go through all nominal corners and build a vector of lines using Eigen::ParametrizedLine<float,2>::Through(p1, p2) . Add then as Wall tuples to the vector.

ii. **Wall get_closest_wall_to_point(const Eigen::Vector2f &p)** Call get_walls(). Use std::ranges::min_element(walls, [p](w1, w2){ return w1.distance(p1) < w2.distance(p2)}) to select the wall that is closest to p.
iii. **Eigen::Vector2f get_projection_of_point_on_closest_wall(const Eigen::Vector2f &p)** Call get_closest_wall_to_point(p) and obtain w. Get projected point as auto pp = w.project(p)

d. In the TURN method, just before returning to GOTO_DOOR, get a COPY of the doors vector defined in specificworker.h. For each door, transform its d1 and d2 coordinates (robot frame) to the room's frame using the direct method. Then call the get_projection_of_point_on_closest_wall(**robot_pose * dx**) method twice.
e. create two new entries in the Door type, p1_global and p2_global of Eigen::Vector2f type. Store the values there.
f. Save the updated **doors** vector IN the nominal_rooms[i] room element. Add the attribute to the class. This version of doors is created only once and stored in the robot's memory. They will be there even if the door_detector fails to detect a door.
g. Draw the door(s) in the right frame

3. Choose a room and set it as current-room
4. Choose a door from the **doors** vector stored in nominal_rooms[current_room]
5. In GOTO_DOOR, get the center_before() points from the selected room. It is in room's coordinates. Transform it to **robot coordinates** using the inverse transform and send it to robot_controller() to make the robot approach the door.
6. Cross from the red room to the green room through the current-room
7. Once in the green room, track the just crossed door until the robot is at the center and aligned with the green patch.
8. Complete the second room coordinates of the tracked door with the new values.
9. Now select the non-used door and repeat the process.

Once the robot can smoothly navigate between doors,

1. Create in a separate file a Graph class with a reasonable API. Use std::map if possible and DO NOT use pointers. Define two types of nodes: rooms and doors. The graph is undirected.
2. Add the necessary calls to the graph from the state machine to insert a new node when a new room is detected or a new door is detected. Connect the nodes properly.
3. Add a new frame in the UI to draw the graph in real-time

Annex 1: State-machine changes to track visited doors and rooms

This code is not intended to be copied directly into your program. If you do that, you'll end up with many errors difficult to find. Study it and make the necessary changes to your running code.

Preliminaries

- specificworker.h

```
#include "door_crossing_tracker.h"

int current_room = -1;
int current_door = -1;
// relocalization for current room
bool localised = false;

// new door crossing detector
DoorCrossing door_crossing; // used the file en beta-robotica-class

// updated signature
std::optional<std::pair<Eigen::Affine2f, loat>> update_robot_pose(int room_index,
                                    const Corners &corners,
                                    const Eigen::Affine2f &r_pose,
                                    bool transform_corners);
```

- In Door.h add the following attributes:

```
bool visited = false;
int connects_to_room = -1; // index of the room this door connects to
int connects_to_door = -1; // index of the door in the connected room
```

- In NominalRoom add

```
bool visited = false;
```

- In specificworker.cpp: update_robot_pose()
  Now, update_robot_pose() computed the match between the measured corners and the nominal corners transformed to the robot's frame

```
std::optional<std::pair<Eigen::Affine2f, float>>
SpecificWorker::update_robot_pose(int room_index,
```

```
const Corners &corners,

const Eigen::Affine2f &r_pose,

bool transform_corners)
{
    // match corners   transforming first nominal corners to
robot's frame
   Match match;
   if (transform_corners)
                          match    =    hungarian.match(corners,
nominal_rooms[room_index].transform_corners_to(r_pose.inverse()
));
   else
                          match    =    hungarian.match(corners,
nominal_rooms[room_index].corners());

   if (match.empty() or match.size() < 3)
       return {};

    const auto max_error_iter = std::ranges::max_element(match,
[](const auto &a, const auto &b)
     { return std::get<2>(a) < std::get<2>(b); });

   const auto max_match_error = std::get<2>(*max_error_iter);

   // create matrices W and b for pose estimation
   auto r = solve_pose(corners, match);
   if (r.array().isNaN().any())
   {
       qWarning() << __FUNCTION__ << "NaN values in r ";
       return {};
   }

   auto r_pose_copy = r_pose;
   r_pose_copy.translate(Eigen::Vector2f(r(0), r(1)));
   r_pose_copy.rotate(r[2]);
   return {{r_pose_copy, max_match_error}};
}
```

- compute()

Now, update_robot_pose() computed the match between the measured corners and the nominal corners transformed to the robot's frame

```
void SpecificWorker::compute()
{
    RoboCompLidar3D::TPoints data = read_data();
    data = door_detector.filter_points(data);
    draw_local_doors(&viewer->scene);

    // compute corners and center
                const      auto      &[corners,      lines]      =
room_detector.compute_corners(data, &viewer->scene);
    const auto center_opt = center_estimator.estimate(data);
    draw_lidar(data, center_opt, &viewer->scene);

    // update robot pose
    float max_match_error = -1;
    //Match match;
    if (localised)
    {
            if (const auto res = update_robot_pose(current_room,
corners, robot_pose, true); res.has_value())
        {
            robot_pose = res.value().first;
            max_match_error = res.value().second;

time_series_plotter->addDataPoint(match_error_graph,max_match_e
rror);
        }
    }

    // Process state machine
            RetVal    ret_val    =    process_state(data,    corners,
&viewer->scene, &viewer_room->scene);
    auto [st, adv, rot] = ret_val;
    state = st;

     // Send movements commands to the robot constrained by the
match_error
    if (not pushButton_stop->isChecked())
        move_robot(adv, rot, max_match_error);

```

```
    draw_nominal_room(current_room, &viewer_room->scene);
    //print_room_data();
                draw_robot_in_viewer(localised,      robot_pose,
robot_room_draw);
    update_gui(adv, rot);

    last_time = std::chrono::high_resolution_clock::now();;
}
```

- Add the new class in door_crossing_tracker. It has two methods. set_entering_data() to initiate the tracking when the robot enters the new room at CROSS_DOOR, and track_entering_door() to be called during the GOTO_TO_CENTER state (TURN won't change the distance). When the robot is located again, you can use its attributes to update the doors and room connection.

- LOCALISE()

  localised is set to false

```
SpecificWorker::RetVal            SpecificWorker::localise(const
RoboCompLidar3D::TPoints &points, QGraphicsScene *scene)
{
    // initialise robot pose at origin. Necessary to reser pose
accumulation
   robot_pose.setIdentity();
   robot_pose.translate(Eigen::Vector2f(0.0,0.0));
   localised = false;

    // if error high but not at room centre, go to centering
step
   // compute mean of LiDAR points as room center estimate

    if(const auto center = center_estimator.estimate(points);
center.has_value())
   {
       if (center.value().norm() > params.RELOCAL_CENTER_EPS )
           return{STATE::GOTO_ROOM_CENTER, 0.0f, 0.0f};

       // If close enough to center -> stop and move to TURN
       if (center.value().norm() < params.RELOCAL_CENTER_EPS )
           return {STATE::TURN, 0.0f, 0.0f};
```

```
    }
     qWarning() << __FUNCTION__ << "Not able to estimate room
center from walls, continue localising.";
    return {STATE::LOCALISE, 0.0f, 0.0f};
}
```

- GOTO_ROOM_CENTER
  add this line just before the last line: return {STATE::GOTO_DOOR_CENTER} so the door_crossing object tracks the position of the just entered door.

```
door_crossing.track_entering_door(door_detector.doors());
```

- TURN

  Use the new check_colour_patch_in_image() that returns the room index (pull beta-robotica-class)
  There are **two** blocks. The first one to save the doors to the nominal room if not previously visited. The second one checks is door_tracking is active and updates final door crossing info using its current value. It sets it to false.

```
SpecificWorker::RetVal SpecificWorker::turn(const Corners
&corners){
////////////////////////////////////////////////////////////////
///
    // check for colour patch in image

////////////////////////////////////////////////////////////////
//
const auto &[success, room_index, left_right] =
image_processor.check_colour_patch_in_image(camera360rgb_proxy,
this->label_img);
    if (success)
    {
        current_room = room_index;
        // update robot pose to have a fresh value
           if (const auto res = update_robot_pose(current_room,
corners, robot_pose, false); res.has_value())
            robot_pose = res.value().first;
                              else return{STATE::TURN, 0.0f,
left_right*params.RELOCAL_ROT_SPEED/2};

```

```
//////////////////////////////////////////////////////////////////
///////
/// save doors to nominal_room if not previously visited

//////////////////////////////////////////////////////////////////
////////////
if (not nominal_rooms[current_room].visited)
{
                              nominal_rooms[current_room].name   =
image_processor.room_name_from_index(current_room);
            auto doors = door_detector.doors();
              if (doors.empty()) { qWarning() << __FUNCTION__ <<
"empty          doors";          return{STATE::TURN,          0.0f,
left_right*params.RELOCAL_ROT_SPEED};}
            for (auto &d : doors)
            {
                                                    d.p1_global    =
nominal_rooms[current_room].get_projection_of_point_on_closest_
wall(robot_pose * d.p1);
                                                    d.p2_global    =
nominal_rooms[current_room].get_projection_of_point_on_closest_
wall(robot_pose * d.p2);
            }
            nominal_rooms[current_room].doors = doors;
            // choose door to go
            current_door = choose_next_door(current_room);
               // we need to match the current selected nominal
door to the successive local doors detected during the approach
               // select the local door closest to the selected
nominal door
                                              const    auto    dn    =
nominal_rooms[current_room].doors[current_door];
            const auto ds = door_detector.doors();
               const auto sd = std::ranges::min_element(ds, [dn,
this](const auto &a, const auto &b)
                     {  return (a.center() - robot_pose.inverse()
* dn.center_global()).norm() <
                                  (b.center() - robot_pose.inverse()
* dn.center_global()).norm(); });
               // sd is the closest local door to the selected
nominal door. Update nominal door with local values
```

```
            nominal_rooms[current_room].doors[current_door].p1 =
sd->p1;
            nominal_rooms[current_room].doors[current_door].p2 =
sd->p2;
           nominal_rooms[current_room].visited = true;
        }
                                                                    //
////////////////////////////////////////////////////////////////////
////////////
// // finish door tracking and update door crossing info
                                                                    //
////////////////////////////////////////////////////////////////////
////////////
if (door_crossing.valid)
{
                  door_crossing.set_entering_data(current_room,
nominal_rooms);

nominal_rooms[door_crossing.leaving_room_index].doors[door_cros
sing.leaving_door_index].connects_to_door                    =
door_crossing.entering_door_index;

nominal_rooms[door_crossing.leaving_room_index].doors[door_cros
sing.leaving_door_index].connects_to_room                    =
door_crossing.entering_room_index;

nominal_rooms[current_room].doors[door_crossing.entering_door_i
ndex].visited = true;

nominal_rooms[current_room].doors[door_crossing.entering_door_i
ndex].connects_to_door = door_crossing.leaving_door_index;

nominal_rooms[current_room].doors[door_crossing.entering_door_i
ndex].connects_to_room = door_crossing.leaving_room_index;
           door_crossing.valid = false;
        }
        localised = true;
        return {STATE::GOTO_DOOR, 0.0f, 0.0f};  // SUCCESS
    }
    // continue turning
                        return       {STATE::TURN,       0.0f,
left_right*params.RELOCAL_ROT_SPEED};
}
```

- GOTO_DOOR

  In this state we differentiate between localised and not localised. If localised, we use the nominal robot transformed coordinates to select the measured door closest to the nominal, target one. If not localised, we choose the measured door better aligned witht he robot's heading.

```
SpecificWorker::RetVal SpecificWorker::goto_door(const
RoboCompLidar3D::TPoints &points, QGraphicsScene *scene)
{
    Doors doors;
    // Exit conditions
    if ( doors = door_detector.doors(); doors.empty())
    {
        qInfo() << __FUNCTION__ << "No doors detected, switching to
UPDATE_POSE";
        return {STATE::GOTO_DOOR, 0.f, 0.f};  // TODO: keep moving for
a while?
    }
    // select from doors, the one closest to the nominal door
    Door target_door;
    if (localised)
    {
        //qInfo() << __FUNCTION__ << "Localised, selecting door
closest to nominal door";
        const auto dn =
nominal_rooms[current_room].doors[current_door];
        const auto sd = std::ranges::min_element(doors, [dn,
this](const auto &a, const auto &b)
                { return (a.center() - robot_pose.inverse() *
dn.center_global()).norm() <
                         (b.center() - robot_pose.inverse() *
dn.center_global()).norm(); });
        target_door = *sd;
    }
    else  // select the one closest to the robot's heading direction
    {
        //qInfo() << __FUNCTION__ << "Not localised, selecting door
closest to robot heading";
        const auto sd = std::ranges::min_element(doors, [](const auto
&a, const auto &b)
                { return abs(a.p1_angle) < abs(b.p1_angle); });
        target_door = *sd;
```

```
    }

    // distance to target is less than threshold, stop and switch to
ORIENT_TO_DOOR
    const auto target =
target_door.center_before(robot_pose.translation(),
params.RELOCAL_MIN_DISTANCE_TO_DOOR);
    const auto dist_to_door = target.norm();

    // draw target
    static QGraphicsItem *door_target_draw = nullptr;
    if (door_target_draw != nullptr)
        scene->removeItem(door_target_draw);
    door_target_draw = scene->addEllipse(-50, -50, 100, 100,
QPen(Qt::magenta), QBrush(Qt::magenta));
    door_target_draw->setPos(target.x(), target.y());

   // Exit condition
    if (dist_to_door < params.DOOR_REACHED_DIST)
    {
        //qInfo() << __FUNCTION__ << "Door reached at distance " <<
dist_to_door << ", switching to ORIENT_TO_DOOR";
        return {STATE::ORIENT_TO_DOOR, 0.f, 0.f};
    }

    //qInfo() << __FUNCTION__ << "moving to door at " << target.x() <<
"," << target.y() << " dist: " << dist_to_door;
    const auto &[adv, rot] = robot_controller(target); // go to first
detected door
    return {STATE::GOTO_DOOR, adv, rot};
}
```

- ORIENT_TO_DOOR

  Here, we repeat the pattern in GOTO_DOOR, selecting with the localised variable.

```
SpecificWorker::RetVal SpecificWorker::orient_to_door(const
RoboCompLidar3D::TPoints &points)
{
    // data
    const auto doors = door_detector.doors();
    if (localised)
    {
        const auto dn =
nominal_rooms[current_room].doors[current_door];
```

```
        const auto sd = std::ranges::min_element(doors, [dn,
this](const auto &a, const auto &b)
            {  return (a.center() - robot_pose.inverse() *
dn.center_global()).norm() <
                      (b.center() - robot_pose.inverse() *
dn.center_global()).norm(); });
        //qInfo() << __FUNCTION__ << "Localised, selecting door
closest to nominal door" << sd->center_angle() <<
params.RELOCAL_MAX_ORIENTED_ERROR << doors.size();
        if ( abs(sd->center_angle()) <
params.RELOCAL_MAX_ORIENTED_ERROR)
            return {STATE::CROSS_DOOR, 0.1, 0.f};
        else
            return {STATE::ORIENT_TO_DOOR, 0.f,
std::get<1>(robot_controller(sd->center()))};
    }
    else  // select the one closest to the robot's heading direction
    {
        qInfo() << __FUNCTION__ << "Not localised, selecting door
closest to robot heading";
        const auto sd = std::ranges::min_element(doors, [](const auto
&a, const auto &b)
                {  return std::fabs(a.center_angle()) <
std::fabs(b.center_angle());} );
        if (abs(sd->center_angle()) <
params.RELOCAL_MAX_ORIENTED_ERROR)
            return {STATE::CROSS_DOOR, 0.5f, 0.f};
        else
            return {STATE::ORIENT_TO_DOOR, 0.f,
std::get<1>(robot_controller(sd->center()))};
    }
}
```

- **CROSS_DOOR**

  This is the trickiest part. When the robot passes to a known room, it can localise there using the new room door's coordinates. It places it self at the door's nominal coordinates and facing towards the color room.

  If this conditions evaluates to true:

  // if entering known room, relocalise the robot
  if (next_room_idx >= 0 and nominal_rooms[next_room_idx].visited)

the robot is entering a known room. current room is updated and its pose is set to the door's pose in the new room with a -500 mm offset closer to the center and a rotation of 0º. Localised is set to true.

If the room is **new** the variable door_crossing is initialized (overwritten), and the door visited flag is set to true.

```
SpecificWorker::RetVal SpecificWorker::cross_door(const
RoboCompLidar3D::TPoints &points)
{
   static bool first_time = true;
   static std::chrono::time_point<std::chrono::system_clock> start;

    // Exit condition: the robot has advanced 1000 or equivalently 2
seconds at 500 mm/s
   if (first_time)
   {
       first_time = false;
       start = std::chrono::high_resolution_clock::now();
       return {STATE::CROSS_DOOR, 500.0f, 0.0f};
   }
   else
   {
        const auto elapsed = std::chrono::high_resolution_clock::now()
- start;
       //qInfo() << __FUNCTION__ << "Elapsed time crossing door: "
                                  //
                                                                  <<
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count(
) << " ms";
                                                                  if
(std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count
() > 3000)
       {
           first_time = true;
                                      const   auto   &leaving_door   =
nominal_rooms[current_room].doors[current_door];
           int next_room_idx = leaving_door.connects_to_room;
           // if entering known room, relocalise the robot
                                    if   (next_room_idx   >=   0   and
nominal_rooms[next_room_idx].visited)
           {
               //qInfo() << __FUNCTION__ << "Entering known room " <<
next_room_idx << ". Skipping LOCALISE.";
```

```
                // Update indices to the new room
                int next_door_idx = leaving_door.connects_to_door;
                current_room = next_room_idx;
                current_door = next_door_idx;

                   // Compute robot pose based on the door in the new
room frame.
                                        const auto &entering_door =
nominal_rooms[current_room].doors[current_door];  // door we are
entering now
                                       Eigen::Vector2f door_center =
entering_door.center_global(); //
                // Vector from door to origin (0,0) is -door_center
                     const float angle = std::atan2(-door_center.x(),
-door_center.y());

                // robot_pose now must be translated so it is drawn in
the new room correctly
                robot_pose.setIdentity();
                  door_center.y() -= 500; // place robot 500 mm inside
the room
                robot_pose.translate(door_center);
                robot_pose.rotate(0);
                  //qInfo() << __FUNCTION__ << "Robot localised in NEW
room " << current_room << " at door " << current_door;
                std::cout << door_center.x() << " " << door_center.y()
<< " " << angle << std::endl;


                localised = true;
                // Continue navigation in the new room
                return {STATE::GOTO_ROOM_CENTER, 0.f, 0.f};
            }
              else // Unknown room. I need to store the door index of
the current door and start tracking the just crossed door,
            {
                            door_crossing = DoorCrossing{current_room,
current_door};

nominal_rooms[current_room].doors[current_door].visited = true; //
exiting door
                 // from here it must be updated until localisation is
achieved again
                return {STATE::LOCALISE, 0.f, 0.f};
            }
```

```
        }
        else // keep crossing
            return {STATE::CROSS_DOOR, 500.f, 0.f};
    }
}
```

Annex 2: DNN model to recognise numbers on the rooms' walls

We will create a NEW separate component to process the camera images and return a 0-9 digit or no-digit. This component will be created using robocompdsl and setting the language to Python. You need to:

- move your existing localiser code in the actividad 4 folder to a sub-folder named localiser. Do this with **git mv** after cleaning the current local version.
- Copy **mnist_detector** from beta_robotica_class to Activity4
- create an IDSL file under ~/robocomp/interfaces/IDSLs/ named **MNIST.idsl** specifying a data structure and a method that will be called from within your localiser component, and that will be implemented in the new dnn_processor component.
- create the new component using **robocompdsl mnist-detector.cdsl**, edit it to adde the Camera360RGB.idsl and MNist.idsl interfaces and then do **robocompdsl dnn_processor.cdsl** to generate the Python code.
- Open in Pycharm-professional or another IDE
- The new component has to:
  - load the trained model (my_network.pt) once
  - In the process_image() method (defined in the MNist.idsl interface and created by robocompdsl in the specificworker.py file)
    - read the image from WebotsBridge
    - pass it to the model
    - return the result
- In your localiser component (C++) edit the .cdsl file to add the new interface **MNist.idsl** and regenerate it. You will get an additional proxy to remotely access
- the new Python component. Set the ports properly.

This Python program downloads the dataset and creates a mosaic visualization of a sample of the 10K images.
You need another Python program to train the dataset and create a my_network.pt file with the network architecture and weights. Specify a 2-layer convolutional net with matching sizes for the input layer (image size) and the output layer (10 elements coding the digits).
Before integrating it in the new component, create another script to test the model with the test images. It should give you a success rate of over 95%.
In **beta-robotica-class** you can find a new Python component, **mnist_detector,** that requires the image from the robot's camera through WebotsBridge. Complete the code in compute() to:

1. load the trained DNN model
2. search for a candidate number region, i.e. a black square.
3. if found, do a forward pass on the detected region.

When the **get_number()** method is called from the localiser component, it will return the current detection number, if found, or -1 otherwise