* [ROS](#ros2)

קישור לתוכנה המלמדת רוס:

<https://www.robotigniteacademy.com/en/course/ros-basics-5-days-c/details/>

UN: [mandel.yonatan@gmail.com](mailto:mandel.yonatan@gmail.com)

PW:  obir7iSt

* [Topic](#topic3)
* [Messages](#Messages3)
* [Nodes and Packages](#Nodes_and_Packages3)
* [Publisher](#Publisher2)
* [Subscriber](#subscriber)
* [Names](#names3)
* [Launch files](#launch2)
* [Parameters](#parameters3)
* [services](#services3)

* [C](#c2)++

* [LINUX](#linux2)

|  |  |  |
| --- | --- | --- |
| [**ROS**](#ros1) | | |
| ~$ roscore | | להתחיל  אם יש קובץ לאונץ להרצה אז אין צורך. |
| ~/name\_of\_ws$ catkin build (catkin make) | | compile our environment |
| $ source ~/catkin\_ws/devel/setup.bash  אופציה נוספת כדי לא לשכוח את זה כל פעם זה להוסיף את השורה הזאת בתוך קובץ ה ~/.bashrc כי הקובץ הזה נקרא בכל פעם שפותחים טרמינל חדש. – זה מה שעשינו | | source the development environment |
| **תחילת עבודה** | | |
| **Lab computer:**   1. ifconfig 2. sudo raspi-config  * Interfacing Options * SSH   **Our computer:**  ssh ubuntu@IP\_address\_lab(192.168.42.43)  pw: ubuntu | | Connect to the raspberry pi using SSH command |
| **הפעלת הרחפן** | | |
| $ roslaunch bebop\_driver bebop\_node.launch | | |
| $ rostopic pub --once /bebop/takeoff std\_msgs/Empty | | Takeoff |
| $ rostopic pub --once /bebop/land std\_msgs/Empty | | Land |
| **הפעלת המצלמה** | | |
| How to run the camera?  **Recommended way:**  source ~/.bashrc  roslaunch rs\_pub publish\_rs.launch  (rostopic list and then rostopic echo)  **Not recommended:**  source ~/.bashrc  roslaunch realsense2\_camera rs\_t265.launch  The pose is published on topic:  /camera/odom/sample | | To show the output in rviz, use the following command |
| **Launch Files – general commands** | | |
| roslaunch <*package\_name*> <*launch\_file*> | | מריץ את הקובץ הספציפי בפרויקט שאחראי לביצוע פקודה שאנחנו רוצים.  חשוב להיות בתוך תיקיית ה launch כשמשתמשים בפקודה הזו. |
| catkin build --dry-run  cd /tmp/ros\_tutorials\_ws #Navigate to workspace  catkin build --dry-run #Show the package build order | | Before actually building anything in the workspace, it is useful to preview which packages will be built and in what order. This can be done with the --dry-run option: |
| **Create a package** | | |
| catkin\_create\_pkg <*package\_name*> <*package\_dependecies*>  full example:  $ catkin\_create\_pkg <*package\_name*> <*package\_dependecies*>  $ cd <*package\_name*>/  $ nano CMakeLists.txt | create a package | |
| **Compiling** | | |
| In CMakeLists.txt  find\_package(catkin REQUIRED COMPONENTS roscpp geometry\_msgs)  In package.xml, we should add elements for the new dependency: <build\_depend>geometry\_msgs</build\_depend>  <run\_depend>geometry\_msgs</run\_depend> | | Declaring message type dependencies:  We must modify the find\_package line in CMakeLists.txt to mention geometry\_msgs in addition to roscpp.  (and also in package.xml)  \*\*\*If you skip (or forget) this step, then catkin\_make may not be able to find the header file geometry\_msgs/Twist.h. When you see errors about missing header files, it’s a good idea to verify the dependencies of your package. |
| roscd; cd ..  catkin build (catkin\_make) | | לקמפל את כל החבילות |
| roscd; cd ..  catkin\_make --only-pkg-with-deps <*package\_name*> | | לקמפל חבילה ספציפית |
| ~/our\_ws/src/our\_pkg/src$ chmod +x file\_name.cpp | | אם אנחנו רושמים קובץ c ואנחנו לא דואגים להפוך את הקובץ ל executable נקבל שגיאה. לכן אם נשתמש בפקודת ls לא נראה את הקובץ שלנו. אם זה המצב זה אומר שצריך להשתמש בפקודת  chmod +x file\_name.cpp |
| **general** | | |
| add\_executable(simple\_topic\_publisher src/simple\_topic\_publisher.cpp)  add\_dependencies(simple\_topic\_publisher ${simple\_topic\_publisher\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})  target\_link\_libraries(simple\_topic\_publisher  ${catkin\_LIBRARIES}) | |  |
| add\_executable(executable-name source-files) target\_link\_libraries(executable-name ${catkin\_LIBRARIES})  add\_executable(hello hello.cpp)  target\_link\_libraries(hello ${catkin\_LIBRARIES}) | | Declaring an executable  The first line declares the name of the executable we want, and a list of source files that should be combined to form that executable. If you have more than one source file, list them all here, separated by spaces. The second line tells CMake to use the appropriate library flags (defined by the find\_package line above) when linking this executable. If your package contains more than one executable, copy and modify these two lines for each executable you have. |
| rospack list | | רשימה של כל הפרויקטים שיש לנו |
| rospack list | grep <*package\_name*> | | נותן לנו להסתכל על פרויקט ספציפי |
| rospack profile | | עושה refresh לכל ה ROS. |
| rosnode list | | רשימה של כל הצמתים הקיימים |
| rosnode info /Obiwan | | לקבל אינפורמציה על צומת ספציפית |
| rosparam list | | רשימת הפרמטרים הקיימים לרוס ספציפי |
| rosparam get <*parameter\_name*> | | ערך של פרמטר ספציפי (של צומת) |
| rosparam set <*parameter\_name*> <*value*> | | לשנות ערך של פרמטר ספציפי (של צומת) |
| export | grep ROS | | משתנים גלובליים הקשורים ללינוקס (של המחשב לא של התוכנה) |
| **Topic** | | |
| rostopic echo <topic\_name> | | The actual messages that are being published on a single topic  Each --- line in the output shows the end of one message and the start of another. In this case, there were three messages. |
| rostopic pub <topic\_name> <message\_type> <value> | | הפקודה שמפרסמת טופיק באופן ידני. |
| rostopic pub -r <rate-in-hz> <topic-name> <message-type> <message-content> | | This command repeatedly publishes the given message on the given topic at the given rate.  The final *message content* parameter should provide values for all of the fields in the  message type, in order. Here’s an example:  rostopic pub -r 1 /bebop/cmd\_vel geometry\_msgs/Twist ’[2, 0, 0]’ ’[0, 0, 0]’ |
| rostopic list | | To get a list of active topics |
| rostopic info <topic\_name> | | To get the type of msg, etc. |
| rostopic -h | | כמו help במטלב |
| **Messages** | | |
| rosmsg show <name\_of\_msg(the full name with pkg)>  rosmsg show geometry\_msgs/Twist | לקבל את רשימת השדות הרלוונטיים | |
| Where can we find all the std\_msgs?  ~/name\_ws/src/name\_pkg$ roscd std\_msgs/  /opt/ros/kinetic/share/std\_msgs$ cd msg  /opt/ros/kinetic/share/std\_msgs$ ls  and then in the cpp file for example, we can write:  #include <std\_msgs/string.h>  …  msg\_to\_publish = String() //this is python  …  msg\_to\_publish.data=”hkjhkj” | | |
| **Nodes and Packages** | | |
| rospack list | | A list of all of the installed ROS packages using this command |
| rospack find <package-name> | | To find the directory of a single package, use the rospack find command: |
| rosls <package-name> | | To view the files in a package directory |
| rosrun <package-name> <executable-name>  rosrun test\_pub\_pkg test\_publisher.py | | The basic command to create a node (also known as “running a ROS pro- gram”)  \***node names** are not necessarily the same as the names of the executables underlying those nodes |
| rosnode list | | To get a list of running nodes |
| rosnode kill <node-name> | | To kill a node |
| **ROS and C++** | | |
| #include <ros/ros.h> | | The header file ros/ros.h includes declarations of the standard ROS classes. You’ll want to include it in every ROS program that you write. |
| ros::init(argc, argv, "my\_node\_name"); | | There are two levels of initialization for a roscpp [Node](http://wiki.ros.org/Nodes). This is the first step.  Node names must be **unique** across the ROS system. If a second node is started with the same name as the first, the first will be shutdown automatically. In cases where you want multiple of the same node running without worrying about naming them uniquely, you may use the init\_options::AnonymousName option described below. |
| ros::NodeHandle nh; | | The second step:  The ros::NodeHandle object is the main mechanism that your program will use to interact with the ROS system.  Creating this object registers your program as a node with the ROS master. The simplest technique is to create a single NodeHandle object to use throughout your program. |
| Classes:  class ExampleClass; | | If the file primarily implements a class, name the file after the class. For example the class ActionServer would live in the file action\_server.h.  Class names (and other type names) are **CamelCased** |
| **A** [**publish****er**](#Publisher) **program** | | |
| Learning the topic | | |
| rostopic info <topic-name> | | learn more about a topic  \*\*The most important part of this output is the very first line, which shows the **message type** of that topic. The word “type” in this context is referring to the concept of a **data type**. It’s important to understand message types because they determine the content of the messages. That is, the message type of a topic tells you what information is included in each message on that topic, and how that information is organized.  \*\*Message type names always contain a slash, and the part before the slash is the name of the containing package:  <package-name>/<type-name>  \*\*when writing ROS programs, we’ll need to declare dependencies on other packages that contain message types that we use. Including the package name as part of the message type name makes these dependencies easier to see. |
| rosnode info <node-name> | | Information about a particular node |
| Creating a publisher object | | |
| #include <package\_name/type\_name.h>  #include <geometry\_msgs/Twist.h> | | Every ROS topic is associated with a message type. Each message type has a corresponding C++ header file. You’ll need to #include this header for every message type used in your program.  \*\*The practical impact of this naming is that when referring to message classes in C++ code, you’ll use the double colon (::)—also called the **scope resolution operator**— to separate the package name from the type name. In our example, the header defines a class called geometry\_msgs::Twist. |
| ros :: Publisher pub = node\_handle.advertise<message\_type>( topic\_name, queue\_size);  ros :: Publisher pub = nh. advertise<geometry\_msgs :: Twist>( "bebop/cmd\_vel", 1000); | | The work of actually publishing the messages is done by an object of class ros::Publisher.  The **node\_handle** is an object of class ros::NodeHandle, one that you created near the start of your program. We’re calling the advertise method of that object.  The **message\_type** part inside the angle brackets—formally called the template parameter—is the data type for the messages we want to publish. This should be the name of the class defined in the header discussed above. In the example, we use the geometry\_msgs::Twist class.  The **topic\_name** is a string containing the name of the topic on which we want to publish. It should match the topic names shown by rostopic list or rqt\_graph, but (usually) without the leading slash (/).  The last parameter to advertise is an integer representing the size of the **message queue** for this publisher. In most cases, a reasonably large value, say 1000, is suit- able. If your program rapidly publishes more messages than the queue can hold, the oldest unsent messages will be discarded. |
| Creating and filling in the message object | | |
| rosmsg show <message-type-name> | | To see details about a message type.  \*\*The format is a list of **fields**, one per line. Each field is defined by a built-in data type (like int8, bool, or string) and a field name. |
| Example:  rosmsg show geometry\_msgs/Twist | | In this case, both linear and angular are **composite fields** whose data type is geometry\_msgs/Vector3. The indentation shows that fields named x, y, and z are members within those two top-level fields. That is, a message with type geometry\_msgs/Twist contains exactly six numbers, organized into two vectors called linear and angular. Each of these numbers has the built-in type float64, which means, naturally, that each is a 64-bit floating point number. |
| geometry\_msgs::Twist msg; msg.linear.x = double(rand())/double(RAND\_MAX); msg.angular.z = 2\*double(rand())/double(RAND\_MAX) - 1; | | Creating and filling in the message object |
| Publishing the message | | |
| pub.publish(msg); | | publish the message, using the publish method of the ros::Publisher object |
| The publishing loop | | |
| ros::Rate rate(2);  while(ros::ok()) {  geometry\_msgs : : Twist msg ; msg. linear .x = ….;  msg.angular.z = ...;  pub.publish(msg ) ;  ROS\_INFO\_STREAM("Sending␣random␣velocity␣command:" << "␣linear=" << msg.linear.x << "␣angular=" << msg.angular.z);  rate . sleep () ;  } | | This example creates a rate object designed to regulate a loop that executes two iterations per second. Near the end of each loop iteration, we call the sleep method of this object:  rate.sleep();  Each call to the this method causes a delay in the program. The duration of the delay is calculated to prevent the loop from iterating faster than the specified rate. Without this kind of control, the program would publish messages as fast as the computer allows, which can overwhelm publish and subscribe queues and waste computation and network resources. |
| **A** **[subscriber](#subscriber2) program** | | |
| Writing a callback function | | |
| void function\_name(const package\_name::type\_name &msg)  {  (printing to screen / reading x,y,z… etc.)  }  void poseMessageReceived(const turtlesim::Pose& msg) { … } | | The package\_name and type\_name are the same as for publishing: They refer to the message class for the topic to which we plan to subscribe.  As always, we must include the appropriate header that defines this class. |
| Creating a subscriber object | | |
| ros::Subscriber sub = node\_handle.subscribe(topic\_name, queue\_size, pointer\_to\_callback\_function);  ( inside the int main(….) )  ros::Subscriber sub = nh.subscribe("turtle1/pose", 1000, &poseMessageReceived ) ;  full example:  int main(int argc, char ∗∗argv) {  // Initialize the ROS system and become a node.  ros::init(argc, argv, "subscribe\_to\_pose"); ros::NodeHandle nh;  // Create a subscriber object .  ros::Subscriber sub = nh.subscribe("turtle1/pose", 1000, &poseMessageReceived ) ;  // Let ROS take over.  ros::spin();  } | | The **node\_handle** is the same node handle object that we’ve seen several times already.  The **topic\_name** is the name of the topic to which we want to subscribe, in the form of a string. Again, we omit the leading slash to make this string a relative name.  The **queue\_size** is the integer size of the message queue for this subscriber. Usually, you can use a large value like 1000 without worrying too much about the queuing process.  The last parameter is a **pointer to the callback function** that ROS should execute when messages arrive. In C++, you can get a pointer to a function using the ampersand (&, “address-of”) operator on the function name. In our example, it looks like this: &poseMessageReceived |
| Giving ROS control  The final complication is that ROS will only execute our callback function when we give it explicit permission to do so. There are actually two slightly different ways to accomplish this. | | |
| ros::spinOnce(); | | This code asks ROS to execute all of the pending callbacks from all of the node’s subscriptions, and *then return control back to us*. |
| ros::spin(); | | This alternative to ros::spinOnce() asks ROS to wait for and execute callbacks *until the node shuts down.* In other words, ros::spin() is roughly equivalent to this loop:  while(ros::ok()) { ros::spinOnce(); } |
| **graph resource names** | | |
| Global names | | |
| Nodes, topics, services, and parameters are collectively referred to as **graph resources**. Every graph resource is identified by a short string called a **graph resource name**. | | |
| /teleop\_turtle  /turtlesim  /turtle1/cmd\_vel /turtle1/pose /run\_id /count\_and\_log/set\_logger\_level | | There are several parts to a global name:  **A leading slash /,** which identifies the name as a global name.  **A sequence of zero or more namespaces,** separated by slashes. Namespaces are used to group related graph resources together. The example names above include two explicit namespaces, called turtle1 and count\_and\_log. Multiple levels of name- spaces are allowed, so this is also a valid (but rather unlikely) global name, consisting of 11 nested namespaces: /a/b/c/d/e/f/g/h/i/j/k/l  Global names that don’t explicitly mention any namespace—including three of the examples above—are said to be in the **global namespace**.  A **base name** that describes the resource itself. The base names in the example above are teleop\_turtle, turtlesim, cmd\_vel, pose, run\_id, and set\_logger\_level. |
| Relative names | | |
| במקום לתת ל ROS להחליט על הדרך המדויקת של השם (עם הסלשים והשם ה״רשמי״) יש אופציה היא לתת ל ROS שם בררת מחדל.  A name that uses this feature is called a **relative graph resource name**, or simply a **relative name**. | | |
| teleop\_turtle turtlesim cmd\_vel turtle1/pose run\_id  count\_and\_log/set\_logger\_level | | The characteristic feature of a relative name is that it lacks a leading slash (/). |
|  | | |
| \_ \_ns:=default-namespace | | Most ROS programs, including all C++ programs that call ros::init, accept a command line parameter called \_ \_ns, which specifies a default namespace for that program. |
| Private names | | |
| **Private names**, which begin with a tilde (∼) character, are the third and final class of graph resource names. Like relative names, private names do not fully specify the namespace in which they live, and instead rely on the ROS client library to resolve the name to a complete global name. The difference is that, instead of using the current default namespace, private names use the name of their node as a namespace. | | |
| For instance, in a node whose global name is /sim1/pubvel, the private name ∼max\_vel would be converted to a global name like this | | |
| Private names are often used for parameters—**roslaunch** has a specific feature for setting parameters that are accessible by private names; see page 113—and **services** that govern the operation of a node. It is usually a mistake to use a private name to refer to a topic because, if we’re keeping our nodes loosely coupled, no topic is “owned” by any particular node. | | |
| [Anonymous names](#Anonymous_names2) | | |
| Anonymous names are specifically used to name **nodes**. The purpose of an anonymous name is to make it easier to obey the rule that each node must have a unique name. The idea is that a node can, during its call to **ros::init**, request that a unique name be assigned automatically.  To request an anonymous name, a node should pass  ros::init\_options::AnonymousName as a fourth parameter to ros::init:  ros::init(argc, argv, base\_name, ros::init\_options::AnonymousName);  The effect of this extra option is to append some extra text to the given base name, ensuring that the node’s name is unique.  \*\*To use an anonymous name from **within a launch file**, use an anon substitution􏰁4 for the name attribute, like this:  name="$(anon base\_name)"  Note, however, that multiple uses of the same base name will generate the same anonymous name. This means that (a) we can refer to that name in other parts of the launch file, but (b) we must be careful to use different base names for each node we want to anonymize. | | |
| **[Launch](#launch1) files** | | |
| Executing launch files | | |
| roslaunch package-name launch-file-name  roslaunch agitr example.launch | | roslaunch enables us to start many nodes at once.  \*\*An important fact about roslaunch—one that can be easy to forget—is that all of the nodes in a launch file are started at roughly the same time. As a result, you cannot be sure about the order in which the nodes will initialize themselves. Well-written ROS nodes don’t care about the order in which they and their siblings start up. |
| Launching nodes | | |
| <node  pkg="package-name"  type="executable-name"  name="node-name"  output="screen"  />  You can also write the closing tag explicitly:  <node pkg=". . . " type=". . . " name=". . . "></node>  In fact, this explicit closing tag is needed if the node has children, such as remap or  param elements.  אסור .cpp !!! | | The heart of any launch file is a collection of node elements, each of which names a single node to launch.   1. **pkg="package\_name"** # Name of the package that contains the code of the ROS program to execute 2. **type="cpp\_executable\_name"** # Name of the cpp executable file that we want to execute 3. **name="node\_name"** # Name of the ROS node that will launch our C++ file. This overrides any name that the node would normally assign to itself in its call to ros::init. 4. **output="type\_of\_output"** # Through which channel you will print the output of the program |
| **6.3 Launching nodes inside a namespace**  לעבור על הפרק הזה אם נצטרך ליצור למשל 2 סימולטורים עם אותו שם. בפרק הזה מוסבר איך לבנות את שני ה node-ים בצורה נכונה. | | |
|  | |  |
|  | |  |
| **Setting parameters in launch files** | | |
| <param name="param-name" value="param-value" /> | | **Setting parameters** To ask roslaunch to set a parameter value |
| **Remapping names** | | |
| <original-name>:=<new-name>  For example, to run a turtlesim instance that publishes its pose data on a topic called /tim instead of /turtle1/pose, use a command like this:  rosrun turtlesim turtlesim\_node turtle1/pose:=tim | | **Creating remappings**  To remap a name when starting a node **from the command line**, give the original name and the new name, separated by a :=, somewhere on the command line. |
| <remap from="original-name" to="new-name" />  <node pkg="turtlesim" type="turtlesim\_node" name="turtlesim" >  <remap from="turtle1/pose" to="tim" />  </node> | | **Creating remappings**  To remap names within a launch file, use a remap element:  If it appears at the top level, as a child of the launch element, this remapping will apply to all subsequent nodes. These remap elements can also appear as children of a node element, like this:  <node node-attributes >  <remap from="original-name" to="new-name" />  ... </node> |
| Including other files | | |
| <include file="path-to-launch-file" /> | | To include the contents of another launch file, including all of its nodes and parameters, use an include element |
| <include file="$(find package-name)/launch-file-name" />  The example uses this technique to include our previous example, doublesim.launch.  <include  file="$(find agitr)/doublesim.launch"  />  <include  file="$(find iri\_wam\_reproduce\_trajectory)/launch/start\_service.launch"  /> | | The file attribute expects the full path to the file we want to include. Because it can be both cumbersome and brittle to enter this information directly, most include elements use a find substitution to search for a package, instead of explicitly naming a directory.  The find argument is expanded, via a string substitution, to the path to the given package. |
| **6.5.2 Launch arguments**  לעבור על הפרק הזה אם רוצים להפוך את הקוד ליעיל יותר מהבחינה של חיסכון בשורת קוד בעיקר. | | |
|  | |  |
| **7.** **Parameters** | | |
| rosparam list | | **Listing parameters** To see a list of all existing parameters |
| rosparam get <parameter\_name>  rosparam get /rosdistro | | **Querying parameters** To ask the parameter server for the value of a parameter |
| rosparam set <parameter\_name> <parameter\_value>  For example, these commands create string parameters that store the wardrobe preferences of a certain group of cartoon ducks:  rosparam set /duck\_colors/huey red  rosparam set /duck\_colors/dewey blue rosparam set /duck\_colors/louie green rosparam set /duck\_colors/webby pink | | **Setting parameters** To assign a value to a parameter |
| rosparam dump <filename> <namespace> | | **Creating parameter files** To store all of the parameters from a namespace, in YAML format, to a file |
| rosparam load <filename> <namespace> | | **Loading parameter files** The opposite of dump is load, which reads parameters from a file and adds them to the  parameter server |
| **7.3 Accessing parameters from C++** | | |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
| **Services** | | |
| * Service calls are **bi-directional**. One node sends information to another node and waits for a response. Information flows in both directions. In contrast, when a message is published, there is no concept of a response, and not even any guarantee that anyone is subscribing to those messages. * Service calls implement **one-to-one** communication. Each service call is initiated by one node, and the response goes back to that same node. On the other hand, each message is associated with a topic that might have many publishers and many subscribers. | | |
| The idea is that a **client** node sends some data called a **request** to a **server** node and waits for a reply. The server, having received this request, takes some action (computing some- thing, configuring hardware or software, changing its own behavior, *etc.*) and sends some data called a **response** back to the client. | | |
| The specific content of the request and response data is determined by the **service data type**, which is analogous to the message types that determine the content of messages. | | |
| many ROS services in general, can be divided into two basic types.   * Some services, such as the get\_loggers and set\_logger\_level services in the list above, are used to get information from or pass information to specific nodes. These kinds of services usually use their node’s name as a namespace to prevent name collisions, and to allow their nodes to offer them via private names like ∼get\_loggers or ∼set\_logger\_level. * Other services represent more general capabilities that are not conceptually tied to any particular node. | | |
| rosservice list | | **Listing all services.**  The output of rosservice list shows the full global name of each service. |
| rosnode info node-name | | **Listing services by node** To see the services offered by one particular node. |
| rosservice node service-name | | **Finding the node offering a service** To perform the reverse query—that is, to see which node offers a given service. |
| rosservice info service-name  rosservice info /spawn  (We can see that the data type of the /spawn service is turtlesim/Spawn. ) | | **Finding the data type of a service** To determine the service data type of a service. |
| As with message types, a service data type has two parts, one naming the package that owns the type, and one naming the type itself: | | |
| rossrv show service-data-type-name  rossrv show turtlesim/Spawn | | **Inspecting service data types** To get some details about service data types.  \*\*In this case, the data before the dashes (---) are the elements of the **request**. This is the information that the client node sends to the server node. Everything after the dashes is the **response**, or information that the server sends back from the client when the server has finished acting on the request. |
| rosservice call service-name request-content  rosservice call /spawn 3 3 0 Mikey  The effect of this service call is to create a new turtle named “Mikey,” at position (*x*, *y*) = (3, 3), facing angle θ = 0, within the existing simulator.  rosservice call /my\_service [TAB]+[TAB] | | **Calling services from the command line** To get a feel for how services work.  The **request content** part should list values for each field of the request, as shown by rossrv show. |
| **A** **[client](#client2) program** | | |
| #include <package\_name/type\_name.h>  #include <turtlesim/Spawn.h> | | **Declaring the request and response types** Just like message types, every service data has an associated C++ header file that we must include. |
| ros::ServiceClient client = node\_handle.serviceClient<service\_type>( service\_name);  ros :: ServiceClient spawnClient = nh.serviceClient<turtlesim::Spawn>("spawn"); | | **Creating a client object** After initializing itself as a node (by calling ros::init and creating a NodeHandle object), our program must create an object of type ros::ServiceClient, whose job is to actually carry out the service call.  The **node\_handle** is the usual ros::NodeHandle object. We’re calling its service- Client method.  The **service\_type** is the name of the service object defined in the header file we included above. In the example, it’s turtlesim::Spawn.  The **service\_name** is a string naming the service that we want to call. This should be a relative name, but can also be a global name. The example uses the relative name "spawn". |
| package\_name::service\_type::Request package\_name::service\_type::Response  turtlesim : : Spawn : : Request req ; turtlesim : : Spawn : : Response resp ; | | **Creating request and response objects**  to create a request object to contain the data to be sent to the server.  (rossrv show) |
| bool success = service\_client.call(request, response);  bool success = spawnClient.call(req, resp); | | **Calling the service** Once we have a ServiceClient, a completed Request, and a Response, we can actually call the service |
| A common mistake is to fail to check the return value of call. This can lead to unexpected problems if the service call fails. It takes only a minute or two to add code to check this value and call ROS\_ERROR\_STREAM when the service call fails. It’s quite likely that this investment of time will be repaid with easier debugging in the future. | | |
| \*\*If you prefer—as the authors of many online tutorials apparently do—you can pass an object of this class to the call method introduced below, instead of separate Request and Response objects.  ?? ככה עושים באתר של יוני ?? | | |
| ROS also supports a concept of persistent service clients, in which the ros::ServiceClient constructor establishes a connection to the server, which is then reused for every subsequent call for that client object. A persistent service client can be created by passing true for the second parameter of the constructor (which we’ve allowed to default to false in the previous examples):  ros::ServiceClient client = node\_handle.advertise<service\_type>( service\_name, true); | | |
| **A** **[server](#server2) program** | | |
| The code for acting as a server is remarkably similar to the code for subscribing to a topic. Aside from differences in names—we must create a ros::ServiceServer instead of a ros::Subscriber—the only difference is that a server can send data back to the client, via both a response object and a boolean indication of success or failure. | | |
| bool function\_name( package\_name::service\_type::Request &req), package\_name::service\_type::Response &resp)  ){ … }  bool toggleForward (  std\_srvs::Empty::Request &req,  std\_srvs::Empty::Response &resp ){ … } | | **Writing a service callback**  The callback’s job is to fill in the data members of the Response object.  The callback function should return true to indicate success or false to indicate failure. |
| ros::ServiceServer server = node\_handle.advertiseService(  service\_name, pointer\_to\_cal lback\_function  );  ros::ServiceServer server = nh.advertiseService( "toggle\_forward", &toggleForward); | | **Creating a server object** To associate the callback function with a service name, and to offer the service to other nodes, we must advertise the service.  \*\*The **service\_name** is a the string name of the service we would like to offer. This should be a relative name, but could also be a global name. |
| In the example, we use ros::spinOnce(), instead of ros::spin(), because we have other work to do—specifically, publishing velocity commands—when there are no incoming service calls to process. | | **Giving ROS control** Don’t forget that ROS will not execute any callback functions until we specifically ask it to, using ros::spin() or ros::spinOnce(). |
|  | | |
| roswtf | | בדיקה באופן כללי אם כל הלאונצים והשמות תקינים. |
| roscd iri\_wam\_aff\_demo/launch  roswtf false\_start\_demo.launch | | To make *roswtf your\_launch\_file.launch* work, you need to go to the path where the file is. That's why you had to use the *roscd* command. |
| **RQT - GRAPH** | | |
| ~$ rosrun rqt\_graph (hit tab again so it will end up as)  !$ rosrun rqt\_graph rqt\_graph | | In order to understand better what is going on. RQT\_GRAPH is a very convenient tool for having a graphical display of what is going on in our project. |
|  | |  |
| how to write cpp program? | | |
| [Run a C/C++ program on terminal using gcc compiler](https://rupinderjeetkaur.wordpress.com/2014/06/20/run-a-cc-program-on-terminal-using-gcc-compiler/)  Follow these steps to run programs on terminal:  **Step 1.** Open terminal.  **Step 2.** Type command to install gcc or g++ complier:  **$ sudo apt-get install build-essential**  This will install the necessary C/C++ development libraries for your Ubuntu to create C/C++ programs.  To check gcc version type this command:  **$ gcc –version or gcc –v**  **Step 3.** Now go to that folder where you will create C/C++ programs. I am creating my programs in Documents directory. Type these commands:  **$ cd Documents/**  **$ sudo mkdir programs**  **$ cd programs/**  **Step 4.** Open a file using any editor.  **$ sudo gedit first.c**(for C programs)  **$ sudo gedit hello.cpp**(for C++ prgrams)  **Step 5.** Add this code in the file:  (i). C program code:  #include<stdio.h>  int main()  {              printf(“\n\nWelcome to my Homepage!!\n\n”);              return 0;  }  (i). C++ program code:  #include<iostream>  using namespace std;  int main()  {              cout<<“\n\nHello World,\nWelcome to my first C ++ program on Ubuntu Linux\n\n”<<endl;              return 0;  }  **Step 6.**Save the file and exit.  **Step 7.** Compile the program using any of the following command:  (i). Compiling C program.  **$ sudo gcc first.c**  It will create an executable file with “.out” extension named as “a.out”.  Or  **$ sudo gcc –o first first.c**  Where **first** is the executable or object file of **first.c** program.  (ii). Compiling C++ program.  **$ sudo g++ hello.cpp**(or)  **$ sudo g++ -o hello hello.cpp**  **[Note:** Make sure you are in the same directory where you have created your program before compiling it.**]**  **Step 8.** To run this program type this command:  (i). For running C program  **$ ./a.out** (If you compiled using first command)  Or  **$ ./first** (If you compiled using second command)  (ii). For running C++ program  **$ ./a.out** (If you compiled using first command)  Or  **$ ./hello** (If you compiled using second command) | | |
|  | |  |

**[Publisher](#Publisher2) full example:**



**[Subscriber](#subscriber) full example:**



**[Anonymous Name](#Anonymous_names1) full example:**





**[Launch](#launch2) file full example:**



**A** [**client**](#client1) **program – full example:**



**A** **[server](#server1) program – full example:**



|  |  |
| --- | --- |
| [C++ ONLY](#c1) | |
| <https://www.w3schools.com/cpp/default.asp> | |
| std = standard namespace:  first option:  second option: include the library:  (written before the int main()). |  |
| type variable=value; | Names must begin with a letter or an underscore(\_)  A Boolean data type is declared with the bool keyword and can only take the values true or false. When the value is returned, true=1 and false=0.  To use strings, you must include an additional header file in the source code, the <string> library.  String index start with 0: [0] is the first character. |
|  | The :: operator for string (and cout) objects |
|  | Ternary Operator:  If you have only one statement to execute, one for if, and on for else, you can put it all on the same line |
|  | function. |
|  | In real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.  attributesand methods are basically variables and functions that belongs to the class (class members).  The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class.  You access method just like you access attributes; by creating an object of the class and by using the dot syntax.  To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifying the name of the class, followed the scope resolution :: operator, followed by the name of the function |
|  | A constructor in C++ is a special method that is automatically called when an object of a class is created.  constructors can also take parameters, which can be useful for setting initial values for attributes.  The constructor has the same name as the class, it is always public, and it does not have any return value. |
|  | constructor inside the class. |
|  | constructor outside the class. |
| **שימוש ב Namespace:**   * כאשר נרשום למשל using System אנחנו אומרים ל C# שאנחנו רוצים להשתמש ב Namespace **System**. * ה Namespace מיועד לדאוג לארגון הקוד וכולל בין השאר מספר רב של Classes, Interfaces, enums, structs, delegates. | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| [LINUX - ONLY](#linux1) | |
| **Linux** is an operating system or a kernel. It is distributed under an open source license. Its functionality list is quite like UNIX. | |
| הממשק של לינוקס נקרא bash, שהוא בעצם ראשי תיבות של Bourne Again Shell. ממשק ברירת המחדל של UNIX היה בזמנו Bourne Shell, והמפתח של bash כתב גרסה דומה, חינמית, ומשופרת של ה shell – בעצם shell שנולד מחדש, חופשי. | |
| **Repository**(*Computing)* a central location in which data is stored and managed: *the metadata will be aggregated in a repository*. | |
| **A kernel** is a program at the heart of any operating system that takes care of fundamental stuff, like letting hardware communicate with software. | |
| **ls command** | |
| ls [options] [fd]  ls Downloads/  ls -l  (stands for long format)  ls -a  (this will give us the hidden files also)  ls -al  (hidden files and long format) | רשימת קבצים בתיקיה  כדי לראות בצורה יותר נוחה את הקבצים, יש להקליד:  צבע הטקסט הוא בעל משמעות לפי סוג הקובץ וגם לפי כמה תכונות שלו:   * כחול – תיקיה * ירוק – קובץ הניתן להרצה (executable) * אדום – קובץ דחוס (rpm, tar, zip) * רקע ירוק עם טקסט שחור אומר שזו תיקיה עם הרשאות כתיבה ל״כולם״. * רקע ירוק עם טקסט כחול הוא מקרה דומה, אבל כאשר ה sticky bit כבוי. |
| ls -lS > name\_of\_file.txt | If we want to save all the results of what we have listed to some file |
| man ls | To read more about the ls command |
| **cd command** | |
| cd  cd ~ | current home directory  ~ is the sign for home  מעבר לתיקיית הבית של המשתמש בנוכחי (כלומר: שלי).  המקבילה בחלונות היא C:\Users\MY\_NAME. |
| cd / | change directory for the root  / is the sign for root |
| cd ..  cd ../.. | the parent directory |
| cd Documents | change directory |
| cd - | מחזירה אותי לתיקיה הקודמת בה הייתי. Bash זוכר היסטוריה של שינויי ספריות (פקודת cd), וחזרה אחורה היא דווקא די נוחה. |
| **התמצאות בסיסית בתיקיות** | |
| החלוקה הראשית של תיקיות בלינוקס היא על פי פרופיל השימוש של הקבצים שיאוחסנו בתוך הספרייה. למשל,   * ב- /usr מאוחסנים ה executable של האפליקציות המשותפות – ולכן אתם רוצים שם קריאה מהירה (שהתוכנות יעלו מהר). * ב- /var שומרים קבצים בעלי גודל משתנה (למשל: קובץ של תוכנת המייל, בה מיילים נוספים ונמחקים כל הזמן) – ובתיקייה זו יהיו הרבה מאוד כתיבות. * /tmp צפויה להרמה I/O מכל הסוגים, בעוד /swap (תיקיה ל paging של ה virtual memory או page file) צפויה ל I/O רק בקונפיגורציות בהן יש לחץ בזיכרון זמין. | |
| pwd | הצגת path מלא של תיקיה נוכחית |
| clear | ניקוי המסך |
|  | |
| source  source devel/setup.bash | * The source command can be used to load any functions file into the current shell script or a command prompt. * It read and execute commands from given FILENAME and return. * The pathnames in $PATH are used to find the directory containing FILENAME. If any ARGUMENTS are supplied, they become the positional parameters when FILENAME is executed. |
| git clone  git clone https://gitlab.com/bradan…. | If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository. |
| **cat command: cat [options] [file1 file 2 file 3 …]** | |
| cat | צפייה בקובץ (הזרמה שלו לתוך ה shell). |
| cat <filename> | less | צפייה בקובץ, עם גלילה |
| cat list1.txt  cat list1.txt list2.txt | Displaying the text  (print on the terminal)  \*\* we have to be on the current directory where the files we wants to display are |
| cat -b list1.txt | Show the line number (skipping blank lines) |
| cat -n list1.txt |  |
| This command can also combine copies of text files and create new text files | |
| to exit the cat command press Ctrl D | |
| **man command** | |
| (עם ניווט) man <command  (עם גלילה) <command> --help | עזרה של פקודה מסוימת |
| **I/O Redirection** | |
| [output] > [file]  ~$ cat > test.txt  line 1  line 2  line 3 [and enter Ctrl D] | This means we want to sent output to a file  line 1  line2  line 3  This will be written to the test.txt file |
| [output] >> [file] | Appending to the existing content of a file |
| cat [file1 file2 …] > [file3]  cat list1.txt list2.txt > out.txt | first is join the files (list1.txt and list2.txt) and then this content to a new file (or existing file but delete it’s old content |
| **mkdir command** | |
| mkdir | יצירת תיקיה |
| mkdir -p a/b/c/d/e | The *-p* (i.e., *parents*) option creates the specified intermediate directories for a new directory if they do not already exist. |
| **rm and rmdir commands:**  **rmdir [options] [directory names]** | |
| rmdir <namd\_of\_folder> | remove a folder (delete), as oppose to mkdir. |
| rmdir -p a/b/c/d/e  (rmdir a/b/c/d/e will remove only the e folder) | The *-p* (i.e., *parents*) option deletes the specified intermediate directories. |
| rmdir -pv a/b/c/d/e | v stands for verbose flag (to get extended information) |
| rm -r [parent directory]  rm -r a/ | r stands for recursive  remove the file and the directories (its parents) |
| **cp command: cp [options] [name\_of\_source] [name\_of\_destination]** | |
| cp file1.txt file2.txt | copy file to new file  (if the destination file does not exist it will create a new file) |
| cp file1.txt dir1  cp file1.txt file2.txt dir2  and then:  ~/Desktop$ ls dir2  file1.txt file2.txt | copy file (or multiple files) to a directory (directory here means folder) |
| cp -i file1.txt file2.txt dir1 | i stands for interactive which ask before overwriting (for example, if file1.txt is already in dir1). |
| cp -R dir1 dir3 | copy directory that contains files (dir1 and dir3 should exist). |
| **mv command: mv [options] [name\_of\_source] [name\_of\_destination]** | |
| mv file1.txt file2.txt  (file1 will be named “file2”) | change a name of a file |
| mv -i file2.txt dir1/  mv dir1 dir2 | move (transfer) file to a directory |
| **less command** | |
| less big.txt  This command will show us the beginning of the file and if we want to see more of it we can hit the down arrow key (and up).  /book  (to go to another “book” in the file we can press n) | less command is useful when we want to read very long files (the cat command fails to do it).  We can also see the content of the file page by page by pressing space. Capital b (B) will return to the previous page.  If we want to jump to the end of the file we can press capital G (shift+g)  If we want to jump to the beginning of the file we can press 1g.  If we want to search something (from up to down) we can type /something\_to\_find  If we want to quit the less command we can press q |
| **touch command** | |
| touch <name\_of\_file>  touch ofir\_launch\_file.launch | create new empty file |
| touch <name\_of\_file>  (if the file exists that the time stamp will change to the current clock time) | change the time stamp of a file |
| **nano command** | |
| nano <name\_of\_file>  nano file1.cpp | create new file but in this case we have to save the file using Ctrl o |
| nano <name\_of\_file>  nano file1.cpp | edit some file |
| **kill command: kill –[flags] [pid]** | |
| pidof unity-control-center | to know the pid of a process (when we know the name of the running process) |
| kill <pid>  (the answer from the previous command) | close the process |
| kill -KILL <pid> | force the process to close  (not recommended) |
| kill -9 <pid> | -9 is call sig kill command. It is a powerful command for process that are not killed by the regular version of this command. |
| How to know the name of a process? How to find out the list of running processes? | |
| ps -ux | list of running processes |
| ps -aux | list of all processes |
| **echo command** | |
| echo <some text>  echo “hello world”  ~$ myvar=”Ofir”  ~$ echo $myvar  Ofir  ~$ x=10  ~$ echo “the value of x is $x”  the value of x is 10 | echoing whatever we write after the word “echo” |
| **chmod command** | |
| chmod o+x <file\_name/directory\_name>  how can we know what are the permissions?  ~/Desktop$ ls -l  -rw -rw -rwx …. file (example)  this means that the group has permission to read and write (in the middle) and others (right) can read, write and execute (user in left).  chmod ug=rwx file | utility to change the permission of a file or directory  **o** stands for others  **g** stands for group  **u** stands for user  **a** stands for user, group and others  **+** stands for adding the permission  **-** Stands for subtracting the permission  **x** is the permission in this case x stands for the executable permission  **w** is the permission in this case w stands for the write permission  **=** this notation is used for changing the permission |
| **How to create a** **[script](#script1) using linux?** | |
| script is a text file that contains a sequence of commands for our linux based operating system. | |
| ~/Desktop$ nano myscript.sh  (.sh stands for script file) | create script file |
|  | Where our bash is located on our system?  using the command which bash  We need to write it on the first line in the script file. |
| **which and whatis command** | |
| which <command\_name>  which bash | The location of the command |
| whatis <command\_name>  whatis ls | display the short manual page discription (short description compared to the man command) |
| **useradd command** | |
| useradd <user\_name> -m -s  useradd ofir -m -s /bin/bash -g users -c “my\_comment” | create user  The -m flag going to create a default home directory for this user.  The -s flag is the command for the default shell we (or the shell after the flag)  The -g flag is the default user flag  The -c flag is for comments |
| **.bashrc file** | |
| The .bashrc file is a script that is executed whenever a new terminal session is started in interactive mode. | |
| **watch command** | |
| watch free -m | run scripts for command at a regular interval or repeatedly |
| **head and tail command** | |
| head <file\_name> | output (or print) the first part of the file |
| tail <file\_name> | output (or print) the last part of the file |
| head -n3 <file\_name> | output (or print) only the top 3 lines of the file (the last 3 lines for tail) |
| **find command** | |
| find <directory> -name <name>  sudo find / -name dmesg (root directory)  find /home/our\_user -name test.\*  find /home/our\_user -name \*.sh | search for files in a directory hierarchy  (the directory parameter is the directory in which we want to find |
| find <directory> -mtime -1 | look at the files which were created \*\* days ago (in the example it is one day ago) |
| **ssh command** | |
| sudo apt-get install openssh-server | יוצר server |
| sudo systemctl status ssh | בודק את הserver |
| sudo raspi-config   * Interfacing Options * SSH |  |
| ifconfig | לבדוק קובץ ה ip |
| ssh ubunto@<ip>  passward: ubuntu |  |
| התקנת אפליקציות בלינוקס | |
| ה- package manager של אובונטו הוא deb ( קרוי גם dpkg), אולם הוא ידוע דווקא בשם APT (קיצור של Advanced Packaging Tool) – שזה בעצם שם המודול האחראי רק לפענוח תלויות והורדת הקבצים ברשת. | |
| sudo apt-get intall vim | הפקודה **apt-get** להפעלת מספר רב של פעולות במערכת APT: התקנה, הסרה, עדכון וכו׳. כלומר: install הוא הפרמטר לפקודת apt-get שאומר שאנו בעצם רוצים לבצע התקנה.  להתקין מה? **vim,** הפרמטר השני של פקודת apt-get. כל אפליקציה רשומה ב repository בשם פשוט, קצר וייחודי. יש מקרים בהם השם להתקנה יהיה שונה מהשם להפעלה של התוכנה, או מהם בו אתם מכירים את התוכנה.  אז מה הוא **sudo**? סביר להניח שלמשתמש הנוכחי אין הרשאות (טכנית, מיד נראה שהוא יכול) להתקין אפליקציות על המערכת. כדי להתקין אפליקציה יש לקרוא להרשאות גבוהות יותר של super user (או root). שם הפקודה sudo הוא קיצור של ״Super User – DO״, כלומר, דלגציה של פקודה כלשהי ל-context של ה – super user. |
| Apt-get update | הורדת חבילת העדכונים  פקודה זו מורידה את ״חבילות העדכון״ של התוכנים ומערכת ההפעלה אשר מותקנים במכונה. |
| Apt-get upgrate | התקנת חבילת העדכונים  פקודה זו מתקינה את ״חבילות העדכון״ של התוכנות ומערכת ההפעלה שר הורדו על ידי הפקודה הקודמת |
| apt-get install X | התקנת החבילה X |
|  | |
| &בסוף הפקודה | האופרטור מריץ את הפקודה מאחורי הקלעים של המערכת, ומציגה את מספר ה- process. פקודה זו יעילה כאשר מריצים סקריפטים או פקודות המריצות שורות רבות ולא חיוניות לנו, על גבי המסך.  כדי לראות את הרדימה המלאה של הפקודות הרצות כרגע ״מאחורי הקלעים״ משתמשים בפקודה jobs. |
| Find -iname FILE\_NAME  find / -size | על מנת לחפש קובץ מסוים במכונה (קובץ שאנו יודעים את שמו או חלק משמו).  אם רוצים לחפש קבצים בגודל מסוים: |
| apt-cache depends <package-name> | In addition to the dpkg method, you can check the dependencies of packages in the repository |
|  |  |

**Full example of** **[script](#script2)**

![A screenshot of a social media post

Description automatically generated]()