FMU Module User Guide

Functional Mock-up Unit, known as FMU, can be used to build connection between different simulators if they support FMU import/export. The updated Shipware demonstration have two sample modules built with FMU: Ship Dynamics and Navigation Control. This guidance will cover the following topics: Requirements and General Information, Navigation Control with FMU, Ship Dynamics with FMU.

1. Requirements and General Information

(1) System Requirements

The updated Shipware is developed on Ubuntu 20.04 and ROS noetic. For installation guide, please refer to the GitHub repository:
https://github.com/Autonoship/Autonoship_simulation

The FMUs can be built with software which supports FMU export, such as MATLAB Simulink, OpenModelica, etc. MATLAB Simulink is used in this demo. A version later than R2020a is recommended.

(2) Introduction to FMI/FMU

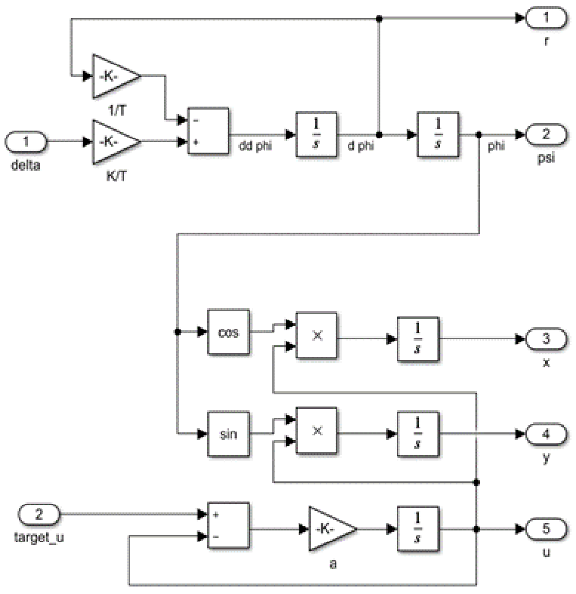
There are two types of FMUs: Model Exchange and Co-Simulation. A Co-Simulation FMU has a solver inside and can calculate the simulation result by itself as a slave simulator, while a Model Exchange FMU relies on the solver of the master simulator to do the calculation.

An FMU file is a zipped file, and you can rename the FMU as "*.zip" and unzip the file to look at the contents inside. The model is contained in the "binary" folder, and sometimes you can find both "*.dll" and "*.so", which means that the FMU can be used on both Windows and Linux systems. The "modelDescription.xml" contains all the information about the FMU, including the generation tool and the variables. You can check the "causality" of each variable to see whether it is an input, output, or parameter. For more information about FMI/FMU, please look at the FMI specifications: https://fmi-standard.org/downloads/. This project is built on FMI 2.0.

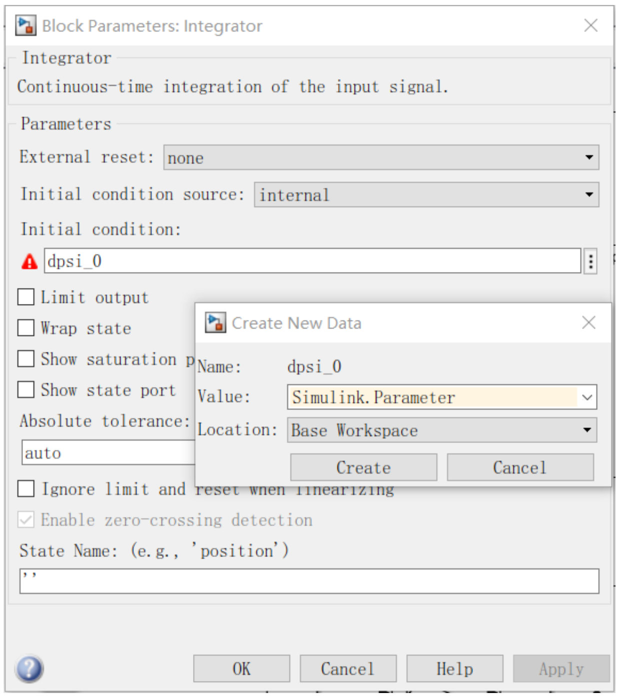
In this guide, all the FMUs are built with MATLAB Simulink, which has the Simulink compiler to export models into Co-Simulation FMU files. The exported FMU does not rely on MATLAB related dependencies, and can be used in a machine without MATLAB. However, the exported FMU can only be opened on the same type of machine, e.g. an FMU exported by MATLAB in a Linux machine can only be opened in a Linux machine. In this demonstration, the FMUs are exported by MATLAB in Ubuntu.

(3) Exporting FMU with Simulink

The following figure is an example of exporting an FMU ship dynamics by MATLAB Simulink:



The blocks “delta” and “target_u” are “In1” blocks and will become input variables after export. The blocks “r”, “psi”, “x”, “y”, and “u” are “Out1” blocks and will become output variables after export. If you want to set some tunable variables, such as the initial value of integrators, it can be set in the initial condition as base workspace parameters.


Block Parameters: Integrator

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: none

Initial condition source: internal

Initial condition:

dpsi_0

☐ Limit output

☐ Wrap state

☐ Show saturation p

☐ Show state port

Absolute tolerance:

auto

☐ Ignore limit and reset when linearizing

✓ Enable zero-crossing detection

State Name: (e.g., 'position')

''

Create New Data

Name: dpsi_0

Value: Simulink.Parameter

Location: Base Workspace

After building the model, you need to select the “Fixed-step” in the “Model Settings”, and then in the drop box of “Save”, select “Standalone FMU” to export to an FMU file.

(4) Usage of an FMU file

While using the FMU file, after initialization stage, at each time step, you can set the input variables by the function "fmi2SetXXX" according to the datatype. Then, you can call the function "fmi2DoStep" to run the simulation. You need to provide the current communication point (the current time in FMU) and the communication step size (the time interval for the simulation). The current communication point mush be equal to the previous communication point plus the previous communication step size (t_current_point = t_prev_point + t_prev_step), and the FMU will do the simulation until the time reaches (t_current_point + t_current_step) and provides the output. The output value can be get by calling the function "fmi2GetXXX" according to the data type.

```
fmi2Status fmi2DoStep(fmi2Component c,
                      fmi2Real    currentCommunicationPoint,
                      fmi2Real    communicationStepSize,
                      fmi2Boolean noSetFMUStatePriorToCurrentPoint);
```

In real case, you don't have to use these functions directly. There is a project called FMI library (https://jmodelica.org/fmil/FMILibrary-2.0.3-htmldoc/index.html), which has organized the original FMU functions very well. In this demonstration, we used two libraries which are built on the FMI library. Details will be covered in the next two sections.

2. Navigation Control with FMU

(1) Introduction to FMI Adapter package

The Navigation Control module is built based on the FMI Adapter package: http://wiki.ros.org/fmi_adapter. The FMI Adapter package is designed to run Co-Simulation FMUs in ROS environment only (not in Gazebo simulation). For testing, after installing the package, one can run the command:

roslaunch fmi_adapter fmi_adapter_node.launch fmu_path:=[PathToTheFMUFile]

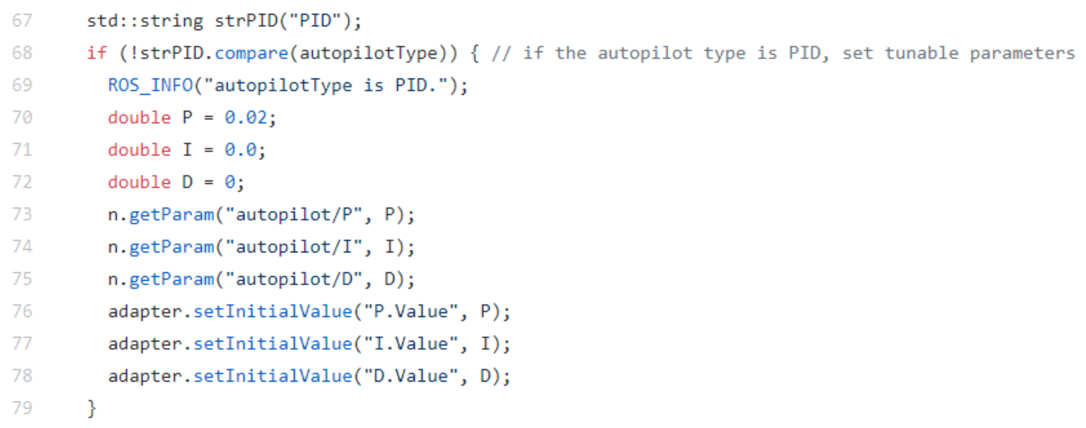
This will run the FMU file with a ROS node, and the connection between ROS and FMU are realized by ROS publisher and subscriber. For more information about publisher and subscriber, please look at:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

You can look at and communicate with the ROS topics with the tool "rostopic": http://wiki.ros.org/rostopic

(2) Autopilot node

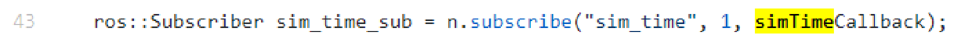
The Navigation Control module is located in the folder "navigation_control". In the "src" folder, you can find the "autopilot.cpp", which defines the autopilot node. The code is based on the fmi_adapter_node:
https://github.com/boschresearch/fmi_adapter/blob/melodic_and_noetic/fmi_adapter/src/fmi_adapter_node.cpp

I added the initialization of PID values by getting parameters from the launch file:



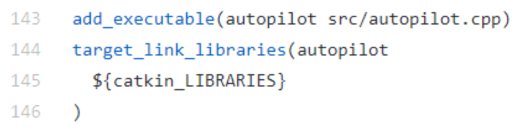
```
std::string strPID("PID");
if (!strPID.compare(autopilotType)) { // if the autopilot type is PID, set tunable parameters
  ROS_INFO("autopilotType is PID.");
  double P = 0.02;
  double I = 0.0;
  double D = 0;
  n.getParam("autopilot/P", P);
  n.getParam("autopilot/I", I);
  n.getParam("autopilot/D", D);
  adapter.setInitialValue("P.Value", P);
  adapter.setInitialValue("I.Value", I);
  adapter.setInitialValue("D.Value", D);
}
```

and aligned the clock of Gazebo simulation and the FMU:



```
ros::Subscriber sim_time_sub = n.subscribe("sim_time", 1, simTimeCallback);
```

If you want to build you own node, you also need to register the new file in the "CMakeLists.txt", as I've done for the autopilot node, and then build with "catkin_make":



```
add_executable(autopilot src/autopilot.cpp)
target_link_libraries(autopilot
  ${catkin_LIBRARIES}
)
```

(3) FMU files

You can find an "fmu" folder, which contains the FMU files for the demonstration. There are two sample FMU files: "PID_controller.fmu" and "nonlinear_controller.fmu". If you want to use your own FMU file, you can name it as "xxxxxx_controller.fmu", and you can direct the "autopilot" node to a specific FMU file in this folder by parsing the argument "autopilot" with roslaunch command.

The two sample FMU files are exported by MATLAB Simulink R2021a in Ubuntu 20.04. The original Simulink files can be found in the folder "simulink_model".

(4) Autopilot launch file

By calling the launch file, you can run the "autopilot" node. You can find the launch file in the "launch" folder, which contains the launch file that runs the "autopilot" node. The following figure shows the launch file of this node:

```xml
<?xml version="1.0"?>

<launch>
  <arg name="autopilot" default="nonlinear"/>s
  <arg name="step_size" default="0.001" />
  <node name="autopilot" pkg="navigation_control" type="autopilot" >
    <remap from="delta" to="u1" />
    <param name="autopilot_type" value="$(arg autopilot)" />
    <param name="fmu_path" value="$(find navigation_control)/fmu/$(arg autopilot)_controller.fmu" />
    <param name="step_size" value="$(arg step_size)" />

    <param name="P" value="0.03" />
    <param name="I" value="0.0001" />
    <param name="D" value="0.0" />

  </node>
</launch>
```

In the sample FMU file, the output variable is "delta". In the simulation, the ship dynamics plugin is subscribing to a topic named "u1". Thus, the variable name "delta" is remapped to "u1" by <remap>.

The argument "autopilot_type" will be part of the fmu_path, indicating the name of the FMU file. For example, when the "autopilot_type" is set to "PID", the launch file will look for an FMU file called "PID_controller.fmu" in the fmu folder.

If the autopilot type is "PID", the "P", "I", and "D" are the parameters to initialize the PID controller. You can change the values in the launch file directly, and the next time you run the simulation, the new values will be used to initialize the PID controller.

(5) Testing

After installation according to the installation guide, the demo simulation can be run by the command:

roslaunch autonoship_simulation autonoship_gazebo scenario:=scenario1 autopilot:=PID

The argument "scenario" tells the simulation which testing scenario to run, and the argument "autopilot" tells the simulation which FMU file to use as the autopilot controller. The default testing scenario is "scenario1", and the default autopilot is "nonlinear".

If you want to develop your own FMU controller and need the linear and angular position feedback of the ship, you can use a subscriber that listens to the topic "/ownship/twist", which is the ground truth of the state of the ownship in the simulation in the message type "geometry_msgs/Twist". If you need the linear speed of the ownship, subscribe to "/ownship/vel_x", which is in the message type "std_msgs/Float64". For more information about the message type "geometry_msgs/Twist", please look at: http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Twist.html.

3. Ship Dynamics with FMU

(1) Introduction to Gazebo simulator

The Autonoship simulator is built with Gazebo simulator (http://gazebosim.org/). ROS provides many sample simulations in Gazebo, and you can call ROS libraries in Gazebo plugins to communicate between the simulation and ROS programs by publishers and subscribers.

Gazebo uses different types of plugins to insert code into simulations (http://gazebosim.org/tutorials/?tut=plugins_hello_world). You can find tutorial for Gazebo plugins in ROS here: http://gazebosim.org/tutorials?tut=ros_gzplugins.

(2) Introduction to gazebo-fmi project

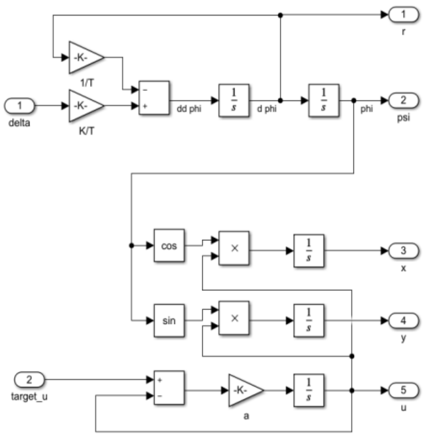
The FMU based ship dynamics plugin are built based on the gazebo-fmi project (https://github.com/robotology/gazebo-fmi). This project is a pure Gazebo related project and is not connected to ROS. In the Autonoship project, ROS functions are used to build connection between the Gazebo simulation and ROS program.

(3) Mechanism of FMU-based ship dynamics

The ship dynamics and sensor plugins are defined in the package "usv_gazebo_plugins". The ship dynamics model used in the simulation is the following:

$$
\begin{aligned}
x_1 &= \dot{\psi} & \qquad \dot{x}_1 &= -\frac{x_1}{T} + \frac{K}{T}\boxed{u_1} \\
x_2 &= \psi & \dot{x}_2 &= x_1 \\
x_3 &= x & \dot{x}_3 &= x_5 \cos\left(x_2\right) \\
x_4 &= y & \dot{x}_4 &= x_5 \sin\left(x_2\right) \\
x_5 &= v & \dot{x}_5 &= a\left(\boxed{u_2} - x_5\right)
\end{aligned}
$$

The FMU is exported by MATLAB Simulink as the following. You can also find the Simulink file in the folder "usv_gazebo_plugins/simulink_model".



For details of building FMUs with Simulink, please look at “Section 1 - Exporting FMU with Simulink”.

To use the FMU ship dynamics in the simulation, you need to call the FMI library in the Gazebo dynamics. The dynamics plugin is defined in “src/FMI_dynamics_plugin.cpp”. In the plugin, you need to declare all the variables of the FMU file as the following:

```
namespace FMIShipDynamicsPluginNS
{
  enum InputIndex
  {
    delta = 0,
    target_u,
    TotalInputs,
  };

  enum OutputIndex
  {
    r = 0,
    psi,
    x,
    y,
    u,
    TotalOutputs,
  };

  enum TunableParameterIndex
  {
    K = 0,
    T,
    a,
    dphi0,
    phi0,
    u0,
    x0,
    y0,
    TotalParameters,
  };
}
```

The order of the parameters should follow the same order in the model description file in the FMU. The tunable parameters are defined in the file “autonoship_simulation/autonoship_gazebo/urdf/FMI_gazebo_dynamics_plugin.xacro”:

```
        <ship_dynamics>
          <name>autonoship_FMI_dynamics</name>
          <link>base_link</link>
          <fmu>ShipWareDynamics.fmu</fmu>
          <tunable_parameter>
            <K>0.1555</K>
            <T>73.77</T>
            <a>0.034296296296</a>
            <dphi0>0.0</dphi0>
            <phi0>0.0</phi0>
            <u0>0.0</u0>
            <x0>0.0</x0>
            <y0>0.0</y0>
          </tunable_parameter>
        </ship_dynamics>
```

The <fmu> tag tells the plugin the name of the FMU file, which is located in the folder "usv_gazebo_plugins/fmu/". These initial values of tunable parameters will be read by the plugin as the SDF parameters as the following:

```
if (_sdf->HasElement("ship_dynamics")) {
  if (_sdf->GetElement("ship_dynamics")->HasElement("tunable_parameter")) {
    sdf::ElementPtr tunable_elem = _sdf->GetElement("ship_dynamics")->GetElement("tunable_parameter");
    K_ = getSdfParamDouble(tunable_elem,"K",K_);
    T_ = getSdfParamDouble(tunable_elem,"T",T_);
    a_ = getSdfParamDouble(tunable_elem,"a",a_);
    dphi0_ = getSdfParamDouble(tunable_elem,"dphi0",dphi0_);
    phi0_ = getSdfParamDouble(tunable_elem,"phi0",phi0_);
    u0_ = getSdfParamDouble(tunable_elem,"u0",u0_);
    x0_ = getSdfParamDouble(tunable_elem,"x0",x0_);
    y0_ = getSdfParamDouble(tunable_elem,"y0",y0_);
    ROS_INFO_STREAM("Set tunable parameters");
  }
}
```

The variable names are configured as the following:

```
// Configure default FMU variable names
m_fmu.m_inputVariablesDefaultNames.resize(FMIShipDynamicsPluginNS::TotalInputs);
m_fmu.m_inputVariablesDefaultNames[FMIShipDynamicsPluginNS::delta] = "delta";
m_fmu.m_inputVariablesDefaultNames[FMIShipDynamicsPluginNS::target_u] = "target_u";

m_fmu.m_tunableParameterDefaultNames.resize(FMIShipDynamicsPluginNS::TotalParameters);
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::K] = "K";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::T] = "T";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::a] = "a";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::dphi0] = "dphi0";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::phi0] = "phi0";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::u0] = "u0";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::x0] = "x0";
m_fmu.m_tunableParameterDefaultNames[FMIShipDynamicsPluginNS::y0] = "y0";

m_fmu.m_outputVariablesDefaultNames.resize(FMIShipDynamicsPluginNS::TotalOutputs);
m_fmu.m_outputVariablesDefaultNames[FMIShipDynamicsPluginNS::r] = "r";
m_fmu.m_outputVariablesDefaultNames[FMIShipDynamicsPluginNS::psi] = "psi";
m_fmu.m_outputVariablesDefaultNames[FMIShipDynamicsPluginNS::x] = "x";
m_fmu.m_outputVariablesDefaultNames[FMIShipDynamicsPluginNS::y] = "y";
m_fmu.m_outputVariablesDefaultNames[FMIShipDynamicsPluginNS::u] = "u";
```

After initializing the FMU, at each step of the simulation, you set the input values, run the simulation in the FMU for a step, and get the output values from the FMU:

```
// Set inputs
m_fmu.inputVarBuffers[FMIShipDynamicsPluginNS::delta] = u1_;
m_fmu.inputVarBuffers[FMIShipDynamicsPluginNS::target_u] = u2_;

bool ok = m_fmu.fmu.setInputVariables(m_fmu.inputVarReferences, m_fmu.inputVarBuffers);
```

```
// Run fmu simulation
ok = ok && m_fmu.fmu.doStep(simulatedTimeInSeconds - stepSizeInSeconds - initialization_time, stepSizeInSeconds);
```

```
// Get ouput
ok = ok && m_fmu.fmu.getOutputVariables(m_fmu.outputVarReferences, m_fmu.outputVarBuffers);

if (!ok)
{
  gzerr << "gazebo_fmi: Failure in simulating single body fluid dynamics forces of link " << link_->GetScopedName() << std::endl;
}

// This order should be coherent with the order defined in Load
psi_ = m_fmu.outputVarBuffers[FMIShipDynamicsPluginNS::psi];
r_ = m_fmu.outputVarBuffers[FMIShipDynamicsPluginNS::r];
u_ = m_fmu.outputVarBuffers[FMIShipDynamicsPluginNS::u];
x_ = m_fmu.outputVarBuffers[FMIShipDynamicsPluginNS::x];
y_ = m_fmu.outputVarBuffers[FMIShipDynamicsPluginNS::y];
```

(4) Update the state of the ship in Gazebo

After getting the output values, the values will be used to calculate the forces applied on the ship (setting the position and velocity of the ship directly in Gazebo sometimes is unstable):

```
  double angular_accel = (r_ - previous_r_)/dt;
  previous_r_ = r_;
  double linear_accel = (u_ - previous_u_)/dt;
  previous_u_ = u_;

// Distribute upward buoyancy force
  float buoy_force;
  double vel_z = vel_linear_body_.Z();
  buoy_force = (water_level_ - twist_.linear.z - 1.0*vel_z)*buoy_factor_;

  link_->AddRelativeForce(ignition::math::Vector3d(linear_accel*143620, u_*r_*143620, buoy_force));
  link_->AddRelativeTorque(ignition::math::Vector3d(0.0, 0.0, angular_accel*1357268841));
```

The buoyancy force is added to the ship. The force and torque are added to the ship (link) by the member functions of the class "Link": "AddRelativeForce" and "AddRelativeTorque". For more information about other member functions, please look at: https://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1physics_1_1Link.html#a83d38e22a943dee130965b034ac59537.

For example, if you want to look at the linear speed of the ship, you can call "RelativeLinearVel" and publish the value to ROS:

```
vel_x_pub_ = rosnode_->advertise<std_msgs::Float64>("vel_x", 1, true);
```

```
// Get body-centered linear and angular rates
vel_linear_body_ = link_->RelativeLinearVel();
```

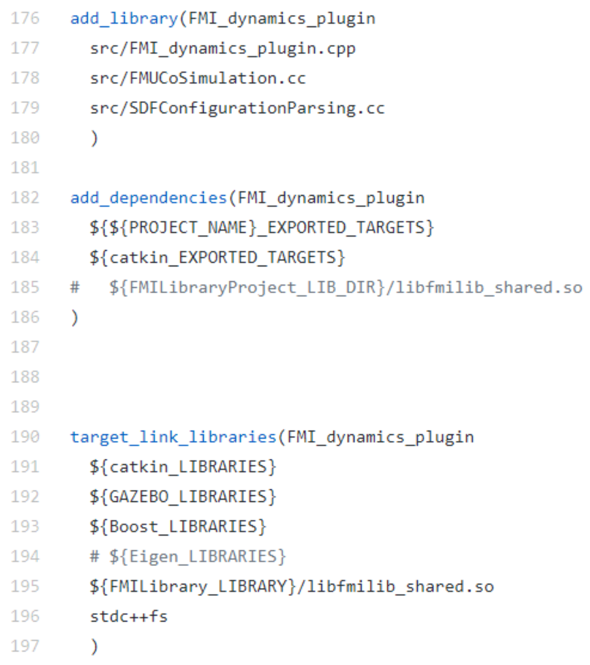
```
std_msgs::Float64 vel_x_;
vel_x_.data = vel_linear_body_.X();
vel_x_pub_.publish(vel_x_);
```

Then the linear speed of the ownship can be found in the topic "ownship/vel_x". Some other existing useful topics are "ownship/yaw" and "ownship/twist".

(5) Write your own dynamics plugin

If you want to use your own FMU file, you can put your FMU file in the folder "usv_gazebo_plugins/fmu/" and then modify the <fmu> tag in the file "autonoship_gazebo/urdf/FMI_gazebo_dynamics_plugin.xacro" to direct the simulation to your own FMU file.

If you want to write your own dynamics plugin, also remember to modify the "usv_gazebo_plugins/CMakeLists.txt". The following shows how the FMI_dynamics_plugin is compiled as a linked library. Then, use "catkin_make" to compile.



```
add_library(FMI_dynamics_plugin
  src/FMI_dynamics_plugin.cpp
  src/FMUCoSimulation.cc
  src/SDFConfigurationParsing.cc
  )

add_dependencies(FMI_dynamics_plugin
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS}
#   ${FMILibraryProject_LIB_DIR}/libfmilib_shared.so
)



target_link_libraries(FMI_dynamics_plugin
  ${catkin_LIBRARIES}
  ${GAZEBO_LIBRARIES}
  ${Boost_LIBRARIES}
  # ${Eigen_LIBRARIES}
  ${FMILibrary_LIBRARY}/libfmilib_shared.so
  stdc++fs
  )
```