Tutorials

Tutorial 1: free fall

python echo=False, results='raw', name='tutorial_01_load_yaml' yaml_data = load_yaml('tutorial_01_falling_ball.yml')

We start by defining the rotation conventions:

python echo=False, results='raw', name='tutorial_01_print_rotation_conventions' print_yaml(yaml_data, 'rotations convention')

Then we give environmental constants:

python echo=False, results='raw', name='tutorial_01_print_environmental_constants' print_yaml(yaml_data, 'environmental constants')

No environmental model (swell, wind, etc.) is required for this simulation:

python echo=False, results='raw', name='tutorial_01_print_environment_models' print_yaml(yaml_data, 'environment models')

We define the position of the reference “body” compared to the mesh:

python echo=False, results='raw', name='tutorial_01_print_position_of_body_frame' print_yaml(yaml_data, 'bodies/0/position of body frame relative to mesh')

The initial conditions are described as follows:

python echo=False, results='raw', name='tutorial_01_print_initial_conditions' print_yaml(yaml_data, 'bodies/0/initial position of body frame relative to NED') print_yaml(yaml_data, 'bodies/0/initial velocity of body frame relative to NED')

Dynamic data includes mass, inertia matrix, added inertias, and center of inertia position:

python echo=False, results='raw', name='tutorial_01_print_dynamics_section' print_yaml(yaml_data, 'bodies/0/dynamics')

Only gravity acts on the solid:

python echo=False, results='raw', name='tutorial_01_print_external_forces_section' print_yaml(yaml_data, 'bodies/0/external forces')

In the end, we get the following file:

python echo=False, results='raw', name='tutorial_01_print_full_yaml' print_yaml_file('tutorial_01_falling_ball.yml')

Launching the simulation

The simulation can run as follows:

python echo=False, results='raw', name='tutorial_01_launch_simulation_csv_output' execCommand('xdyn tutorial_01_falling_ball.yml --dt 0.01 --tend 1 -o out.csv')

To have output on the console, you can do:

python echo=False, results='raw', name='tutorial_01_launch_simulation_console_output' execCommand('xdyn tutorial_01_falling_ball.yml --dt 1 --tend 5 -o tsv')

tsv stands for “tab-separated values” here.

You can also change the initial time (it being understood that the initial conditions defined in the YAML file apply to this initial time, whatever it is, and not to t = 0):

python echo=False, results='raw', name='tutorial_01_sun_with_modified_initial_conditions' execCommand('xdyn tutorial_01_falling_ball.yml --dt 0.01 --tstart 2 --tend 3 -o out.csv')

We can choose the solver:

python echo=False, results='raw', name='tutorial_01_change_solver' execCommand('xdyn tutorial_01_falling_ball.yml --dt 0.01 --tend 1 -s rk4 -o out.csv')

The list of all options is available by running:

python echo=False, results='raw', name='tutorial_01_xdyn_help' execCommand('xdyn -h', echo_output=True)

Results

Here is a plot of elevation over time:

python echo=False, results='raw', name='tutorial_01_plot_results' data = csv('out.csv') plot = prepare_plot_data(data, x='t', y='z(ball)', name='Résultat') g = cartesian_graph([plot], x='t (s)', y='Élévation (m)') create_layout(g, title='Élévation au cours du temps')

Tutorial 2: oscillations in immersion

This tutorial aims to illustrate the use of hydrostatic models and to briefly compare non-linear hydrostatic (exact) and non-linear hydrostatic (fast) models .

python echo=False, results='raw', name='tutorial_02_load_yaml'
yaml_data_exact_hs = load_yaml('tutorial_02_exact_hydrostatic.yml')
yaml_data_fast_hs = load_yaml('tutorial_02_fast_hydrostatic.yml')

description of the problem

In this example, we consider a ship subjected only to gravity and hydrostatic forces, without damping. The ship is released without initial speed above the free surface (assumed flat) and will therefore perform undamped oscillations in immersion.

Writing the simulator configuration file

We document here only the changes compared to tutorial 1.

The environment is defined as follows:

python echo=False, results='raw', name='tutorial_02_print_environment_models'
print_yaml(yaml_data_exact_hs, 'environment models')

As described in the documentation of the input file, this means that the free surface is perfectly flat and horizontal, at the height *'z* = 0' in the NED coordinate system.

Compared to tutorial 1, the position of the “body” marker relative to the mesh is important here since an STL file is provided:

python echo=False, results='raw', name='tutorial_02_print_position_of_body'
print_yaml(yaml_data_exact_hs, 'bodies/0/position of body frame relative to mesh')

We describe in the initial conditions the fact that the boat is released at 5 m above the water level (the z axis of the NED frame being oriented downwards, negative values correspond to points above the free surface):

python echo=False, results='raw', name='tutorial_02_print_initial_conditions'
print_yaml(yaml_data_exact_hs, 'bodies/0/initial position of body frame relative to NED') print_yaml(yaml_data_exact_hs, 'bodies/0/initial velocity of body frame relative to NED')

Dynamic data includes mass, inertia matrix, added inertias, and center of inertia position:

python echo=False, results='raw', name='tutorial_02_print_dynamics_section'
print_yaml(yaml_data_exact_hs, 'bodies/0/dynamics')

We first use the approximate hydrostatic model whose documentation is described here:

python echo=False, results='raw', name='tutorial_02_print_external_forces'
print_yaml(yaml_data_fast_hs, 'bodies/0/external forces')

In the end, we get the following file:

python echo=False, results='raw', name='tutorial_02_print_full_yaml'
print_yaml(yaml_data_fast_hs)

Launching the simulation

The simulation can now be started as follows:

python echo=False, results='raw', name='tutorial_02_run_simulation' execCommand('xdyn
tutorial_02_fast_hydrostatic.yml --dt 0.1 --tend 10 -o fast.csv')
execCommand('xdyn tutorial_02_exact_hydrostatic.yml --dt 0.1 --tend 10 -o exact.csv')

Results

Here are the results :

python echo=False, results='raw', name='tutorial_02_plot_elevations' fast_data =
csv('fast.csv') exact_data = csv('exact.csv') fast_plot = prepare_plot_data(fast_data,
x='t', y='z(TestShip)', name='Modèle hydrostatique rapide') exact_plot =
prepare_plot_data(exact_data, x='t', y='z(TestShip)', name='Modèle hydrostatique
exact') g = cartesian_graph([fast_plot, exact_plot], x='t (s)', y='Élévation (m)')
create_layout(graph=g, title='Élévation au cours du temps')

We can also represent displacements along the *'y'* axis :

python echo=False, results='raw', name='tutorial_02_plot_y' fast_plot =
prepare_plot_data(fast_data, x='t', y='y(TestShip)', name='Modèle hydrostatique
rapide') exact_plot = prepare_plot_data(exact_data, x='t', y='y(TestShip)', name='Modèle hydrostatique
exact') g = cartesian_graph([fast_plot, exact_plot], x='t (s)', y='y (m)')
create_layout(graph=g, title='Embardée au cours du temps')

Tutorial 3: wave generation on a mesh

The simulator is intended to represent the behavior of solids in a fluid environment, but it can also be used to simulate an environment, without any solid. This can be interesting, for example, to generate wave fields in order to test wave motion prediction algorithms. This tutorial explains how to use the simulator for this type of simulation.

description of the problem

In this example, we will simulate an Airy swell consisting of the sum of two directional spectra:

- one with JONSWAP spectral density and mono-directional • the other monochromatic with cos2s dispersion

It is also assumed to have a depth of 100 m.

In this example, we are limited to two spectra, but the simulator makes it possible to sum as many as we wish (we are only limited by the memory of the machine and by the time available).

Writing the simulator configuration file

python echo=False, results='raw', name='tutorial_03_load_yaml' yaml_data = load_yaml('tutorial_03_waves.yml')

The environment models section is much more extensive than for previous tutorials .

We start by defining the discretization. Currently, the number of pulses is equal to the number of directions: this is a limitation of the code.

python echo=False, results='raw', name='tutorial_03_print_wave_discretization' print_yaml(yaml_data, 'environment models/0/discretization')

We will therefore sum <% yaml_data['environment models'][0]['discretization']['n'] %> pulsations and <% yaml_data['environment models'][0]['discretization']['n '] %> directions, i.e. <% yaml_data['environment models'][0]['discretization']['n']yaml_data['environment *models'][0]['discretization']['n'] % > dots. However, the spatial discretization of monochromatic spectra and monodirectional dispersions is reduced to one point. We also specify that we want to represent <%yaml_data['environment models'][0]['discretization']['energy fraction']100* %> % of the total energy, the other components not being retained.

The first spectrum is defined as follows:

python echo=False, results='raw', name='tutorial_03_print_first_spectrum' print_yaml(yaml_data, 'environment models/0/spectra/0')

For the second spectrum, we write:

python echo=False, results='raw', name='tutorial_03_print_second_spectrum' print_yaml(yaml_data, 'environment models/0/spectra/1')

The outputs are defined as follows:

python echo=False, results='raw', name='tutorial_03_print_outputs_section' print_yaml(yaml_data, 'environment models/0/output')

Ultimately, the environment is defined as follows:

python echo=False, results='raw', name='tutorial_03_print_environment_yaml' print_yaml(yaml_data, 'environment models')

As we do not simulate a body, the input file is reduced to:

python echo=False, results='raw', name='tutorial_03_print_full_yaml' print_yaml_file('tutorial_03_waves.yml')

Launching the simulation

The simulation can now be started as follows:

python echo=False, results='raw', name='tutorial_03_launch_simulation'
execCommand('xdyn tutorial_03_waves.yml --dt 1 --tend 1 -w tutorial_03_results.h5')

The result file is here tutorial_03_results.h5.

Results

We get an hdf5 file that can be opened with different software like HDFView. In the "outputs" group, there is a "waves" group which contains four data sets named t, x, y and z.

- t gives the time steps of the simulation • x gives the coordinates according to x of the points where the elevation is calculated. Each line corresponds to a time step.
- y gives the coordinates along y of the points where the elevation is calculated. Each line corresponds to a time step.
- z gives the elevation at the points defined by x and y. Each slice corresponds at a time step.

The description of this file is made in the documentation of YAML files.

Elevations can be obtained in any xdyn frame (NED or linked to a solid). If the coordinate system is linked to a solid, we obtain x and y coordinates that change over time.

Tutorial 6: propulsion

So far we have only simulated environmental stresses. In this tutorial, we simulate a thruster.

description of the problem

The vessel is operating in a swell-free environment. It is subject to the following five efforts:

- Gravity • Hydrostatic forces (rapid and not exact) • Viscous damping • Propulsion force due to a propeller • Resistance to progress

We can also be satisfied with only 2 efforts of resistance and propulsion.

Writing the simulator configuration file

Changes from tutorial 2 are the addition of damping and resistance forces, an external forces section and a

section commands.

We start by defining the characteristics of the thruster:

python echo=False, results='raw', name='tutorial_06_load_yaml' yaml_data = load_yaml('tutorial_06_1D_propulsion.yml')

python echo=False, results='raw', name='tutorial_06_print_external_forces_section' print_yaml(yaml_data, 'bodies/0/external forces')

The commands are defined at the root of the YAML:

python echo=False, results='raw', name='tutorial_06_print_commands_section' print_yaml(yaml_data, 'commands')

Ultimately, the input file is:

python echo=False, results='raw', name='tutorial_06_print_full_yaml' print_yaml(yaml_data)

Launching the simulation

The simulation can now be started as follows:

python echo=False, results='raw', name='tutorial_06_launch_simulation' execCommand('xdyn tutorial_06_propulsion.yml --dt 0.1 --tend 20 -o out.csv')

Results

Here is the time evolution of the forward speed:

python echo=False, results='raw', name='tutorial_06_plot_results' data = csv('out.csv') plot = prepare_plot_data(data, x = 't', y = 'u(TestShip)', name="Vitesse d'avance") g = cartesian_graph([plot], x='t (s)', y='U (m/s)') create_layout(graph=g, title="Vitesse d'avance longitudinale")

Tutorial 9: Using a remote wave model

This tutorial explains how to use an external swell model in xdyn.

We will use Docker compose to launch the client (xdyn) and the swell server. This is not mandatory (you can do without Docker and Docker Compose to make the whole thing work), but using Docker greatly simplifies the implementation.

Overview

For this tutorial, you need:

- a wave model

- an xdyn data entry

The wave model can be implemented in Python. In order to simplify its implementation , one can use the repository https://gitlab.sirehna.com/sirehna/demo_ docker_grpc which already contains a sample swell server in Python.

xdyn data setting

In a YAML file (named tutorial_09_gRPC_wave_model.yml in this example) we write:

python echo=False, results='raw', name='tutorial_09_load_yaml' yaml_data = load_yaml('tutorial_09_gRPC_wave_model.yml')

python echo=False, results='raw', name='tutorial_09_print_yaml' print_yaml(yaml_data)

Writing the wave model

In a Python file (named airy.py in this example) we write:

“‘python evaluate=False, results=‘hidden’ """“Airy wave model. As implemented in xdyn."""

import math import yaml import numpy as np import waves

def pdyn_factor(k, z, eta): return 0 if (eta != 0 and z < eta) else math.exp(-k * z)

class Airy(waves.AbstractWaveModel): def init(self): self.psi0 = None self.jonswap_parameters = {‘sigma_a’: 0.07, ‘sigma_b’: 0.09} self.directional_spectrum = {}

def set_parameters(self, parameters): param
    = yaml.safe_load(parameters)
    self.jonswap_parameters['t_p'] = param['Tp']
    self.jonswap_parameters['gamma'] = param['gamma']
    self.directional_spectrum['omega'] = param['omega']
    self.directional_spectrum['psi'] = \
        [param['waves propagating to']*math.pi/180]
    self.jonswap_parameters['hs_square'] = param['Hs']*param['Hs']
    self.jonswap_parameters['omega0'] = 2*math.pi/param['Tp']
    self.jonswap_parameters['coeff'] = 1-0.287*math.log(param['gamma'])
    self.directional_spectrum['si'] = [self.jonswap(omega) for omega in param['omega']]

    self.directional_spectrum['dj'] = [1]
    self.directional_spectrum['psi'] = [1]
    self.directional_spectrum['k'] = [omega*omega/9.81 for omega in
                                      param['omega']]
    phases = np.random.uniform(low=0,

```
                                                high=2*math.pi,
                                                size=(len(param['omega']),))
        self.directional_spectrum['phase'] = phases

def jonswap(self, omega): sigma_a
        = self.jonswap_parameters['sigma_a'] sigma_b =
        self.jonswap_parameters['sigma_b'] omega0 =
        self.jonswap_parameters['omega0'] hs_square =
        self.jonswap_parameters['hs_square'] coeff =
        self.jonswap_parameters['coeff'] gamma =
        self.jonswap_parameters['coeff'] sigma = sigma_a if
        omega <= omega0 else sigma_b ratio = omega0/omega alpha =
        ratio*ratio*ratio*ratio awm_5
        = coeff*5.0/16.0*alpha/omega*hs_square
        bwm_4 = 1.25*alpha kappa = (omega-omega0)/(sigma*omega0)
        return awm_5*math.exp(-
        bwm_4)*math.pow(gamma, math.exp(-0.5*kappa*kappa))



def elevation(self, x, y, t):
        zeta = 0
        dir_spec = self.directional_spectrum psi =
        dir_spec['psi'][0] for s_i, k, omega,
        phase in zip(dir_spec['si'], dir_spec['k'], dir_spec['omega'],
                                                dir_spec['phase']):
                                                k_x_cos_psi_y_sin_psi =
                                                k * (x * math.cos(psi) + y *
                math.sin(psi)) zeta -= s_i * math.sin(-omega*t + k_x_cos_psi_y_sin_psi + phase)

        return zeta

def dynamic_pressure(self, x, y, z, t):
        dir_spec = self.directional_spectrum eta =
        self.elevation(x, y, t) acc = 0 psi =

        dir_spec['psi'][0] for s_i, k, omega,
        phase in zip(['si'], dir_spec['k'], dir_spec['omega'],
                                                dir_spec['phase']):
                                                k_x_cos_psi_y_sin_psi =
                                                k * (x * math.cos(psi) + y *
                math.sin(psi)) acc -= s_i * pdyn_factor(k, z, eta)*math.sin(-omega*t


                                                        + k_x_cos_psi_y_sin_psi +
                                                        phase)
```

```
        return 1000*9.81*acc

def orbital_velocity(self, x, y, z, t):
        dir_spec = self.directional_spectrum eta =
        self.elevation(x, y, t) v_x = 0 v_y = 0 v_z
        = 0 psi =


        dir_spec['psi'][0] for s_i, k, omega,
        phase in zip([']; si'], dir_spec['k'], dir_spec['omega'],
                                                dir_spec['phase']):
                                                pdyn_factor =
                                                self.pdyn_factor(k, z, eta)
                pdyn_factor_sh = pdyn_factor k_x_cos_psi_y_sin_psi =
                k*(x*math. cos(psi) + y * math.sin(psi))
                theta = -omega * t + k_x_cos_psi_y_sin_psi + phase cos_theta =
                                                  math.cos(theta) sin_theta =
                math.sin(theta) a_affected = s_i * k / omega
                a_affected_pdyn_factor_sin_theta =
                a_affected *pdyn_factor\*sin_theta



                v_x += a_involved_pdyn_factor_sin_theta * math.cos(psi) v_y +=
                a_involved_pdyn_factor_sin_theta * math.sin(psi) v_z += a_involved *
                pdyn_factor_sh * cos_theta

        return {'vx': v_x, 'vy': v_y, 'vz': v_z}

def angular_frequencies_for_rao(self):
        return self.directional_spectrum['omegas']

def directions_for_rao(self): return
        self.directional_spectrum['psis']

def spectrum(self, x, y, t): return
        self.directional_spectrum

if name == 'main': waves.serve(Airy())
```

Launching the simulation

We start by retrieving the sample swell model:

```bash
git clone git@gitlab.sirehna.com:sirehna/demo_docker_grpc.git
```

We then write a docker-compose.yml file:

```
version: '3'
services:
  waves-server:
    build: waves_grpc/python_server entrypoint:
    ["/bin/bash", "/entrypoint.sh", "/work/airy.py"] working_dir: /work volumes:


    - .:/work - ./
    waves_grpc:/proto xdyn:
  image:
    sirehna/xdyn working_dir: /
    data entrypoint: xdyn
    tutorial_09_gRPC_wave_model.yml --dt 0.1 --tend 1 -o tsv volumes:

    - .:/data
    depends_on:
    - waves-server
```

We can then run the simulation as follows:

```
docker-compose up
```

Without Docker

If you don't use Docker, you have to start the swell server manually:

```
python3 airy.py
```

Then you have to edit the xdyn input YAML file by replacing:

```
environment models:
  - model: grpc
    url: waves-server:50051
```

about

```
environment models:
  - model: grpc url:
    localhost:50051
```

We can then launch xdyn normally:

```
./xdyn tutorial_09_gRPC_wave_model.yml --dt 0.1 --tend 1 -o tsv
```

Tutorial 10: Using a Remote Effort Model

This tutorial explains how to use an external effort model in xdyn.

We will use Docker compose to launch the client (xdyn) and the server (the effort model). This is not mandatory (you can do without Docker and

Docker Compose to make it all work), but using Docker greatly simplifies the implementation.

Overview

For this tutorial, you need:

- a force model (implemented in Python in this example) • an xdyn data entry (a YAML file)

xdyn data setting

In a YAML file (named tutorial_10_gRPC_force_model.yml in this example) we write:

python echo=False, results='raw', name='tutorial_10_load_yaml' yaml_data = load_yaml('tutorial_10_gRPC_force_model.yml')

python echo=False, results='raw', name='tutorial_10_print_yaml' print_yaml(yaml_data)

We create a file containing the commands for the gRPC model (tutorial_10_gRPC_force_model_commands.yml in this example):

python echo=False, results='raw', name='tutorial_10_load_yaml_commands' yaml_cmds = load_yaml('tutorial_10_gRPC_force_model_commands.yml')

python echo=False, results='raw', name='tutorial_10_print_yaml_commands' print_yaml(yaml_cmds)

The remote effort model connection is defined in the following section:

python echo=False, results='raw', name='tutorial_10_print_yaml_subsection' print_yaml(yaml_data, 'bodies/0/external forces')

- model: grpc tells xdyn that this is a remote force model • url: force-model:9902 gives the network address at which the force model can be reached. Using docker-compose here allows us to specify an address equal to the template name
- name: parametric oscillator is an arbitrary name that the user gives in his YAML file in order to be able to match any commands (commands section of the YAML file) to this force model.

All other lines are sent to the effort model as a parameter, without being interpreted by xdyn. In the present case, the model has two parameters *'k'* and *'c'* whose value is given once and for all at the start of the simulation.

Writing the effort model

This is a damped harmonic oscillator model:

F_x = -k\cdot x - c\cdot u
F_y = c\cdot \overline{v}
F_z = 0
M_x = 0
M_y = 0
M_z = 0

The force on the Y axis is proportional to the filtered speed. The definition of this filtering is done in the filtered states section of the xdyn YAML file.

In a Python file (named harmonic_oscillator.py in this example) we write:

“‘python evaluate=False, results=‘hidden’ """“Damped harmonic oscillator model."""

import yaml import grpcforce

class HarmonicOscillator(grpcforce.Model): def init(self): self.k = None self.c = None

def model_needs_wave_outputs(self):
return False

def set_parameters(self, parameters): param =
yaml.safe_load(parameters) self.k = param['k']
self.c = param['c']

def force(self, t, states, _, _, filtered_states): # The
index in brackets corresponds to the position in # the history of states (starting
with the most recent # value) and the corresponding instant is given by #
states.t(i) . force = {'Fx': -self.k*states.x(0) - self.c*states.u(0), 'Fy':
self.c*filtered_states.v
return {'forces': forces, ' extra outputs': {}}

if **name** == **‘main’:** grpcforce.serve(HarmonicOscillator())

Launching the simulation

We start by retrieving the sample swell model:

```bash
git clone git@gitlab.sirehna.com:root/xdyn.git
```

We then write a docker-compose.yml file:

```
version: '3'
services:
  force-model:
```

```
    build: xdyn/grpc_force_python_server entrypoint:
    ["/bin/bash", "/entrypoint.sh", "/work/harmonic_oscillator.py"] working_dir: /work volumes:


    - .:/work - ./
    xdyn:/proto xdyn:
  image:
    sirehna/xdyn working_dir: /
    data entrypoint: xdyn
    tutorial_10_gRPC_force_model.yml tutorial_10_gRPC_force_model_commands  volumes:

    - .:/data
    depends_on: -
    force-model
```

We can then run the simulation as follows:

docker-compose up

Without Docker

If you don't use Docker, you have to start the swell server manually:

python3 harmonic_oscillator.py

Then you have to edit the xdyn input YAML file by replacing:

```
external forces: -
  model: grpc url:
    force-model:9002
```

about

```
external forces:
  - model: grpc url:
    localhost:50051
```

We can then launch xdyn normally:

./xdyn tutorial_10_gRPC_force_model.yml tutorial_10_gRPC_force_model_commands.yml --dt 0.1

changequote({{',}}')

Tutorial 11: Using a Remote Controller

This tutorial explains how to use an external controller in xdyn.

We will use Docker compose to launch the client (xdyn) and the server (the effort model). This is not mandatory (you can do without Docker and Docker Compose to make the whole thing work), but using Docker greatly simplifies the implementation.

Overview

For this tutorial, you need:

- a controller (implemented in Python in this example) • an xdyn data entry (a YAML file)

xdyn data setting

In a YAML file (named tutorial_11_gRPC_controller.yml in this example) we write:

python echo=False, results='raw', name='tutorial_11_load_yaml' yaml_data = load_yaml('tutorial_11_gRPC_controller.yml')

python echo=False, results='raw', name='tutorial_11_print_yaml' print_yaml(yaml_data)

This example contains two controllers:

- a remote PID controller (gRPC) implemented in Python and clocked at 0.3 seconds
- an internal xdyn PID controller, clocked at 0.7 seconds

These controllers are, in principle, implemented in the same way (with the differences of programming language). Two heading instructions are given:

- 30° from t = 0 to t = 250 seconds •
  45° from t = 250 seconds of simulation

The remote controller section is:

python echo=False, results='raw', name='tutorial_11_print_yaml_subsection' print_yaml(yaml_data, 'controllers/0')

- type: grpc tells xdyn that it is a remote controller • url: pid:9002 gives
  the network address at which the controller can be reached. Using docker-compose here allows us to specify an address equal to the name of the model
  • name: portside controller is an
  arbitrary name that the user gives in his YAML file in order to be able to access any outputs of the controller.

All other lines are sent to the controller as a parameter, without being interpreted by xdyn. In this case, the model has gains and weights assigned to states (same parameterization as xdyn's PID model).

Controller write

In a Python file (named pid_controller.py in this example) we write

python evaluate=False, results='hidden' include({{pid_controller.py}})

Launching the simulation with docker-compose

We start by retrieving the sample swell model:

```
git clone git@gitlab.com:sirehna_naval_group/sirehna/interfaces.git
```

We then write a docker-compose.yml file:

```
include({{docker-compose.yml}})
```

This file was created to be used in the source folder of xdyn and therefore the path must be adapted by replacing context: ../../interfaces by context: interfaces.

We can then run the simulation as follows:

```
CURRENT_UID=$(id -u):$(id -g) docker-compose up
```

The CURRENT_UID=$(id -u):$(id -g) part is simply used to ensure that any files generated are generated with the permissions of the current user.

Launching the simulation without Docker

If you don't use Docker, you have to start the swell server manually:

```
python3 pid_controller.py
```

Then you have to edit the xdyn input YAML file by replacing:

```
controllers:
    - type: grpc
      name: portside controller url:
      pid:9002
```

about

```
controllers:
    - type: grpc
      name: portside controller url:
      localhost:9002
```

We can then launch xdyn normally:

```
./xdyn tutorial_11_gRPC_controller.yml --dt 0.1 --tend 1 -o tsv
```

Launching the simulation from the xdyn repository

The tutorial can be launched directly from the xdyn repository by running make from the grpc_tests/controller directory. The simulation will then be launched and a CSV file will be generated. The convergence of the heading controller is then evaluated (at 30° between 0 and 250 seconds, then at 45° between 250 and 500 seconds).

Tutorial 12: using PRECAL result files

This tutorial explains how to use files generated by PRECAL_R instead of HDB files for added masses.

Preparation: generation of PRECAL_R output files

The calcAmasDampCoefInfFreq flag must have the value true in the PRECAL_R input file (it is false by default). It is located in the section sim > parHYD > calcAmasDampCoefInfFreq. For more details, one can refer to the theoretical manual of PRECAL_R version 18.1.3 (Report No. 21447-7-RD, sections 2.3 and 2.4) and to its user manual (section 3.3.2, p. 25).

Configuration d'xdyn

If the PRECAL_R output file is called ONRT_SIMMAN.raodb.ini, the following bodies[0]/ dynamics/added mass matrix at the center of gravity and projected in the body frame section is used :

```
added mass matrix at the center of gravity and projected in the body frame:
    from PRECAL_R: ONRT_SIMMAN.raodb.ini
```

Here is the full example file:

python echo=False, results='raw', name='tutorial_12_load_yaml' yaml_data = load_yaml('tutorial_12_precal_r.yml') print_yaml(yaml_data)

In the output file of PRECAL_R, the section defining the mass matrix added to infinite frequency has the following appearance:

```
[added_mass_damping_matrix_inf_freq]

total_added_mass_matrix_inf_freq_U1_mu1 =
{ 0.110E+06,-0.888E-01,0.226E+06,-0.144E+00,0.270E+08,0.551E+01
-0.122E-01,0.344E+07,-0.563E-02,-0.113E+07,0.157E+02,0.497E+08
0.227E+06,-0.898E+00,0.129E+08,0.763E+01,0.844E+08,0.130E+01
0.183E+00,-0.123E+07,0.251E+01,0.498E+08,0.104E+03,0.338E+09
0.270E+08,0.106E+01,0.845E+08,-0.431E+02,0.119E+11,-0.341E+03
0.164E+01,0.497E+08,0.101E+02,0.345E+09,-0.390E+03,0.522E+10 }
```

Tutorial 13: Using Potential Code Results with Remote Effort Patterns (gRPC)

This tutorial explains how to use the results of potential codes (HDB or PRECAL_R format) in gRPC effort models.

Configuration d’xdyn

python echo=False, results='raw', name='tutorial_13_load_yaml' yaml_data = load_yaml('tutorial_13_hdb_force_model.yml') yaml_data['output'][0]['data'] = yaml_data['output'][0]['data'][1:3]

To use an HDB or PRECAL_R file, add the (optional) hdb (or, respectively, raodb) key to the gRPC effort section:

python echo=False, results='raw', name='tutorial_13_grpc_yaml' print_yaml(yaml_data['bodies'][0]['external forces'][0])

The complete file is:

python echo=False, results='raw', name='tutorial_13_print_full_yaml' print_yaml(yaml_data)

xdyn does not guarantee that the HDB or PRECAL_R files used in the YAML are consistent: it is quite possible to use different files for the internal efforts and for the external efforts.

In the effort model (here a Python code), we can write:

```
class HDBForceModel(force.Model):
    """Outputs data from HDB in extra_observations."""

    def __init__(self, _, body_name, pot): """Initialize
        parameters from gRPC's set_parameters."""  self.body_name = body_name
        self.pot = pot


    def get_parameters(self):
        """Parameter k is stiffness and c is damping."""  return
        {'max_history_length': 0, 'needs_wave_outputs': False,
                    'frame': self.body_name, 'x': 0, 'y': 0, 'z': 0, 'phi': 0, 'theta': 0, 'psi': 0,
                    'required_commands': []}

    def force(self, states,  _, __):
        """Force model."""
        extra_observations = {}
        extra_observations['Ma(0,0)'] = self.pot.Ma[0][0]  return {'Fx': 0, 'Fy':
        0, 'Fz': 0, 'Mx': 0, 'My':
                    0, 'Mz': 0,



                    'extra_observations': extra_observations }
```

```
if __name__ == '__main__':
    force.serve(HDBForceModel)
```

The data is provided once, during model initialization, in the third parameter of the effort model constructor. All data is stored in Numpy types (ndarray) to simplify and speed up numerical processing.

This information can be written to the xdyn output files using the following output section:

python echo=False, results='raw', name='tutorial_13_outputs'
print_yaml(yaml_data['output']) changequote({{',}}')

Tutorial 14: Using Filtered States

This tutorial explains how to use filtered states in remote effort models and how to write them to xdyn output files.

Configuration d'xdyn

python echo=False, results='raw', name='tutorial_14_load_yaml' yaml_data = load_yaml('tutorial_14_filtered_states.yml')

To use filtered states, we define them in the filtered states section, defined for each body at the same level as the name or position of body frame relative to mesh keys, for example:

python echo=False, results='raw', name='tutorial_14_filtered_states_yaml' y = {'filtered states': yaml_data['bodies'][0]['filtered states']} print_yaml(y)

You can retrieve the filtered states in the xdyn output files using the following output section:

python echo=False, results='raw', name='tutorial_14_outputs'
print_yaml(yaml_data['output'])

The complete file is:

python echo=False, results='raw', name='tutorial_14_print_full_yaml' print_yaml(yaml_data)

If a state is not in the filtered states section, its filtered value will be the same as its unfiltered value.

In a Python file (named filtered_force.py in this example) we write

python evaluate=False, results='hidden' include({{filtered_force.py}})

We then write a docker-compose.yml file:

include({{docker-compose-filtered-states.yml}})

We can then run the simulation as follows:

```
CURRENT_UID=$(id -u):$(id -g) docker-compose up
```

The CURRENT_UID=$(id -u):$(id -g) part is simply used to ensure that any files generated are generated with the permissions of the current user.

Simulation results can be seen in the filtered_states.csv file:

```
python echo=False, results='raw', name='tutorial_14_plots' data =
csv('filtered_states.csv') xplot = prepare_plot_data(data, x='t', y='x(TestShip)',
name='État x non filtré') xfilteredplot = prepare_plot_data(data, x='t',
y='x_filtered(TestShip)', name='État x filtré') gx = cartesian_graph([xplot, xfilteredplot],
x='t (s)', y='Cavalement (m)') create_layout(graph=gx, title='Position x au cours
 du temps') zplot = prepare_plot_data(data, x='t', y='z(TestShip)', name='État z non
filtré') zfilteredplot = prepare_plot_data(data, x='t', y='z_filtered(TestShip)',
name='État z filtré') gz = cartesian_graph([zplot, zfilteredplot], x='t (s)', y='Pillonement (m)')
create_layout(graph=gz, title='Position z au cours du temps')
```

We observe that :

- the values x_filtered and x are identical (since no filtering is defined for this degree of freedom) • the filtering is well taken into account in z