NEMO5 - Machine Learning demonstrator

In this tutorial we will run a hybrid experiment for NEMO5. Objective is to deploy the Python Machine Learning (ML) model proposed by Bodner et al. (BBZ24) within the standard ORCA2_ICE_PISCES config.

Prerequisites to the turorial:

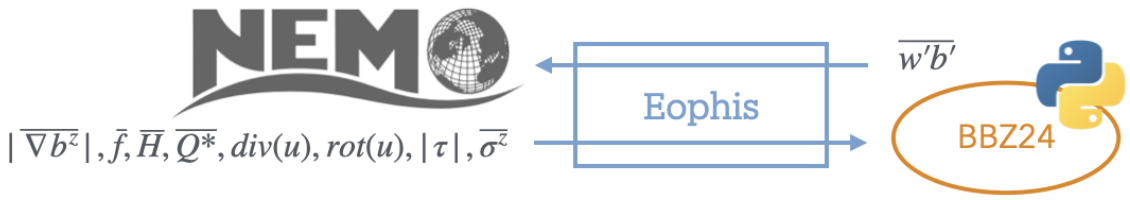
- Operating NEMO environment with XIOS (see NEMO doc for help)
- Operating Python environment (>= 3.10)

Introduction

ORCA2 is a reference configuration for the global ocean with a 2°x2° curvilinear horizontal mesh and 31 vertical levels. ORCA is the generic name given to global ocean Mercator mesh, with two poles in the northern hemisphere so that the ratio of anisotropy is nearly one everywhere. For this demonstrator, the configuration uses the ocean dynamical core OCE and the SI3 thermodynamic-dynamic sea ice model.

The work of BBZ24 takes place in the context of parameterizing the Mixed Layer Eddies (MLE) with a trained Convolutional Neural Network (CNN). The ML model infers the subgrid vertical buoyancy fluxes (VBF) from relevant surface large scale variables. We wish to use VBF from BBZ24 to compute the streamfunctions representing MLE-induced transport in NEMO.

Here is the difficulty: the ML model is written with native Python libraries while NEMO is written in Fortran. Thus, an interface is required to make both communicate and exchange data. Eophis is a Python library designed to ease the deployment and configuration of the OASIS API in Python scripts. Since NEMO has an OASIS interface, we can use Eophis to couple an external Python script to NEMO that will contain BBZ24 model.



The experiment will execute the following steps:

- NEMO is modeling ORCA2_ICE global circulation
- It sends the surface fields to a Python script that contains BBZ24
- BBZ24 computes subgrid VBF from received inputs
- Results are sent back to NEMO and written in an output file with XIOS
- VBF is used to compute MLE streamfunctions

1. Software environment

Compile OASIS_v5.0

OASIS is the coupling library on which both NEMO and Eophis rely to perform field exchanges. OASIS_v5.0 is the minimal required version and must be dynamically compiled. See OASIS documentation for more details.

```
# Clone OASIS_v5.0
cd ~/
```

```
git clone https://gitlab.com/cerfacs/oasis3-mct.git
cd ~/oasis3-mct
git checkout OASIS3-MCT_5.0
cd util/make_dir
```

Edit your own `make.<YOUR_ARCH>` file. Pay attention to the following important variables:

```
# Dynamic flags
DYNOPT = -fPIC
LDDYNOPT = -shared
# inc and lib dir
NETCDF_INCLUDE = /PATH/TO/NETCDF/include
NETCDF_LIBRARY = -L/PATH/TO/NETCDF/lib -lnetcdf -lnetcdff
MPI_INCLUDE = /PATH/TO/MPI/include
MPILIB = -L/PATH/TO/MPI/lib -lmpi
# Compilers and linker
F90 = # mpifort -I$(MPI_INCLUDE) , ftn , mpiifort ...
CC =  # mpicc , cc , mpiicc ...
LD = $(F90) $(MPILIB)
# Compilation flags - adapt with your compilers
FCBASEFLAGS = -O2 ...
CCBASEFLAGD = -O2 ...
```

```
# Link your architecture file for compilation
echo "include ~/oasis3-mct/util/make_dir/make.<YOUR_ARCH>"  >  make.inc

# Compile dynamic libraries
make -f TopMakefileOasis3 pyoasis

# Libraries should be there
ls ~/oasis3-mct/BLD/lib/
libmct.so   libmpeu.so   liboasis.cbind.so   libpsmile.MPI1.so   libscrip.so
```

Activate OASIS Python API. The best is to put this command in your `bash_profile`:

```
source ~/oasis3-mct/BLD/python/init.sh
```

Compile XIOS with OASIS

XIOS is used by NEMO to write results. It must be compiled with the abovementioned OASIS libraries. See XIOS documentation for more details about compilation of XIOS with OASIS.

The most important thing is to include OASIS libraries directories and bindings in `arch-<YOUR_MACHINE>.path`:

```
# edit arch files
vi archs/arch-<YOUR_MACHINE>.path
    # ...
    OASIS_INCDIR="-I~/oasis3-mct/BLD/include/ -I~/oasis3-mct/BLD/build-shared/lib/cbindings/"
    OASIS_LIBDIR="-L~/oasis3-mct/BLD/lib"
    OASIS_LIB="-loasis.cbind -lpsmile.MPI1 -lscrip -lmct -lmpeu"
```

and compile with OASIS flag:

```
./make_xios --full --prod --arch <YOUR_MACHINE> --use_oasis oasis3_mct
```

2. Experiment environment

NEMO and Python material are provided for this demonstrator. Start by cloning the following repository:

```
cd ~/
git clone https://github.com/morays-community/NEMO-ORCA2_MLE.git
```

The repository contains a README with informations about experiment context and motivations.

The other informations listed in README are the software requirements:

- **Compilation:** ocean code version to compile and with which potential additionnal material
- **Python:** Eophis version to install and potential additional Python material
- **Run:** submission tools to manage the experiment execution
- **Post-Process:** post-processing libraries and plotting tools

In accordance with README content, we must install *NEMO_v5.0.1*, *Eophis_v1.0.1*, BBZ24 package and download ORCA2 input files. Other tools and scripts are already contained in the repository.

```
# Clone NEMO_v5.0.1
cd ~/
git clone --branch 5.0.1 https://forge.nemo-ocean.eu/nemo/nemo.git nemo_v5.0.1

# Clone and install Eophis_v1.0.1
cd ~/
git clone --branch v1.0.1 https://github.com/meom-group/eophis eophis_v1.0.1
cd eophis_v1.0.1
pip install .

# README instructions for BBZ24 package
cd ~/NEMO-ORCA2_MLE/ORCA2.BBZ24/INFERENCES/BBZ24_MLE
pip install -e .

# README instructions for input files
cd ~/
wget "https://gws-access.jasmin.ac.uk/public/nemo/sette_inputs/r5.0.0/ORCA2_ICE_v5.0.0.tar.gz"
tar -xf ORCA2_ICE_v5.0.0.tar.gz
```

We will now browse the directories of the ORCA2_MLE.BBZ24 experiment to deploy the test case.

```
ls ~/NEMO-ORCA2_MLE/ORCA2_MLE.BBZ24/
CONFIG  INFERENCES  POST-PROCESS  RES  RUN
```

3. CONFIG - NEMO case

We prepare a new NEMO experiment following the standard configuration structure:

```
# Create NEMO test case
echo "ORCA2_MLE OCE ICE" >> ~/nemo_v5.0.1/cfgs/work_cfgs.txt
mkdir -p ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXPREF
mkdir -p ~/nemo_v5.0.1/cfgs/ORCA2_MLE/MY_SRC
```

A list of active CPP keys is given in the CONFIG directory:

```
# Copy CPP keys
cp ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/CONFIG/cpp_DINO_GZ21.fcm   ~/nemo_v5.0.1/cfgs/ORCA2_MLE/
```

An architecture file is compulsory to compile NEMO. A template for the experiment is also given in CONFIG. Depending on your hardware environment, it might not be suitable. Feel free to copy and edit it in accordance

with your machine. Be sure to have the OASIS and XIOS paths corresponding to those compiled with OASIS_v5.0 libraries.

Morays patch

README mentionned that NEMO must be patched with Morays sources. Those are the minimal NEMO modifications to create an Python communication module through OASIS. We can obtain them by cloning this repository:

```
cd ~/
git clone https://github.com/morays-community/Patches-NEMO.git
```

We transfer the Morays sources for NEMO_v5.0.0 to our custom test case. Only the sources of the OCE module are needed:

```
# Copy Morays sources
cp ~/Patches-NEMO/NEMO_v5.0.0/OCE/*    ~/nemo_v5.0.1/cfgs/ORCA2_MLE/MY_SRC/
```

Experiment patch

README also specified to patch NEMO with the experiment specific sources. Morays sources do not configure the Python communication module in accordance with ORCA2_MLE.BBZ24 but experiment patch does. They also contain the code to use the outsourced fields and are stored in CONFIG:

```
# Copy experiment sources
cp ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/CONFIG/my_src/*    ~/nemo_v5.0.1/cfgs/ORCA2_MLE/MY_SRC/
```

4. RUN - NEMO settings

This directory contains all the production material, such as XIOS configuration files and NEMO namelists. We need them obviously:

```
# Copy xml files and namelists
cp ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/RUN/NAMELISTS/*    ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXPREF/
cp ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/RUN/XML/*    ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXPREF/
```

We have everything we need to compile NEMO:

```
cd ~/nemo_v5.0.1
./makenemo -m "ORCA2_BBZ24_GCC" -r ORCA2_MLE -n "ORCA2_MLE" -j 8
```

In RUN directory is also contained the execution material for the experiment. Except job.ksh, no particular tool to manage the run is specified in README:

```
cp ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/RUN/job.ksh    ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXP00/
```

Script job.ksh assumes that NEMO will run on a HPC system via a SBATCH scheduler. Adapt script content or remove SBATCH header if necessary. Finally copy inputs file in NEMO execution directory.

```
cp ~/ORCA2_ICE_v5.0.0/*    ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXP00/
```

5. INFERENCES - Python material

This directory contains the Python scripts for hybrid modeling. It also includes additional packages for the Python model (if necessary). Here, BBZ24 model sources and weights are provided. The model is imported and used in accordance with the experiment objectives in `models.py`. Finally, `main.py` contains the Eophis instructions to couple `models.py` with NEMO. Let's copy them in the test case directory:

```
cp ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/INFERENCES/*.py   ~/nemo_v5.0.1/cfgs/ORCA2_MLE/MY_DINO_GZ21/EXP00/
```

Weights are stored in the `BBZ24_MLE` model folder. Adapt line 12 et 13 in `models.py` to get right path towards trained models and normalization files.

Since we already installed BBZ24 package in section **2. Experiment environment**, it may be tested by running `models.py` as a standalone script:

```
cd ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/INFERENCES/
python3 ./models.py
# Should print "Test successful"
```

Python sources description

Let's summarize coupling. Python needs to receive some surface fields from NEMO on ORCA2 grid `(180,148,1)` at each time step, and to send back results from BBZ24 at same frequency and on the same grid. BBZ24 model requires 3 extra halos cells at the edges of the inputs grid. Those must respect global grid boundary conditions (East-West periodic and folded at north pole). Exchanges and their constraints are specified in Eophis script `main.py` at line 14:

```
tunnel_config.append( { 'label' : 'TO_NEMO_FIELDS', \
                        'grids' : { 'ORCA2' : {'npts' : (180,148) , 'halos' : 3, 'bnd' : ('cyclic','NFold'), 'folding' : ('T','T')} }, \
                        'exchs' : [ {'freq' : 10800, 'grd' : 'ORCA2', 'lvl' : 1, \
                                     'in'  : ['grad_B','FCOR','HML','TAU','Q','div','vort','strain'], \
                                     'out' : ['w_b']} ] }
                    )
```

Here, ORCA2 grid is defined with desired number of halos and boundary treatment. The `exchs` line may be read as:

> every `10800` seconds, execute the receiving of field `grad_B`, `FCOR`, `HML`, `TAU`, `Q`, `div`, `vort`, `strain` and the sending back of fields `tau`, `w_b` on the first level of grid `ORCA2`.

Configuration of OASIS and MPI communications for those exchanges is automatically done by Eophis. From line 72 is defined how the fields to be sent and received are connected:

```
from models import vert_buoyancy_flux_CNN
# ...
def loop_core(**inputs):
    arrays = ( inputs['grad_B'] , inputs['FCOR'], -inputs['HML'], inputs['TAU'], \
                inputs['Q'], inputs['div'], inputs['vort'], inputs['strain']  )

    outputs = {}
    outputs['w_b'] = vert_buoyancy_flux_CNN( arrays , tmask=tmask )

    return outputs
```

`w_b` is the results of ML imported function in which all the received fields have been passed as arguments. Eophis can generate OASIS namelist `namcouple` with:

```
python3 ./main.py --exec preprod
```

Eophis log also produced useful informations to configure Python communication module in NEMO:

```
cat eophis.out
# [...]
-------- Tunnel TO_NEMO_FIELDS registered --------
namcouple variable names
Earth side:
- grad_B -> E_OUT_0
- FCOR -> E_OUT_1
- HML -> E_OUT_2
- TAU -> E_OUT_3
- Q -> E_OUT_4
- div -> E_OUT_5
- vort -> E_OUT_6
- strain -> E_OUT_7
- w_b -> E_IN_0
```

NEMO sources description

Coupling is defined in Python side. Let's now take a look on the NEMO sources we copied in `MY_SRC`. `extfld.F90` defines arrays in which fields returned from Python will be stored. They are available from anywhere in NEMO. `extcom.F90` is the Python communication module. It starts be defining IDs for fields to exchange:

```
! line 36
INTEGER, PARAMETER ::   jps_tmask = 1   ! t-grid mask
INTEGER, PARAMETER ::   jps_gradb = 2   ! depth-averaged buoyancy gradient magnitude on t-grid
INTEGER, PARAMETER ::   jps_fcor = 3    ! Coriolis parameter
INTEGER, PARAMETER ::   jps_hml = 4     ! mixed-layer-depth on t-grid
INTEGER, PARAMETER ::   jps_tau = 5     ! surface wind stress magnitude on t-grid
INTEGER, PARAMETER ::   jps_q = 6       ! surface heat flux
INTEGER, PARAMETER ::   jps_div = 7     ! depth-averaged horizontal divergence
INTEGER, PARAMETER ::   jps_vort = 8    ! depth-averaged vertical vorticity
INTEGER, PARAMETER ::   jps_strain = 9  ! depth-averaged strain magnitude
INTEGER, PARAMETER ::   jps_ext = 9  ! total number of sendings

INTEGER, PARAMETER ::   jpr_wb  = 1    ! depth-averaged subgrid vertical buoyancy flux on t-grid
INTEGER, PARAMETER ::   jpr_ext = 1    ! total number of receptions
```

Note that IDs names mirror those used in Eophis exchanges definition. Those IDs are used to set up OASIS configuration from NEMO side with previous Eophis informations:

```
! Line 142
! sending gradb, FCOR, HML, TAU, Q, div, vort, strain
ssnd(nmodext)%fld(jps_gradb)%clname = 'E_OUT_0'
ssnd(nmodext)%fld(jps_fcor)%clname = 'E_OUT_1'
ssnd(nmodext)%fld(jps_hml)%clname = 'E_OUT_2'
ssnd(nmodext)%fld(jps_tau)%clname = 'E_OUT_3'
ssnd(nmodext)%fld(jps_q)%clname = 'E_OUT_4'
ssnd(nmodext)%fld(jps_div)%clname = 'E_OUT_5'
ssnd(nmodext)%fld(jps_vort)%clname = 'E_OUT_6'
ssnd(nmodext)%fld(jps_strain)%clname = 'E_OUT_7'
ssnd(nmodext)%fld(jps_tmask)%clname = 'E_OUT_8'

! reception of vertical buoyancy fluxes
srcv(nmodext)%fld(jpr_wb)%clname = 'E_IN_0'
```

A bit further, contents of the fields that wil be sent to Python are defined:

```
! Line 190
! gradB
CALL calc_2D_scal_gradient( bz, zdatx, zdaty )
extsnd(jps_gradb)%z3(:,:,ssnd(nmodext)%fld(jps_gradb)%nlvl) = SQRT( zdatx(:,:)**2 + zdaty(:,:)**2 )
! FCOR
extsnd(jps_fcor)%z3(:,:,ssnd(nmodext)%fld(jps_fcor)%nlvl) = ff_t(:,:)
! HML
extsnd(jps_hml)%z3(:,:,ssnd(nmodext)%fld(jps_hml)%nlvl) = hmld(:,:)
```

Some of them are specifically computed for the test case and some others are standard already available NEMO fields. Finally, receptions and update of `extfld.F90` are done here:

```
! Line 273
! wb
ext_wb(:,:) = extrcv(jpr_wb)%z3(:,:,srcv(nmodext)%fld(jpr_wb)%nlvl)
```

The last important part is to call the Python communication module to perform sendings and receivings. This is done in `tramle.F90` during computation of MLE parameterization:

```
! Line 250
!                    !==  External computation of MLE ==!
CALL ext_comm( kt , 0, 0, 0, zmld, zbm, zum, zvm )
! [...]
zpsim_u(:,:) = ext_psiuf(:,:) * e2u(:,:)    ! replace by external values
zpsim_v(:,:) = ext_psivf(:,:) * e1v(:,:)
```

This implementation and use of the Python communication module is of course a template and can be modified in a finer way.

6. Running the experiment

Everything is ready to execute the hybrid experiment. Submit the run with `job.ksh` in accordance with your computing environment. If you are not running NEMO on a HPC system, execute commands below inside the experiment directory:

```
cd ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXP00/

# clean working directory
touch namcouple
rm namcouple*

# execute eophis in preproduction mode to generate OASIS namcouple
python3 ./main.py --exec preprod

# save eophis preproduction logs
mv eophis.out eophis_preprod.out
mv eophis.err eophis_preprod.err

# run coupled NEMO-Python
mpirun -np 1 ./nemo : -np 1 python3 ./main.py --exec prod
```

If run is going well, Eophis log should contain messages like these:

```
Iteration 6: 54000s -- 15:00:00
   Treating grad_B, FCOR, HML, TAU, Q, div, vort, strain received through tunnel TO_NEMO_FIELDS
   Sending back w_b through tunnel TO_NEMO_FIELDS
Iteration 7: 64800s -- 18:00:00
```

```
   Treating grad_B, FCOR, HML, TAU, Q, div, vort, strain received through tunnel TO_NEMO_FIELDS
   Sending back w_b through tunnel TO_NEMO_FIELDS
Iteration 8: 75600s -- 21:00:00
   Treating grad_B, FCOR, HML, TAU, Q, div, vort, strain received through tunnel TO_NEMO_FIELDS
   Sending back w_b through tunnel TO_NEMO_FIELDS
```

This means that the exchanges are well performed.

> **Note**
>
> If your computing environment is able to give you an access to a CUDA-compatible GPU, `models.py` will automatically execute the prediction on the GPU while NEMO and Eophis will be executed on CPUs.

7. POST-PROCESS and RES

POST-PROCESS directory contains material and/or scripts to compute and plot results. The latter are stored in the RES directory to be available for consultation. In this tutorial, Python scripts `plots_res.py` and `plots_diff.py` should be executed to make figures. Required dependencies are also given.

```
# install dependencies, if necessary
cd ~/NEMO-ORCA2_MLE/ORCA_MLE.BBZ24/POST-PROCESS/
pip install -r requirements.txt
cp plots_*  ~/nemo_v5.0.1/cfgs/ORCA2_MLE/EXP00/

# Plot
cd ~/nemo_v5.0.1/cfgs/ORCA2_MLE//EXP00/
python3 ./plots_res.py && python3 ./plots_diff.py
```

If everything went good, we should have similar figures than those stored in RES.

Going further

We invite you to read Eophis tutorial and configuration guide for NEMO to discover advanced coupling features.

This GitHub platform also hosts hybrid NEMO-ML experiments that can be deployed following the same demonstrator procedure. There are also tutorials to build hybrid NEMO experiments from scratch.