DFI in OOPS, Report on stay, Brussels, 17-21 Aug 2015

Martina Tudor

10. prosinca 2015.

Sadržaj

1 Abstract **1**

2 Introduction **1**

3 Some basics on the toy model **2**
3.1 The contents of the xml files 3
3.2 Plotting 4
3.3 Other OOPS source 4

4 DFI in OOPS toy model **5**

5 Code changes **5**
5.1 ExternalDFI.h 5

6 Appendix OOPS - Object Oriented Prediction System **8**

1 Abstract

This document describes some features of DFI (Digital Filter Initialization) as implemeted in the OOPS (Object oriented prediction system) toy model and attempts to widen its functionality to resemble the usage in ALADIN code.

The toy model described below including the modified code can be found on ecgate under the user hr4:

/home/ms/cr/hr4/oops

The original versions of the modified code have additional extension .orig. In the directory /home/ms/cr/hr4/oops/source/oops/src/oops/runs there are several versions of ExternalDFI.h. The one that was designed for backward-forward DFI is ExternalDFI.h.bf. The original one is ExternalDFI.h.orig. Version that does forward DFI and forecast or backward DFI and backward integration is ExternalDFI.h.borf (not much different than the original).

2 Introduction

ARPEGE/IFS code has become rather complex and consequently more difficult to maintain. The idea is to make it more simple by making the code more object oriented. This means that the routines at the control level would be (perhaps already are) rewritten in C(++) language. The purpose of this stay was to examine DFI in the OOPS layer of a toy model. The toy model is a quasi-geostrophic model that has plenty of features of the complete system but requires much less time to compute (and therefore test it). The features include 4DVAR, ensemble forecasts and DFI.

The idea was to enable the Toy model to compute the forecast using "standard" DFI as it is used in the ALADIN code. First backward integration, the filtering, then forward integration from the filtered state, then filtering again and then the usual forecast run. In aLADIN it is possible to use different timesteps for each of these 3 runs. Apparently, it is not possible in the toy model as I did not find tha way how to change the timestep in the execution of the toy model.

It is possible to run DFI forward and theno forecast forward using the same timestep. Attempt to run DFI with one timestep and forecast with another failed (it run the forecast using the timestep for DFI).

It is also possible to run DFI forward and integrade model backward in time using the same timestep. Just be careful to put negative forecast time, otherwise the model just might compute backwards indefinetly (it took some effort to kill the process).

3 Some basics on the toy model

There are 3 subdirectories in the directory of the toy model:

- source contains the source code for the quasigeostrophic model and the control routines in the oops layer.
- build contains executables, after code modivications, go to build directory and type "make -j3". The compilation will yield a number of executables. Apparently each mode of functionality is represented by a different executable.
- test_qg contains "namelists", here the experiments can be run and the input data are taken from a subdirectory and the output data are also stored into a subdirectory. To run tests with DFI, type "../build/bin/qg_dif.x dfi.xml" (path to the specific executable and the specific xml file).

This is the list of executables in the build/bin directory:

- l95_4dvar.x 4DVAR for the Lorenz 1995 model.
- l95_forecast.x run forecast using the Lorenz 1995 model.
- l95_genpert.x generate perturbations for the Lorenz 1995 model.
- l95_hofx.x the Lorenz 1995 model.
- l95_makeobs.x the Lorenz 1995 model.
- qg_4dvar.x 4DVAR for the quasi-geostrophic model.
- qg_dfi.x DFI for the quasi-geostrophic model.
- qg_forecast.x run forecast for the quasi-geostrophic model.
- qg_genpert.x generate perturbations for the quasi-geostrophic model.
- qg_makeobs.x for the quasi-geostrophic model.
- qg_test.x for the quasi-geostrophic model.

3.1 The contents of the xml files

Currently there are two xml files I have worked with. The truth.xml file contains set-up for a plain forecast. These are the contents and some explanations:

```
<config>
  <logging> # here we define what to print in the log and what not
    <categories>Info, Test, Warning, Error, Fatal</categories>
  </logging>

#  LOGS(Info, Test) means that the ouput shall be printed for each of the categories listed

  <resolution> # define the domain
    <nx>40</nx> # number of gridpoints in x direction
    <ny>20</ny> # number of gridpoints in y direction
    <bc>1</bc> # boundary condition (lateral)
  </resolution>

  <model> # define model configuration
    <tstep>PT10M</tstep> # this sis the timestep for the model forecast
    <top_layer_depth>   6000.0</top_layer_depth> # there are two layers in the m
odel
    <bottom_layer_depth>4000.0</bottom_layer_depth>
  </model>

  <initial> # define initial conditions
    <date>2009-12-15T00:00:00Z</date> # initial date
    <read_from_file>0</read_from_file> # is it read from file?
    <top_layer_depth>   6000.0</top_layer_depth> # there are two layers in the model
    <bottom_layer_depth>4000.0</bottom_layer_depth>
    <perturb>0</perturb>
  </initial>

  <forecast_length>P18D</forecast_length> # forecast is run for 18 days

  <output> # define writing of output files
    <frequency>PT12H</frequency> # how often - 12 hours
    <datadir>Data</datadir> # subdirectory
    <exp>truth</exp> # name of experiment: filenames begin with "truth"
    <date>2009-12-15T00:00:00Z</date> # filenam also contains this date (should be analysis
    <type>fc</type> # filename begins with ep.type, so truth.fc in this example.
  </output>

# filename truth.fc.2009-12-15T00:00:00Z.P9DT12H is for the exp, date and type as defined a

  <prints>
    <frequency>PT3H</frequency> # print output on screen with 3 hourly interval.
  </prints>

</config>
```

3.2 Plotting

There are also several python scripts in the directory oops/source/oops/qg/scripts to plot the output:

- qg/scripts/plotFields.py - plots a QG model state.
- qg/scripts/plotDiffs.py - plots the difference between two states.

I did not try to plot.

3.3 Other OOPS source

The main routines (there is a separate executable created for each of these programs) are in oops/source/oops/q subdirectory that contains:

- qgDFI.cc main program to run DFI, the usefull lines are

```
#include "mains/RunQg.h"
#include "model/QgTraits.h"
#include "oops/runs/ExternalDFI.h"

int main(int argc, char ** argv) {
  qg::RunQg< oops::ExternalDFI<qg::QgTraits> > run(argc, argv);
  int info = run.execute();
  return info;
};
```

- qgForecast.cc main program to run forecast (the source as above, if "ExtarnalDFI" is replaced by "Forecast".
- qgMakeObs.cc make observations (from forecast run actually).
- qgGenEnsPertB.cc generate perturbations for ensemble.
- qgMain4dvar.cc run 4DVAR.
- qgTest.cc run a test of the code.

The header files (the .h files) for these main programs are in directory oops/source/oops/src/oops/runs (sorry for so many oops-es, interesting directory tree). Perhaps it is useful to mention here that the rule that .h and .cc files have the same name is not obeyed for the main programs (also they are not stored in the same directory).

- Forecast.h to run a simple forecast run.
- ExternalDFI.h to run a DFI and a subsequent forecast.
- MakeObs.h make observations (from forecast run actually).
- GenEnsPertB.h generate perturbations for ensemble.
- EnsForecasts.h generate ensemble forecast.
- Variational.h run 4DVAR.

Other routines related to the oops layer are in directory oops/source/oops/src/oops/base and some of these have been modified (the original versions have .orig extension). Most of the modifications are related to additional prints.

The source code for the quasi-geostrophic model is in the directory oops/source/oops/qg/model. There are, however other places with the control routines and additional auxiliary functions. For example, routines that define the configuration, date and time are in oops/source/oops/src/util and some of them are in c++ and other in Fortran.

4 DFI in OOPS toy model

The model execution when DFI is invoked is controlled in the ExternalDFI.h file. The toy model was written so that it would run only forward DFI and then the forecast. Therefore the model would first integrate for the time equal to dfispan, filter the output and set it to time equal half the span. Then the forecast is run starting with these filtered fields and from time dfispan/2 until the forecast length. It is important to note that the toy model requires the forecast length to be longer than the filter span. Disabling this allowed backward DFI and then integrating the model backwards.

5 Code changes

Here some changes to the code are listed and explained. The code changes related to additional prins are ommited.

- ModelState.h `while (this->validTime() < end)` changed to `while (this->validTime() != end)`
- PostTimer.cc `if (now >= bgn_ && now <= end_)` condition was removed (put under comment).
- DolphChebyshev.cc `const int nstep = window.toSeconds() / dt.toSeconds();` changed to `const int nstep = abs(window.toSeconds() / dt.toSeconds());` and `ASSERT(tt > 1);` changed to `ASSERT(tt != 0);`
- WeightedMean.h One needs to reset the sum of weights to zero: `sum_ = 0;` `LOG(Info) << "Reset s`
  One also needs to compute the wights for the negative timestep so the following was introduced:

```
if (!linit_ && end <= bgn_ && bgn >= end_) {
  LOG(Info) << "bgn>=end=" << bgn_ << " weights computed using tstep=" << tstep;
    weights_ = wfct_->setWeights(end_, bgn_, -tstep);
    linit_ = true;
  }
```

5.1 ExternalDFI.h

First the model geometry is set-up (the part under resolution in the xml file).

```
    const util::Config resolConfig(fullConfig, "resolution");
    const Geometry_ resol(resolConfig);
```

then the model configuration is read (this is where the timestep is defined)

```
    const util::Config modelConfig(fullConfig, "model");
//  const ModelConfig_ model(resol, modelConfig); // have to declare it later
```

but the second line was commented to be able to set the timestep for DFI. Then the configuration for DFI is red (the dfi block in the xml file):

```
const util::Config dfiConf(fullConfig, "dfi");
```

Then the model configuration is copied and the DFI configuration is copied

```
util::Config modelConfigDfiBack(modelConfig); // not constant, because modified later
util::Config dfiConfBack(dfiConf); // not constant, because modified later
```

change the timestep of modelConfigDfiBack to the one from dfi in the xml file. This part is successful, the model is run with the timestep defined as tstep_back, but whatever I did it was not possible to run the model using any other timestep later (in forward DFI or forecast).

```
modelConfigDfiBack.setElement("tstep",dfiConf.getData("tstep_back"));

const ModelConfig_ model(resol, modelConfig);
ModelConfig_ modelDfiBack(resol, modelConfigDfiBack);
```

Setup initial state for the DFI backward integration from the initial file.

```
const util::Config initialConfig(fullConfig, "initial");
LOG(Configs) << "Initial configuration is:\n" << initialConfig;
ModelState<MODEL> xx(modelDfiBack, initialConfig);
LOGS(Info, Test) << "Initial state:" << xx;
```

Setup augmented state

```
ModelAuxCtrl_ moderr(initialConfig, resol);
```

Setup post-processing

```
PostProcessor<State_> post;

const util::Config prtConfig(fullConfig, "prints", true);
post.enrollProcessor(new StateInfo<State_>("fc", prtConfig));
```

Setup DFI is done in a way that the post processing setup is copied.

```
PostProcessor<State_> pp(post);
```

The initial time for DFI run, timestep and the date at which the filtered output is valid are computed: dfitimm is for the backward DFI and dfitime for the forward one.

```
const util::DateTime bgndfi(xx.validTime());
const util::Duration dfispan(dfiConf.getData("filter_span"));
const util::Duration dfistep(dfiConf.getData("tstep_back"));
const util::DateTime dfitimm(bgndfi-dfispan/2);
const util::DateTime dfitime(dfitimm+dfispan/2);
LOG(Info) << "dfispan:" << dfispan << " dfitime:" << dfitime
          << " dfitimm:" << dfitimm << " dfistep:" << dfistep;
boost::shared_ptr< WeightedMean<MODEL, State_> >
  pdfi(new WeightedMean<MODEL, State_>(dfitimm, -dfispan, resol, dfiConf));
pp.enrollProcessor(pdfi);
```

Run DFI forecast backward

```
xx.forecast(moderr, -dfispan, pp);
```

Retrieve initialized state from backward DFI for forward DFI.

```
boost::scoped_ptr<State_> xdfi(pdfi->releaseMean());
LOGS(Info, Test) << "Filtered state:" << *xdfi;
```

Repeat the excersize for the forward DFI.

```
const util::Config fdfiConf(fullConfig, "fdfi");
util::Config modelConfigDfiFor(modelConfig); // not constant, because modified later
util::Config dfiConfFor(fdfiConf); // not constant, because modified later

modelConfigDfiFor.setElement("tstep",fdfiConf.getData("tstep_forward"));
ModelConfig_ modelDfiFor(resol, modelConfigDfiFor);

const util::Duration dfistef(fdfiConf.getData("tstep_forward"));
LOG(Info) << "dfispan:" << dfispan << " dfitime:" << dfitime
          << " dfitimm:" << dfitimm << " dfistep:" << dfistef;

boost::shared_ptr< WeightedMean<MODEL, State_> >
  rdfi(new WeightedMean<MODEL, State_>(dfitime, dfispan, resol, fdfiConf));
pp.enrollProcessor(rdfi);
```

Run DFI forecast

```
ModelState<MODEL> yy(*xdfi, model);
yy.forecast(moderr, dfispan, pp);
```

is fails during the forecast because it is actually running backwrds since ethe timestep remained negative. Retrieve initialized state after forward DFI for the forecast

```
boost::scoped_ptr<State_> ydfi(rdfi->releaseMean());
LOGS(Info, Test) << "Filtered state:" << *ydfi;

ModelState<MODEL> zz(*ydfi, model);
```

Setup times for the forecast run

```
const util::Duration fclength(fullConfig.getData("forecast_length"));
util::DateTime bgndate(zz.validTime());
const util::DateTime enddate(bgndate + fclength);
LOG(Info) << "Running forecast from " << bgndate << " to " << enddate;
```

Setup forecast outputs

```
const util::Config outConfig(fullConfig, "output");
post.enrollProcessor(new StateWriter<State_>(bgndate, outConfig));
```

and finally, run the forecast

```
zz.forecast(moderr, fclength, post);
LOGS(Info, Test) << "Final state:" << zz;
```

Except the timestep, there are "bgn" and "end" variables set and reset at different places so when the forward run started at one point they assume the values for backward DFI and then turn back to the values for the forward one.

6 Appendix OOPS - Object Oriented Prediction System

Object oriented programing includes several essential elements, such as encapsulation, inheritance and polymorphism.

Encapsulation basically groups similar stuff into one unit. It can be done in a simple way as to assign a name to some constant or function or more complicated high level way as to define a class that consists of the variables and the methods that can be applied to them. Private variables can be modified only by the methods from the same class and public data can be accessed from other classes. However there are also protected data.

Inheritance means that one object (a class with data and methods) can acquire properties from another object (with more general data and methods).

Polymorphism means that one name can be used for a set of functions. An object can have some default behaviour but this can be manipulated at run-time when the specific method is determined.

In the toy model a class is defined in a header file with an extension .h, and if there are any methods associated with it, these are defined in a .cc file with the same name.