SICOPOLIS

Ralf Greve, Shreyas Sunil Gaikwad, Sri Hari Krishna Narayanan, L

Mar 25, 2023

CONTENTS:

CHAPTER

ONE

FILES AND DIRECTORIES IN SICOPOLIS

- `runs` :
  1. Configuration file `sico_configs.sh`.
  2. Shell script (bash) `sico.sh` for running a single simulation.
  3. Shell scripts (bash) `multi_sico_1.sh` and `multi_sico_2.sh` for running multiple simulations by repeated calls of `sico.sh`.
  4. Subdirectory `headers`: specification files `sico_specs_{run_name}.h` ({run_name}: name of run) for a number of computationally rather inexpensive test runs.
     - `v5_vialov3d25`
       * 3-d version of the 2-d Vialov profile
       * SIA, resolution 25 km, $t = 0 \ldots 100\,\mathrm{ka}$.
       * Similar to the EISMINT Phase 1 fixed-margin experiment
       * citep{huybrechts_etal_1996}, but without thermodynamics. Instead, isothermal conditions with $T = -10^{\circ}$ C everywhere are assumed.
     - `v5_emtp2sge25_expA`
       * EISMINT Phase 2 Simplified Geometry Experiment A
       * SIA, resolution 25 km, $t = 0 \ldots 200\,\mathrm{ka}$ citep{payne_etal_2000}.
       * The thermodynamics solver for this run is the one-layer melting-CTS enthalpy scheme (ENTM), while all other runs employ the polythermal two-layer scheme (POLY) citep{greve_blatter_2016}.
     - `v5_grl16_bm5_ss25ka`
       * Greenland ice sheet, SIA, resolution 16 km, short steady-state run ($t = 0 \ldots 25\,\mathrm{ka}$) for modern climate conditions (unpublished).
     - `v5_ant40_b2_ss25ka`
       * Antarctic ice sheet without ice shelves, SIA, resolution 40 km, short steady-state run ($t = 0 \ldots 25\,\mathrm{ka}$) for modern climate conditions (unpublished).
     - `v5_grl20_b2_paleo21`
       * Greenland ice sheet, SIA, resolution 20 km, $t = -140 \ldots 0\,\mathrm{ka}$, basal sliding ramped up during the first 5 ka
       * modified, low-resolution version of the spin-up for ISMIP6 InitMIP citet{greve_etal_2017a}
     - `v5_ant64_b2_spinup09_init100a`

    * v5_ant64_b2_spinup09_fixtopo, v5_ant64_b2_spinup09 and v5_ant64_b2_future09_ctrl
    * Antarctic ice sheet with hybrid shallow-ice–shelfy-stream dynamics citep{bernales_etal_2017} and ice shelves (SSA)
    * Resolution 64 km, $t = -140.1 \ldots -140.0\,\mathrm{ka}$ for the init run without basal sliding (..._init100a)
    * $t = -140 \ldots 0\,\mathrm{ka}$ for the run with almost fixed topography (..._fixtopo)
    * basal sliding ramped up during the first 5 ka
    * $t = -0.5 \ldots 0\,\mathrm{ka}$ for the final, freely-evolving-topography part of the (..._spinup09),
    * $t = 0 \ldots 100\,\mathrm{a}$ for the constant-climate control run (..._future09_ctrl)
    * 64-km version of the spin-up and the constant-climate control run for ISMIP6 InitMIP; Greve and Galton-Fenzi (pers.comm.2017).
  - v5_asf2_steady and v5_asf2_surge
    * Austfonna, SIA, resolution 2 km, $t = 0 \ldots 10\,\mathrm{ka}$
    * Similar to citeauthor{dunse_etal_2011}'s (citeyear{dunse_etal_2011}) Exp. 2 (steady fast flow) and Exp. 5 (surging-type flow), respectively
  - v5_nmars10_steady, v5_smars10_steady
    * North-/south-polar cap of Mars, SIA, resolution 10 km, $t = -10\,\mathrm{Ma} \ldots 0$
    * Steady-state runs by citet{greve_2007b}
  - v5_nhem80_nt012_new
    * northern hemisphere, SIA, resolution 80 km, $t = -250 \ldots 0\,\mathrm{ka}$
    * Similar to run nt012 by citet{greve_etal_1999a}
  - v5_heino50_st
    * ISMIP HEINO standard run ST
    * SIA, resolution 50 km, $t = 0 \ldots 200\,\mathrm{ka}$ citep{calov_etal_2010}.

- src :
  - Directory that contains the main program file sicopolis.F90.
  1. Subdirectory subroutines/general : general subroutines, for any modelled domain.
  2. Subdirectory subroutines/ant : subroutines specific for the Antarctic ice sheet.
  3. Subdirectory subroutines/emtp2sge : subroutines specific for the EISMINT Phase 2 Simplified Geometry Experiments.
  4. Subdirectory subroutines/grl : subroutines specific for the Greenland ice sheet.
  5. Accordingly subdirectories subroutines/asf, nhem, scand, tibet, nmars and smars for Austfonna, the northern hemisphere, Scandinavia, Tibet and the north and south polar caps of Mars, respectively.
  6. Subdirectory subroutines/tapenade : AD specific subroutines and files.
  7. Subdirectory subroutines/xyz : Framework to create new domains, this directory is empty by default.
- sico_in :
  - Directory that contains input data files for SICOPOLIS.
  1. Subdirectory general : general input files, for any modelled domain.

2. Subdirectory `ant` : input files specific for the Antarctic ice sheet.
3. Subdirectory `emtp2sge` : input files specific for the EISMINT Phase 2 Simplified Geometry Experiments.
4. Subdirectory `grl`: input files specific for the Greenland ice sheet.
5. Accordingly subdirectories asf, nhem, scand, tibet, nmars and smars for Austfonna, the northern hemisphere, Scandinavia, Tibet and the north and south polar caps of Mars, respectively.
6. Subdirectory textbf{xyz}: Framework to create new domains, place your input files here.

- `test_ad` :
  - AD specific utilities and CI testing framework
- `sico_out` :
  - Empty directory into which output files of SICOPOLIS simulations are written.
- `docs` :
  - Documentation with quick-start manual, sphinx docs, JOSS paper, doxygen, etc.
- `tools` :
  - Tools to help with forward modeling, eg - `resolution_doubler` , `make_ismip_output` , etc.

1. Program `make_ismip_output`
   - Generating ISMIP output (see http://tinyurl.com/clic-ismip6) from the NetCDF time-slice files produced by SICOPOLIS
   - For simulation run `./tools.sh -p make_ismip_output -m run_name`
   - For further options, try `./tools.sh -h`
2. Program `resolution_doubler`
   - Doubling the horizontal resolution of a NetCDF time-slice output file produced by SICOPOLIS
   - For simulation run name, to be executed by `./tools.sh -p resolution_doubler -m run_name`
   - For further options, try `./tools.sh -h`
   - For example, run `v5_grl10_b2_paleo21` (10 km resolution) requires the resolution doubled output of run `v5_grl20_b2_paleo21` (20 km resolution) for $t = -9\mathrm{ka}$ as initial condition. In order to create it, execute the resolution doubler for run `v5_grl20_b2_paleo21` (i.e., with the option `-m v5_grl20_b2_paleo21`) and enter
     - Number of time-slice file (with leading zeros, 4 digits) > 0004
   - This will convert the original time-slice file `v5_grl20_b2_paleo210004.nc` to the resolution-doubled file `v5_grl20_b2_paleo21_dbl_0004.nc` that serves as initial conditions for run `v5_grl10_b2_paleo21` .

CHAPTER

TWO

USING AUTOMATIC DIFFERENTIATION

2.1 Contributors

- Shreyas Sunil Gaikwad - Principal developer and maintainer of SICOPOLIS-AD v2
- Laurent Hascoet - Developer of the open-source AD tool Tapenade
- Sri Hari Krishna Narayanan - Provided guidance on application of AD tools for both SICOPOLIS-AD v1 and SICOPOLIS-AD v2
- Liz Curry-Logan - Principal developer of SICOPOLIS-AD v1
- Ralf Greve - Developer of the SICOPOLIS ice sheet model
- Patrick Heimbach - PI of the NSF-supported project that funds this initiative

2.2 How to contribute?

SICOPOLIS-AD v2 is an open source project that relies on the participation of its users, and we welcome contributions. Users can contribute using the usual pull request mechanisms in git, and if the contribution is substantial, they can contact us to discuss gaining direct access to the repository.

If you think you've found a bug, please check if you're using the latest version of the model. If the bug is still present, then think about how you might fix it and file a ticket in the Gitlab issue tracker (you might need to request membership access on Gitlab, which we can approve). Your ticket should include: what the bug does, the location of the bug: file name and line number(s), and any suggestions you have for how it might be fixed.

To request a new feature, or guidance on how to implement it yourself, please open a ticket with a clear explanation of what the feature will do.

You can also directly contact Shreyas Gaikwad (shreyas.gaikwad@utexas.edu) for any of the above.

2.3 Introduction and features

Previously, OpenAD have been used to get the adjoint of the SICOPOLIS code (Logan et. al, 2020). The current implementation with Tapenade (SICOPOLIS-AD v2) has the following advantages over the previous implementation -

1. It is up-to-date with the latest SICOPOLIS code
2. The AD tool Tapenade is open-source and actively maintained
3. A new tangent linear code generation capability is introduced (Forward Mode)

- This is useful for adjoint validation, and from a physical perspective, also useful for Bayesian UQ and inverse modeling.

4. We are now able to deal with inputs in the NetCDF format
5. We have now correctly incorporated the external LIS solver, its tangent linear code, and its adjoint which improve the simulation of Antarctic ice shelves and Greenland outlet glaciers. (subject to more testing)
6. We leverage continuous integration and the pytest framework in order to track changes in the trunk that "break" the AD- based code generation - precludes the need for constant monitoring.
7. We "show" the entire code to Tapenade, including the initialization subroutines, thus avoiding cumbersome maintenance of subroutines OpenAD used to initialize for adjoint runs.
8. We have provided convenient Python scripts to make I/O with the differentiated variables easier.

In addition we also have the following previously available features -

1. The adjoint mode is available, like before along with the capability to do Finite Differences validation of the gradient computed using the adjoint mode.

The code has the following capabilities or possible applications -

1. Paleoclimatic inversions using the adjoint generated gradients as part of a model calibration exercise.
2. Uncertainty Quantification of calibrated parameters by leveraging the new tangent linear mode and adjoint mode.
3. Sensitivity analysis of various quantities of interest (QoI) to state parameters.
4. Optimal Experimental Design - where should sensors be placed such that the newly collected data optimally informs our uncertain parameters.

2.4 Installation

SICOPOLIS-AD v2 requires the installation of Tapenade as well as SICOPOLIS. It is mandatory to install the external libraries such as NetCDF, LIS to access the full functionality of the code, as well as git, to be able to clone and contribute to the repository.

2.4.1 Open source AD Tool TAPENADE

TAPENADE is an Automatic Differentiation Engine developed at Inria at Sophia Antipolis by the Tropics then Ecuador teams. TAPENADE takes as input a computer source program, plus a request for differentiation. TAPENADE builds and returns the differentiated source program, that evaluates the required derivatives.

Building TAPENADE (latest 3.16)

While the SICOPOLIS source files are prepared to generate adjoint sensitivities, they will not be able to do so without an operable installation of Tapenade. Fortunately the Tapenade installation procedure is straight forward.

We detail the instructions here, but the latest instructions can always be found here.

Prerequisites for Linux or Mac OS X

Before installing Tapenade, you must check that an up-to-date Java Runtime Environment is installed. Tapenade will not run with older Java Runtime Environment.

NOTE: Alternatively, a Tapenade version that works correctly with SICOPOLIS-AD v2 is always available in the `test_ad/tapenade_supported` directory.

Steps for Mac OS

Tapenade 3.16 distribution does not contain a fortranParser executable for MacOS. It uses a docker image from here. You need docker on your Mac to run the Tapenade distribution with Fortran programs. Details on how to build fortranParser is here. You may also build Tapenade on your Mac from the gitlab repository.

Steps for Linux

1. Read the Tapenade license.
2. Download tapenade_3.16.tar into your chosen installation directory *install_dir*.
3. Go to your chosen installation directory *install_dir*, and extract Tapenade from the tar file :

```
% tar xvfz tapenade_3.16.tar
```

4. On Linux, depending on your distribution, Tapenade may require you to set the shell variable `JAVA_HOME` to your java installation directory. It is often `JAVA_HOME=/usr/java/default`. You might also need to modify the `PATH` by adding the bin directory from the Tapenade installation. An example can be found *here*.

Prerequisites for Windows

NOTE: Although Tapenade can be built on Windows, SICOPOLIS requires a Unix-like system (e.g., Linux), as mentioned *here*.

Before installing Tapenade, you must check that an up-to-date Java Runtime Environment is installed. Tapenade will not run with older Java Runtime Environment. The Fortran parser of Tapenade uses cygwin.

Steps for Windows

1. Read the Tapenade license.
2. Download tapenade_3.16.zip into your chosen installation directory *install_dir*.

NOTE: Alternatively, a Tapenade version that works correctly with SICOPOLIS-AD v2 is always available in the `test_ad/tapenade_supported` directory.

3. Go to your chosen installation directory *install_dir*, and extract Tapenade from the zip file.
4. Save a copy of the `install_dir\tapenade_3.16\bin\tapenade.bat` file and modify `install_dir\tapenade_3.16\bin\tapenade.bat` according to your installation parameters:

replace `TAPENADE_HOME=..` by `TAPENADE_HOME="install_dir"\tapenade_3.16` replace `JAVA_HOME="C:\Progra~1\Java\jdkXXXX"` by your current java directory replace `BROWSER="C:\Program Files\Internet Explorer\iexplore.exe"` by your current browser.

NOTE: Every time you wish to use the adjoint capability of SICOPOLIS-AD, you must re-source the environment. We recommend that this be done automatically in your bash or c-shell profile upon login. An example of an addition to a `.bashrc` file from a Linux server is given below. Luckily, shell variable `JAVA_HOME` was not required to be explicitly set for this particular Linux distribution, but might be necessary for some other distributions.

```
##set some env variables for SICOPOLIS tapenade

export TAPENADE_HOME="/home/shreyas/tapenade_3.16"
export PATH="$PATH:$TAPENADE_HOME/bin"

##Modules

module use /share/modulefiles/
module load java/jdk/16.0.1 # Java required by Tapenade
```

You should now have a working copy of Tapenade.

For more information on the tapenade command and its arguments, type :

```
tapenade -?
```

2.4.2 Ice sheet model SICOPOLIS

Prerequisites for SICOPOLIS

The official SICOPOLIS Quick Start Manual is generally excellent to get started with SICOPOLIS. However, we will mention some steps here since using the Automatic Differentiation (AD) capabilities with Tapenade requires a slightly modified setup.

The satisfaction of the following prerequisites is highly recommended to access all the features of the code. Details can differ from the SICOPOLIS Quick Start Manual, since there are multiple ways to do things. We detail one of them here.

GNU GCC Compiler (gfortran+gcc) or Intel Compiler (ifort+icc)

We have tested the software on gfortran/gcc v5.4.0, v7.2.0 and v8.5.0, any intermediate versions should work just as well. We have also tested the software on ifort/icc v18.0.0 (however, it should be noted that we have not tested the external LIS solver with Intel compilers).

Git

The Git repository of SICOPOLIS is kindly hosted by the GitLab system of the Alfred Wegener Institute for Polar and Marine Research (AWI) in Bremerhaven, Germany. Front page here .

Cloning the latest develop or ad revision:

::
    git clone –branch develop https://gitlab.awi.de/sicopolis/sicopolis.git

::
    git clone –branch ad https://gitlab.awi.de/sicopolis/sicopolis.git

(Cloning with SSH instead of HTTPS is also available. See the above GitLab front page link for details.)

You should then have a new directory `sicopolis` that contains the entire program package.

LIS (1.4.43 or newer)

LIS installation is mandatory to use shallow shelf/shelfy stream dynamics in simulations. Install LIS as explained in the Quick Start Manual. The following commands might be helpful, they are written for the latest version at the time of writing.

```
wget https://www.ssisc.org/lis/dl/lis-2.0.30.zip
unzip lis-2.0.30.zip
cd lis-2.0.30
./configure --enable-fortran --prefix=/lis-2.0.30/installation/ --enable-shared && make &
↪& make check && make install
```

For AD purposes, we compile the code using the `src/MakefileTapenade` makefile. This makefile requires the following environment variables set -

1. `LISDIR` - The installation directory for the LIS version to be used. You can change this variable, either in the Makefile directly, or automatically in your bash or c-shell profile upon login (for example, `.bashrc`). Examples for both are shown here.

`src/MakefileTapenade`

```
ifndef LISDIR
LISDIR=/home/shreyas/lis-2.0.30/installation
endif
```

`.bashrc`

```
export LISDIR="/home/shreyas/lis-2.0.30/installation"
```

2. `LIBLIS` - Absolute path to `liblis.so`. By default in `src/MakefileTapenade`, it is `${LISDIR}/lib/liblis.so`. If you follow the original instructions to install LIS, this should work, else one can set it manually within `src/MakefileTapenade`.
3. `LIBLISFLAG` - Add directory using `-L` to be searched for `-llis`. By default in `src/MakefileTapenade`, it is `-L${LISDIR}/lib -llis`. If you follow the original instructions to install LIS, this should work, else one can set it manually within `src/MakefileTapenade`.
4. `LISFLAG` - This flag declares directories to be searched for LIS `#include` header file `lisf.h`, as well as defines the `BUILD_LIS` as a macro with value 1. By default in `src/MakefileTapenade`, it is `-DBUILD_LIS -I${LISDIR}/include/`. If you follow the original instructions to install LIS, this should work, else one can set it manually within `src/MakefileTapenade`.

NOTE: Some users have reported needing to extend their `LD_LIBRARY_PATH` with the location of `${LISDIR}/lib` in order to find `liblis.so.0`.

NetCDF (3.6.x or newer)

NetCDF installation is mandatory since it is a powerful library with widespread use for I/O with a machine-independent data format. Install NetCDF as explained in the Quick Start Manual. In some cases, for example while working on a shared server which uses a module manager or Docker container, thing have to be set up differently. `src/MakefileTapenade` needs either the `NETCDF_FORTRAN_DIR` macro set or both `NETCDF_F90_FLAG` and `LIB_NETCDF_F90_FLAG` set (see code snippet from `src/MakefileTapenade` here).

```
ifndef NETCDF_F90_FLAG
ifndef LIB_NETCDF_F90_FLAG
```

(continues on next page)

(continued from previous page)

```
ifdef NETCDF_FORTRAN_DIR
LIB_NETCDF_F90=${NETCDF_FORTRAN_DIR}/lib/libnetcdff.so
LIB_NETCDF_F90_FLAG=-L${NETCDF_FORTRAN_DIR}/lib -lnetcdff
NETCDF_F90_FLAG=-I${NETCDF_FORTRAN_DIR}/include/
endif
endif
endif
```

1. NETCDF_FORTRAN_DIR - The installation directory for netcdf-fortran. You can change this variable, either in the Makefile directly, or automatically in your bash or c-shell profile upon login (for example, .bashrc). Examples for both are shown here.

src/MakefileTapenade

```
ifndef NETCDF_FORTRAN_DIR
NETCDF_FORTRAN_DIR=/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4
endif
```

.bashrc

```
export NETCDF_FORTRAN_DIR="/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4"
```

2. LIB_NETCDF_F90 - Absolute path to libnetcdff.so. By default in src/MakefileTapenade, it is {NETCDF_FORTRAN_DIR}/lib/libnetcdff.so.
3. LIB_NETCDF_F90_FLAG - Add directory using -L to be searched for -lnetcdff. By default in src/MakefileTapenade, it is -L${NETCDF_FORTRAN_DIR}/lib -lnetcdff. See some examples below where this has to be set explicitly in case of a Docker container.
4. NETCDF_F90_FLAG - This flag declares directories to be searched for netcdf-fortran #include header files. By default in src/MakefileTapenade, it is -I${NETCDF_FORTRAN_DIR}/include/. See some examples below where this has to be set explicitly in case of a Docker container.

For a server that uses modules, you can load the relevant modules using commands like these (can also make permanent by adding to login script like .bashrc -

```
% module use /share/modulefiles/
% module load openmpi
% module load netcdf-fortran
% module load netcdf
```

You then have to give the NETCDF_FORTRAN_DIR macro to src/MakefileTapenade, either by adding to a login script or directly inside the makefile. If your system uses a module manager, you can query to find the exact directory location.

```
% module show netcdf-fortran
-------------------------------------------------------------
/opt/ohpc/pub/moduledeps/gnu-openmpi/netcdf-fortran/4.4.4:
-------------------------------------------------------------

whatis("Name: NETCDF_FORTRAN built with gnu toolchain ")
whatis("Version: 4.4.4 ")
whatis("Category: runtime library ")
whatis("Description: Fortran Libraries for the Unidata network Common Data Form ")
whatis("http://www.unidata.ucar.edu/software/netcdf/ ")
prepend_path("PATH","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/bin")
```

(continues on next page)

(continued from previous page)

```
prepend_path("MANPATH","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/share/man")
prepend_path("INCLUDE","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/include")
prepend_path("LD_LIBRARY_PATH","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/lib")
setenv("NETCDF_FORTRAN_DIR","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4")
setenv("NETCDF_FORTRAN_BIN","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/bin")
setenv("NETCDF_FORTRAN_LIB","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/lib")
setenv("NETCDF_FORTRAN_INC","/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4/include
↪")
help([[
This module loads the NetCDF Fortran API built with the gnu compiler toolchain.

Note that this build of NetCDF leverages the HDF I/O library and requires linkage
against hdf5 and the native C NetCDF library. Consequently, phdf5 and the standard C
version of NetCDF are loaded automatically via this module. A typical compilation
example for Fortran applications requiring NetCDF is as follows:

]])
```

In this case `NETCDF_FORTRAN_DIR=/opt/ohpc/pub/libs/gnu/openmpi/netcdf-fortran/4.4.4`.

For a Docker container, for example one with a `centos:8` distribution, and the `dnf` package manager, NetCDF is typically installed as follows -

```
RUN dnf install -y https://github.com/openhpc/ohpc/releases/download/v1.3.GA/ohpc-
↪release-1.3-1.el7.x86_64.rpm

# Add some packages
RUN dnf -y install epel-release
RUN dnf -y install dnf-plugins-core
RUN dnf config-manager --set-enabled powertools
RUN dnf -y install make which git
RUN dnf -y install diffutils
RUN dnf -y install vim
RUN dnf -y install autoconf automake
RUN dnf -y install valgrind-ohpc
RUN dnf -y install gnu8-compilers-ohpc
RUN dnf -y install gsl-gnu8-ohpc hdf5-gnu8-ohpc
RUN dnf -y install openmpi-devel
RUN dnf -y install bc wget zlib-devel perl-Digest-MD5
RUN dnf -y --enablerepo=powertools install netcdf-fortran netcdf-devel # NetCDF␣
↪installation
RUN dnf -y install netcdf-fortran-devel # NetCDF installation
```

In this case, you will find that the `./usr/lib64/gfortran/modules/netcdf.mod` exists in your docker environment. In this case, you can just directly set `NETCDF_F90_FLAG=-I/usr/lib64/gfortran/modules` either the makefile or the login script (no need to set `NETCDF_FORTRAN_DIR` macro).

You can also confirm that the files `/usr/lib64/libnetcdff.so*` and `/usr/lib64/libnetcdf.so*` exist, which means you have to set `LIB_NETCDF_F90_FLAG=-L/usr/lib64 -lnetcdff`.

The Quick Start manual, and these two cases should help cover most of the issues with the installation of NetCDF.

Unix-like system

A Unix-like system, e.g. Linux (Ubuntu, CentOS, Fedora, Redhat, etc.), MacOS is required to run both SICOPOLIS and SICOPOLIS-AD v2.

Setting up SICOPOLIS (latest v5.3)

The Git repository of SICOPOLIS is kindly hosted by the GitLab system of the Alfred Wegener Institute for Polar and Marine Research (AWI) in Bremerhaven, Germany.

- Front page: Front page: https://gitlab.awi.de/sicopolis/sicopolis/
- Cloning the latest `ad` (the branch most relevant to us) revision with Git:

```
git clone --branch ad \
https://gitlab.awi.de/sicopolis/sicopolis.git
```

Cloning with SSH instead of HTTPS is also available. See the above GitLab link for details.

- Tagged versions of SICOPOLIS (latest: v5.3, 2021-06-11) can be accessed from the archive.

More details can be found here.

SICOPOLIS and SICOPOLIS-AD v2 applications are built using a configuration header file in `runs/headers`. A typical user setup involves copying over example configuration files from `runs/headers/templates` (see below), and suitably modifying one of them for custom runs.

Initial configuration

In addition to the steps above, the following steps need to be performed from the root of the repository-

- Copy template header files from `runs/headers/templates` to `runs/headers` so that SICOPOLIS can read one of these header files for the simulations desired by the user. Also, one can modify them suitably for their own custom simulations. The original files are always stored in `runs/headers/templates` for reference. Run the following command from the root directory of the repository.

```
./copy_templates.sh
```

- Get the input data files needed for both Greenland and Antarctica. These files are stored on a server and needed for various inputs such as geothermal heat flux, physical parameters, height of the ice base and lithosphere, precipitation, definition of regions for heterogenous basal sliding, etc. Run the following command from the root directory of the repository.

```
./get_input_files.sh
```

- Locate the file `sico_environment.sh` in the directory `sicopolis/runs`, open it with a text editor, and replace the "Default" entry for `SICO_INSTITUTION` by the name of your institution (max. 256 characters). This is just for bookkeping purposes.

Now, you are ready to use SICOPOLIS-AD v2, as described in *Running SICOPOLIS-AD v2*!

2.5 Theory and tutorials

2.5.1 Some FAQs

Does the Finite Differences use AD at all?

The Finite Differences does not use AD at all, it simply relies on multiple non-linear forward model runs to calculate the gradient. Thus Tapenade is not even required to differentiate the code. To activate the finite differences capabilities, the `ALLOW_GRDCHK` flag must be set, either within the compilation command itself or in the header file.

Why use the Finite Differences when the adjoint is much faster?

The finite differences is supposed to be a validation test for the gradient generated using the adjoint mode. For fine grids, it only makes sense to use FD to validate the values of the adjoint at a few points since FD is expensive, requiring 2 non-linear forward model runs per evaluation of one component of the gradient.

2.5.2 Tutorials

Tutorial 1 : Writing your own tangent-linear and adjoint code

In this tutorial, we essentially follow the **Appendix A** of Marotzke et al 1999. The difference is that we will be doing this in Python. The paper does not give the tangent linear code. However, we have provided one here, since it is helpful to write the adjoint code. The tape (storage) for the adjoint is mimicked using a list object in Python as a stack.

We start by importing the only library we need, which is numpy.

[1]:
```
import numpy as np
```

Forward model

The forward model (not to be confused with tangent linear model) performs non-linear quadratic operations on the input $\mathbf{x}_0$, which has been copied to x. This is followed by "convective adjustment". $\mathbf{x}_0$ is our control variable, and for the given choice of $\mathbf{x}_0 = (1, 3, 3)$, the cost function fc evaluates to 40.25.

We call the function **forward_without_tape** because we are not storing the interim vectors to tape, which would then aid in adjoint calculations. Instead we have a separate function **forward** for that, which is illustrated further below.

[2]:
```
def forward_without_tape(x0 = np.array([1, 3, 3])):

    x = np.zeros(3)

    for i in range(3):
        x[i] = x0[i]

    y = x[0]**2

    for i in range(3):

        x[i] = y + x[i]**2
```

(continues on next page)

(continued from previous page)

```
    # Convective adjustment

    for i in range(2):

        if (x[i] < x[i+1]):
            x[i] = 0.5*(x[i] + x[i+1])
            x[i+1] = x[i]

    # Cost function

    fc = (x[0]-5.5)**2 + 2.*x[1] + 3.*x[2]

    return fc

print(f"Cost function for given x0: {forward_without_tape()}")
```

```
Cost function for given x0: 40.25
```

Getting the finite difference gradient from the Forward model

We can use finite differences with the central differencing scheme to get the gradient evaluated at the given value of $\mathbf{x}_0$. For a N dimensional control vector, the forward model needs to be called $2N$ times to get an approximate gradient. This is because we can only evaluate directional derivatives, which we need to do N times to get the gradient. Each directional derivative evaluation, in turn, requires 2 forward model evaluations.

Mathematically, this can be formulated as

$$(\nabla_{\mathbf{x}_0} fc(\mathbf{x}_0), \hat{\mathbf{e}}) \approx \frac{fc\,(\mathbf{x}_0 + \epsilon\hat{\mathbf{e}}) - fc\,(\mathbf{x}_0 - \epsilon\hat{\mathbf{e}})}{2\epsilon} + \mathcal{O}(\epsilon^2)$$

Here $(.,.)$ indicates the normal inner product of two discrete vectors in $\mathbb{R}^N$.

The left side is the directional derivative of fc at $\mathbf{x}_0$ in the direction $\hat{\mathbf{e}}$, while the right side is the central differences approximation of the directional derivative.

The choice of ϵ can be critical, however we do not discuss that here and simply choose a value $\epsilon = 0.001$. In general, one would perform a convergence analysis with a range of values of $\epsilon = 0.1, 0.01, 0.001, ...$

To get the first component of the gradient at $\mathbf{x}_0$, we choose $\epsilon = 0.001, \hat{\mathbf{e}} = \hat{\mathbf{e}}_1 = (1, 0, 0)$. To get the second component of the gradient at $\mathbf{x}_0$, we choose $\epsilon = 0.001, \hat{\mathbf{e}} = \hat{\mathbf{e}}_2 = (0, 1, 0)$. To get the third component of the gradient at $\mathbf{x}_0$, we choose $\epsilon = 0.001, \hat{\mathbf{e}} = \hat{\mathbf{e}}_3 = (0, 0, 1)$.

[3]:
```
eps = 0.001
x0 = np.array([1,3,3])
e1 = np.array([1,0,0])
e2 = np.array([0,1,0])
e3 = np.array([0,0,1])
```

[4]:
```
g1 = (forward_without_tape(x0 = x0 + eps*e1) - forward_without_tape(x0 = x0 - eps*e1))/
↪(2*eps)

print(f"First component of gradient for given x0: {g1:.2f}")
```

```
First component of gradient for given x0: 15.50
```

[5]:
```
g2 = (forward_without_tape(x0 = x0 + eps*e2) - forward_without_tape(x0 = x0 - eps*e2))/
↪(2*eps)

print(f"Second component of gradient for given x0: {g2:.2f}")
```

```
Second component of gradient for given x0: 10.50
```

[6]:
```
g3 = (forward_without_tape(x0 = x0 + eps*e3) - forward_without_tape(x0 = x0 - eps*e3))/
↪(2*eps)

print(f"Third component of gradient for given x0: {g3:.2f}")
```

```
Third component of gradient for given x0: 15.00
```

[7]:
```
from IPython.display import Markdown as md
md("Therefore the approximated finite differences gradient at $\mathbf{x}_0 = (1,3,3)$␣
↪is given by $\mathbf{g} = (%.2f, %.2f, %.2f)$"%(g1,g2, g3))
```

[7]: Therefore the approximated finite differences gradient at $\mathbf{x}_0 = (1, 3, 3)$ is given by $\mathbf{g} = (15.50, 10.50, 15.00)$

Tangent linear mode

We now illustrate how to write the tangent linear code for the forward model shown above. Mathematically, a tangent linear model is a linearization of a non-linear forward model.

If the forward model is $\mathbf{x}_{\text{new}} = A(\mathbf{x})$, where A can be non-linear, the linearized statement would read - $\mathbf{dx}_{\text{new}} = \mathbf{B}\mathbf{dx}$, where $\mathbf{B}(i,j) = \frac{\partial [A(\mathbf{x})]_i}{\partial \mathbf{x}_j}$ is the jacobian of the non-linear operator A.

An important subtlety here is that $\mathbf{B}$ depends on $\mathbf{x}$, and not $\mathbf{x}_{\text{new}}$. From a coding perspective, this means that the linearized statements are always called before the original non-linear ones. This is because we generally need the old values of the control vector for the linearized statement. If the non-linear statements are called first, they will generally update $\mathbf{x}$ to $\mathbf{x}_{\text{new}}$.

One would expect a linearized model to be cheaper computationally than its non-linear counterpart. However, the computations for the linearized model require variables from the original non-linear model and hence one has to evaluate the non-linear forward model too in order to evaluate the linearized forward model. This means the cost is approximately 2 times that of the original non-linear model.

This can be clearly seen below. For each statement in the non-linear forward model, the tangent linear model **forward_d** has 2 statements. It is helpful to look at each line of code as a general non-linear operation $\mathbf{x}_{\text{new}} = A(\mathbf{x})$, which we linearize to $\mathbf{dx}_{\text{new}} = \mathbf{B}\mathbf{dx}$. The jacobian matrices $\mathbf{B}$, which are functions of $\mathbf{x}$ are mentioned in the comments above the repsective code statements.

[8]:
```
def forward_d(x0 = np.array([1, 3, 3]), x0d = [1, 0, 0]):

    x = np.zeros(3)
    xd = np.zeros(3)



    for i in range(3):

        # [xd[i]]  = [0. 1.] [xd]
        # [x0d[i]] = [0. 1.] [x0d[i]]
```

(continues on next page)

(continued from previous page)

```
        xd[i] = x0d[i]
        x[i] = x0[i]

    # [yd]    = [0. 2*x[0]] [yd]
    # [xd[0]] = [0.     1.] [xd[0]]

    yd = 2*x[0]*xd[0]
    y = x[0]**2

    for i in range(3):

        # [xd[i]]  = [2*x[i]  1.] [xd[i]]
        # [yd]     = [0.      1.] [yd]

        xd[i] = yd + 2*x[i]*xd[i]
        x[i] = y + x[i]**2

    # Convective adjustment

    for i in range(2):

        if (x[i] < x[i+1]):

            # [xd[i]]   = [0.5 0.5] [xd[i]]
            # [xd[i+1]] = [0.   1.] [xd[i+1]]

            xd[i] = 0.5*(xd[i] + xd[i+1])
            x[i] = 0.5*(x[i] + x[i+1])

            # [xd[i]]   = [1. 0.] [xd[i]]
            # [xd[i+1]] = [1. 0.] [xd[i+1]]

            xd[i+1] = xd[i]
            x[i+1] = x[i]

    # [xd[0]] = [1.           0. 0. 0.] [xd[0]]
    # [xd[1]] = [0.           1. 0. 0.] [xd[1]]
    # [xd[2]] = [0.           0. 1. 0.] [xd[2]]
    # [fcd]   = [2*(x[0]-5.5) 2. 3. 0.] [fcd]

    fcd = 2*(x[0]-5.5)*xd[0] + 2.*xd[1] + 3.*xd[2]
    fc = (x[0]-5.5)**2 + 2.*x[1] + 3.*x[2]


    return fcd
```

The tangent linear model for a given $\mathbf{dx}_0$, represented by $x0d$ in the code above, returns a scalar value fcd, which is directional derivative in the direction $\mathbf{dx}_0$. To get the gradient, we have to evaluate the directional derivative in all of the basis directions $\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3$, just like we did for the finite differences gradient.

[9]:
```
(g1, g2, g3) = forward_d(x0d = [1, 0, 0]), forward_d(x0d = [0, 1, 0]), forward_d(x0d =␣
```

(continues on next page)

(continued from previous page)

```
↪[0, 0, 1])
```

[10]:
```
from IPython.display import Markdown as md
md("Therefore the tangent linear evaluated gradient at $\mathbf{x}_0 = (1,3,3)$ is given␣
↪by $\mathbf{g} = (%.2f, %.2f, %.2f)$"%(g1,g2, g3))
```

[10]: Therefore the tangent linear evaluated gradient at $\mathbf{x}_0 = (1, 3, 3)$ is given by $\mathbf{g} = (15.50, 10.50, 15.00)$

Adjoint mode

We now illustrate how to write the adjoint code for the forward model shown above. To better understand this, we consider a non-linear forward model $\mathbf{y} = A(\mathbf{x})$, where the non-linear operator A can be further seen as a composition of functions - $A = A_3 \circ A_2 \circ A_1$. This is helpful to see how the adjoint is reverse in the nature of its flow.

$$\therefore \mathbf{y} = A_3(A_2(A_1(\mathbf{x})))$$

One can even interpret A_1, A_2, A_3 as being individual lines of code. We also define the interim variables as $\mathbf{x}_1 = A_1(\mathbf{x}), \mathbf{x}_2 = A_2(A_1(\mathbf{x})) = A_2(\mathbf{x}_1), \mathbf{y} = A_3(A_2(A_1(\mathbf{x}))) = A_3(\mathbf{x}_2)$.

Let us assume that the cost function $\mathcal{J}$ is a function of $\mathbf{y}$, therefore $\mathcal{J} = \mathcal{J}(\mathbf{y})$. We wish to evaluate the sensitivity of $\mathcal{J}$ to $\mathbf{x}$, i.e $\nabla_{\mathbf{x}} J$, which can be evaluated using a simple chain rule. Both the LHS and RHS are row vectors here.

$$\mathbf{g}^T = \nabla_{\mathbf{x}}\mathcal{J}^T = \frac{d\mathcal{J}}{d\mathcal{J}} \nabla_{\mathbf{y}}\mathcal{J}^T \, \mathbf{B}_3(\mathbf{x}_2) \, \mathbf{B}_2(\mathbf{x}_1) \, \mathbf{B}_1(\mathbf{x}) = 1 \times \nabla_{\mathbf{y}}\mathcal{J}^T \, \mathbf{B}_3(\mathbf{x}_2) \, \mathbf{B}_2(\mathbf{x}_1) \, \mathbf{B}_1(\mathbf{x})$$

Here $\mathbf{B}_i$ is the jacobian matrix for A_i. There are two ways to calculate this sensitivity:

1. Take inner product of LHS and RHS by unit basis vectors $\hat{\mathbf{e}}_i$. Do this N times to get all the N components of the gradient.

$$\mathbf{g}_i = \nabla_{\mathbf{x}}\mathcal{J}^T \hat{\mathbf{e}}_i = 1 \times \nabla_{\mathbf{y}}\mathcal{J}^T \, \mathbf{B}_3(\mathbf{x}_2) \, \mathbf{B}_2(\mathbf{x}_1) \, \mathbf{B}_1(\mathbf{x}) \hat{\mathbf{e}}_i$$

This is exactly what the tangent linear model is doing. If you go back to the discussion of the tangent linear model, you see that we substitute $\mathbf{dx}$ as $\hat{\mathbf{e}}_i$ and get the directional derivative N times to get the gradient.

2. Transpose both the LHS and the RHS.

$$\mathbf{g} = \nabla_{\mathbf{x}}\mathcal{J} = \mathbf{B}_1^T(\mathbf{x}) \, \mathbf{B}_2^T(\mathbf{x}_1) \, \mathbf{B}_3^T(\mathbf{x}_2) \, \nabla_{\mathbf{y}}\mathcal{J} \times 1$$

We see the advantage here clearly, if we multiply starting from the right, we only ever have to do matrix vector multiplications, and we get the entire gradient in one single go. This is what the adjoint model does.

The reverse nature of the adjoint model is clearly seen here. In the non-linear forward model and tangent linear model, $A_1, \mathbf{B}_1$ act first respectively followed by $A_2, \mathbf{B}_2$, and finally $A_3, \mathbf{B}_3$. For the adjoint model, $\mathbf{B}_3^T$ acts first, followed by $\mathbf{B}_2^T$ and finally, $\mathbf{B}_1^T$. Furthermore, the adjoint calculation requires $\mathbf{y}$ first even though it is computed last in the forward mode, followed by $\mathbf{x}_2, \mathbf{x}_1, \& \ \mathbf{x}$.

Therefore, we need to run the non-linear forward mode, and store (push to a stack named tape) $\mathbf{x}, \mathbf{x}_2, \mathbf{x}_3, \mathbf{y}$ to tape, and then retrieve them (pop from a stack named tape) in the reverse order during the adjoint computation.

All of this can be generalized to $A = A_M \circ A_{M-1} ... \circ A_1$. If we view $\mathbf{x}_i = A_i(\mathbf{x}_{i-1})$ as one line of code, $d\mathbf{x}_i = \mathbf{B}_i(\mathbf{x}_{i-1}) d\mathbf{x}_{i-1}$ is the linearization of that one line of code and $\nabla_{\mathbf{x}_{i-1}}\mathcal{J} = \mathbf{B}_i^T \nabla_{\mathbf{x}_i}\mathcal{J}$ is the adjoint of that line of code. By definition, within the code $xib = \nabla_{\mathbf{x}_i}\mathcal{J}$. Therefore, $Jb = \frac{d\mathcal{J}}{d\mathcal{J}} = 1$.

Now we see why doing the Tangnt Linear code was useful. The jacobian matrices computed and illustrated in the comments there just need to be transposed and used for the adjoint code.

Forward mode with tape used for adjoint

We now define the forward model with tape functionality included, which is useful to compute the adjoint. The tape is essentially a stack (it actually has all the properties of a stack data structure such as being Last-In-First-Out (LIFO) and First-In-Last-Out (FILO)) that stores the vectors that are needed during the computation of adjoint, which has a reverse flow when compared to the forward and tangent linear models.

[11]:
```
def forward(x0 = np.array([1, 3, 3])):

    x = np.zeros(3)
    tape = []
    for i in range(3):
        x[i] = x0[i]

    y = x[0]**2

    tape.append(np.copy(x))

    for i in range(3):

        x[i] = y + x[i]**2

    # Convective adjustment

    for i in range(2):

        tape.append(np.copy(x))

        if (x[i] < x[i+1]):
            x[i] = 0.5*(x[i] + x[i+1])
            x[i+1] = x[i]

    tape.append(np.copy(x))

    fc = (x[0]-5.5)**2 + 2.*x[1] + 3.*x[2]

    return fc, tape

print(f"Cost function for given x0: {forward()[0]}")
print(f"Tape for given x0: {forward()[1]}")
```

```
Cost function for given x0: 40.25
Tape for given x0: [array([1., 3., 3.]), array([ 2., 10., 10.]), array([ 6.,  6., 10.]),␣
↪array([6., 8., 8.])]
```

Code for adjoint mode

In this case, all the adjoint variables are suffixed with b. The comments show the adjoint matrices used for the operations. They are exactly the transpose of the jacobian matrices mentioned in the tangent linear code comments. We pop the required value of $\mathbf{x}$ from tape whenever needed.

There is an important subtlety here as well. Sometimes, you have to avoid doing the matrix vector product shown in the comments in the exact same order. Look in the comments below where it says *LOOK HERE!* to understand why.

[12]:
```
_, tape = forward(x0 = np.array([1, 3, 3]))

def forward_b(x0 = np.array([1, 3, 3]), tape = tape):

    xb = np.zeros(3)
    x0b = np.zeros(3)
    yb = 0.0

    fcb = 1.0

    # [xb[0]] = [1. 0. 0. 2*(x[0]-5.5)] [xb[0]]
    # [xb[1]] = [0. 1. 0. 2.          ] [xb[1]]
    # [xb[2]] = [0. 0. 1. 3.          ] [xb[2]]
    # [fcb]   = [0. 0. 0. 0.          ] [fcb]

    x = tape.pop()

    xb[0] = xb[0] +  2*(x[0]-5.5) * fcb
    xb[1] = xb[1] +  2.           * fcb
    xb[2] = xb[2] +  3.           * fcb

    for i in range(1,-1,-1):

        x = tape.pop()

        if (x[i] < x[i+1]):

            # [xb[i]]   = [1. 1.] [xb[i]]
            # [xb[i+1]] = [0. 0.] [xb[i+1]]

            xb[i] = xb[i] + xb[i+1]
            xb[i+1] = 0.

            # [xb[i]]   = [0.5 0.] [xb[i]]
            # [xb[i+1]] = [0.5 1.] [xb[i+1]]
            # LOOK HERE!
            # We first compute xb[i+1] even though it is the second entry in the matrix.
            # This is because it depends on old value of xb[i].
            # If we compute xb[i] first, we lose that old value which is needed for␣
↪xb[i+1].

            xb[i+1] = 0.5*xb[i] + xb[i+1]
            xb[i]   = 0.5*xb[i]
```

(continues on next page)

(continued from previous page)

```
    x = tape.pop()

    for i in range(2,-1,-1):

        # [xb[i]]  = [2*x[i] 0.] [xb[i]]
        # [yb]     = [1.      1.] [yb]

        yb = yb + xb[i]
        xb[i] = 2*x[i]*xb[i]

    # [yb]    = [0.     0.] [yb]
    # [xb[0]] = [2*x[0] 1.] [xb[0]]

    xb[0] = xb[0] + 2*x[0]*yb
    yb    = 0.0

    for i in range(2,-1,-1):

        # [xb[i]]  = [0. 0.] [xb]
        # [x0b[i]] = [1. 1.] [x0b[i]]

        x0b[i] = x0b[i] + xb[i]
        xb[i]  = 0.0

    return x0b
```

As mentioned above, we have to store the required variables to tape by running the non-linear forward code first. Then we use this tape for our adjoint calculations.

[13]:
```
_, tape = forward(x0 = np.array([1, 3, 3]))
(g1, g2, g3) = forward_b(x0 = np.array([1, 3, 3]), tape = tape)
```

[14]:
```
from IPython.display import Markdown as md
md("Therefore the adjoint evaluated gradient at $\mathbf{x}_0 = (1,3,3)$ is given by $\
↪mathbf{g} = (%.2f, %.2f, %.2f)$"%(g1,g2, g3))
```

[14]: Therefore the adjoint evaluated gradient at $\mathbf{x}_0 = (1, 3, 3)$ is given by $\mathbf{g} = (15.50, 10.50, 15.00)$

We see that we get the same gradient using finite differences, tangent linear model and adjoint model. Adjoint model is very efficient computationally when the size of the control vector is extremely large.

Acknowledgments

Thanks to An T. Nguyen for fruitful discussions regarding this code.

Tutorial 2 : Using Tapenade for a mountain glacier model

In this tutorial, we will describe the steps to use Tapenade to develop the tangent linear and adjoint codes for a simple PDE-based mountain glacier model. The model itself is inspired from the book Fundamentals of Glacier Dynamics, by CJ van der Veen and has been used for glaciology summer schools before, for example here , although they used a different tool to develop the adjoint model.

As prerequisites, ensure you have Tapenade installed and gfortran compiler available.

Mountain glacier model

Model description

The equations for a simple, 1D mountain glacier are described here.

$$\frac{\partial H}{\partial t} = -\frac{\partial}{\partial x}\left(-D(x)\frac{\partial h}{\partial x}\right) + M$$

where

$$D(x) = CH^{n+2}\left|\frac{\partial h}{\partial x}\right|^{n-1}$$

and

$$C = \frac{2A}{n+2}(\rho g)^n$$

Here $H(x,t)$ is the thickness of the ice sheet, $h(x,t)$ is the height of the top surface of the ice sheet (taken from some reference height), $b(x)$ is the height of the base of the ice sheet (taken from the same reference height). These three parameters can be related as

$$H(x,t) = h(x,t) - b(x)$$

In essence it's a highly non-linear diffusion equation with a source term. Furthermore, the boundary conditions are given by

$$H_{\text{left}} = 0$$

and

$$H_{\text{right}} = 0$$

Model parameters

We assume that the basal topography is linear in x such that $\frac{\partial b}{\partial x} = -0.1$. M is the accumulation rate and is modeled to be constant in time, but varying linearly in space.

$$M(x) = M_0 - xM_1$$

where $M_0 = 4.0\,\mathrm{m/yr},\ M_1 = 0.0002\,\mathrm{yr}^{-1}$.

The acceleration due to gravity $g = 9.2\,\mathrm{m/s}^2$. The density of ice $\rho = 920\,\mathrm{kg/m}^3$. The exponent $n = 3$ comes from Glen's flow law. We take $A = 10^{-16}\,\mathrm{Pa}^{-3}\mathrm{a}^{-1}$. Our domain length is $L = 30\,\mathrm{km}$. The total time we run the simulation for is $T = 5000$ years.

Discretization and Finite Volumes solution

The equations above cannot be solved analytically. Here we discuss a Finite Volumes method to solve them. We use a staggered grid such that D is computed at the centres of the grid, and H, h are computed at the vertices.

$$D\left(x_{j+1/2}\right) = C\left(\frac{H_j + H_{j+1}}{2}\right)^{n+2}\left(\frac{h_{j+1} - h_j}{\Delta x}\right)^{n-1}$$

$$\phi_{j+1/2} = D\left(x_{j+1/2}\right)\frac{\partial h}{\partial x}, \text{ where } \frac{\partial h}{\partial x} = \frac{h_{j+1} - h_j}{\Delta x}$$

$$\frac{\partial H_j}{\partial t} = -\frac{\phi_{j+1/2} - \phi_{j-1/2}}{\Delta x} + M(x_j)$$

Here, ϕ indicates flux and is just an intermediate variable.

We step forward in time using the simple Euler forward scheme. Here we choose $\Delta x = 1.0\,\text{km}, \Delta t = 1.0\,\text{month} = \frac{1}{12}\,\text{years}$.

What sensitivities are we interested in?

Our Quantity of Interest (QoI) is the total volume of the ice sheet after 5000 years (defined as `V` in the code). The control variables are the spatially varying accumulation rate. We are interested in the sensitivities of the QoI to the perturbations in the accumulation rate at various locations along the ice sheet. (defined as `xx` in the code)

Forward Model code

The code for the non-linear forward model is given here.

`forward.f90`

```
module forward

implicit none
     real(8), parameter :: xend = 30
     real(8), parameter :: dx = 1
     integer, parameter :: nx = int(xend/dx)


contains

     subroutine forward_problem(xx,V)

             implicit none

             real(8), parameter :: rho = 920.0
             real(8), parameter :: g = 9.2
             real(8), parameter :: n = 3
             real(8), parameter :: A = 1e-16
             real(8), parameter :: dt = 1/12.0
             real(8), parameter :: C = 2*A/(n+2)*(rho*g)**n*(1.e3)**n
             real(8), parameter :: tend = 5000
             real(8), parameter :: bx = -0.0001
             real(8), parameter :: M0 = .004, M1 = 0.0002
```

(continues on next page)

(continued from previous page)

```
            integer, parameter :: nt = int(tend/dt)
            real(8), dimension(nx+1,nt+1) :: h
            real(8), dimension(nx+1,nt+1) :: h_capital
            integer :: t,i
            real(8), dimension(nx+1) :: xarr
            real(8), dimension(nx+1) :: M
            real(8), dimension(nx+1) :: b
            real(8), dimension(nx) :: D, phi
            real(8), intent(in), dimension(nx+1) :: xx
            real(8), intent(out) :: V

            xarr = (/ ((i-1)*dx, i=1,nx+1) /)
            M = (/ (M0-(i-1)*dx*M1, i=1,nx+1) /)
            b = (/ (1.0+bx*(i-1)*dx, i=1,nx+1) /)
            M = M + xx
            h(1,:) = b(1)
            h(:,1) = b
            h(nx+1,:) = b(nx+1)
            h_capital(1,:) = h(1,:) - b(1)
            h_capital(nx+1,:) = h(nx+1,:) - b(nx+1)
            h_capital(:,1) = h(:,1) - b

            do t = 1,nt
                    D(:) = C * ((h_capital(1:nx,t)+h_capital(2:nx+1,t))/2)**(n+2) *␣
↪((h(2:nx+1,t) - h(1:nx,t))/dx)**(n-1)
                    phi(:) = -D(:)*(h(2:nx+1,t)-h(1:nx,t))/dx
                    h(2:nx,t+1) = h(2:nx,t) + M(2:nx)*dt - dt/dx * (phi(2:nx)-phi(1:nx-
↪1))

                    where (h(:,t+1) < b)
                            h(:,t+1) = b
                    end where

                    h_capital(:,t+1) = h(:,t+1) - b

            end do

            V = 0.


            open (unit = 2, file = "results_forward_run.txt", action="write",status=
↪"replace")
            write(2,*) "         #                H                h                   b"
            write(2,*) "_____________________________________________________________________
↪_____________"

            do i = 1, size(h_capital(:,nt+1))
                 V = V + h_capital(i,nt+1)*dx

                 write(2,*) i, "    ", h_capital(i,nt+1), "    ", h(i,nt+1), "    ",␣
↪b(i)
            end do
```

(continues on next page)

(continued from previous page)

```
     close(2)
     end subroutine forward_problem

end module forward
```

Finite Differences for validation

We can use finite differences with the forward differencing scheme (or alternatively, a more accurate central differencing scheme mentioned in *Tutorial 1*). to get the gradient evaluated at the given value of the control vector $\mathbf{x}$. For a N dimensional control vector, the forward model needs to be called $N+1$ times in the forward differencing scheme to get an approximate gradient. This is because we can only evaluate directional derivatives, which we need to do N times to get the gradient. Each directional derivative evaluation, in turn, requires 1 perturbed forward model evaluation and 1 unperturbed forward model evaluation (this can be done only once and stored).

Mathematically, this can be formulated as ($\mathcal{J}$ is the objective function, in our case the total volume denoted by `V` in the code)

$$(\nabla_{\mathbf{x}}\mathcal{J}(\mathbf{x}), \hat{\mathbf{e}}) \approx \frac{\mathcal{J}(\mathbf{x}+\epsilon\hat{\mathbf{e}}) - \mathcal{J}(\mathbf{x})}{\epsilon} + \mathcal{O}(\epsilon)$$

Here $(.,.)$ indicates the normal inner product of two discrete vectors in $\mathbb{R}^N$.

The left side is the directional derivative of $\mathcal{J}$ at $\mathbf{x}$ in the direction $\hat{\mathbf{e}}$, while the right side is the finite differences approximation of the directional derivative.

The choice of ϵ can be critical, however we do not discuss that here and simply choose a value $\epsilon = 1.e-7$. In general, one would perform a convergence analysis with a range of values of $\epsilon = 0.1, 0.01, 0.001, ...$

The code for the finite differences can be found in the *Driver Routine*.

Tangent Linear model

The tangent linear model is described in detail in *Tutorial 1*. Another excellent resource is the MITgcm documentation.

Generating Tangent Linear model code using Tapenade

It is pretty simple to generate the tangent linear model for our forward model.

```
% tapenade -tangent -tgtmodulename %_tgt -head "forward_problem(V)/(xx)" forward.f90
```

`-tangent` tells Tapenade that we want a tangent linear code.

`-tgtmodulename %_tgt` tells Tapenade to suffix the differentiated modules with `_tgt`.

`-head "forward_problem(V)/(xx)` tells Tapenade that the head subroutine is `forward_problem`, the dependent variable or the objective function is `V` and the independent variable is `xx`.

This generates 2 files -

- `forward_d.f90` - This file contains the tangent linear module (called `forward_tgt`) as well as the original forward code module. `forward_tgt`) contains the `forward_problem_d` subroutine which is our tangent linear model. All the differential variables are suffixed with the alphabet `d`.

```
!        Generated by TAPENADE     (INRIA, Ecuador team)
!  TAPENADE 3.15 (master) - 15 Apr 2020 13:12
!
MODULE FORWARD_TGT
  IMPLICIT NONE
  REAL*8, PARAMETER :: xend=30
  REAL*8, PARAMETER :: dx=1
  INTEGER, PARAMETER :: nx=INT(xend/dx)

CONTAINS
!  Differentiation of forward_problem in forward (tangent) mode:
!   variations   of useful results: v
!   with respect to varying inputs: xx
!   RW status of diff variables: v:out xx:in
  SUBROUTINE FORWARD_PROBLEM_D(xx, xxd, v, vd)
    IMPLICIT NONE
    REAL*8, PARAMETER :: rho=920.0
    REAL*8, PARAMETER :: g=9.2
    REAL*8, PARAMETER :: n=3
    REAL*8, PARAMETER :: a=1e-16
    REAL*8, PARAMETER :: dt=1/12.0
    REAL*8, PARAMETER :: c=2*a/(n+2)*(rho*g)**n*1.e3**n
    REAL*8, PARAMETER :: tend=5000
    REAL*8, PARAMETER :: bx=-0.0001
    REAL*8, PARAMETER :: m0=.004, m1=0.0002
    INTRINSIC INT
    INTEGER, PARAMETER :: nt=INT(tend/dt)
    REAL*8, DIMENSION(nx+1, nt+1) :: h
    REAL*8, DIMENSION(nx+1, nt+1) :: hd
    REAL*8, DIMENSION(nx+1, nt+1) :: h_capital
    REAL*8, DIMENSION(nx+1, nt+1) :: h_capitald
    INTEGER :: t, i
    REAL*8, DIMENSION(nx+1) :: xarr
    REAL*8, DIMENSION(nx+1) :: m
    REAL*8, DIMENSION(nx+1) :: md
    REAL*8, DIMENSION(nx+1) :: b
    REAL*8, DIMENSION(nx) :: d, phi
    REAL*8, DIMENSION(nx) :: dd, phid
    REAL*8, DIMENSION(nx+1), INTENT(IN) :: xx
    REAL*8, DIMENSION(nx+1), INTENT(IN) :: xxd
    REAL*8, INTENT(OUT) :: v
    REAL*8, INTENT(OUT) :: vd
    INTRINSIC SIZE
    REAL*8, DIMENSION(nx) :: pwx1
    REAL*8, DIMENSION(nx) :: pwx1d
    REAL*8 :: pwy1
    REAL*8, DIMENSION(nx) :: pwr1
    REAL*8, DIMENSION(nx) :: pwr1d
    REAL*8, DIMENSION(nx) :: pwx2
```

(continues on next page)

(continued from previous page)

```
    REAL*8, DIMENSION(nx) :: pwx2d
    REAL*8 :: pwy2
    REAL*8, DIMENSION(nx) :: pwr2
    REAL*8, DIMENSION(nx) :: pwr2d
    REAL*8, DIMENSION(nx) :: temp
    xarr = (/((i-1)*dx, i=1,nx+1)/)
    m = (/(m0-(i-1)*dx*m1, i=1,nx+1)/)
    b = (/(1.0+bx*(i-1)*dx, i=1,nx+1)/)
    md = xxd
    m = m + xx
    h(1, :) = b(1)
    h(:, 1) = b
    h(nx+1, :) = b(nx+1)
    h_capital(1, :) = h(1, :) - b(1)
    h_capital(nx+1, :) = h(nx+1, :) - b(nx+1)
    h_capital(:, 1) = h(:, 1) - b
    h_capitald = 0.0_8
    hd = 0.0_8
    DO t=1,nt
      pwx1d = (h_capitald(1:nx, t)+h_capitald(2:nx+1, t))/2
      pwx1 = (h_capital(1:nx, t)+h_capital(2:nx+1, t))/2
      pwy1 = n + 2
      WHERE (pwx1 .LE. 0.0 .AND. (pwy1 .EQ. 0.0 .OR. pwy1 .NE. INT(pwy1)&
&         ))
        pwr1d = 0.0_8
      ELSEWHERE
        pwr1d = pwy1*pwx1**(pwy1-1)*pwx1d
      END WHERE
      pwr1 = pwx1**pwy1
      pwx2d = (hd(2:nx+1, t)-hd(1:nx, t))/dx
      pwx2 = (h(2:nx+1, t)-h(1:nx, t))/dx
      pwy2 = n - 1
      WHERE (pwx2 .LE. 0.0 .AND. (pwy2 .EQ. 0.0 .OR. pwy2 .NE. INT(pwy2)&
&         ))
        pwr2d = 0.0_8
      ELSEWHERE
        pwr2d = pwy2*pwx2**(pwy2-1)*pwx2d
      END WHERE
      pwr2 = pwx2**pwy2
      dd(:) = c*(pwr2*pwr1d+pwr1*pwr2d)
      d(:) = c*pwr1*pwr2
      temp = h(2:nx+1, t) - h(1:nx, t)
      phid(:) = -(temp*dd(:)/dx+d(:)*(hd(2:nx+1, t)-hd(1:nx, t))/dx)
      phi(:) = -(d(:)/dx*temp)
      hd(2:nx, t+1) = hd(2:nx, t) + dt*md(2:nx) - dt*(phid(2:nx)-phid(1:&
&       nx-1))/dx
      h(2:nx, t+1) = h(2:nx, t) + m(2:nx)*dt - dt/dx*(phi(2:nx)-phi(1:nx&
&       -1))
      WHERE (h(:, t+1) .LT. b)
        hd(:, t+1) = 0.0_8
        h(:, t+1) = b
      END WHERE
```

(continues on next page)

(continued from previous page)

```
      h_capitald(:, t+1) = hd(:, t+1)
      h_capital(:, t+1) = h(:, t+1) - b
    END DO
    v = 0.
    OPEN(unit=2, file='results_forward_run.txt', action='write', status=&
&  'replace')
    WRITE(2, *) &
&   '         #                H                h                   b'
    WRITE(2, *) '_____________________________________________________&
&_____________________'
    vd = 0.0_8
    DO i=1,SIZE(h_capital(:, nt+1))
      vd = vd + dx*h_capitald(i, nt+1)
      v = v + h_capital(i, nt+1)*dx
      WRITE(2, *) i, '    ', h_capital(i, nt+1), '    ', h(i, nt+1), &
&     '    ', b(i)
    END DO
    CLOSE(2)
  END SUBROUTINE FORWARD_PROBLEM_D

  SUBROUTINE FORWARD_PROBLEM(xx, v)
    IMPLICIT NONE
    REAL*8, PARAMETER :: rho=920.0
    REAL*8, PARAMETER :: g=9.2
    REAL*8, PARAMETER :: n=3
    REAL*8, PARAMETER :: a=1e-16
    REAL*8, PARAMETER :: dt=1/12.0
    REAL*8, PARAMETER :: c=2*a/(n+2)*(rho*g)**n*1.e3**n
    REAL*8, PARAMETER :: tend=5000
    REAL*8, PARAMETER :: bx=-0.0001
    REAL*8, PARAMETER :: m0=.004, m1=0.0002
    INTRINSIC INT
    INTEGER, PARAMETER :: nt=INT(tend/dt)
    REAL*8, DIMENSION(nx+1, nt+1) :: h
    REAL*8, DIMENSION(nx+1, nt+1) :: h_capital
    INTEGER :: t, i
    REAL*8, DIMENSION(nx+1) :: xarr
    REAL*8, DIMENSION(nx+1) :: m
    REAL*8, DIMENSION(nx+1) :: b
    REAL*8, DIMENSION(nx) :: d, phi
    REAL*8, DIMENSION(nx+1), INTENT(IN) :: xx
    REAL*8, INTENT(OUT) :: v
    INTRINSIC SIZE
    REAL*8, DIMENSION(nx) :: pwx1
    REAL*8 :: pwy1
    REAL*8, DIMENSION(nx) :: pwr1
    REAL*8, DIMENSION(nx) :: pwx2
    REAL*8 :: pwy2
    REAL*8, DIMENSION(nx) :: pwr2
    xarr = (/((i-1)*dx, i=1,nx+1)/)
    m = (/(m0-(i-1)*dx*m1, i=1,nx+1)/)
    b = (/(1.0+bx*(i-1)*dx, i=1,nx+1)/)
```

(continues on next page)

(continued from previous page)

```
    m = m + xx
    h(1, :) = b(1)
    h(:, 1) = b
    h(nx+1, :) = b(nx+1)
    h_capital(1, :) = h(1, :) - b(1)
    h_capital(nx+1, :) = h(nx+1, :) - b(nx+1)
    h_capital(:, 1) = h(:, 1) - b
    DO t=1,nt
      pwx1 = (h_capital(1:nx, t)+h_capital(2:nx+1, t))/2
      pwy1 = n + 2
      pwr1 = pwx1**pwy1
      pwx2 = (h(2:nx+1, t)-h(1:nx, t))/dx
      pwy2 = n - 1
      pwr2 = pwx2**pwy2
      d(:) = c*pwr1*pwr2
      phi(:) = -(d(:)*(h(2:nx+1, t)-h(1:nx, t))/dx)
      h(2:nx, t+1) = h(2:nx, t) + m(2:nx)*dt - dt/dx*(phi(2:nx)-phi(1:nx&
&       -1))
      WHERE (h(:, t+1) .LT. b) h(:, t+1) = b
      h_capital(:, t+1) = h(:, t+1) - b
    END DO
    v = 0.
    OPEN(unit=2, file='results_forward_run.txt', action='write', status=&
&  'replace')
    WRITE(2, *) &
&   '         #                H                h                  b'
    WRITE(2, *) '_____________________________________________________&
&_____________________'
    DO i=1,SIZE(h_capital(:, nt+1))
      v = v + h_capital(i, nt+1)*dx
      WRITE(2, *) i, '    ', h_capital(i, nt+1), '    ', h(i, nt+1), &
&     '    ', b(i)
    END DO
    CLOSE(2)
  END SUBROUTINE FORWARD_PROBLEM

END MODULE FORWARD_TGT
```

- `forward_d.msg` - Contains warning messages. TAPENADE can be quite verbose with the warnings. None of the warnings below are much important.

```
1 Command: Procedure forward_problem understood as forward.forward_problem
2 forward_problem: (AD03) Varied variable h[i,nt+1] written by I-O to file 2
3 forward_problem: (AD03) Varied variable h_capital[i,nt+1] written by I-O to file 2
~
```

Adjoint Model

The adjoint model is described in detail in *Tutorial 1*. Another excellent resource is the MITgcm documentation.

Generating Adjoint model code using Tapenade

It is pretty simple to generate the tangent linear model for our forward model.

```
% tapenade -reverse -head "forward_problem(V)/(xx)" forward.f90
```

`-reverse` tells Tapenade that we want a reverse i.e. adjoint code.

`-head "forward_problem(V)/(xx)` tells Tapenade that the head subroutine is `forward_problem`, the dependent variable or the objective function is `V` and the independent variable is `xx`.

This generates 2 files -

- `forward_b.f90` - This file contains the adjoint module (called `forward_diff`) as well as the original forward code module. `forward_diff` contains the `forward_problem_b` subroutine which is our adjoint model. Notice how it requires values for `xx, V` which have to be obtained from a forward solve (calling the `forward` subroutine). We also will define `Vb = 1` and propagate the adjoint sensitivities backwards to evaluate `xxb`. If you look at the second `DO` loop in `forward_problem_b`, it indeed runs backwards in time. Note that all the adjoint variables in the code are simply suffixed with the alphabet `b`. There are also a lot of `PUSH` and `POP` statements, they are essentially pushing and popping the stored variables from a stack just like we did manually in *Tutorial 1*.

```
!  TAPENADE 3.15 (master) - 15 Apr 2020 13:12
!
MODULE FORWARD_DIFF
  IMPLICIT NONE
  REAL*8, PARAMETER :: xend=30
  REAL*8, PARAMETER :: dx=1
  INTEGER, PARAMETER :: nx=INT(xend/dx)

CONTAINS
!  Differentiation of forward_problem in reverse (adjoint) mode:
!   gradient     of useful results: v
!   with respect to varying inputs: v xx
!   RW status of diff variables: v:in-zero xx:out
  SUBROUTINE FORWARD_PROBLEM_B(xx, xxb, v, vb)
    IMPLICIT NONE
    REAL*8, PARAMETER :: rho=920.0
    REAL*8, PARAMETER :: g=9.2
    REAL*8, PARAMETER :: n=3
    REAL*8, PARAMETER :: a=1e-16
    REAL*8, PARAMETER :: dt=1/12.0
    REAL*8, PARAMETER :: c=2*a/(n+2)*(rho*g)**n*1.e3**n
    REAL*8, PARAMETER :: tend=5000
    REAL*8, PARAMETER :: bx=-0.0001
    REAL*8, PARAMETER :: m0=.004, m1=0.0002
    INTRINSIC INT
    INTEGER, PARAMETER :: nt=INT(tend/dt)
    REAL*8, DIMENSION(nx+1, nt+1) :: h
    REAL*8, DIMENSION(nx+1, nt+1) :: hb
```

(continues on next page)

(continued from previous page)

```
    REAL*8, DIMENSION(nx+1, nt+1) :: h_capital
    REAL*8, DIMENSION(nx+1, nt+1) :: h_capitalb
    INTEGER :: t, i
    REAL*8, DIMENSION(nx+1) :: xarr
    REAL*8, DIMENSION(nx+1) :: m
    REAL*8, DIMENSION(nx+1) :: mb
    REAL*8, DIMENSION(nx+1) :: b
    REAL*8, DIMENSION(nx) :: d, phi
    REAL*8, DIMENSION(nx) :: db, phib
    REAL*8, DIMENSION(nx+1), INTENT(IN) :: xx
    REAL*8, DIMENSION(nx+1) :: xxb
    REAL*8 :: v
    REAL*8 :: vb
    INTRINSIC SIZE
    LOGICAL, DIMENSION(nx+1) :: mask
    REAL*8, DIMENSION(nx) :: temp
    REAL*8, DIMENSION(nx) :: temp0
    REAL*8, DIMENSION(nx) :: tempb
    REAL*8, DIMENSION(nx) :: tempb0
    REAL*8, DIMENSION(nx-1) :: tempb1
    INTEGER :: ad_to
    m = (/(m0-(i-1)*dx*m1, i=1,nx+1)/)
    b = (/(1.0+bx*(i-1)*dx, i=1,nx+1)/)
    m = m + xx
    h(1, :) = b(1)
    h(:, 1) = b
    h(nx+1, :) = b(nx+1)
    h_capital(1, :) = h(1, :) - b(1)
    h_capital(nx+1, :) = h(nx+1, :) - b(nx+1)
    h_capital(:, 1) = h(:, 1) - b
    DO t=1,nt
      CALL PUSHREAL8ARRAY(d, nx)
      d(:) = c*((h_capital(1:nx, t)+h_capital(2:nx+1, t))/2)**(n+2)*((h(&
&       2:nx+1, t)-h(1:nx, t))/dx)**(n-1)
      phi(:) = -(d(:)*(h(2:nx+1, t)-h(1:nx, t))/dx)
      CALL PUSHREAL8ARRAY(h(2:nx, t+1), nx - 1)
      h(2:nx, t+1) = h(2:nx, t) + m(2:nx)*dt - dt/dx*(phi(2:nx)-phi(1:nx&
&       -1))
      CALL PUSHBOOLEANARRAY(mask, nx + 1)
      mask(:) = h(:, t+1) .LT. b
      CALL PUSHREAL8ARRAY(h(:, t+1), nx + 1)
      WHERE (mask(:)) h(:, t+1) = b
      CALL PUSHREAL8ARRAY(h_capital(:, t+1), nx + 1)
      h_capital(:, t+1) = h(:, t+1) - b
    END DO
    DO i=1,SIZE(h_capital(:, nt+1))

    END DO
    CALL PUSHINTEGER4(i - 1)
    h_capitalb = 0.0_8
    CALL POPINTEGER4(ad_to)
    DO i=ad_to,1,-1
```

(continues on next page)

(continued from previous page)

```
      h_capitalb(i, nt+1) = h_capitalb(i, nt+1) + dx*vb
    END DO
    hb = 0.0_8
    mb = 0.0_8
    DO t=nt,1,-1
      CALL POPREAL8ARRAY(h_capital(:, t+1), nx + 1)
      hb(:, t+1) = hb(:, t+1) + h_capitalb(:, t+1)
      h_capitalb(:, t+1) = 0.0_8
      CALL POPREAL8ARRAY(h(:, t+1), nx + 1)
      CALL POPBOOLEANARRAY(mask, nx + 1)
      phib = 0.0_8
      CALL POPREAL8ARRAY(h(2:nx, t+1), nx - 1)
      db = 0.0_8
      temp = (h(2:nx+1, t)-h(1:nx, t))/dx
      temp0 = (h_capital(1:nx, t)+h_capital(2:nx+1, t))/2
      WHERE (mask(:)) hb(:, t+1) = 0.0_8
      hb(2:nx, t) = hb(2:nx, t) + hb(2:nx, t+1)
      mb(2:nx) = mb(2:nx) + dt*hb(2:nx, t+1)
      tempb1 = -(dt*hb(2:nx, t+1)/dx)
      hb(2:nx, t+1) = 0.0_8
      phib(2:nx) = phib(2:nx) + tempb1
      phib(1:nx-1) = phib(1:nx-1) - tempb1
      db(:) = -((h(2:nx+1, t)-h(1:nx, t))*phib(:)/dx)
      tempb = -(d(:)*phib(:)/dx)
      hb(2:nx+1, t) = hb(2:nx+1, t) + tempb
      hb(1:nx, t) = hb(1:nx, t) - tempb
      CALL POPREAL8ARRAY(d, nx)
      WHERE (temp0 .LE. 0.0 .AND. (n + 2 .EQ. 0.0 .OR. n + 2 .NE. INT(n &
&         + 2)))
        tempb = 0.0_8
      ELSEWHERE
        tempb = (n+2)*temp0**(n+1)*temp**(n-1)*c*db(:)/2
      END WHERE
      WHERE (temp .LE. 0.0 .AND. (n - 1 .EQ. 0.0 .OR. n - 1 .NE. INT(n -&
&         1)))
        tempb0 = 0.0_8
      ELSEWHERE
        tempb0 = (n-1)*temp**(n-2)*temp0**(n+2)*c*db(:)/dx
      END WHERE
      hb(2:nx+1, t) = hb(2:nx+1, t) + tempb0
      hb(1:nx, t) = hb(1:nx, t) - tempb0
      h_capitalb(1:nx, t) = h_capitalb(1:nx, t) + tempb
      h_capitalb(2:nx+1, t) = h_capitalb(2:nx+1, t) + tempb
    END DO
    xxb = 0.0_8
    xxb = mb
    vb = 0.0_8
  END SUBROUTINE FORWARD_PROBLEM_B

  SUBROUTINE FORWARD_PROBLEM(xx, v)
    IMPLICIT NONE
    REAL*8, PARAMETER :: rho=920.0
```

(continues on next page)

(continued from previous page)

```
    REAL*8, PARAMETER :: g=9.2
    REAL*8, PARAMETER :: n=3
    REAL*8, PARAMETER :: a=1e-16
    REAL*8, PARAMETER :: dt=1/12.0
    REAL*8, PARAMETER :: c=2*a/(n+2)*(rho*g)**n*1.e3**n
    REAL*8, PARAMETER :: tend=5000
    REAL*8, PARAMETER :: bx=-0.0001
    REAL*8, PARAMETER :: m0=.004, m1=0.0002
    INTRINSIC INT
    INTEGER, PARAMETER :: nt=INT(tend/dt)
    REAL*8, DIMENSION(nx+1, nt+1) :: h
    REAL*8, DIMENSION(nx+1, nt+1) :: h_capital
    INTEGER :: t, i
    REAL*8, DIMENSION(nx+1) :: xarr
    REAL*8, DIMENSION(nx+1) :: m
    REAL*8, DIMENSION(nx+1) :: b
    REAL*8, DIMENSION(nx) :: d, phi
    REAL*8, DIMENSION(nx+1), INTENT(IN) :: xx
    REAL*8, INTENT(OUT) :: v
    INTRINSIC SIZE
    LOGICAL, DIMENSION(nx+1) :: mask
    xarr = (/((i-1)*dx, i=1,nx+1)/)
    m = (/(m0-(i-1)*dx*m1, i=1,nx+1)/)
    b = (/(1.0+bx*(i-1)*dx, i=1,nx+1)/)
    m = m + xx
    h(1, :) = b(1)
    h(:, 1) = b
    h(nx+1, :) = b(nx+1)
    h_capital(1, :) = h(1, :) - b(1)
    h_capital(nx+1, :) = h(nx+1, :) - b(nx+1)
    h_capital(:, 1) = h(:, 1) - b
    DO t=1,nt
      d(:) = c*((h_capital(1:nx, t)+h_capital(2:nx+1, t))/2)**(n+2)*((h(&
&       2:nx+1, t)-h(1:nx, t))/dx)**(n-1)
      phi(:) = -(d(:)*(h(2:nx+1, t)-h(1:nx, t))/dx)
      h(2:nx, t+1) = h(2:nx, t) + m(2:nx)*dt - dt/dx*(phi(2:nx)-phi(1:nx&
&       -1))
      mask(:) = h(:, t+1) .LT. b
      WHERE (mask(:)) h(:, t+1) = b
      h_capital(:, t+1) = h(:, t+1) - b
    END DO
    v = 0.
    OPEN(unit=2, file='results_forward_run.txt', action='write', status=&
&  'replace')
    WRITE(2, *) &
&   '         #                H                h                   b'
    WRITE(2, *) '_____________________________________________________&
&_____________________'
    DO i=1,SIZE(h_capital(:, nt+1))
      v = v + h_capital(i, nt+1)*dx
      WRITE(2, *) i, '    ', h_capital(i, nt+1), '    ', h(i, nt+1), &
&     '    ', b(i)
```

(continues on next page)

(continued from previous page)

```
    END DO
    CLOSE(2)
  END SUBROUTINE FORWARD_PROBLEM

END MODULE FORWARD_DIFF
```

- `forward_b.msg` - Contains warning messages. Tapenade can be quite verbose with the warnings. None of the warnings below are much important.

```
1 Command: Procedure forward_problem understood as forward.forward_problem
2 forward_problem: Command: Input variable(s) v have their derivative modified in␣
↪forward_problem: added to independents
3 forward_problem: (AD03) Varied variable h[i,nt+1] written by I-O to file 2
4 forward_problem: (AD03) Varied variable h_capital[i,nt+1] written by I-O to file 2
```

Driver Routine

The driver routine evaluates the sensitivities of our cost function `V` to the independent variable `xx` in three ways - Finite Differences, Tangent Linear Model (forward mode), Adjoint Model (reverse mode). Both the TLM and adjoint methods should give sensitivities that agree within some tolerance with the Finite Differences sensitivities. One can see that the TLM and finite differences have to be called multiple times (as many times as the number of control variables) while the adjoint model only has to be called once to evaluate the entire gradient, making it much more efficient.

```
program driver
     !!Essentially both modules had variables with the same name
     !!so they had to be locally aliased for the code to compile and run
     !!Obviously only one set of variables is needed so the other is useless

     use forward_diff, n => nx, fp => forward_problem !! Alias to a local variable
     use forward_tgt, n_useless => nx, fp_useless => forward_problem

     implicit none

     real(8), dimension(n+1) :: xx=0., xx_tlm =0., xxb =0., xx_fd, accuracy
     real(8) :: V=0., Vb = 1., V_forward, Vd = 0.
     real(8), parameter :: eps = 1.d-7
     integer :: ii

     call fp(xx,V)

     !! Forward run
     V_forward = V


     !! Adjoint run
     xx = 0.
     V = 0.
     call fp(xx,V)
     call forward_problem_b(xx,xxb,V,Vb)
```

(continues on next page)

(continued from previous page)

```
    open (unit = 1, file = "results.txt", action="write",status="replace")

    write(1,*) "          #                Reverse                              FD",&
                    "                          Tangent                        Relative␣
↪accuracy"
    write(1,*) "_____________________________________________________________________",&
                    "_____________________________________________________________________
↪____"

    !! Finite differences and Tangent Linear Model
    do ii = 1, n+1

            xx = 0.
            V = 0.
            xx_tlm = 0.
            xx_tlm(ii) = 1.

            !! TLM
            call fp(xx,V)
            call forward_problem_d(xx,xx_tlm,V,Vd)


            !! FD
            xx = 0.
            V = 0.
            xx(ii) = eps
            call fp(xx,V)
            xx_fd(ii) =  (V - V_forward)/eps

       if ( xx_fd(ii).NE. 0. ) then
           accuracy(ii) = 1.d0 - Vd/xx_fd(ii)
       else
           accuracy(ii) = 0.
       end if
       write(1,*) ii, "    ", xxb(ii), "    ", xx_fd(ii),"    ", Vd,"    ", accuracy(ii)
    end do

    close(1)

    call execute_command_line('gnuplot plot.script')
end program driver
```

Combining all compilation commands into a Makefile

Makefiles are extremely useful to automate the compilation process. The Makefile shown below generates both the tangent linear and adjoint codes, and bundles everything together with the driver routine and a special file provided by the Tapenade developers, called `adStack.c` together into a single executable named `adjoint` (unfortunately, a slight misnomer since it also executes FD and TLM codes). `adStack.c` contains the definitions of the `POP, PUSH` mechanisms that Tapenade uses in its generated codes.

The `adStack.c` file can be found in `test_ad/tapenade_supported/ADFirstAidKit` sub-directory in the `ad` branch of the SICOPOLIS repository. The Makefile assumes that you have the entire `ADFirstAidKit` directory present, so it is better to copy the entire directory to work as is with this Makefile.

More information on Makefiles can be found here . Makefiles are extremely sensitive to whitespaces both at the start and end of any line, so you should be very careful with that.

- `Makefile`:

```
EXEC := adjoint
SRC  := $(wildcard *.f90)
OBJ  := $(patsubst %.f90, %.o, $(SRC))
# NOTE - OBJ will not have the object files of c codes in it, this needs to be improved␣
↪upon.
# Options
F90  := gfortran
CC           := gcc
TAP_AD_kit:= ./ADFirstAidKit

# Rules

$(EXEC): $(OBJ) adStack.o forward_b.o forward_d.o
             $(F90) -o $@ $^

%.o: %.f90
             $(F90) -c $<

driver.o: forward_b.f90 forward_diff.mod forward_tgt.mod
forward_diff.mod: forward_b.o
forward_tgt.mod: forward_d.o

adStack.o :
             $(CC) -c $(TAP_AD_kit)/adStack.c

forward_b.f90: forward.f90
             tapenade -reverse -head "forward_problem(V)/(xx)" forward.f90
forward_d.f90: forward.f90
             tapenade -tangent -tgtmodulename %_tgt -head "forward_problem(V)/(xx)"␣
↪forward.f90
# Useful phony targets

.PHONY: clean

clean:
     $(RM) $(EXEC) *.o $(MOD) $(MSG) *.msg *.mod *_db.f90 *_b.f90 *_d.f90 *~
```

To compile everything, just type -

```
% make
```

To clean up the compilation, just type -

```
% make clean
```

Results

The results are shown here. The adjoint and the tangent linear model agree with each other up to 12 decimal places in most cases. The agreement of both of these with the Finite differences is also extremely good, as seen from the `Relative accuracy`.

```
        #               Reverse                         FD                      
↪Tangent                    Relative accuracy
_____________________________________________________________________________________________
↪_____________________________________________
         1        0.0000000000000000             0.0000000000000000             0.
↪0000000000000000             0.0000000000000000
         2        7.7234861280779530             7.7234756545863092             7.
↪7234861280783651            -1.3560594380734869E-006
         3        13.803334061615192             13.803310956461701             13.
↪803334061616136            -1.6738849475395057E-006
         4        19.668654270913329             19.668613369105969             19.
↪668654270914736            -2.0795471444845504E-006
         5        25.460720864192346             25.460657031572964             25.
↪460720864194293            -2.5071081728444966E-006
         6        31.276957320291768             31.276862468843092             31.
↪276957320293935            -3.0326395730195799E-006
         7        37.209906557896431             37.209769434554119             37.
↪209906557899728            -3.6851436515661362E-006
         8        43.370516019133923             43.370320437219334             43.
↪370516019138648            -4.5095797620575695E-006
         9        49.919848601374966             49.919570930256896             49.
↪919848601380266            -5.5623699923845749E-006
        10        57.138817544000176             57.138414888413536             57.
↪138817544006628            -7.0470207107486971E-006
        11        65.642858080405333             65.642249307273914             65.
↪642858080414797            -9.2741054322775796E-006
        12        77.452386462370669             77.451334217215617             77.
↪452386462382293            -1.3585888187783723E-005
        13        139.25939689848738             139.25460375929788             139.
↪25939689851842            -3.4419969546117812E-005
        14        149.07659819564697             149.07253577334245             149.
↪07659819568059            -2.7251313040821401E-005
        15        153.96500702363139             153.96133390410682             153.
↪96500702366376            -2.3857415779593438E-005
        16        156.68784327845535             156.68444133254411             156.
↪68784327848971            -2.1712085237490797E-005
        17        158.00297057747002             157.99977889585648             158.
↪00297057750697            -2.0200545043813634E-005
        18        158.20831395372016             158.20529343457679             158.
↪20831395375782            -1.9092402760101379E-005
```

(continues on next page)

(continued from previous page)

```
        19        157.42238310624799             157.41950864622822             157.
↪42238310628576           -1.8259871868764321E-005
        20        155.66584913271916             155.66310167969277             155.
↪66584913275506           -1.7649995616375591E-005
        21        152.88684508060106             152.88421169046273             152.
↪88684508064247           -1.7224735965992721E-005
        22        148.96151788159625             148.95898933886542             148.
↪96151788163439           -1.6974757818921660E-005
        23        143.67549543577167             143.67306516049894             143.
↪67549543580796           -1.6915316077614762E-005
        24        136.67934044641444             136.67700816455408             136.
↪67934044645017           -1.7064186050186336E-005
        25        127.39349079030032             127.39126539429435             127.
↪39349079032215           -1.7468984399471310E-005
        26        114.79185160452661             114.78974569101297             114.
↪79185160454820           -1.8345833267208178E-005
        27        96.827067402857665             96.825136655098731             96.
↪827067402878811           -1.9940563440234982E-005
        28        68.425966236870423             68.424387524856911             68.
↪425966236889820           -2.3072358993792008E-005
        29        0.0000000000000000             0.0000000000000000             0.
↪0000000000000000             0.0000000000000000
        30        0.0000000000000000             0.0000000000000000             0.
↪0000000000000000             0.0000000000000000
        31        0.0000000000000000             0.0000000000000000             0.
↪0000000000000000             0.0000000000000000
```

A plot of these results is also shown here. To the naked eye, these results are identical.

A note on Binomial Checkpointing

What is Checkpointing?

What is Binomial Checkpointing?

How to use Binomial Checkpointing with Tapenade?

Using binomial checkpointing is extremely simple with Tapenade. First we add the following line just before our time loop starts.

```
!$AD BINOMIAL-CKP nt+1 20 1
do t = 1,nt
    !!!! Stuff inside the time loop.
end do
```

Notice that for the fortran compiler the line we just added is a comment, but when Tapenade parses the code, it knows it needs to use binomial checkpointing. The line essentially tells Tapenade the following - "Use binomial checkpointing for this loop which has nt+1 steps (if you count the exit iteration as well), checkpoint every 20 steps, with the first step numbered as 1." Tapenade uses some subroutines whose names start with `ADBINOMIAL` to manage the checkpointing and thus we need to link to the `adBinomial.c` file in the Makefile so that we tell the compiler what these subroutines are. (essentially the same thing we did with `adStack.c`)

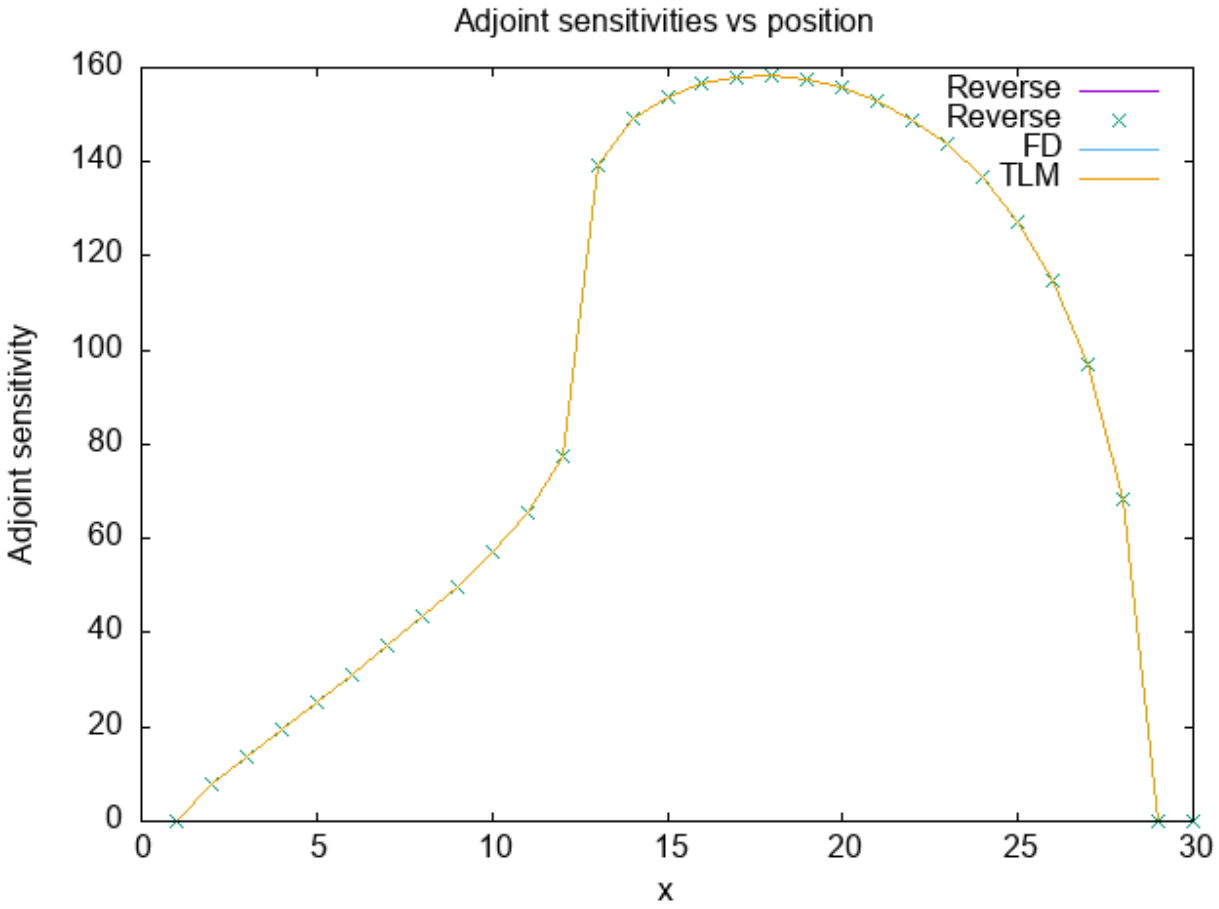


Fig. 1: `xxb` vs `x` for all methods used to calculate the sensitivities.

Tutorial 3 : Validating results of SICOPOLIS adjoint and tangent linear mode with Finite differences

In this tutorial, we will discuss how results we get from the adjoint and tangent linear modes (TLM) can be validated with a Finite Differences (FD) run. we use the header file `v5_grl16_bm5_ss25ka` provided as a reference template in SICOPOLIS. We shorten, however, the total time to 100 years, to keep the computational cost of the tangent linear and finite differences code reasonable. Our objective or cost function is the total volume of the ice sheet at the end of the run (`fc`). The sensitivity is evaluated with respect to the geothermal heat flux, `q_geo`, a 19,186-dimensional field.

There are two ways to accomplish this - using the Python utilities or manually performing all 3 runs separately. We will discuss only the first one here. The second one is easily accomplished as well - by compiling using the `MakefileTapenade` and adding I/O instructions as added by the Python scripts manually.

As prerequisites, ensure you have set up everything correctly as described in the Configuration section for both SICOPOLIS and Tapenade.

Validation using automated Python script

The python script automatically inserts I/O statements correctly where necessary for validation. All we need to do is provide the correct `test_ad/inputs.json` file. The file looks something like this -

- `test_ad/inputs.json`

```
{
    "json": "inputs.json",
    "header": "v5_grl16_bm5_ss25ka",
    "domain": "grl",
    "dep_var": "fc",
    "ind_var": "q_geo",
    "perturbation": 0.001,
    "checkpoint": 4,
    "run": true,
    "dimension": 2,
    "z_co_ord": null
}
```

```
cd test_ad
python tapenade_config.py -jsf inputs.json
```

That's it! Although, this will only run the FD and TLM at 5 select points for validation. Since the adjoint computes the full gradient in one run, this is not an issue with the adjoint. In order to run the FD and TLM on the entire grid, we need to modify `test_ad/tapenade_config.py` slightly. We modify the `limited_or_block_or_full` option in the call to the `simulation` function from `limited` to `full`. This is what it should look like -

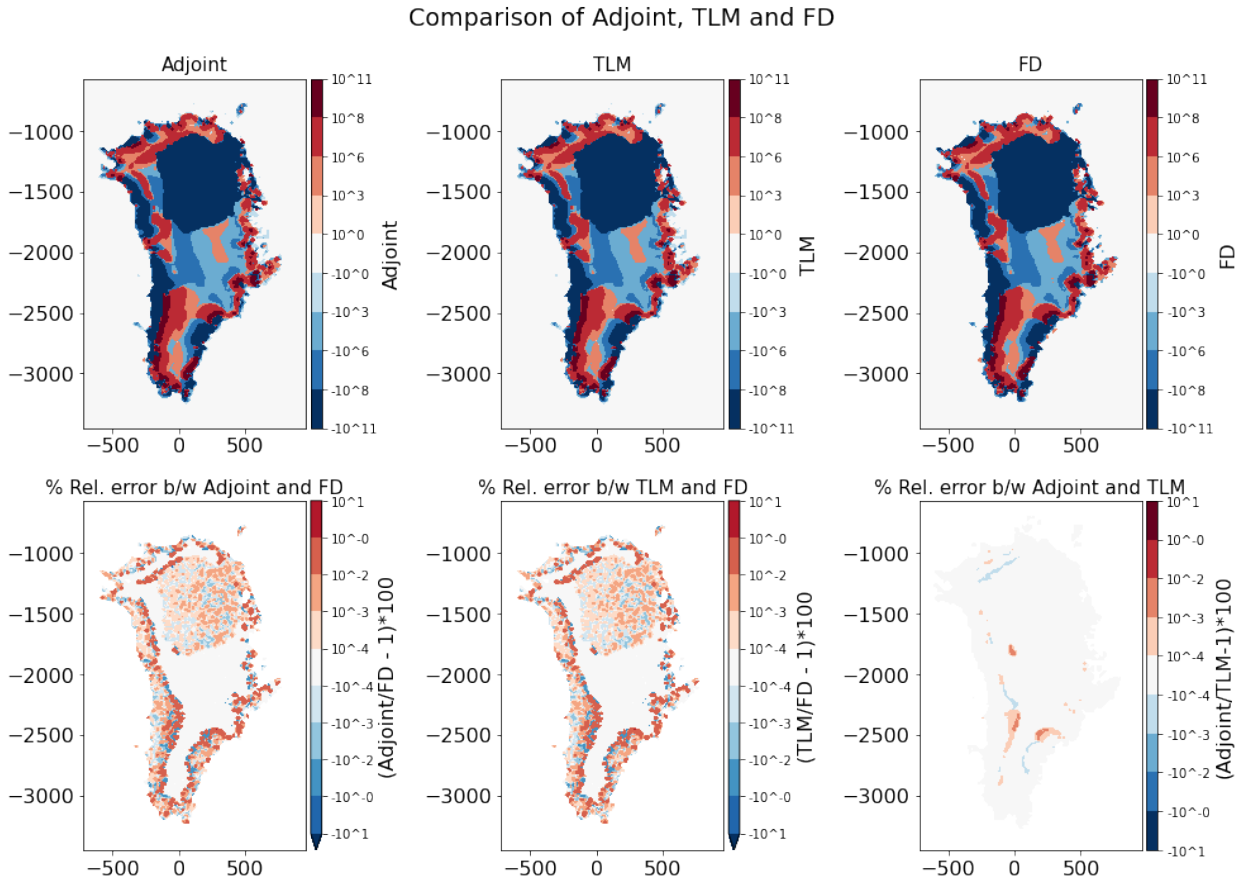
```
simulation(mode = mode, header = args.header, domain = args.domain,
                  ind_var = args.ind_var, dep_var = args.dep_var,
                  limited_or_block_or_full = 'full',
                  ind_var_dim = args.dimension, ind_var_z_co_ord = args.z_co_ord,
                  perturbation = args.perturbation,
                  run_executable_auto = args.run,
                  output_vars = args.output_vars, output_iters = args.output_iters,␣
↪output_dims = args.output_dims,
                  output_adj_vars = args.output_adj_vars, output_adj_iters = args.output_
↪adj_iters, output_adj_dims = args.output_adj_dims,
                  ckp_status = ckp_status, ckp_num = args.checkpoint)
```

This run will generate the following files, which are used to plot the results below: `src/GradientVals_q_geo_1.00E-03_v5_grl16_bm5_ss25ka_{limited_or_full}.dat`, `src/ForwardVals_q_geo_v5_grl16_bm5_ss25ka_{limited_or_full}.dat`, `src/AdjointVals_q_geob_v5_grl16_bm5_ss25ka.dat`.

Results

The results are shown in the plot below. The table below shows a comparison of the time taken by various methods for gradient calculation to evaluate the gradient for a scalar objective function with respect to a 19,186-dimensional 2D field (16 km mesh) in a typical SICOPOLIS run. Model setup and I/O times are not included. The runs are performed on Intel Xeon CPU E5-2695 v3 nodes (2.30 GHz clock rate, 35.84 MB L3 cache, 63.3 GB memory), on the Sverdrup cluster.

Gradient calculation method	Time (in seconds) for 16 km mesh
Finite Differences	1.640×10^5
Tangent Linear Model	9.793×10^4
Adjoint Model	2.214×10^1



Validation exercise for Adjoint/Tangent Linear (TLM) modes using the Finite Differences (FD) results for the sensitivity of `fc` with respect to `q_geo`. The upper row shows the sensitivities computed using the Adjoint, Tangent Linear mode, and Finite Differences respectively. The bottom run illustrates the relative error between (Adjoint, FD), (TLM, FD), and (Adjoint, TLM) respectively.

2.6 SICOPOLIS-AD v2

The code for SICOPOLIS-AD v2 is kept mostly independent from the base SICOPOLIS code, allowing non-AD users to avoid it completely. All of the AD-related support routines and data files can be found in `src/subroutines/tapenade`. Similarly, all utilities and testing files for AD simulations are stored in the `test_ad` directory. A separate Makefile is provided for AD purposes - `src/MakefileTapenade`.

`SICOPOLIS-AD v2` can be run directly by interacting with the Makefile and the `Fortran` code. The user has to prepare a suitable header file for the base `SICOPOLIS` code and add a few more preprocessing directives to run the adjoint code with the same header file. This header file, along with a set of dependent and independent variables, is given to the Makefile. The Makefile executes the workflow for differentiating and compiling the code depending on the mode selected by the user (tangent linear, adjoint, finite differences, code coverage evaluation). The user must then insert the I/O statements in the Tapenade-generated code depending on what they wish to analyze. This is followed by recompilation and execution of the code.

Alternatively, the user can use the functions in `test_ad/tapenade_config.py` to automatically customize the setup. These functions are written for automated sensitivity studies. They can be easily modified to serve other purposes such as model calibration and UQ.

2.6.1 Running SICOPOLIS-AD v2

SICOPOLIS-AD v2 runs almost independently of the setup for SICOPOLIS. It has its own Makefile `src/MakefileTapenade`, typically one runs it using the following generic command (assuming the current working directory is `src/`) -

```
% make -f MakefileTapenade clean
% make -f MakefileTapenade driver{mode} HEADER={header} DOMAIN_SHORT={domain} DEP_VAR=
↪{dep_var} IND_VARS={ind_vars} DISC_AND_GRL={disc_and_grl}
#### Add I/O commands in the differentiated code if needed and recompile
% make -f MakefileTapenade driver{mode} HEADER={header} DOMAIN_SHORT={domain} DEP_VAR=
↪{dep_var} IND_VARS={ind_vars} DISC_AND_GRL={disc_and_grl}
% ./driver{mode}
```

Here, `{mode}` refers to one of these options - `normal, grdchk, adjoint, forward` for normal (vanilla `SICOPOLIS` run), finite differences, adjoint and tangent linear mode respectively. `{header}` refers to the latter half of the name of the header file. If the header file is `sico_specs_v5_grl20_ss25ka.h`, then the `{header}` is `v5_grl20_ss25ka`. `{domain}` can either be `grl` or `ant`, depending on Greenland or Antarctica. `{dep_var}` refers to the cost function / objective function. `{ind_vars}` is a list of independent variables for which we calculate the sensitivity of the objective function. `{disc_and_grl}` is either 0 or 1. It is 1 if the `DISC` flag in the header file is greater than 0 and the domain is Greenland.

The `finite differences` mode can only have one indpendent variable at a time, unlike the `adjoint` and `forward` modes.

The limitation here is that since Tapenade freshly generates the codes, the I/O for the differentiatied variables has to be done manually, at least for the current version of the setup. To tackle this, we have some Python utilities to automate this to a large extent.

Note that most of the Python utilities that have been developed for convenience have only been tested with one variable in `{ind_vars}`.

Activating variables

There are some variables in the SICOPOLIS code, for example `c_slide`, the basal sliding coefficient, that are constant in time and reinitialized at each time step in the subroutines in `src/subroutines/general/calc_vxy_m.F90`. This reinitialization has an "erasing" effect on the the dependency of cost functions to such variables, resulting in identically zero adjoint fields. From a physics standpoint, it is pretty clear that `c_slide` should have affect various kinds of cost functions.

The workaround to this issue is to complete the following steps (the code snippets are based on the `c_slide` example) -

1. Declare a dummy variable in `src/subroutines/general/sico_variables_m.F90`.

```
real(dp), dimension(0:JMAX,0:IMAX) :: xxc_slide
```

2. Initialize the dummy variable to 0. at the end of `src/subroutines/{domain}/sico_init_m.F90`. Here `{domain}` can either be `grl` or `ant`.

```
xxc_slide = 0.
```

3. Add the dummy variable to the actual variable. In the case of `c_slide`, add it at the point where it's initialization is complete.

```
c_slide = c_slide + xxc_slide
```

4. Evaluate sensitivities to the dummy variable i.e. the `{ind_vars}` should contain `xxc_slide` for a non-trivial senstivity with respect to `c_slide`.

This procedure has the effect of continuity of the variable `xxc_slide` in time, preventing the "erasing" effect caused by the reinitialization of `c_slide` to the same value for each time step. Also, since `xxc_slide` is initialized to 0, the results in the normal simulations are unaffected, because for the purposes of the non-linear forward model, all we did was add zeros to `c_slide`.

2.6.2 Utilities

Continuous Integration(CI)

We leverage the Continuous Integration (CI) infrastructure of Gitlab-CI. We have a suite of regression tests in our repository at `test_ad/tests.py` which leverage the `pytest` module in Python. The CI infrastructure allows us to automate the tests. Essentially, these tests are automatically triggered after new code is pushed to the remote repository. They run on an online `gitlab runner` which uses a virtual Docker environment that is correctly set up to run the SICOPOLIS code. The instructions that run in the Docker environment can be found in a file name `.gitlab-ci.yml` in the root directory. The status of tests can be seen with the help of the build badge in `README.md` in the `ad` branch.

Code coverage

The information in this section is based on the slides of Dr. Karl Schulz.

It is good practice to include the use of a code-coverage tool. Although it has not been used with Tapenade generated code, we can nonetheless use it with the original SICOPOLIS code to see what parts of this code are covered by our regression testing. We use the code coverage tool `gcov` and it's graphical front-end, `lcov`. It aggregates gcov output to generate html containing the source code annotated with coverage information. More details can be found here and here.

Let's say the user wants to generate the code coverage data from running two configurations, `v5_grl20_ss25ka` and `v5_ant40_ss25ka`. Of course, since we only care about code coverage, the total time does not need to be 25000 years for the simulations. 100-200 years will work fine too.

First we generate the `gcov` output for `v5_grl20_ss25ka`.

```
% make -f MakefileTapenade clean; make -f MakefileTapenade drivergrdchk_cov HEADER=v5_
↪grl20_ss25ka DOMAIN_SHORT=grl
% ./drivergrdchk_cov
% make -f MakefileTapenade cov
% lcov --capture --directory . --output-file coverage_v5_grl20_ss25ka.info
```

Next, we generate the `gcov` output for `v5_ant40_ss25ka`.

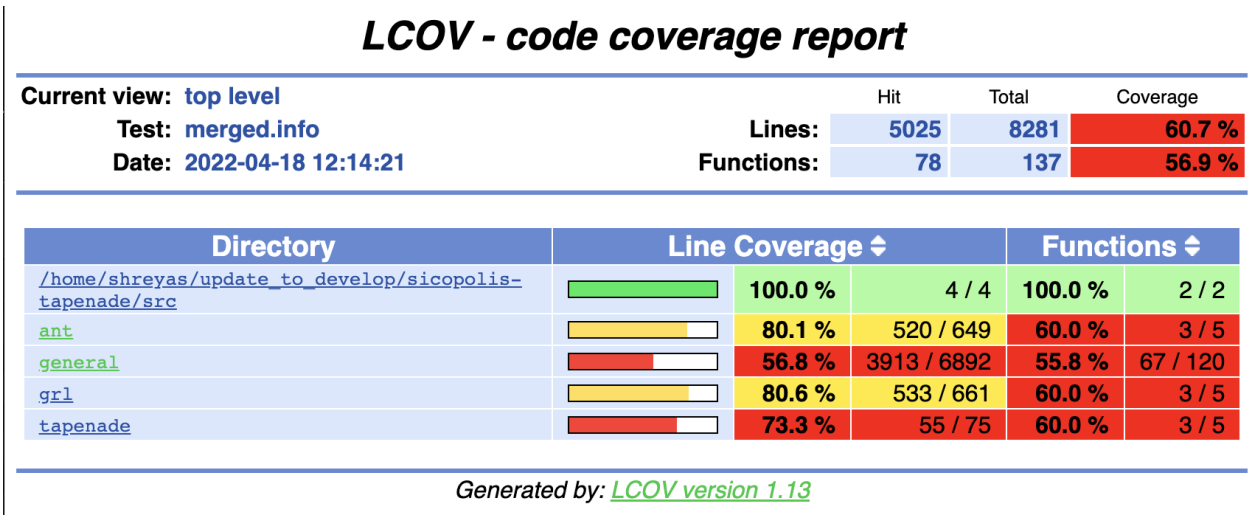
```
% make -f MakefileTapenade clean; make -f MakefileTapenade drivergrdchk_cov HEADER=v5_
↪ant40_ss25ka DOMAIN_SHORT=ant
% ./drivergrdchk_cov
% make -f MakefileTapenade cov
% lcov --capture --directory . --output-file coverage_v5_ant40_ss25ka.info
```

Finally, we need to aggregate and visualize the results of the two `info` files we created.

```
% lcov --add-tracefile coverage_v5_grl20_ss25ka.info -a coverage_v5_ant40_ss25ka.info -o␣
↪merged.info
% genhtml merged.info --output-directory out
```

This will create a `src/out` directory. Open `src/out/index.html` to get your interactive, graphical output for code coverage.

You can click further on the hyperlinks to see individual files and lines.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	merged.info	**Lines:**	5025	8281	60.7 %
Date:	2022-04-18 12:14:21	**Functions:**	78	137	56.9 %

Directory	Line Coverage			Functions	
/home/shreyas/update_to_develop/sicopolis-tapenade/src		100.0 %	4 / 4	100.0 %	2 / 2
ant		80.1 %	520 / 649	60.0 %	3 / 5
general		56.8 %	3913 / 6892	55.8 %	67 / 120
grl		80.6 %	533 / 661	60.0 %	3 / 5
tapenade		73.3 %	55 / 75	60.0 %	3 / 5

Generated by: LCOV version 1.13

Fig. 2: Example of graphical output for code coverage

Automation using Python scripts

`test_ad/tapenade_config.py` contains a few helpful functions for automating the setup and I/O of desired variables in the finite differences, tangent linear and adjoint mode.

The Python functions get updated frequently, therfore the sections that follow are only supposed to serve as a basic guideline on what is possible using these scripts. With very little modifications, these scripts can be used to serve other purposes too.

Automated Normal Mode

To run the SICOPOLIS code normally (within the AD context, this is generally used to evaluate the cost function `fc` during calibration),

1. Remove the old output file that `SICOPOLIS` generates.

```
process = subprocess.run (['rm', '-r', f'../sico_out/N_{header}'])
```

2. Compile the code

```
compile_code(mode = 'normal', header = header, domain = domain,
             clean = True, dep_var=dep_var, ind_vars = ind_var)
```

3. Run the executable

```
run_executable('normal')
```

All of the above steps are bundled within the `simulation` function, which can be run as follows -

```
simulation(mode = 'normal', header = header, domain = domain,
                  run_executable_auto = True)
```

Automated Finite Differences

A typical finite differences simulation requires perturbing the correct independent variable in the `grdchk_main` subroutine in `src/subroutines/tapenade/tapenade_m.F90`. This is achieved as follows -

1. Copy the correct header file to `src/sico_specs.h`

```
copy_file(f'../runs/headers/sico_specs_{header}.h', 'sico_specs.h')
```

2. Perturb the correct independent variable in `grdchk_main` subroutine in `src/subroutines/tapenade/tapenade_m.F90`.

This step involves the following sub-steps:

- Decide which of the three modes is most appropriate - `limited, block, full`. `limited` means that we compute the finite differences sensitivity at 5 selected points only. `block` allows the sensitivity to be computed at a block of points within *block_imin, block_imax* and *block_jmin, block_jmax*. `full` computes the sensitivities for all points, which can be prohibitively expensive.
- Copy the adjoint template file `test_ad/tapenade_m_adjoint_template.F90` to `src/subroutines/tapenade/tapenade_m.F90`. This template file contains useful directives (these directives are comments to F90 compilers, but serve as reference strings to locate the correct line for our Python scripts) which the Python functions can leverage to perturb the correct independent variable in the right direction and

compute the sensitivities, for example `!@ python_automated_grdchk limited_or_block_or_full @ ! @ python_automated_grdchk @`. The following code snippet performs this task.

```
setup_grdchk(ind_var = ind_var, header = header, domain = domain,
             dimension = ind_var_dim,
             z_co_ord = ind_var_z_co_ord,
             perturbation = perturbation,
             limited_or_block_or_full = limited_or_block_or_full,
             block_imin = block_imin, block_imax = block_imax, block_jmin = block_jmin,␣
↪block_jmax = block_jmax,
             tapenade_m_file = tapenade_m_file,
             unit = unit)
```

3. Compile the code

```
compile_code(mode = 'grdchk', header = header, domain = domain,
             clean = True, dep_var=dep_var, ind_vars = ind_var)
```

4. Run the executable

```
run_executable('grdchk')
```

All of the above steps are bundled within the `simulation` function, which can be run as follows -

```
simulation(mode = 'grdchk', header = header, domain = domain,
                   ind_var = ind_var, dep_var = dep_var,
                   limited_or_block_or_full = limited_or_block_or_full,
                   block_imin = block_imin, block_imax = block_imax,
                   block_jmin = block_jmin, block_jmax = block_jmax,
                   ind_var_dim = dimension, ind_var_z_co_ord = z_co_ord,
                   perturbation = perturbation,
                   run_executable_auto = True)
```

Automated Tangent Linear Mode

A typical tangent linear mode simulation is set up by giving the correct dependent and independent variables to Tapenade, using the correct I/O for the differentiated variables, and compiling correctly. This can be done easily using the Python functions in `test_ad/tapenade_config.py`.

1. Copy the correct header file to `src/sico_specs.h`

```
copy_file(f'../runs/headers/sico_specs_{header}.h', 'sico_specs.h')
```

2. Set up the I/O for the differentiated variable

This step involves the following sub-steps:

- Decide which of the three modes is most appropriate - `limited, block, full`. `limited` means that we compute the finite differences sensitivity at 5 selected points only. `block` allows the sensitivity to be computed at a block of points within *block_imin, block_imax* and *block_jmin, block_jmax*. `full` computes the sensitivities for all points, which can be prohibitively expensive.
- Copy the TLM template file `test_ad/tapenade_m_tlm_template.F90` to `src/subroutines/tapenade/tapenade_m.F90`. This template file contains useful directives (these directives are comments to F90 compilers, but serve as reference strings to locate the correct line for our Python scripts) which the Python functions

can use to correctly set up the loop for getting the directional derivatives as well as the I/O, for example `! @ python_automated_tlm dep_vard @ !@ python_automated_tlm limited_or_block_or_full @`. The following code snippet performs this task.

```
setup_forward(ind_var = ind_var, header = header, domain = domain,
              dimension = ind_var_dim,
              z_co_ord = ind_var_z_co_ord, limited_or_block_or_full = limited_or_block_
↪or_full,
              block_imin = block_imin, block_imax = block_imax,
              block_jmin = block_jmin, block_jmax = block_jmax,
              tapenade_m_file = tapenade_m_file,
              unit = unit)
```

3. Compile the code.

```
compile_code(mode = 'forward', header = header, domain = domain,
             clean = True, dep_var=dep_var, ind_vars = ind_var)
```

4. Run the executable.

```
run_executable('forward')
```

All of the above steps are bundled within the `simulation` function, which can be run as follows -

```
simulation(mode = 'forward', header = header, domain = domain,
                  ind_var = ind_var, dep_var = dep_var,
                  limited_or_block_or_full = limited_or_block_or_full,
                  block_imin = block_imin, block_imax = block_imax,
                  block_jmin = block_jmin, block_jmax = block_jmax,
                  ind_var_dim = dimension, ind_var_z_co_ord = z_co_ord,
                  run_executable_auto = True)
```

NOTE: While Tapenade can accept multiple independent variables at once, this automated script at least for now accepts only one independent variable at a time.

Automated Adjoint Mode

The adjoint mode has the most possible options of what can be done with it. A typical adjoint simulation is set up by giving the correct dependent and independent variables to Tapenade, using the correct I/O for the differentiated variables, and compiling correctly. This can be done easily using the Python functions in `test_ad/tapenade_config.py`. In addition, we can get the outputs of other adjoint variables, normal variables, both 2D and 3D at different time steps using the python script.

1. Set up checkpointing for the time loop, if necessary.

```
setup_binomial_checkpointing(status = True, number_of_steps = ckp_num)
```

2. Copy the correct header file to `src/sico_specs.h`

```
copy_file(f'../runs/headers/sico_specs_{header}.h', 'sico_specs.h')
```

3. Compile the code once.

```
compile_code(mode = mode, header = header, domain = domain,
        clean = True, dep_var=dep_var, ind_vars = ind_var)
```

4. Set up the I/O for the differentiated variable, as well as other variables the user might specify.

This step involves the following sub-steps:

- Copy the adjoint template file `test_ad/tapenade_m_adjoint_template.F90` to `src/subroutines/tapenade/tapenade_m.F90`. Set up I/O for the independent variable.
- Modify `src/sico_main_loop_m_cpp_b.f90` to write the variables the user specifies to appropriate files at correct times.
  - **NOTE** - This implementation is a bit dependent on the strings in the differentiated code. For now, the Python script searches for certain strings in the differentiated code to decide where to add the I/O statements. Depending on configurations, these strings might not even be present in `src/sico_main_loop_m_cpp_b.f90`, in which case the user would have to modify the script suitably after taking a look at `src/sico_main_loop_m_cpp_b.f90`.

```
setup_adjoint(ind_vars = [ind_var], header = header, domain = domain, ckp_status = ckp_
↪status,
              numCore_cpp_b_file = numCore_cpp_b_file,
              sico_main_loop_m_cpp_b_file = sico_main_loop_m_cpp_b_file,
              dimensions = [ind_var_dim],
              z_co_ords = [ind_var_z_co_ord],
              output_vars = output_vars, output_iters = output_iters, output_dims =␣
↪output_dims,
              output_adj_vars = output_adj_vars, output_adj_iters = output_adj_iters,
              output_adj_dims = output_adj_dims)
```

Here, `output_vars, output_iters, output_dims` are user specified normal variables to be output to a file. Similarly, `output_adj_vars, output_adj_iters, output_adj_dims` are user specified adjoint variables to be output to a file.

5. Compile the code again. Note that the `clean` flag is set to False, since we want the changes we made to stay.

```
compile_code(mode = mode, header = header, domain = domain,
clean = False, dep_var=dep_var, ind_vars = ind_var)
```

6. Run the executable -

```
run_executable('adjoint')
```

All of the above steps are bundled within the `simulation` function, which can be run as follows -

```
simulation(mode = 'adjoint', header = header, domain = domain,
      ind_var = ind_var, dep_var = dep_var,
      ind_var_dim = dimension, ind_var_z_co_ord = z_co_ord,
      run_executable_auto = True,
      output_vars = output_vars, output_iters = output_iters, output_dims = output_dims,
      output_adj_vars = output_adj_vars, output_adj_iters = output_adj_iters,
      output_adj_dims = output_adj_dims, ckp_status = ckp_status, ckp_num = checkpoint)
```

NOTE: While Tapenade can accept multiple independent variables at once, and this automated script accepts multiple independent variables too, we have only tested it with one independent variable at a time.

Input options

While executing the Python script the following input options are available to the users.

```
"-jsf", "--json", help="name of json data file", type=str
"-head", "--header", help="name of header file", type=str
"-dom", "--domain", help="short name of domain, either grl or ant", type = str
"-dv", "--dep_var", help="name of dependent variable", type=str
"-iv", "--ind_var", help="name of independent variable", type=str
"-delta", "--perturbation", help="value of perturbation for grdchk", type=float
"-ckp", "--checkpoint", help="number of steps in checkpointing", type=int
"--travis", help="travis setup", action="store_true"
"-dim", "--dimension", help="2D or 3D independent variable, default 2D", type=int
"-z", "--z_co_ord", help="z co-ordinate if 3D variable", type=int
'-ov','--output_vars', nargs='+', help='List the fields you want to output'
'-od', '--output_dims', nargs='+', help='List the z-coord of output vars, -1 if 2D'
'-oi', '--output_iters', nargs='+', help='List the iter num of output vars, -1 if␣
↪itercount_max'
'-oav','--output_adj_vars', nargs='+', help='List the adjoint fields you want to output'
'-oad', '--output_adj_dims', nargs='+', help='List the z-coord of adjoint output vars, -
↪1 if 2D'
'-oai', '--output_adj_iters', nargs='+', help='List the iter num of adjoint output vars,␣
↪-1 if itercount_max'
```

Using all of these options on terminal can get cumbersome. Alternatively, one can use an input `json` file as explained below.

Reading inputs from a file

We use the "header" files in the `json` format to provide inputs to `test_ad/tapenade_config.py`. Note that this is an alternative to writing the entire python command on the terminal, which can get cumbersome and unwieldy. Note that one can also provide some options on terminal, and some in the `inputs.json` file. If an option is specified both on the terminal and in the json file, the value specified on the terminal takes precedence.

Sample inputs.json file

A sample `inputs.json` file is provided here that acts as a "header" file for our AD workflow.

- `test_ad/inputs.json`

```
{
    "json": "inputs.json",
    "header": "v5_grl20_ss25ka",
    "domain": "grl",
    "dep_var": "fc",
    "ind_var": "H",
    "perturbation": 0.001,
    "checkpoint": 4,
    "travis": false,
    "dimension": 2,
    "z_co_ord": null,
    "output_vars": ["H", "vx_c", "vy_c", "H", "vx_c", "vy_c", "H", "vx_c", "vy_c", "H",
```

(continues on next page)

(continued from previous page)

```
↪"vx_c", "vy_c"],
    "output_dims": [-1, 40, 40, -1, 40, 40, -1, 40, 40],
    "output_iters": ["-1", "-1", "-1", "1", "1", "1", "2", "2", "2"],
    "output_adj_vars": ["H", "H", "H", "vx_c", "vx_c", "vx_c", "vy_c", "vy_c", "vy_c"],
    "output_adj_dims": ["-1", "-1", "-1", "40", "40", "40", "40", "40", "40"],
    "output_adj_iters": ["1", "2", "-1", "1", "2", "-1", "1", "2", "-1"]
}
```

Validation

Validation of AD (`adjoint, forward`) with finite differences (`grdchk`) can be performed as follows (within the defined tolerance `TOL`) -

```
validate_FD_AD(grdchk_file, ad_file, tolerance = TOL)
```

2.7 Papers

2.8 Acknowledgements

This work was supported in part by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357, and by National Science Foundation OPP/P2C2 grant #1903596.

BIBLIOGRAPHY

[LNGH20] L. C. Logan, S. H. K. Narayanan, R. Greve, and P. Heimbach. Sicopolis-ad v1: an open-source adjoint modeling framework for ice sheet simulation enabled by the algorithmic differentiation tool openad. *Geoscientific Model Development*, 13(4):1845–1864, 2020. URL: https://gmd.copernicus.org/articles/13/1845/2020/, doi:10.5194/gmd-13-1845-2020.