Dedalus Project Documentation

Dedalus Collaboration

Nov 27, 2021

CONTENTS

Dedalus is a flexible framework for solving partial differential equations using spectral methods. The code is open-source and developed by a team of researchers studying astrophysical, geophysical, and biological fluid dynamics.

Dedalus is written primarily in Python and features an easy-to-use interface with symbolic equation entry. Our numerical algorithm produces sparse systems for a wide variety of equations and spectrally-discretized domains. These systems are efficiently solved using compiled libraries and are automatically parallelized using MPI.

CHAPTER

ONE

DOC CONTENTS

1.1 Installing Dedalus

Dedalus is a Python 3 package that includes custom C-extensions (compiled with Cython) and that relies on MPI, FFTW (linked to MPI), HDF5, and a basic scientific-Python stack: numpy, scipy, mpi4py (linked to the same MPI), and h5py.

We recommend using conda to build a Python environment with all the necessary prerequisites, as described in the conda instructions below. This procedure can be easily customized to link to existing MPI/FFTW/HDF5 libraries, which may be preferable when installing Dedalus on a cluster.

Once you have the necessary C dependencies (MPI, FFTW+MPI, and HDF5), as well as Python 3, you should be able to install Dedalus from PyPI or build it from source.

1.1.1 Conda installation (recommended)

We recommend installing Dedalus via a conda script that will create a new conda environment with a complete Dedalus installation. The script allows you to link against custom MPI/FFTW/HDF5 libraries or opt for builds of those packages that are available through conda.

First, install conda/miniconda for your system if you don't already have it, following the instructions from conda. Then download the Dedalus conda installation script from this link or using:

```
curl https://raw.githubusercontent.com/DedalusProject/dedalus_conda/master/install_conda.
↪sh --output install_conda.sh
```

Modify the options at the top of the script to change the name of the resulting conda environment, link against custom MPI/FFTW/HDF5 libraries, choose between OpenBLAS and MKL-based numpy/scipy, and more. Then activate the base conda environment and run the script to build a new conda environment with Dedalus and its dependencies, as requested:

```
conda activate base
bash install_conda.sh
```

To use Dedalus, you simply need to activate the new environment. You can test the installation works using the command-line interface:

```
conda activate dedalus
python3 -m dedalus test
```

The Dedalus package within the environment can be updated using pip as described below.

1.1.2 Installing the Dedalus package

Once the necessary C dependencies and Python 3 are present, Dedalus can be installed from PyPI or built from source using pip.

Note: the instructions in this section assume the `pip3` command is hitting the right Python 3 installation. You can check this by making sure that `which pip3` and `which python3` reside in the same location. If not, you may need to use `pip` or `python3 -m pip` instead of `pip3` in the following commands.

Installing from PyPI

We currently only provide Dedalus on PyPI as a source distribution so that the Cython extensions are properly linked to your FFTW/MPI libraries at build-time. To install Dedalus from PyPI, first set the `FFTW_PATH` and `MPI_PATH` environment variables to the prefix paths for FFTW/MPI and then install using pip:

```
export FFTW_PATH=/path/to/your/fftw_prefix
export MPI_PATH=/path/to/your/mpi_prefix
pip3 install dedalus
```

Building from source

Alternately, to build the lastest version of Dedalus from source: clone the repository, set FFTW/MPI paths, and install using pip:

```
git clone https://github.com/DedalusProject/dedalus
cd dedalus
export FFTW_PATH=/path/to/your/fftw_prefix
export MPI_PATH=/path/to/your/mpi_prefix

pip3 install .
```

Updating Dedalus

If Dedalus was installed using the conda script or manually from PyPI, it can be updated pip:

```
pip3 install --upgrade dedalus
```

Note: any custom FFTW/MPI paths set in the conda script or during the original installation will also need to be exported for the update command to work.

If Dedalus was built from source, it can be updated by first pulling new changes from the source repository, and then reinstalling with pip:

```
cd /path/to/dedalus/repo
git pull
export FFTW_PATH=/path/to/your/fftw_prefix
export MPI_PATH=/path/to/your/mpi_prefix
pip3 install --upgrade --force-reinstall .
```

Uninstalling Dedalus

If Dedalus was installed using pip, it can be uninstalled using:

```
pip3 uninstall dedalus
```

1.1.3 Alternative installation procedures

Note: We strongly recommend installing Dedalus using conda, as described above. These alternative procedures may be out-of-date and are provided for historical reference and expert use.

Full-stack installation shell script

This all-in-one installation script will build an isolated stack containing a Python installation and the other dependencies needed to run Dedalus. In most cases, the script can be modified to link with system installations of FFTW, MPI, and linear algebra libraries.

You can get down the installation script using:

```
wget https://raw.githubusercontent.com/DedalusProject/dedalus/master/docs/install.sh
```

and execute it using:

```
bash install.sh
```

The installation script has been tested on a number of Linux distributions and OS X in the past, but is generally no longer supported. Please consider using the conda-based installation instead.

Install notes for PSC/Bridges: Intel stack

Here we build using the recommended Intel compilers. Bridges comes with python 3.4 at present, but for now we'll maintain a boutique build to keep access to python >=3.5 and to tune numpy performance by hand (though the value proposition of this should be tested).

Then add the following to your `.bash_profile`:

```
# Add your commands here to extend your PATH, etc.

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH
export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH

export CC=mpiicc

export I_MPI_CC=icc

#pathing for Dedalus
export LOCAL_PYTHON_VERSION=3.5.1
export LOCAL_NUMPY_VERSION=1.11.0
export LOCAL_SCIPY_VERSION=0.17.0
export LOCAL_HDF5_VERSION=1.8.16
```

(continues on next page)

(continued from previous page)

```
export LOCAL_MERCURIAL_VERSION=3.7.3

export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME
```

Python stack

Here we use the recommended Intel mpi compilers, rather than our own openmpi.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz --no-check-certificate
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION
./configure --prefix=$BUILD_HOME \
                     OPT="-w -vec-report0 -opt-report0" \
                     FOPT="-w -vec-report0 -opt-report0" \
                     CFLAGS="-mkl -O3 -ipo -xCORE-AVX2 -fPIC" \
                     CPPFLAGS="-mkl -O3 -ipo -xCORE-AVX2 -fPIC" \
                     F90FLAGS="-mkl -O3 -ipo -xCORE-AVX2 -fPIC" \
                     CC=mpiicc  CXX=mpiicpc F90=mpiifort  \
                     LDFLAGS="-lpthread"
make -j
make install
```

The previous intel patch is no longer required.

Installing pip

Python 3.4+ now automatically includes pip.

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

We suggest doing the following immediately to suppress version warning messages:

```
pip3 install --upgrade pip
```

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

This required setting the `I_MPI_CC=icc` envirnoment variable above; otherwise we keep hitting gcc.

Installing FFTW3

We build our own FFTW3:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpiicc        CFLAGS="-O3 -xCORE-AVX2" \
                      CXX=mpiicpc CPPFLAGS="-O3 -xCORE-AVX2" \
                      F77=mpiifort  F90FLAGS="-O3 -xCORE-AVX2" \
                      MPICC=mpiicc MPICXX=mpiicpc \
                      LDFLAGS="-lmpi" \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make -j
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library.

Building numpy against MKL

Now, acquire `numpy`. The login nodes for Bridges are 14-core Haswell chips, just like the compute nodes, so let's try doing it with the normal numpy settings (no patching to adjust the compiler commands in distutils for cross-compiling). Ah shoots. Nope. The numpy distutils only employs xSSE4.2 and none of the AVX2 arch flags, nor a basic xhost. Well. On we go. Change `-xSSE4.2` to `-xCORE-AVX2` in `numpy/distutils/intelccompiler.py` and `numpy/distutils/fcompiler/intel.py`. We should really put in a PR and an ability to pass flags via `site.cfg` or other approach.

Here's an automated way to do this, using `numpy_intel.patch`.:

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-$LOCAL_
↪NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_intel.patch
patch -p1 < numpy_intel.patch
```

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cat >> site.cfg << EOF
[mkl]
library_dirs = /opt/packages/intel/compilers_and_libraries/linux/mkl/lib/intel64
include_dirs = /opt/packages/intel/compilers_and_libraries/linux/mkl/include
mkl_libs = mkl_rt
lapack_libs =
EOF
```

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

Numpy has changed the location of _dotblas, so our old test doesn't work. From the dot product speed, it looks like we have succesfully linked against fast BLAS and the test results look relatively normal, but this needs to be looked in to.

Python library stack

After numpy has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Scipy now is no longer hosted at sourceforge for anything past v0.16, so lets try git:

```
git clone git://github.com/scipy/scipy.git scipy
cd scipy
# fall back to stable version
git checkout tags/v$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_ext␣
↪\
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day. Again. Still.

Installing matplotlib

This should just be pip installed. In versions of matplotlib>1.3.1, Qhull has a compile error if the C compiler is used rather than C++, so we force the C complier to be icpc

```
export CC=icpc
pip3 install matplotlib
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O. Intel compilers are failing on this when done with mpi-sgi, and on NASA's recommendation we're falling back to gcc for this library:

```
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-$LOCAL_
↪HDF5_VERSION.tar.gz
tar xzvf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME CC=mpiicc CXX=mpiicpc F77=mpiifort \
                    --enable-shared --enable-parallel
make
make install
```

H5PY via pip

This can now just be pip installed (>=2.6.0):

```
pip3 install h5py
```

For now we drop our former instructions on attempting to install parallel h5py with collectives. See the NASA/Pleiades repo history for those notes.

Installing Mercurial

Here we install mercurial itself. Following NASA/Pleiades approaches, we will use gcc. I haven't checked whether the default bridges install has mercurial:

```
cd $BUILD_HOME
wget http://mercurial.selenic.com/release/mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
tar xvf mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
cd mercurial-$LOCAL_MERCURIAL_VERSION
module load gcc
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your `~/.hgrc`:: cat >> ~/.hgrc << EOF [ui] username = <your bitbucket username/e-mail address here> editor = emacs

[extensions] graphlog = color = convert = mq = EOF

Dedalus

Preliminaries

Then do the following:

```
cd $BUILD_HOME
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
# this has some issues with mpi4py versioning --v
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Bridges

Our scratch disk system on Bridges is `/pylon1/group-name/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /pylon1/group-name/user-name workdir
```

Long-term spinning storage is on `/pylon2` and is provided by allocation request.

Install notes for Compute Canada's clusters

Notes

The following instructions have been provided by Maxime Boissonneault, staff at Compute Canada. Last updated 2021-01-11.

Instructions

Compute Canada already pre-builds Dedalus on demand. You can contact https://docs.computecanada.ca/wiki/Technical_support in order to request a new version. In your home:

```
module purge
module load StdEnv/2020 fftw-mpi mpi4py hdf5-mpi/1.12 python
```

Now build and activate the virtual environment for your installation. You should also update pip as soon as the environment is activated:

```
virtualenv --no-download python_env
source python_env/bin/activate
pip install --no-index --upgrade pip
```

Now install dedalus:

```
pip install --no-index dedalus
```

Compute Canada recommends creating your virtual environment and loading your modules in your job scripts. For more information about doing so, please see https://docs.computecanada.ca/wiki/Python#Creating_and_using_a_virtual_environment

Install notes for MIT Engaging Cluster

This installation uses the Python, BLAS, and MPI modules available on Engaging, while manually building HDF5 and FFTW. Be sure to login to Engaging through the `eofe7` head-node.

Modules and paths

The following commands should be added to your `~/.bashrc` file to setup the correct modules and paths. Modify the `HDF5_DIR`, `FFTW_PATH`, and `DEDALUS_REPO` environment variables as desired to change the build locations of these packages.

```
# Basic modules
module load gcc
module load slurm

# Python from modules
module load engaging/python/2.7.10
module load engaging/python/3.6.0
export PATH=~/.local/bin:${PATH}

# BLAS from modules
```

(continues on next page)

(continued from previous page)

```
module load engaging/OpenBLAS/0.2.14
export BLAS=/cm/shared/engaging/OpenBLAS/0.2.14/lib/libopenblas.so

# MPI from modules
module load engaging/openmpi/2.0.3
export MPI_PATH=/cm/shared/engaging/openmpi/2.0.3

# HDF5 built from source
export HDF5_DIR=~/software/hdf5
export HDF5_VERSION=1.10.1
export HDF5_MPI="ON"
export PATH=${HDF5_DIR}/bin:${PATH}
export LD_LIBRARY_PATH=${HDF5_DIR}/lib:${LD_LIBRARY_PATH}

# FFTW built from source
export FFTW_PATH=~/software/fftw
export FFTW_VERSION=3.3.6-pl2
export PATH=${FFTW_PATH}/bin:${PATH}
export LD_LIBRARY_PATH=${FFTW_PATH}/lib:${LD_LIBRARY_PATH}

# Dedalus from mercurial
export DEDALUS_REPO=~/software/dedalus
export PYTHONPATH=${DEDALUS_REPO}:${PYTHONPATH}
```

Build procedure

Source your ~/.bashrc to activate the above changes, or re-login to the cluster, before running the following build procedure.

```
# Python basics
/cm/shared/engaging/python/2.7.10/bin/pip install --ignore-installed --user pip
/cm/shared/engaging/python/3.6.0/bin/pip3 install --ignore-installed --user pip
pip2 install --user --upgrade setuptools
pip2 install --user mercurial
pip3 install --user --upgrade setuptools
pip3 install --user nose cython

# Python packages
pip3 install --user --no-use-wheel numpy
pip3 install --user --no-use-wheel scipy
pip3 install --user mpi4py

# HDF5 built from source
mkdir -p ${HDF5_DIR}
cd ${HDF5_DIR}
wget https://support.hdfgroup.org/ftp/HDF5/current/src/hdf5-${HDF5_VERSION}.tar
tar -xvf hdf5-${HDF5_VERSION}.tar
cd hdf5-${HDF5_VERSION}
./configure --prefix=${HDF5_DIR} \
    CC=mpicc \
    CXX=mpicxx \
```

(continues on next page)

(continued from previous page)

```
    F77=mpif90 \
    MPICC=mpicc \
    MPICXX=mpicxx \
    --enable-shared \
    --enable-parallel
make
make install
pip3 install --user --no-binary=h5py h5py

# FFTW built from source
mkdir -p ${FFTW_PATH}
cd ${FFTW_PATH}
wget http://www.fftw.org/fftw-${FFTW_VERSION}.tar.gz
tar -xvzf fftw-${FFTW_VERSION}.tar.gz
cd fftw-${FFTW_VERSION}
./configure --prefix=${FFTW_PATH} \
    CC=mpicc \
    CXX=mpicxx \
    F77=mpif90 \
    MPICC=mpicc \
    MPICXX=mpicxx \
    --enable-shared \
    --enable-mpi \
    --enable-openmp \
    --enable-threads
make
make install

# Dedalus from mercurial
hg clone https://bitbucket.org/dedalus-project/dedalus ${DEDALUS_REPO}
cd ${DEDALUS_REPO}
pip3 install --user -r requirements.txt
python3 setup.py build_ext --inplace
```

Notes

Last updated on 2017/09/18 by Keaton Burns.

Install notes for CU/Janus

As with NASA/Pleiades, an initial Janus environment is pretty bare-bones. There are no modules, and your shell is likely a bash varient. Here we'll do a full build of our stack, using only the prebuilt openmpi compilers. Later we'll try a module heavy stack to see if we can avoid this.

Add the following to your `.my.bash_profile`:

```
# Add your commands here to extend your PATH, etc.

module load intel
```

(continues on next page)

(continued from previous page)

```
export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH

export CC=mpicc

#pathing for Dedalus
export LOCAL_MPI_VERSION=openmpi-1.8.5
export LOCAL_MPI_SHORT=v1.8
export LOCAL_PYTHON_VERSION=3.4.3
export LOCAL_NUMPY_VERSION=1.9.2
export LOCAL_SCIPY_VERSION=0.15.1
export LOCAL_HDF5_VERSION=1.8.15


export MPI_ROOT=$BUILD_HOME/$LOCAL_MPI_VERSION
export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME
```

Do your builds on the janus compile nodes (see MOTD). As a positive note, Janus compile nodes have access to the internet (e.g., wget), so you can download and compile on-node. For now we're using stock Pleiades compile flags and patch files. With intel 15.0.1 the cython install is now working well, as does h5py.

Building Openmpi

Tim Dunn has pointed out that we may (may) be able to get some speed improvements by building our own openmpi. Why not give it a try! Compiling on the janus-compile nodes seems to do a fine job, and unlike Pleiades we can grab software from the internet on the compile nodes too. This streamlines the process.:

```
cd $BUILD_HOME
wget http://www.open-mpi.org/software/ompi/$LOCAL_MPI_SHORT/downloads/$LOCAL_MPI_VERSION.
↪tar.gz
tar xvf $LOCAL_MPI_VERSION.tar.gz
cd $LOCAL_MPI_VERSION
./configure \
    --prefix=$BUILD_HOME \
    --with-slurm \
    --with-threads=posix \
    --enable-mpi-thread-multiple \
    CC=icc CXX=icpc FC=ifort

make -j
make install
```

Config flags thanks to Tim Dunn; the CFLAGS etc are from Pleiades and should be general.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION

./configure --prefix=$BUILD_HOME \
                    CC=mpicc         CFLAGS="-mkl -O3 -axAVX -xSSE4.1 -fPIC -ipo" \
                    CXX=mpicxx CPPFLAGS="-mkl -O3 -axAVX -xSSE4.1 -fPIC -ipo" \
                    F90=mpif90  F90FLAGS="-mkl -O3 -axAVX -xSSE4.1 -fPIC -ipo" \
                    --enable-shared LDFLAGS="-lpthread" \
                    --with-cxx-main=mpicxx --with-system-ffi

make -j
make install
```

The former patch for Intel compilers to handle ctypes is no longer necessary.

Installing pip

Python 3.4 now automatically includes pip.

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b, or under openmpi:

```
 cd $BUILD_HOME
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar xvzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                     CC=mpicc        CFLAGS="-O3 -axAVX -xSSE4.1" \
                     CXX=mpicxx CPPFLAGS="-O3 -axAVX -xSSE4.1" \
                     F77=mpif90  F90FLAGS="-O3 -axAVX -xSSE4.1" \
                     MPICC=mpicc MPICXX=mpicxx \
```

(continues on next page)

(continued from previous page)

```
                    --enable-shared \
                    --enable-mpi --enable-openmp --enable-threads

make -j
make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Cython is now working (intel 15.0/openmpi-1.8.5).

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the OpenBLAS libraries.

All of the intel patches, etc. are unnecessary in the gcc stack.

Building numpy against MKL

Now, acquire `numpy` (1.9.0):

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-$LOCAL_
↪NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_janus_intel_patch.
↪tar
tar xvf numpy_janus_intel_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_janus_intel_patch.tar` which is auto-deployed within the `numpy-$LOCAL_NUMPY_VERSION` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

This differs from prior versions in that "-xhost" is replaced with "-axAVX -xSSE4.1".

We'll now need to make sure that numpy is building against the MKL libraries. Start by making a site.cfg file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit site.cfg in the [mkl] section; modify the library directory so that it correctly point to TACC's $MKLROOT/lib/intel64/. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /curc/tools/x_86_64/rh6/intel/15.0.1/composer_xe_2015.1.133/mkl/lib/
↪intel64
include_dirs = /curc/tools/x_86_64/rh6/intel/15.0.1/composer_xe_2015.1.133/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script numpy_test_full. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After numpy has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/$LOCAL_SCIPY_VERSION/scipy-$LOCAL_
↪SCIPY_VERSION.tar.gz
tar -xvf scipy-$LOCAL_SCIPY_VERSION.tar.gz
cd scipy-$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_ext␣
↪\
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed. In versions of matplotlib>1.3.1, Qhull has a compile error if the C compiler is used rather than C++, so we force the C complier to be icpc

```
export CC=icpc
pip3 install matplotlib
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-$LOCAL_
↪HDF5_VERSION.tar.gz

tar xvzf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME \
                    CC=mpicc         CFLAGS="-O3 -axAVX -xSSE4.1" \
                    CXX=mpicxx CPPFLAGS="-O3 -axAVX -xSSE4.1" \
                    F77=mpif90  F90FLAGS="-O3 -axAVX -xSSE4.1" \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel

make -j
make install
```

Installing h5py

This now can be pip installed:

```
pip3 install hypy
```

Installing Mercurial

On Janus, we need to install mercurial itself. I can't get mercurial to build properly on intel compilers, so for now use gcc. Ah, and we also need python2 for the mercurial build (only):

```
module unload openmpi intel
module load gcc/gcc-4.9.1
module load python/anaconda-2.0.0
wget http://mercurial.selenic.com/release/mercurial-3.4.tar.gz
tar xvf mercurial-3.4.tar.gz
```

(continues on next page)

(continued from previous page)

```
cd mercurial-3.4
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your ~/.hgrc:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus

Preliminaries

With the modules set as above, set:

```
export BUILD_HOME=$BUILD_HOME
export FFTW_PATH=$BUILD_HOME
export MPI_PATH=$BUILD_HOME/$LOCAL_MPI_VERSION
```

Pull the dedalus repository::

```
hg clone https://bitbucket.org/dedalus-project/dedalus
```

Then change into your root dedalus directory and run:

```
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Janus

Our scratch disk system on Pleiades is `/lustre/janus_scratch/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /lustre/janus_scratch/bpbrown workdir
```

I also suggest you move your stack to the `projects` directory, `/projects/user-name`. There, I bring back a symbolic link:

ln -s /projects/bpbrown projects ln -s projects/build build

Install notes for Mac OS X (10.9)

These instructions assume you're starting with a clean Mac OS X system, which will need `python3` and all scientific packages installed.

Mac OS X cookbook

```
#!bash

# Homebrew
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew update
brew doctor
# ** Fix any errors raised by brew doctor before proceeding **

# Prep system
brew install gcc
brew install swig

# Python 3
brew install python3

# Scientific packages for Python 3
brew tap homebrew/science
brew install suite-sparse
pip3 install nose
pip3 install numpy
pip3 install scipy
brew install libpng
brew install freetype
pip3 install matplotlib

# MPI
brew install openmpi
pip3 install mpi4py

# FFTW
brew install fftw --with-mpi

# HDF5
brew install hdf5
pip3 install h5py

# Dedalus
# ** Change to the directory where you want to keep the Dedalus repository **
brew install hg
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Detailed install notes for Mac OS X (10.9)

Preparing a Mac system

First, install Xcode from the App Store and seperately install the Xcode Command Line Tools. To install the command line tools, open Xcode, go to `Preferences`, select the `Downloads` tab and `Components`. These command line tools install `make` and other requisite tools that are no longer automatically included in Mac OS X (as of 10.8).

Next, you should install the Homebrew package manager for OS X. Run the following from the Terminal:

```
#!bash
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew update
brew doctor
```

Cleanup any problems identified by `brew doctor` before proceeding.

To complete the `scipy` install process, we'll need `gfortran` from `gcc` and `swig`, which you can install from Homebrew:

```
#!bash
brew install gcc
brew install swig
```

Install Python 3

Now, install python3 from Homebrew:

```
#!bash
brew install python3
```

Scientific packages for Python3

Next install the numpy and scipy scientific packages. To adequately warn you before proceeding, properly installing numpy and scipy on a Mac can be a frustrating experience.

Start by proactively installing UMFPACK from suite-sparse, located in homebrew-science on https://github.com/Homebrew/homebrew-science. Failing to do this may lead to a series of perplexing UMFPACK errors during the scipy install.

```
#!bash
brew tap homebrew/science
brew install suite-sparse
```

Now use pip, the (the standard Python package management system, installed with Python via Homebrew) to install `nose`, `numpy`, and `scipy` in order:

```
#!bash
pip3 install nose
pip3 install numpy
pip3 install scipy
```

The `scipy` install can fail in a number of surprising ways. Be especially wary of custom settings to `LDFLAGS`, `CPPFLAGS`, etc. within your shell; these may cause the `gfortran` compile step to fail spectacularly.

Also install `matplotlib`, the main Python plotting library, along with its dependencies, using Homebrew and pip:

```
#!bash
brew install libpng
brew install freetype
pip3 install matplotlib
```

Other libraries

Dedalus is parallelized using MPI, and we recommend using the Open MPI library on OS X. The Open MPI library and Python wrappers can be installed using Homebrew and pip:

```
#!bash
brew install openmpi
pip3 install mpi4py
```

Dedalus uses the FFTW library for transforms and parallelized transposes, and can be installed using Homebrew:

```
#!bash
brew install fftw --with-mpi
```

Dedalus uses HDF5 for data storage. The HDF5 library and Python wrappers can be installed using Homebrew and pip:

```
#!bash
brew install hdf5
pip3 install h5py
```

Installing the Dedalus package

Dedalus is managed using the Mercurial distributed version control system, and hosted online though Bitbucket. Mercurial (referred to as `hg`) can be installed using homebrew, and can then be used to download the latest copy of Dedalus (note: you should change to the directory where you want the put the Dedalus repository):

```
#!bash
brew install hg
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
```

A few other Python packages needed by Dedalus are listed in the `requirements.txt` file in the Dedalus repository, and can be installed using pip:

```
#!bash
pip3 install -r requirements.txt
```

You then need to build Dedalus's Cython extensions from within the repository using the `setup.py` script. This step should be perfomed whenever updates are pulled from the main repository (but it is only strictly necessary when the Cython extensions are modified).

```
#!bash
python3 setup.py build_ext --inplace
```

Finally, you need to add the Dedalus repository to the Python search path so that the `dedalus` package can be imported. To do this, add the following to your `~/.bash_profile`, substituting in the path to the Dedalus repository you cloned using Mercurial:

```
# Add Dedalus repository to Python search path
export PYTHONPATH=<PATH/TO/DEDALUS/REPOSITORY>:$PYTHONPATH
```

Other resources

http://www.lowindata.com/2013/installing-scientific-python-on-mac-os-x/

http://stackoverflow.com/questions/12574604/scipy-install-on-mountain-lion-failing

https://github.com/jonathansick/dotfiles/wiki/Notes-for-Mac-OS-X

Install notes for NASA/Discover

This installation is fairly straightforward because most of the work has already been done by the NASA/Discover staff, namely Jules Kouatchou.

First, add the following lines to your `~/.bashrc` file and source it:

```
module purge
module load other/comp/gcc-4.9.1
module load lib/mkl-15.0.0.090
module load other/Py3Dist/py-3.4.1_gcc-4.9.1_mkl-15.0.0.090
module load other/mpi/openmpi/1.8.2-gcc-4.9.1

export BUILD_HOME=$HOME/build
export PYTHONPATH=$HOME/dedalus2
```

This loads the gcc compiler, MKL linear algebra package, openmpi version 1.8.2, and crucially various python3 libraries. To see the list of python libraries,

```
listPyPackages
```

We actually have all the python libraries we need for Dedalus. However, we still need fftw. To install fftw,

```
mkdir build

cd $BUILD_HOME
wget http://www.fftw.org/fftw-3.3.4.tar.gz
tar -xzf fftw-3.3.4.tar.gz
cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F77=mpif90 \
                    MPICC=mpicc MPICXX=mpicxx \
```

(continues on next page)

(continued from previous page)

```
                    --enable-shared \
                    --enable-mpi --enable-openmp --enable-threads

make
make install
```

All that remains is to pull Dedalus down from Bitbucket and install it.

```
cd $HOME
hg clone https://bitbucket.org/dedalus-project/dedalus2

export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME
export MPI_DIR=/usr/local/other/SLES11.1/openMpi/1.8.2/gcc-4.9.1
cd $HOME/dedalus2
python3 setup.py build_ext --inplace
```

Install notes for NASA/Pleiades

Best performance is coming from our newly developed Pleaides/Intel/MKL stack; we've retained our gcc/openblas build for future use.

Install notes for NASA/Pleiades: Intel stack

An initial Pleiades environment is pretty bare-bones. There are no modules, and your shell is likely a csh varient. To switch shells, send an e-mail to support@nas.nasa.gov; I'll be using `bash`.

Then add the following to your `.profile`:

```
# Add your commands here to extend your PATH, etc.

module load comp-intel
module load git
module load openssl
module load emacs

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/nasa/openssl/1.0.1h/lib/:$LD_LIBRARY_PATH

export CC=mpicc

#pathing for Dedalus
export LOCAL_MPI_VERSION=openmpi-1.10.1
export LOCAL_MPI_SHORT=v1.10

# falling back to 1.8 until we resolve tcp wireup errors
# (probably at runtime with MCA parameters)
```

(continues on next page)

(continued from previous page)

```
export LOCAL_MPI_VERSION=openmpi-1.8.6
export LOCAL_MPI_SHORT=v1.8

export LOCAL_PYTHON_VERSION=3.5.0
export LOCAL_NUMPY_VERSION=1.10.1
export LOCAL_SCIPY_VERSION=0.16.1
export LOCAL_HDF5_VERSION=1.8.15-patch1
export LOCAL_MERCURIAL_VERSION=3.6

export MPI_ROOT=$BUILD_HOME/$LOCAL_MPI_VERSION
export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME

# Openmpi forks:
export OMPI_MCA_mpi_warn_on_fork=0

# don't mess up Pleiades for everyone else
export OMPI_MCA_btl_openib_if_include=mlx4_0:1
```

Doing the entire build took about 2 hours. This was with several (4) open ssh connections to Pleaides to do poor-mans-parallel building (of python, hdf5, fftw, etc.), and one was on a dev node for the openmpi compile. The openmpi compile is time intensive and mus be done first. The fftw and hdf5 libraries take a while to build. Building scipy remains the most significant time cost.

Python stack

Interesting update. Pleiades now appears to have a python3 module. Fascinating. It comes with matplotlib (1.3.1), scipy (0.12), numpy (1.8.0) and cython (0.20.1) and a few others. Very interesting. For now we'll proceed with our usual build-it-from-scratch approach, but this should be kept in mind for the future. No clear mpi4py, and the `mpi4py` install was a hangup below for some time.

Building Openmpi

The suggested `mpi-sgi/mpt` MPI stack breaks with mpi4py; existing versions of openmpi on Pleiades are outdated and suffer from a previously identified bug (v1.6.5), so we'll roll our own. This needs to be built on a compute node so that the right memory space is identified.:

```
# do this on a main node (where you can access the outside internet):
cd $BUILD_HOME
wget http://www.open-mpi.org/software/ompi/$LOCAL_MPI_SHORT/downloads/$LOCAL_MPI_VERSION.
↪tar.gz
tar xvf $LOCAL_MPI_VERSION.tar.gz

# get ivy-bridge compute node
qsub -I -q devel -l select=1:ncpus=24:mpiprocs=24:model=has -l walltime=02:00:00

# once node exists
cd $BUILD_HOME
```

(continues on next page)

(continued from previous page)

```
cd $LOCAL_MPI_VERSION
./configure \
    --prefix=$BUILD_HOME \
    --enable-mpi-interface-warning \
    --without-slurm \
    --with-tm=/PBS \
    --without-loadleveler \
    --without-portals \
    --enable-mpirun-prefix-by-default \
    CC=icc CXX=icc FC=ifort

make -j
make install
```

These compilation options are based on `/nasa/openmpi/1.6.5/NAS_config.sh`, and are thanks to advice from Daniel Kokron at NAS. Compiling takes about 10-15 minutes with make -j.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz --no-check-certificate
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION

./configure --prefix=$BUILD_HOME \
                    OPT="-mkl -O3 -axCORE-AVX2 -xSSE4.2 -fPIC -ipo -w -vec-report0 -opt-
↪report0" \
                    FOPT="-mkl -O3 -axCORE-AVX2 -xSSE4.2 -fPIC -ipo -w -vec-report0 -
↪opt-report0" \
                    CC=mpicc CXX=mpicxx F90=mpif90 \
                    LDFLAGS="-lpthread" \
                    --enable-shared --with-system-ffi \
                    --with-cxx-main=mpicxx --with-gcc=mpicc

make
make install
```

The previous intel patch is no longer required.

Installing pip

Python 3.4 now automatically includes pip. We suggest you do the following immediately to suppress version warning messages:

```
pip3 install --upgrade pip
```

On Pleiades, you'll need to edit `.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.trust.crt
```

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b, or under openmpi:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpicc        CFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      CXX=mpicxx CPPFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      F77=mpif90  F90FLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the Intel MKL BLAS.

Building numpy against MKL

Now, acquire `numpy` (1.9.2):

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-$LOCAL_
↪NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_pleiades_intel_
↪patch.tar
tar xvf numpy_pleiades_intel_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_pleiades_intel_patch.tar` which is auto-deployed within the `numpy-$LOCAL_NUMPY_VERSION` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

This differs from prior versions in that "-xhost" is replaced with "-axCORE-AVX2 -xSSE4.2". NOTE: this is now updated for Haswell.

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Note: If you're doing many different builds, it may be helpful to have the numpy site.cfg shared between builds. If so, you can edit ~/.numpy-site.cfg instead of site.cfg. This is per site.cfg.example.

Edit `site.cfg` in the `[mkl]` section; modify the library directory so that it correctly point to TACC's `$MKLROOT/lib/intel64/`. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/lib/intel64/
include_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/$LOCAL_SCIPY_VERSION/scipy-$LOCAL_
↪SCIPY_VERSION.tar.gz
tar -xvf scipy-$LOCAL_SCIPY_VERSION.tar.gz
cd scipy-$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_ext␣
↪\
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed. However, we're hitting errors with qhull compilation in every part of the 1.4.x branch, so we fall back to 1.3.1:

```
pip3 install matplotlib==1.3.1
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-$LOCAL_
↪HDF5_VERSION.tar.gz
tar xzvf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME \
                    CC=mpicc         CFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                    CXX=mpicxx CPPFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                    F77=mpif90  F90FLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel
make
make install
```

H5PY via pip

This can now just be pip installed (>=2.6.0):

```
pip3 install h5py
```

For now we drop our former instructions on attempting to install parallel h5py with collectives. See the repo history for those notes.

Installing Mercurial

On NASA Pleiades, we need to install mercurial itself. I can't get mercurial to build properly on intel compilers, so for now use gcc:

```
cd $BUILD_HOME
wget http://mercurial.selenic.com/release/mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
tar xvf mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
cd mercurial-$LOCAL_MERCURIAL_VERSION
module load gcc
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your `~/.hgrc`:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[web]
cacerts = /etc/ssl/certs/ca-bundle.crt

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus

Preliminaries

Then do the following:

```
cd $BUILD_HOME
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Pleiades

Our scratch disk system on Pleiades is `/nobackup/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /nobackup/bpbrown workdir
```

Long-term mass storage is on LOU.

Install notes for NASA/Pleiades: Intel stack with MPI-SGI

Here we build using the recommended MPI-SGI environment, with Intel compilers. An initial Pleiades environment is pretty bare-bones. There are no modules, and your shell is likely a csh varient. To switch shells, send an e-mail to support@nas.nasa.gov; I'll be using `bash`.

Then add the following to your `.profile`:

```
# Add your commands here to extend your PATH, etc.

module load mpi-sgi/mpt
module load comp-intel
module load git
module load openssl
module load emacs
```

(continues on next page)

(continued from previous page)

```
export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/nasa/openssl/1.0.1h/lib/:$LD_LIBRARY_PATH

# proper wrappers for using Intel rather than GNU compilers,
# Thanks to Daniel Kokron at NASA.
export MPICC_CC=icc
export MPICXX_CXX=icpc

export CC=mpicc

#pathing for Dedalus
export LOCAL_PYTHON_VERSION=3.5.0
export LOCAL_NUMPY_VERSION=1.10.1
export LOCAL_SCIPY_VERSION=0.16.1
export LOCAL_HDF5_VERSION=1.8.15-patch1
export LOCAL_MERCURIAL_VERSION=3.6

export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME

# Pleaides workaround for QP errors 8/25/14 from NAS (only for MPI-SGI)
export MPI_USE_UD=true
```

Python stack

Here we use the recommended MPI-SGI compilers, rather than our own openmpi.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz --no-check-certificate
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION
./configure --prefix=$BUILD_HOME \
                     OPT="-w -vec-report0 -opt-report0" \
                     FOPT="-w -vec-report0 -opt-report0" \
                     CFLAGS="-mkl -O3 -ipo -axCORE-AVX2 -xSSE4.2 -fPIC" \
                     CPPFLAGS="-mkl -O3 -ipo -axCORE-AVX2 -xSSE4.2 -fPIC" \
                     F90FLAGS="-mkl -O3 -ipo -axCORE-AVX2 -xSSE4.2 -fPIC" \
                     CC=mpicc  CXX=mpicxx F90=mpif90  \
```

(continues on next page)

(continued from previous page)

```
                     --with-cxx-main=mpicxx --with-gcc=mpicc \
                     LDFLAGS="-lpthread" \
                    --enable-shared --with-system-ffi

make
make install
```

The previous intel patch is no longer required.

Installing pip

Python 3.4 now automatically includes pip.

On Pleiades, you'll need to edit `.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.trust.crt
```

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

We suggest doing the following immediately to suppress version warning messages:

```
pip3 install --upgrade pip
```

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

version >=2.0.0 seem to play well with mpi-sgi.

Installing FFTW3

We build our own FFTW3:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                     CC=icc        CFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                     CXX=icpc CPPFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                     F77=ifort  F90FLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                     MPICC=icc MPICXX=icpc \
                     LDFLAGS="-lmpi" \
                     --enable-shared \
                     --enable-mpi --enable-openmp --enable-threads
```

(continues on next page)

(continued from previous page)

```
make -j
make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the OpenBLAS libraries.

All of the intel patches, etc. are unnecessary in the gcc stack.

Building numpy against MKL

Now, acquire `numpy` (1.10.1):

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-$LOCAL_
↪NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_pleiades_intel_
↪patch.tar
tar xvf numpy_pleiades_intel_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_pleiades_intel_patch.tar` which is auto-deployed within the `numpy-$LOCAL_NUMPY_VERSION` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

This differs from prior versions in that "-xhost" is replaced with "-axCORE-AVX2 -xSSE4.2". I think this could be handled more gracefully using a extra_compile_flag option in the site.cfg.

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit `site.cfg` in the `[mkl]` section; modify the library directory so that it correctly point to TACC's `$MKLROOT/lib/intel64/`. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/lib/intel64/
include_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/$LOCAL_SCIPY_VERSION/scipy-$LOCAL_
↪SCIPY_VERSION.tar.gz
tar -xvf scipy-$LOCAL_SCIPY_VERSION.tar.gz
cd scipy-$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_ext␣
↪\
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed. In versions of matplotlib>1.3.1, Qhull has a compile error if the C compiler is used rather than C++, so we force the C complier to be icpc

```
export CC=icpc
pip3 install matplotlib
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O. Intel compilers are failing on this when done with mpi-sgi, and on NASA's recommendation we're falling back to gcc for this library:

```
export MPICC_CC=
export MPICXX_CXX=
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-$LOCAL_
↪HDF5_VERSION.tar.gz
tar xzvf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME CC=mpicc CXX=mpicxx F77=mpif90 \
                    --enable-shared --enable-parallel
make
make install
```

H5PY via pip

This can now just be pip installed (>=2.6.0):

    pip3 install h5py

For now we drop our former instructions on attempting to install parallel h5py with collectives. See the repo history for those notes.

Installing Mercurial

On NASA Pleiades, we need to install mercurial itself. I can't get mercurial to build properly on intel compilers, so for now use gcc:

```
cd $BUILD_HOME
wget http://mercurial.selenic.com/release/mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
tar xvf mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
cd mercurial-$LOCAL_MERCURIAL_VERSION
module load gcc
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your `~/.hgrc`:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[web]
cacerts = /etc/ssl/certs/ca-bundle.crt

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus

Preliminaries

Then do the following:

```
cd $BUILD_HOME
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Pleiades

Our scratch disk system on Pleiades is `/nobackup/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /nobackup/bpbrown workdir
```

Long-term mass storage is on LOU.

Install notes for NASA/Pleiades: gcc stack

Note: Warning. These instructions for a gcc stack are quite outdated and have not been tested in well over a year. A lot has shifted in the stack since then (e.g., h5py, matplotlib) and using these is at your own risk. We have been using the intel compilers exclusively on Pleiades, so please see those instructions. These gcc instructions are kept for posterity and future use.

Old instructions

An initial Pleiades environment is pretty bare-bones. There are no modules, and your shell is likely a csh varient. To switch shells, send an e-mail to support@nas.nasa.gov; I'll be using `bash`.

Then add the following to your `.profile`:

```
# Add your commands here to extend your PATH, etc.

module load gcc
module load git
module load openssl

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH

export CC=mpicc

#pathing for Dedalus2
export MPI_ROOT=$BUILD_HOME/openmpi-1.8
export PYTHONPATH=$BUILD_HOME/dedalus2:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
```

Note: We are moving here to a python 3.4 build. Also, it looks like scipy-0.14 and numpy 1.9 are going to have happier sparse matrix performance.

Doing the entire build took about 1 hour. This was with several (4) open ssh connections to Pleaides to do poor-mans-parallel building (of openBLAS, hdf5, fftw, etc.), and one was on a dev node for the openmpi and openblas compile.

Python stack

Interesting update. Pleiades now appears to have a python3 module. Fascinating. It comes with matplotlib (1.3.1), scipy (0.12), numpy (1.8.0) and cython (0.20.1) and a few others. Very interesting. For now we'll proceed with our usual build-it-from-scratch approach, but this should be kept in mind for the future. No clear mpi4py, and the `mpi4py` install was a hangup below for some time.

Building Openmpi

The suggested `mpi-sgi/mpt` MPI stack breaks with mpi4py; existing versions of openmpi on Pleiades are outdated and suffer from a previously identified bug (v1.6.5), so we'll roll our own. This needs to be built on a compute node so that the right memory space is identified.:

```
# do this on a main node (where you can access the outside internet):
cd $BUILD_HOME
wget http://www.open-mpi.org/software/ompi/v1.8/downloads/openmpi-1.8.tar.gz
tar xvf openmpi-1.7.3.tar.gz
```

(continues on next page)

(continued from previous page)

```
# get ivy-bridge compute node
qsub -I -q devel -l select=1:ncpus=20:mpiprocs=20:model=ivy -l walltime=02:00:00

# once node exists
cd $BUILD_HOME
cd openmpi-1.7.3
./configure \
    --prefix=$BUILD_HOME \
    --enable-mpi-interface-warning \
    --without-slurm \
    --with-tm=/PBS \
    --without-loadleveler \
    --without-portals \
    --enable-mpirun-prefix-by-default \
    CC=gcc

make
make install
```

These compilation options are based on `/nasa/openmpi/1.6.5/NAS_config.sh`, and are thanks to advice from Daniel Kokron at NAS.

We're using openmpi 1.7.3 here because something substantial changes in 1.7.4 and from that point onwards instances of mpirun hang on Pleiades, when used on more than 1 node worth of cores. I've tested this extensively with a simple hello world program (http://www.dartmouth.edu/~rc/classes/intro_mpi/hello_world_ex.html) and for now suggest we move forward until this is resolved.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/3.4.0/Python-3.4.0.tgz --no-check-certificate
tar xzf Python-3.4.0.tgz
cd Python-3.4.0

./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F90=mpif90 \
                    --enable-shared LDFLAGS="-lpthread" \
                    --with-cxx-main=mpicxx --with-system-ffi

make
make install
```

All of the intel patches, etc. are unnecessary in the gcc stack.

Note: We're getting a problem on `_curses_panel` and on `_sqlite3`; ignoring for now.

Installing pip

Python 3.4 now automatically includes pip.

On Pleiades, you'll need to edit `.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.crt
```

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

Note: Test that this works by doing a:

from mpi4py import MPI

This will segfault on sgi-mpi, but appears to work fine on openmpi-1.8, 1.7.3, etc.

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpicc \
                      CXX=mpicxx \
                      F77=mpif90 \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

The Feb 11, 2014 update to cython (0.20.1) seems to work with gcc.

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the OpenBLAS libraries.

All of the intel patches, etc. are unnecessary in the gcc stack.

Building OpenBLAS

From Stampede instructions:

```
# this needs to be done on a frontend
cd $BUILD_HOME
git clone git://github.com/xianyi/OpenBLAS

# suggest doing this build on a compute node, so we get the
# right number of openmp threads and architecture
cd $BUILD_HOME
cd OpenBLAS
make
make PREFIX=$BUILD_HOME install
```

Here's the build report before the `make install`:

```
OpenBLAS build complete. (BLAS CBLAS LAPACK LAPACKE)

OS               ... Linux
Architecture     ... x86_64
BINARY           ... 64bit
C compiler       ... GCC  (command line : mpicc)
Fortran compiler ... GFORTRAN  (command line : gfortran)
Library Name     ... libopenblas_sandybridgep-r0.2.9.rc2.a (Multi threaded; Max num-
↪threads is 40)
```

Building numpy against OpenBLAS

Now, acquire numpy (1.8.1):

```
wget http://sourceforge.net/projects/numpy/files/NumPy/1.8.1/numpy-1.8.1.tar.gz
tar xvf numpy-1.8.1.tar.gz
cd numpy-1.8.1
```

Create `site.cfg` with information for the OpenBLAS library directory

Next, make a site specific config file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit `site.cfg` to uncomment the `[openblas]` section; modify the library and include directories so that they correctly point to your `~/build/lib` and `~/build/include` (note, you may need to do fully expanded paths). With my account settings, this looks like:

```
[openblas]
libraries = openblas
library_dirs = /u/bpbrown/build/lib
include_dirs = /u/bpbrown/build/include
```

where `$BUILD_HOME=/u/bpbrown/build`. We may in time want to consider adding fftw as well. Now build:

```
python3 setup.py config build_clib build_ext install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After numpy has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/0.13.3/scipy-0.13.3.tar.gz
tar -xvf scipy-0.13.3.tar.gz
cd scipy-0.13.3
python3 setup.py config build_clib build_ext install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed:

```
pip3 install matplotlib
```

Installing sympy

This should just be pip installed:

```
pip3 install sympy
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.12.tar
tar xvf hdf5-1.8.12.tar
cd hdf5-1.8.12
./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F77=mpif90 \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel
make
make install
```

Next, install h5py. For reasons that are currently unclear to me, this cannot be done via pip install.

Installing h5py with collectives

We've been exploring the use of collectives for faster parallel file writing.

git is having some problems, especially with it's SSL version. I suggest adding the following to `~/.gitconfig`:

```
[http]
sslCAinfo = /etc/ssl/certs/ca-bundle.crt
```

This is still not working, owing (most likely) to git being built on an outdated SSL version. Here's a short-term hack:

```
export GIT_SSL_NO_VERIFY=true
```

To build that version of the h5py library:

```
git clone git://github.com/andrewcollette/h5py
cd h5py
git checkout mpi_collective
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

Here's the original h5py repository:

```
git clone git://github.com/h5py/h5py
cd h5py
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

Note: This is ugly. We're getting a "-R" error at link, triggered by distutils not recognizing that mpicc is gcc or something like that. Looks like we're failing `if self._is_gcc(compiler)` For now, I've hand-edited unixccompiler.py in `lib/python3.3/distutils` and changed this line:

> **def _is_gcc(self, compiler_name):** return "gcc" in compiler_name or "g++" in compiler_name

to:

> **def _is_gcc(self, compiler_name):** return "gcc" in compiler_name or "g++" in compiler_name or "mpicc" in compiler_name

This is a hack, but it get's us running and alive!

Note: Ahh… I understand what's happening here. We built with `mpicc`, and the test `_is_gcc` looks for whether gcc appears anywhere in the compiler name. It doesn't in `mpicc`, so the `gcc` checks get missed. This is only ever used in the `runtime_library_dir_option()` call. So we'd need to either rename the mpicc wrapper something like `mpicc-gcc` or do a test on `compiler --version` or something. Oh boy. Serious upstream problem for mpicc wrapped builds that cythonize and go to link. Hmm…

Installing Mercurial

On NASA Pleiades, we need to install mercurial itself:

```
wget http://mercurial.selenic.com/release/mercurial-2.9.tar.gz
tar xvf mercurial-2.9.tar.gz
cd mercurial-2.9
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your ~/.hgrc:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[web]
cacerts = /etc/ssl/certs/ca-bundle.crt

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus2

Preliminaries

With the modules set as above, set:

```
export BUILD_HOME=$BUILD_HOME
export FFTW_PATH=$BUILD_HOME
export MPI_PATH=$BUILD_HOME/openmpi-1.8
```

Then change into your root dedalus directory and run:

```
python setup.py build_ext --inplace
```

further packages needed for Keaton's branch:

```
pip3 install tqdm
pip3 install pathlib
```

Running Dedalus on Pleiades

Our scratch disk system on Pleiades is `/nobackup/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /nobackup/bpbrown workdir
```

Long-term mass storage is on LOU.

Install notes for BRC HPC SAVIO cluster

Installing on the SAVIO cluster is pretty straightforward, as many things can be loaded via modules. First, load the following modules.

```
module purge
module load intel
module load openmpi
module load fftw/3.3.4-intel
module load python/3.2.3
module load nose
module load numpy/1.8.1
module load scipy/0.14.0
module load mpi4py
module load pip
module load virtualenv/1.7.2
module load mercurial/2.0.2
module load hdf5/1.8.13-intel-p
```

We next need to make a virtual environment in which to build the rest of the Dedalus depencencies.

```
virtualenv python_build
source python_build/bin/activate
```

The rest of the depencies will be pip-installed. However, because we are using intel compilers, we need to specify the compiler and some how to link things properly.

```
export CC=icc
export LDFLAGS="-lirc -limf"
```

Now we can use pip to install most of the remaining dependencies.

```
pip-3.2 install cython
pip-3.2 install h5py
pip-3.2 install matplotlib==1.3.1
```

Dedalus itself can be pulled down from Bitbucket.

```
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip-3.2 install -r requirements.txt
```

To build Dedalus, you must specify the locations of FFTW and MPI.

```
export FFTW_PATH=/global/software/sl-6.x86_64/modules/intel/2013_sp1.4.211/fftw/3.3.4-
↪intel
export MPI_PATH=/global/software/sl-6.x86_64/modules/intel/2013_sp1.2.144/openmpi/1.6.5-
↪intel
python3 setup.py build_ext --inplace
```

Using Dedalus

To use Dedalus, put the following in your `~/.bashrc` file:

```
module purge
module load intel
module load openmpi
module load fftw/3.3.4-intel
module load python/3.2.3
module load numpy/1.8.1
module load scipy/0.14.0
module load mpi4py
module load mercurial/2.0.2
module load hdf5/1.8.13-intel-p
source python_build/bin/activate
export PYTHONPATH=$PYTHONPATH:~/dedalus
```

Install notes for TACC/Stampede

Install notes for building our python3 stack on TACC/Stampede, using the intel compiler suite. Many thanks to Yaakoub El Khamra at TACC for help in sorting out the python3 build and numpy linking against a fast MKL BLAS.

On Stampede, we can in principle either install with a `gcc/mpvapich2/fftw3` stack with `OpenBLAS`, or with an `intel/mvapich2/fftw3 stack` with `MKL`. Mpvaich2 is causing problems for us, and this appears to be a known issue with `mvapich2/1.9`, so for now we must use the `intel/mvapich2/fftw3` stack, which has `mvapich2/2.0b`. The intel stack should also, in principle, allow us to explore auto-offloading with the Xenon MIC hardware accelerators. Current `gcc` instructions can be found under NASA Pleiades.

Modules

Here is my current build environment (from running `module list`)

1) TACC-paths
2) Linux
3) cluster-paths
4) TACC
5) cluster
6) intel/14.0.1.106
7) mvapich2/2.0b

Note: To get here from a gcc default do the following:

module unload mkl module swap gcc intel/14.0.1.106

In the `intel` compiler stack, we need to use `mvapich2/2.0b`, which then implies `intel/14.0.1.106`. Right now, TACC has not built `fftw3` for this stack, so we'll be doing our own FFTW build.

See the Stampede user guide for more details. If you would like to always auto-load the same modules at startup, build your desired module configuration and then run:

```
module save
```

For ease in structuring the build, for now we'll define:

```
export BUILD_HOME=$HOME/build_intel
```

Python stack

Building Python3

Create ~\`build_intel` and then proceed with downloading and installing Python-3.3:

```
cd ~/build_intel
wget http://www.python.org/ftp/python/3.3.3/Python-3.3.3.tgz
tar -xzf Python-3.3.3.tgz
cd Python-3.3.3

# make sure you have the python patch, put it in Python-3.3.3
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/python_intel_patch.tar
tar xvf python_intel_patch.tar

./configure --prefix=$BUILD_HOME \
                    CC=icc CFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                    CXX=icpc CPPFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                    F90=ifort F90FLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                    --enable-shared LDFLAGS="-lpthread" \
                    --with-cxx-main=icpc --with-system-ffi

make
make install
```

To successfully build `python3`, the key is replacing the file `ffi64.c`, which is done automatically by downloading and unpacking this crude patch `python_intel_patch.tar` in your `Python-3.3.3` directory. Unpack it in `Python-3.3.3` (`tar xvf python_intel_patch.tar` line above) and it will overwrite `ffi64.c`. If you forget to do this, you'll see a warning/error that `_ctypes` couldn't be built. This is important.

Here we are building everything in `~/build_intel`; you can do it whereever, but adjust things appropriately in the above instructions. The build proceeeds quickly (few minutes).

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b:

```
wget http://www.fftw.org/fftw-3.3.3.tar.gz
tar -xzf fftw-3.3.3.tar.gz
cd fftw-3.3.3

./configure --prefix=$BUILD_HOME \
                      CC=mpicc \
                      CXX=mpicxx \
                      F77=mpif90 \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
make
make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Updating shell settings

At this point, `python3` is installed in `~/build_intel/bin/`. Add this to your path and confirm (currently there is no `python3` in the default path, so doing a `which python3` will fail if you haven't added `~/build_intel/bin`).

On Stampede, login shells (interactive connections via ssh) source only `~/.bash_profile`, `~/.bash_login` or `~/.profile`, in that order, and do not source `~/.bashrc`. Meanwhile non-login shells only launch `~/.bashrc` (see Stampede user guide).

In the bash shell, add the following to `.bashrc`:

```
export PATH=~/build_intel/bin:$PATH
export LD_LIBRARY_PATH=~/build_intel/lib:$LD_LIBRARY_PATH
```

and the following to `.profile`:

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

(from bash reference manual) to obtain the same behaviour in both shell types.

Installing pip

We'll use `pip` to install our python library depdencies. Instructions on doing this are available here and summarized below. First download and install setup tools:

```
cd ~/build
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py
python3 ez_setup.py
```

Then install `pip`:

```
wget --no-check-certificate https://raw.github.com/pypa/pip/master/contrib/get-pip.py
python3 get-pip.py --cert /etc/ssl/certs/ca-bundle.crt
```

Now edit ~/.pip/pip.conf:

```
[global]
cert = /etc/ssl/certs/ca-bundle.crt
```

You will now have pip3 and pip installed in ~/build/bin. You might try doing pip -V to confirm that pip is built against python 3.3. We will use pip3 throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Numpy and BLAS libraries

Building numpy against MKL

Now, acquire numpy (1.8.0):

```
cd ~/build_intel
wget http://sourceforge.net/projects/numpy/files/NumPy/1.8.0/numpy-1.8.0.tar.gz
tar -xvf numpy-1.8.0.tar.gz
cd numpy-1.8.0
wget http://lcd-www.colorado.edu/bpbrown/dedalus_documentation/_downloads/numpy_intel_
↪patch.tar
tar xvf numpy_inte_patch.tar
```

This last step saves you from needing to hand edit two files in numpy/distutils; these are intelccompiler.py and fcompiler/intel.py. I've built a crude patch, numpy_intel_patch.tar which can be auto-deployed by within the numpy-1.8.0 directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

We'll now need to make sure that numpy is building against the MKL libraries. Start by making a site.cfg file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit site.cfg in the [mkl] section; modify the library directory so that it correctly point to TACC's $MKLROOT/lib/intel64/. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /opt/apps/intel/13/composer_xe_2013_sp1.1.106/mkl/lib/intel64
include_dirs = /opt/apps/intel/13/composer_xe_2013_sp1.1.106/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://lcd-www.colorado.edu/bpbrown/dedalus_documentation/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

or do so manually by launching `python3` and then doing:

```
import numpy as np
np.__config__.show()
```

If you've installed `nose` (with `pip3 install nose`), we can further test our numpy build with:

```
np.test()
np.test('full')
```

We fail `np.test()` with two failures, while `np.test('full')` has 3 failures and 19 errors. But we do successfully link against the fast BLAS libraries (look for `FAST BLAS` output, and fast dot product time).

Note: We should check what impact these failed tests have on our results.

Python library stack

After `numpy` has been built (see links above) we will proceed with the rest of our python stack. Right now, all of these need to be installed in each existing virtualenv instance (e.g., `openblas`, `mkl`, etc.).

For now, skip the venv process.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Download an install in your appropriate `~/venv/INSTANCE` directory:

```
wget http://sourceforge.net/projects/scipy/files/scipy/0.13.2/scipy-0.13.2.tar.gz
tar -xvf scipy-0.13.2.tar.gz
cd scipy-0.13.2
```

Then run

```
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_ext␣
↪\
                                        --compiler=intelem --fcompiler=intelem install
```

Installing mpi4py

This should just be pip installed:

```
pip3 install mpi4py==2.0.0
```

Note: If we use use

```
pip3 install mpi4py
```

then stampede tries to pull version 0.6.0 of mpi4py. Hence the explicit version pull above.

Installing cython

This should just be pip installed:

```
pip3 install -v https://pypi.python.org/packages/source/C/Cython/Cython-0.20.tar.gz
```

The Feb 11, 2014 update to cython (0.20.1) seems to have broken (at least with intel compilers).:

```
pip3 install cython
```

Installing matplotlib

This should just be pip installed:

```
pip3 install -v https://downloads.sourceforge.net/project/matplotlib/matplotlib/
↪matplotlib-1.3.1/matplotlib-1.3.1.tar.gz
```

Note: If we use use

```
pip3 install matplotlib
```

then stampede tries to pull version 1.1.1 of matplotlib. Hence the explicit version pull above.

Installing sympy

Do this with a regular pip install:

```
pip3 install sympy
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.12.tar
tar xvf hdf5-1.8.12.tar
cd hdf5-1.8.12
./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F77=mpif90 \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel
make
make install
```

Installing h5py

Next, install h5py. We wish for full HDF5 parallel goodness, so we can do parallel file access during both simulations and post analysis as well. This will require building directly from source (see Parallel HDF5 in h5py for further details). Here we go:

```
git clone https://github.com/h5py/h5py.git
cd h5py
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

After this install, `h5py` shows up as an `.egg` in `site-packages`, but it looks like we pass the `suggested demo2.py` test from Parallel HDF5 in h5py.

Installing h5py with collectives

We've been exploring the use of collectives for faster parallel file writing. To build that version of the h5py library:

```
git clone https://github.com/andrewcollette/h5py.git
cd h5py
git checkout mpi_collective
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
```

(continues on next page)

(continued from previous page)

```
python3 setup.py build
python3 setup.py install
```

To enable collective outputs within dedalus, edit `dedalus2/data/evaluator.py` and replace:

```
# Assemble nonconstant subspace
subshape, subslices, subdata = self.get_subspace(out)
dset = task_group.create_dataset(name=name, shape=subshape, dtype=dtype)
dset[subslices] = subdata
```

with

```
# Assemble nonconstant subspace
subshape, subslices, subdata = self.get_subspace(out)
dset = task_group.create_dataset(name=name, shape=subshape, dtype=dtype)
with dset.collective:
    dset[subslices] = subdata
```

Alternatively, you can see this same edit in some of the forks (Lecoanet, Brown).

Note: There are some serious problems with this right now; in particular, there seems to be an issue with empty arrays causing h5py to hang. Troubleshooting is ongoing.

Dedalus2

With the modules set as above, set:

```
export BUILD_HOME=$HOME/build_intel
export FFTW_PATH=$BUILD_HOME
export MPI_PATH=$MPICH_HOME
export HDF5_DIR=$BUILD_HOME
export CC=mpicc
```

Then change into your root dedalus directory and run:

```
python setup.py build_ext --inplace
```

Our new stack (`intel/14`, `mvapich2/2.0b`) builds to completion and runs test problems successfully. We have good scaling in limited early tests.

Running Dedalus on Stampede

Source the appropriate virtualenv:

```
source ~/venv/openblas/bin/activate
```

or:

```
source ~/venv/mkl/bin/activate
```

grab an interactive dev node with `idev`. Play.

Skipped libraries

Installing freetype2

Freetype is necessary for matplotlib

```
cd ~/build
wget http://sourceforge.net/projects/freetype/files/freetype2/2.5.2/freetype-2.5.2.tar.gz
tar -xvf freetype-2.5.2.tar.gz
cd freetype-2.5.2
./configure --prefix=$HOME/build
make
make install
```

Note: Skipping for now

Installing libpng

May need this for matplotlib?:

```
cd ~/build
wget http://prdownloads.sourceforge.net/libpng/libpng-1.6.8.tar.gz
./configure --prefix=$HOME/build
make
make install
```

Note: Skipping for now

UMFPACK

We may wish to deploy UMFPACK for sparse matrix solves. Keaton is starting to look at this now. If we do, both numpy and scipy will require UMFPACK, so we should build it before proceeding with those builds.

UMFPACK requires AMD (another package by the same group, not processor) and SuiteSparse_config, too.

If we need UMFPACK, we can try installing it from `suite-sparse` as in the Mac install. Here are links to UMFPACK docs and Suite-sparse

Note: We'll check and update this later. (1/9/14)

All I want for christmas is suitesparse

Well, maybe :) Let's give it a try, and lets grab the whole library:

```
wget http://www.cise.ufl.edu/research/sparse/SuiteSparse/current/SuiteSparse.tar.gz
tar xvf SuiteSparse.tar.gz

<edit SuiteSparse_config/SuiteSparse_config.mk>
```

Note: Notes from the original successful build process:

Just got a direct call from Yaakoub. Very, very helpful. Here's the quick rundown.

He got _ctypes to work by editing the following file:

vim /work/00364/tg456434/yye00/src/Python-3.3.3/Modules/_ctypes/libffi/src/x86/ffi64.c

Do build with intel 14 use mvapich2/2.0b Will need to do our own build of fftw3

set mpicc as c compiler rather than icc, same for CXX, FC and others, when configuring python. should help with mpi4py.

in mpi4py, can edit mpi.cfg (non-pip install).

Keep Yaakoub updated with direct e-mail on progress.

Also, Yaakoub is spear-heading TACCs efforts in doing auto-offload to Xenon Phi.

Beware of disk quotas if you're trying many builds; I hit 5GB pretty fast and blew my matplotlib install due to quota limits :)

Installing virtualenv (skipped)

In order to test multiple numpys and scipys (and really, their underlying BLAS libraries), we will use `virtualenv`:

```
pip3 install virtualenv
```

Next, construct a virtualenv to hold all of your python modules. We suggest doing this in your home directory:

```
mkdir ~/venv
```

Python3

Note: With help from Yaakoub, we now build `_ctypes` successfully.

Also, the mpicc build is much, much slower than icc. Interesting. And we crashed out. Here's what we tried with mpicc:

```
./configure --prefix=$BUILD_HOME \
                CC=mpicc CFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                CXX=mpicxx CPPFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                F90=mpif90 F90FLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                --enable-shared LDFLAGS="-lpthread" \
                --with-cxx-main=mpicxx --with-system-ffi
```

Install notes for Trestles

Note: These are a very old set of installation instructions. They likely no longer work.

Make sure to do

$ module purge

first.

Modules

This is a minimalist list for now:

- gnu/4.8.2 (this is now the default gnu module)
- openmpi_ib
- fftw/3.3.3 (make sure to do this one last, as it's compiler/MPI dependent)

Building Python3

I usually build everything in *~/build*, but you can do it whereever. Download Python-3.3. Once loading the above modules, in the Python-3.3 source directory, do

$ ./configure –prefix=$HOME/build

followed by the usual *make -j4; make install* (the *-j4* tells make to use 4 cores). You may get something like this:

```
Python build finished, but the necessary bits to build these modules were not found:
_dbm               _gdbm              _lzma
_sqlite3
To find the necessary bits, look in setup.py in detect_modules() for the module's name.
```

I think this should be totally fine.

At this point, make sure the python3 you installed is in your path!

Installing virtualenv

In order to test multiple numpys and scipys (and really, their underlying BLAS libraries), I am using virtualenv. In order install virtulenv, once Python-3.3 is installed, you first need to install pip manuall. Follow the steps here http://www.pip-installer.org/en/latest/installing.html for "Install or Upgrade Setuptools" and then "Install or Upgrade pip". Briefly, you need to download and run ez_setup.py and then get-pip.py. This should run without incident. Once *pip* is installed, do

$ pip install virtualenv

Building OpenBLAS

To build openBLAS, first do *$ git clone https://github.com/xianyi/OpenBLAS.git* to get OpenBLAS. Then, with the modules loaded, do *make -j4*; and *make PREFIX=path/to/build/dir install*

Building numpy

First construct a virtualenv to hold all of your python modules. I like to do this right in my home directory. For example,

$ mkdir ~/venv (assuming you don't already have *~/venv*) *$ cd ~/venv $ virtualenv openblas*

will create an *openblas* directory, with a *bin* subdirectory. You "activate" the virtual env by doing *$ source path/to/virtualenv/bin/activate*. This will change all of your environment variables so that the active python will see whatever modules are in that directory. **Note that this messes with modules!** To be safe, I'd recommend *$ module purge; module load gnu openmpi_ib* afterwards.

- *$ cp site.cfg.example site.cfg*

edit *site.cfg* to uncomment the [openblas] section and fill in the include and library dirs to whereever you installed openblas.

- *$ python setup.py config*

to make sure that the numpy build has FOUND your openblas install. If it did, you should see something like this:

```
(openblas)trestles-login1:/home/../numpy-1.8.0 [10:15]$ python setup.py config
Running from numpy source directory.
F2PY Version 2
blas_opt_info:
blas_mkl_info:
  libraries mkl,vml,guide not found in ['/home/joishi/venv/openblas/lib', '/usr/local/
↪lib64', '/usr/local/lib', '/usr/lib64', '/usr/lib', '/usr/lib/']
  NOT AVAILABLE

openblas_info:
  FOUND:
    language = f77
    library_dirs = ['/home/joishi/build/lib']
    libraries = ['openblas', 'openblas']

  FOUND:
    language = f77
    library_dirs = ['/home/joishi/build/lib']
    libraries = ['openblas', 'openblas']

non-existing path in 'numpy/lib': 'benchmarks'
lapack_opt_info:
  FOUND:
    language = f77
    library_dirs = ['/home/joishi/build/lib']
    libraries = ['openblas', 'openblas']

/home/joishi/build/lib/python3.3/distutils/dist.py:257: UserWarning: Unknown␣
↪distribution option: 'define_macros'
  warnings.warn(msg)
```

(continues on next page)

(continued from previous page)

```
running config
```

- *$ python setup.py build*

if successful,

- *$ python setup.py install*

Installing Scipy

Scipy is easier, because it just gets its config from numpy.

- *$ python setup.py config*

This notes a lack of UMFPACK…will that make a speed difference? Nevertheless, it works ok.

Do

- *$ python setup.py build*

if successful,

- *$ python setup.py install*

Installing mpi4py

This should just be pip installed, *$ pip install mpi4py*

Installing cython

This should just be pip installed, *$ pip install cython*

Installing matplotlib

This should just be pip installed, *$ pip install matplotlib*

UMFPACK

Requires AMD (another package by the same group, not processor) and SuiteSparse_config, too.

Dedalus2

With the modules set as above (for NOW), set *$ export FFTW_PATH=/opt/fftw/3.3.3/gnu/openmpi/ib* and *$ export MPI_PATH=/opt/openmpi/gnu/ib*. Then do *$ python setup.py build_ext –inplace*.

1.2 Tutorials & Examples

1.2.1 Tutorial Notebooks

This tutorial on using Dedalus consists of four short IPython notebooks, which can be downloaded and ran interactively, or viewed on-line through the links below.

The notebooks cover the basics of setting up and interacting with the primary facets of the code, culminating in the setup and simulation of the 1D KdV-Burgers equation.

Tutorial 1: Bases and Domains

Overview: This tutorial covers the basics of setting up and interacting with basis and domain objects in Dedalus. Dedalus uses spectral discretizations to represent fields and solve PDEs. These discretizations are specified by selecting a spectral basis for each spatial dimension, and combining them to form a domain. When running in parallel, domains are automatically distributed using a block decomposition.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from dedalus import public as de
```

[2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

1.1: Bases

Creating a basis

Each type of basis in Dedalus is represented by a separate class. These classes define the corresponding spectral operators and transforms between the "grid space" and "coefficient space" representations of functions in that basis. The most commonly used bases are:

- `Fourier` for periodic functions on an interval.
- `SinCos` for functions with definite parity (even or odd symmetry) around the endpoints of an interval.
- `Chebyshev` for general functions on an interval.

When instantiating one of these bases, you provide a name for the spatial dimension, the number of modes, and the endpoints of the basis interval. Optionally, you can specify a dealiasing scale factor, indicating how much to pad the included modes when transforming to grid space. To properly dealias quadratic nonlinearities, for instance, you would need a scaling factor of 3/2.

[3]:
```
xbasis = de.Chebyshev('x', 32, interval=(0,1), dealias=3/2)
```

Other less commonly used bases include:

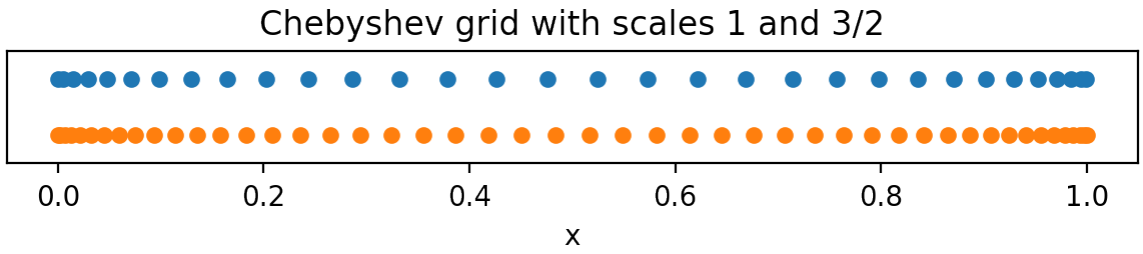
- `Legendre` for general functions on an interval (but without fast transforms).
- `Hermite` for decaying functions on the real line.
- `Laguerre` for decaying functions on the half-line.

See the basis.py API documentation for more information about using these other bases.

Basis grids and scale factors

Each basis has a corresponding grid that can be used for tasks like initializing and plotting fields. The global (non-distributed) grid of a basis can be accessed using the basis object's `grid` method, where you'll have to pass a scale factor determining the number of points in the grid, relative to the number of basis modes. Let's look at the Chebyshev grids with scaling factors of 1 and 3/2.

[4]:
```
grid_normal = xbasis.grid(scale=1)
grid_dealias = xbasis.grid(scale=3/2)

plt.figure(figsize=(6, 1.5), dpi=100)
plt.plot(grid_normal, 0*grid_normal+1, 'o', markersize=5)
plt.plot(grid_dealias, 0*grid_dealias-1, 'o', markersize=5)
plt.xlabel('x')
plt.title('Chebyshev grid with scales 1 and 3/2')
plt.ylim([-2, 2])
plt.gca().yaxis.set_ticks([]);
plt.tight_layout()
```



Note that the Chebyshev grids are non-equispaced: the grid points cluster quadratically towards the ends of the interval, which can be very useful for resolving sharp features like boundary layers.

Compound bases

It may be useful to add resolution at a particular intermediate position in an interval. In this case, a compound basis consisting of stacked Chebyshev segments (or other polynomial basis segments) can be constructed simply by grouping a set of individual bases defined over adjacent intervals into a `Compound` basis object.

[5]:
```
xb1 = de.Chebyshev('x1', 16, interval=(0,2))
xb2 = de.Chebyshev('x2', 32, interval=(2,8))
xb3 = de.Chebyshev('x3', 16, interval=(8,10))
xbasis = de.Compound('x', (xb1, xb2, xb3))
```

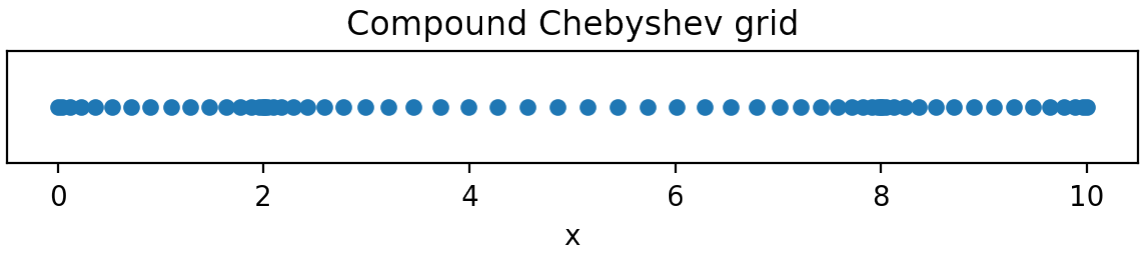
Since we use the interior roots grid for the Chebyshev polynomials, the grid of the compound bases is simply the union of the subbasis grids. When solving a problem with Dedalus, the continuity of fields across the subbasis interfaces is automatically enforced.

[6]:
```
compound_grid = xbasis.grid(scale=1)

plt.figure(figsize=(6, 1.5), dpi=100)
plt.plot(compound_grid, 0*compound_grid, 'o', markersize=5)
plt.xlabel('x')
plt.title('Compound Chebyshev grid')
```

(continues on next page)

(continued from previous page)

```
plt.gca().yaxis.set_ticks([]);
plt.tight_layout()
```



1.2: Domains

Creating a domain

`Domain` objects represent physical domains which are discretized by the direct product of a set of bases. To build a domain, we pass a list of the composite bases, specify the (grid space) datatype, and optionally specify a process mesh for parallelization. Let's construct a domain representing a 3D slab that is periodic in the x and y directions, and non-periodic in z.

[7]:
```
xbasis = de.Fourier('x', 32, interval=(0,2), dealias=3/2)
ybasis = de.Fourier('y', 32, interval=(0,2), dealias=3/2)
zbasis = de.Chebyshev('z', 32, interval=(0,1), dealias=3/2)
domain = de.Domain([xbasis, ybasis, zbasis], grid_dtype=np.complex128)
```

Dedalus currently supports explicit computations on arbitrary N-dimensional domains. However, to solve PDEs with Dedalus, the first (N-1) dimensions of the domain must be separable, meaning that the linear parts of the PDE do not couple different modes in these dimensions. In most cases, this means the first (N-1) dimensions of the domain need to be either `Fourier` or `SinCos` bases.

Parallelization & process meshes

When running in an MPI environment, Dedalus uses block-distributed domain decompositions to parallelize computation. By default, problems are distributed across a 1-dimensional mesh of all the available MPI processes (i.e. a "slab" decomposition). However, arbitrary (N-1)-dimensional process meshes (i.e. "pencil" decompositions) can be used by specifying a mesh shape with the `mesh` keyword when instantiating a domain. The current MPI environment must have the same number of processes as the product of the mesh shape.

Layouts

The domain object builds the machinery necessary for the parallelized allocation and transformation of fields. This includes an ordered set of `Layout` objects describing the necessary transform/distribution states of the data between coefficient space and grid space. Subsequent layouts are connected by basis transforms, which must be performed locally, and global transposes (rearrangements of the data distribution across the process mesh) to achieve the required locality.

The general algorithm starts from coefficient space (layout 0), with the last axis local (non-distributed). It proceeds towards grid space by transforming the last axis into grid space (layout 1). Then it globally transposes the data (if

necessary) such that the penultimate axis is local, transforms that axis into grid space, etc., until all axes have been transformed to grid space (the last layout).

Let's examine the layouts for the domain we just constructed. For serial computations such as this, no global transposes are necessary (all axes are always local), and the paths between layouts consist of coefficient-to-grid transforms, backwards from the last axis:

[8]:
```
for layout in domain.distributor.layouts:
    print('Layout {}:  Grid space: {}  Local: {}'.format(layout.index, layout.grid_space,
↪ layout.local))
```

```
Layout 0:  Grid space: [False False False]  Local: [ True  True  True]
Layout 1:  Grid space: [False False  True]  Local: [ True  True  True]
Layout 2:  Grid space: [False  True  True]  Local: [ True  True  True]
Layout 3:  Grid space: [ True  True  True]  Local: [ True  True  True]
```

To see how things work for a distributed simulation, we'll change the specified process mesh shape and rebuild the layout objects, circumventing the internal checks on the number of available processes, etc.

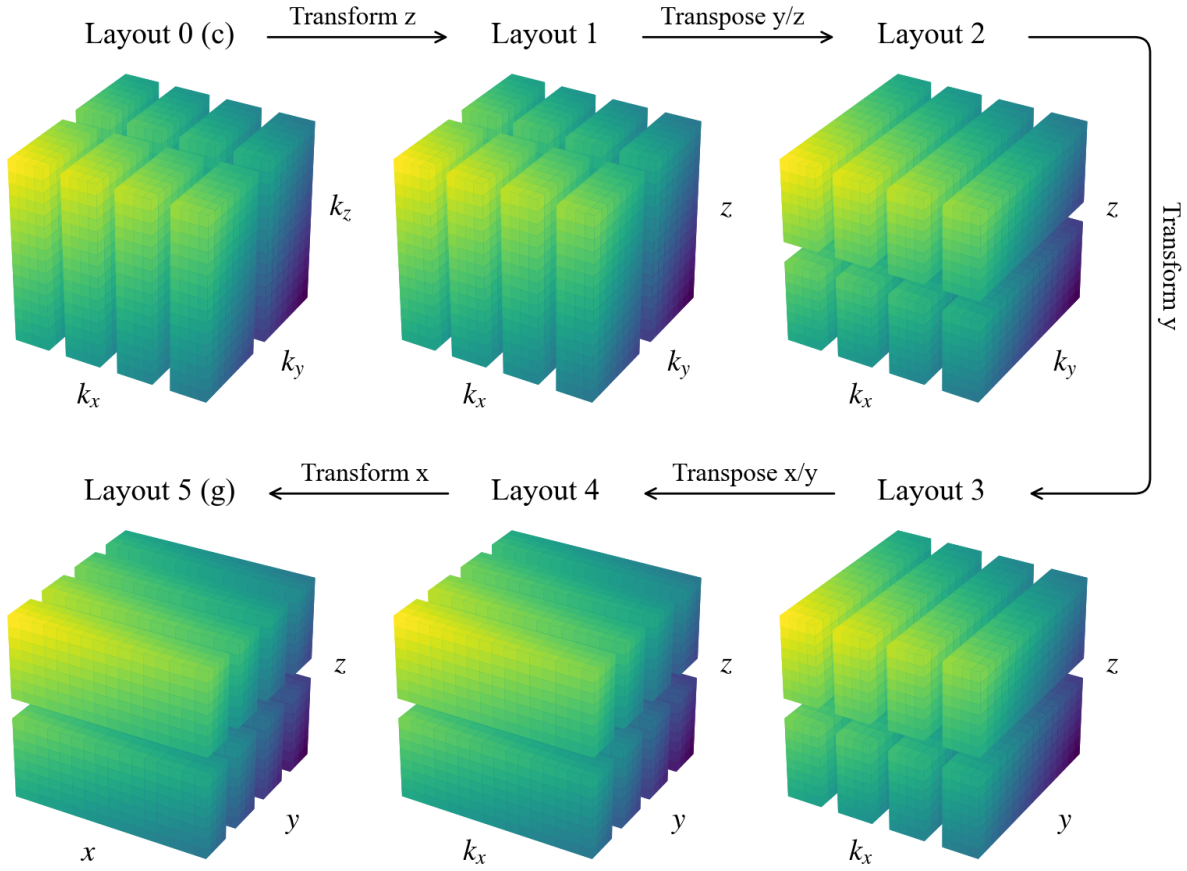
Note this is for demonstration only… messing with these attributes will generally break things.

[9]:
```
# Don't do this. For illustration only.
domain.distributor.mesh = np.array([4, 2])
domain.distributor.coords = np.array([0, 0])
domain.distributor._build_layouts(domain, dry_run=True)
```

[10]:
```
for layout in domain.distributor.layouts:
    print('Layout {}:  Grid space: {}  Local: {}'.format(layout.index, layout.grid_space,
↪ layout.local))
```

```
Layout 0:  Grid space: [False False False]  Local: [False False  True]
Layout 1:  Grid space: [False False  True]  Local: [False False  True]
Layout 2:  Grid space: [False False  True]  Local: [False  True False]
Layout 3:  Grid space: [False  True  True]  Local: [False  True False]
Layout 4:  Grid space: [False  True  True]  Local: [ True False False]
Layout 5:  Grid space: [ True  True  True]  Local: [ True False False]
```

We can see that there are now two additional layouts, corresponding to the transposed states of the mixed-transform layouts. Two global transposes are necessary here in order for the y and x axes to be stored locally, which is required to perform the respective basis transforms. Here's a sketch of the data distribution in the different layouts:



Interacting with the layout objects directly is typically not necessary, but being aware of this system for controlling the distribution and tranformation state of data is important for interacting with field objects, as we'll see in future notebooks.

Distributed grid and element arrays

To help with creating field data, Dedalus domain objects provide properly oriented arrays representing the local portions of the basis grids and elements (wavenumbers or polynomial indices). The local grids (distributed according to the last layout, grid space) are accessed using the `domain.grid` method and specifying an axis and scale factors:

[11]:
```
local_x = domain.grid(0, scales=1)
local_y = domain.grid(1, scales=1)
local_z = domain.grid(2, scales=1)
print('Local x shape:', local_x.shape)
print('Local y shape:', local_y.shape)
print('Local z shape:', local_z.shape)
```

```
Local x shape: (32, 1, 1)
Local y shape: (1, 8, 1)
Local z shape: (1, 1, 16)
```

The local x grid is the full Fourier grid for the x-basis, and will be the same on all processes, since the first axis is local in grid space. The local y and local z grids will generally differ across processes, since they contain just the local portions of the y and z basis grids, distributed across the specified process mesh (4 blocks in y and 2 blocks in z).

The local elements (distributed according to layout 0) are accessed using the `domain.elements` method and specifying an axis:

[12]:
```
local_kx = domain.elements(0)
local_ky = domain.elements(1)
local_nz = domain.elements(2)
print('Local kx shape:', local_kx.shape)
print('Local ky shape:', local_ky.shape)
print('Local nz shape:', local_nz.shape)
```

```
Local kx shape: (8, 1, 1)
Local ky shape: (1, 16, 1)
Local nz shape: (1, 1, 32)
```

The local kx and local ky elements will now differ across processes, since they contain just the local portions of the x and y wavenumbers, which are distributed in coefficient space. The local nz elements are the full set of Chebyshev modes, which are always local in coefficient space.

These local arrays can be used to form parallel-safe initial conditions for fields, in grid or coefficient space, as we'll see in the next notebook.

Tutorial 2: Fields and Operators

Overview: This tutorials covers the basics of setting up and interacting with field and operator objects in Dedalus. Dedalus uses field and operator abstractions to implement a symbolic algebra system for representing mathematical expressions and PDEs.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from dedalus import public as de
from dedalus.extras.plot_tools import plot_bot_2d
figkw = {'figsize':(6,4), 'dpi':100}
```

[2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

2.1: Fields

Creating a field

Field objects in Dedalus represent scalar-valued fields defined over a domain. A field can be directly instantiated from the `Field` class by passing a domain object, or using the `domain.new_field` method. Let's set up a 2D domain and build a field:

[3]:
```
xbasis = de.Fourier('x', 64, interval=(-np.pi, np.pi), dealias=3/2)
ybasis = de.Chebyshev('y', 64, interval=(-1, 1), dealias=3/2)
domain = de.Domain([xbasis, ybasis], grid_dtype=np.float64)
f = de.Field(domain, name='f')
```

We also gave the field a name – something which is automatically done for the state fields when solving a PDE in Dedalus (we'll see more about this in the next notebook), but we've just done it manually, for now.

Manipulating field data

Field objects have a variety of methods for transforming their data between different layouts (i.e. grid and coefficient space, and all the layouts in-between). The `layout` attribute of each field points to the layout object describing its current transform and distribution state. We can see that fields are instantiated in coefficient space:

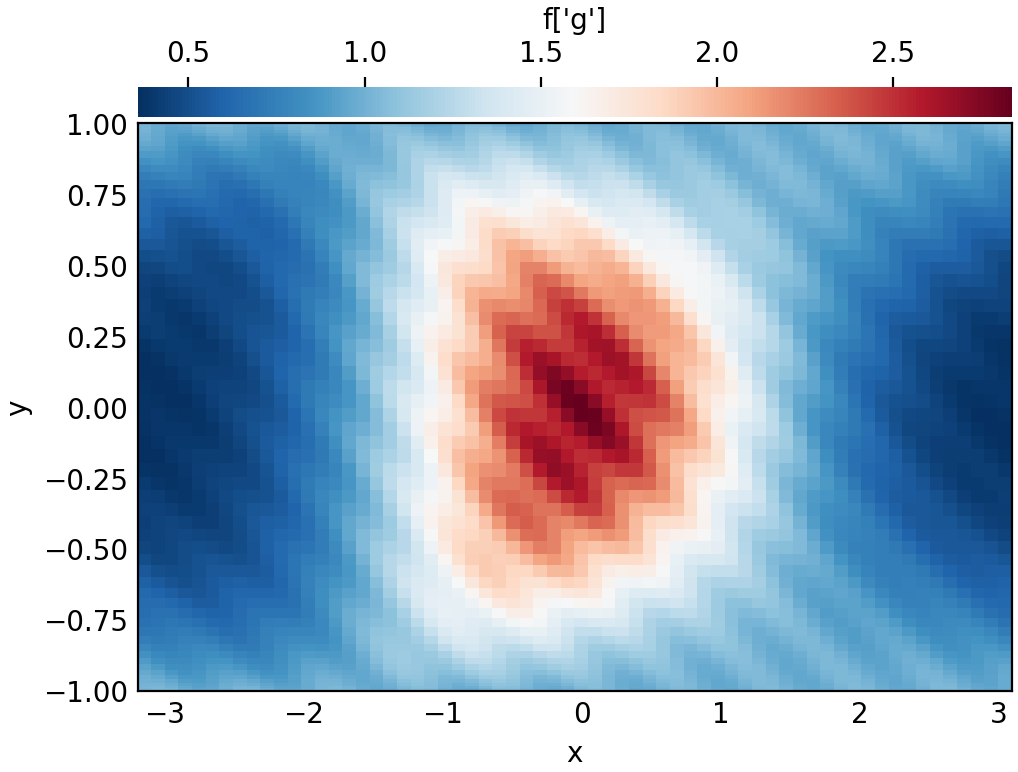
[4]:
```
f.layout.grid_space
```

[4]:
```
array([False, False])
```

Field data can be assigned and retrieved in any layout by indexing a field with that layout object. In most cases, mixed layouts aren't needed, and it's just the full grid and full coefficient data that are most useful to interact with. These layouts can be easily accessed using `'g'` and `'c'` keys as shortcuts.

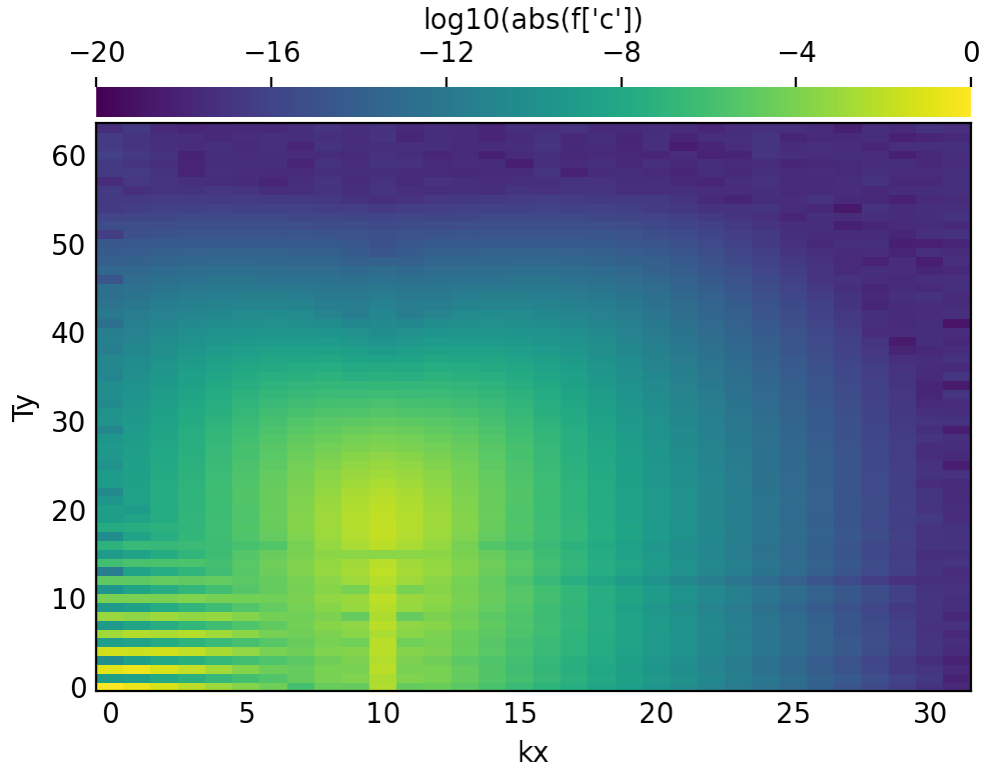
When accessing field data in parallel, each process manipulates just the local data of the globally distributed dataset. We can therefore easily set a field's grid data in a parallel-safe fashion by using the local grids provided by the domain object:

[5]:
```
x, y = domain.grids(scales=1)
f.set_scales(1)
f['g'] = np.exp((1-y**2)*np.cos(x+np.cos(x)*y**2)) * (1 + 0.05*np.cos(10*(x+2*y)))

# Plot grid values
plot_bot_2d(f, figkw=figkw, title="f['g']");
```



We can convert a field to spectral coefficients by requesting the field's data in coefficient space. This internally triggers an in-place multidimensional spectral transform on the field's data.

[6]:
```
f['c']

# Plot log magnitude of spectral coefficients
log_mag = lambda xmesh, ymesh, data: (xmesh, ymesh, np.log10(np.abs(data)))
plot_bot_2d(f, func=log_mag, clim=(-20, 0), cmap='viridis', title="log10(abs(f['c'])",␣
↪figkw=figkw);
```



Examining the spectral coefficients of fields is very useful, since the amplitude of the highest modes indicate the truncation errors in the spectral discretizations of fields. If these modes are small, like here, then we know the field is well-resolved.

Field scale factors

The `set_scales` method on a field is used to change the scaling factors used when transforming the field's data into grid space. When setting a field's data using grid arrays, shape errors will result if there is a mismatch between the field and grid's scale factors.

Large scale factors can be used to interpolate the field data onto a high-resolution grid, while small scale factors can be used to view a lower-resolution grid representation of a field. **Beware:** using scale factors less than 1 will result in a loss of data when transforming to grid space.

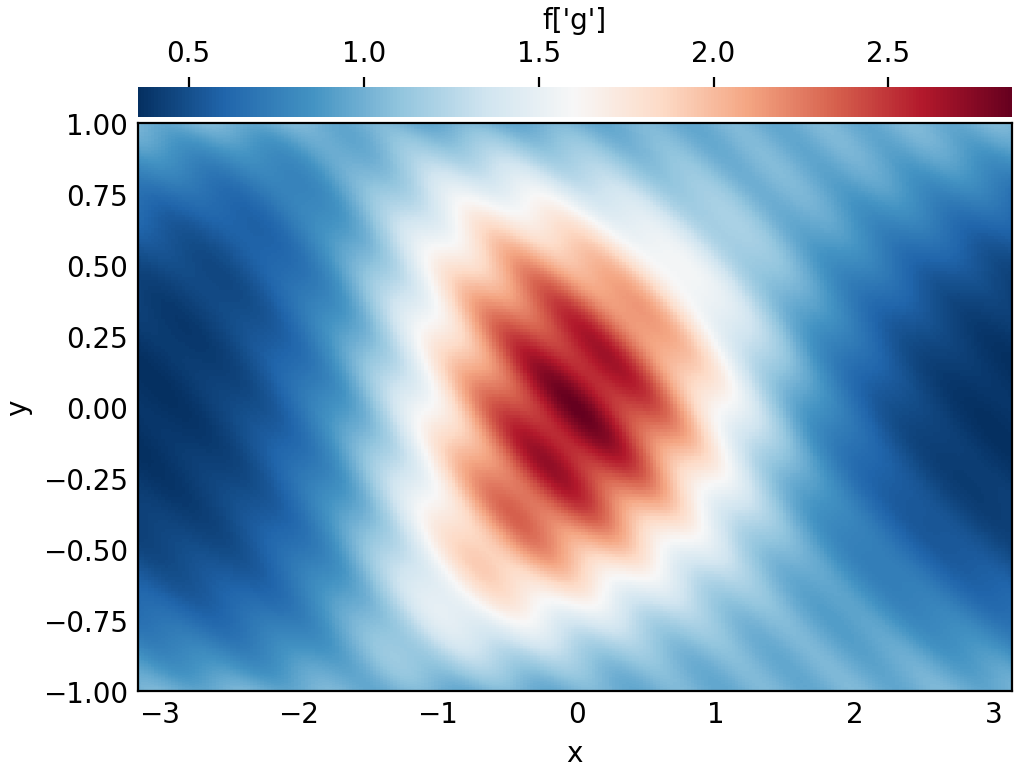
Let's take a look at a high-resolution sampling of our field, by increasing the scales.

[7]:
```
f.set_scales(4)

# Plot grid values
f['g']
```

(continues on next page)

(continued from previous page)

```
plot_bot_2d(f, title="f['g']", figkw=figkw);
```



2.2: Operators

Arithmetic with fields

Mathematical operations on fields, including arithmetic, differentiation, integration, and interpolation, are represented by `Operator` classes. An instance of an operator class represents a specific mathematical expression, and provides an interface for the deferred evaluation of that expression with respect to it's potentially evolving arguments.

Arithmetic operations between fields, or fields and scalars, are produced simply using Python's infix operators for arithmetic. Let's start with a simple affine transformation of our field:

[8]:
```
g_op = 1 - 2*f
print(g_op)
```

```
1 + (-1*(2*f))
```

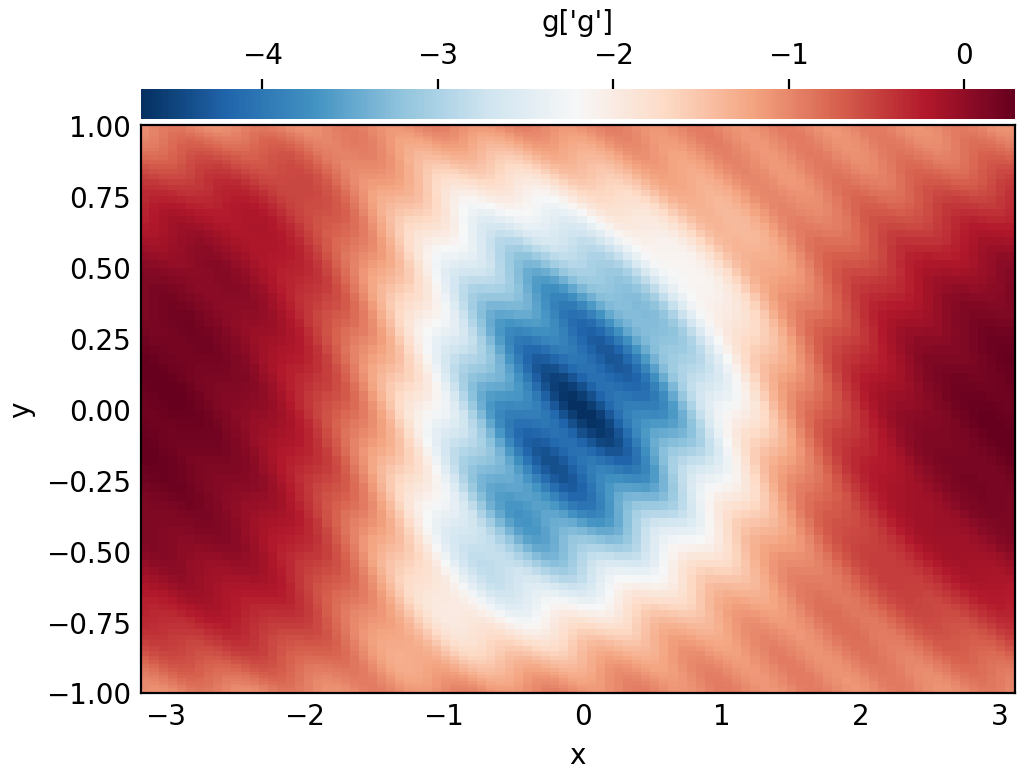
The object we get is not another field, but an operator object representing the addition of 1 to the multiplication of -1, 2, and our field. To actually compute this operation, we use the `evaluate` method, which returns a new field with the result. The dealias scale factors set during basis instantiation are used for the evaluation of all operators.

[9]:
```
g = g_op.evaluate()

# Plot grid values
```

(continues on next page)

(continued from previous page)

```
g['g']
plot_bot_2d(g, title="g['g']", figkw=figkw);
```



Building expressions

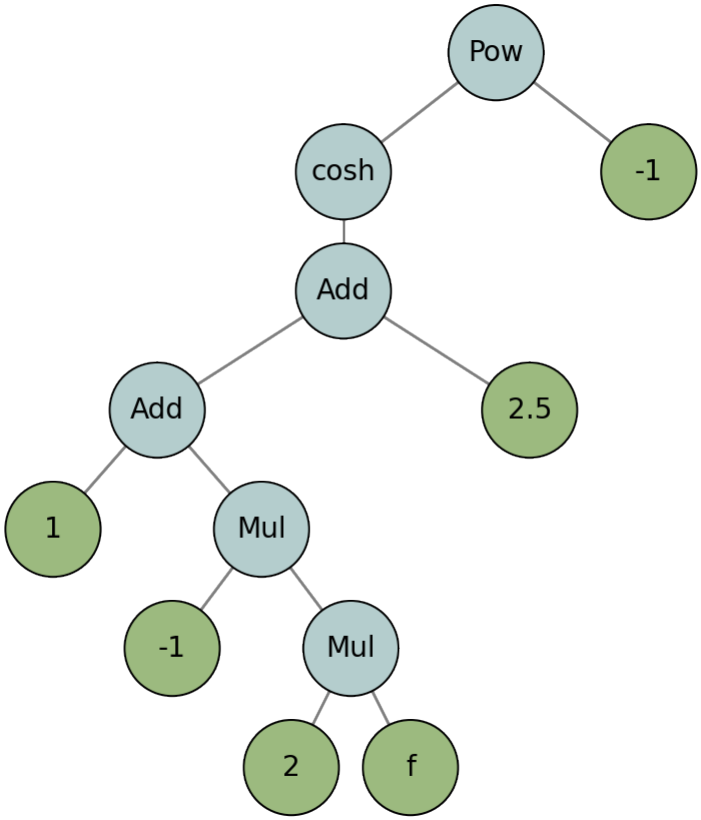
Operator instances can be passed as arguments to other operators, building trees that represent more complicated expressions:

[10]:
```
h_op = 1 / np.cosh(g_op + 2.5)
print(h_op)
```

```
cosh((1 + (-1*(2*f))) + 2.5)**-1
```

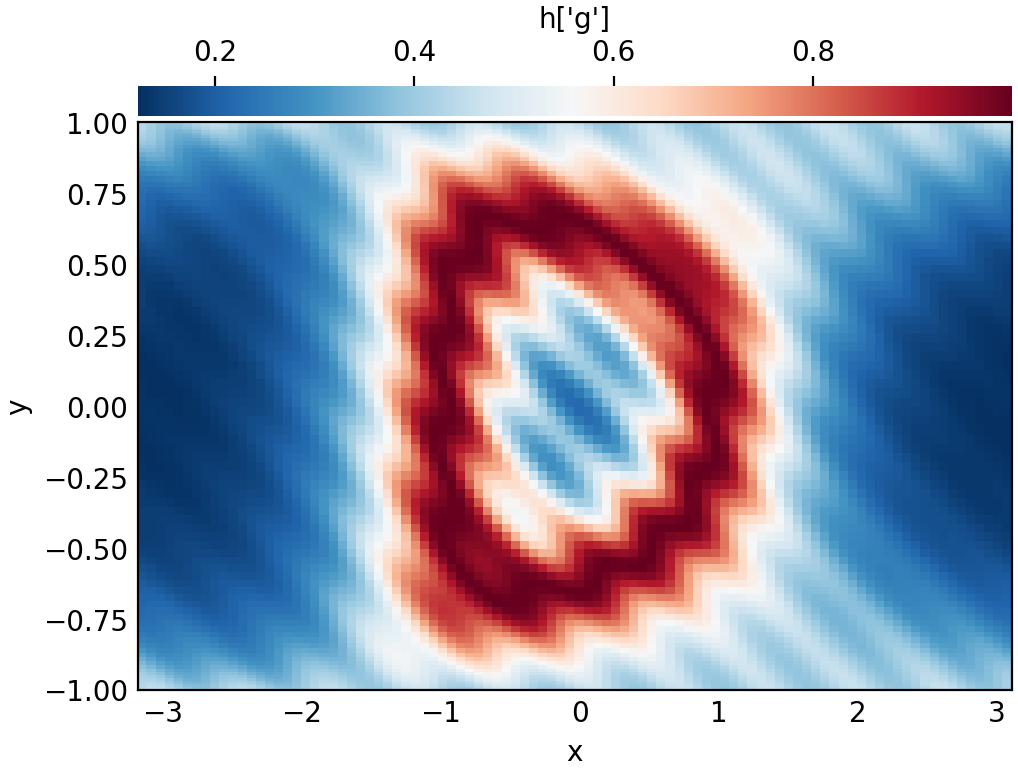
Reading these signatures can be a little cumbersome, but we can plot the operator's structure using a helper from `dedalus.tools`:

[11]:
```
from dedalus.tools.plot_op import plot_operator
plot_operator(h_op, figsize=6, fontsize=14, opsize=0.4)
```



And evaluating it:

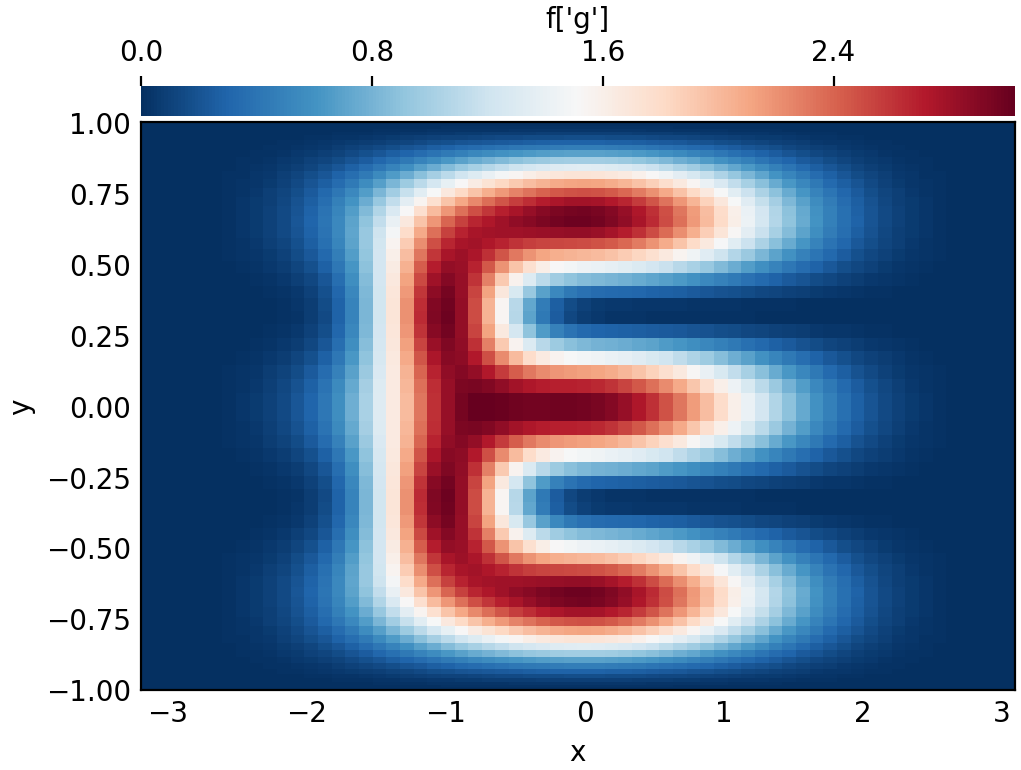
[12]:
```
h = h_op.evaluate()

# Plot grid values
h['g']
plot_bot_2d(h, title="h['g']", figkw=figkw);
```



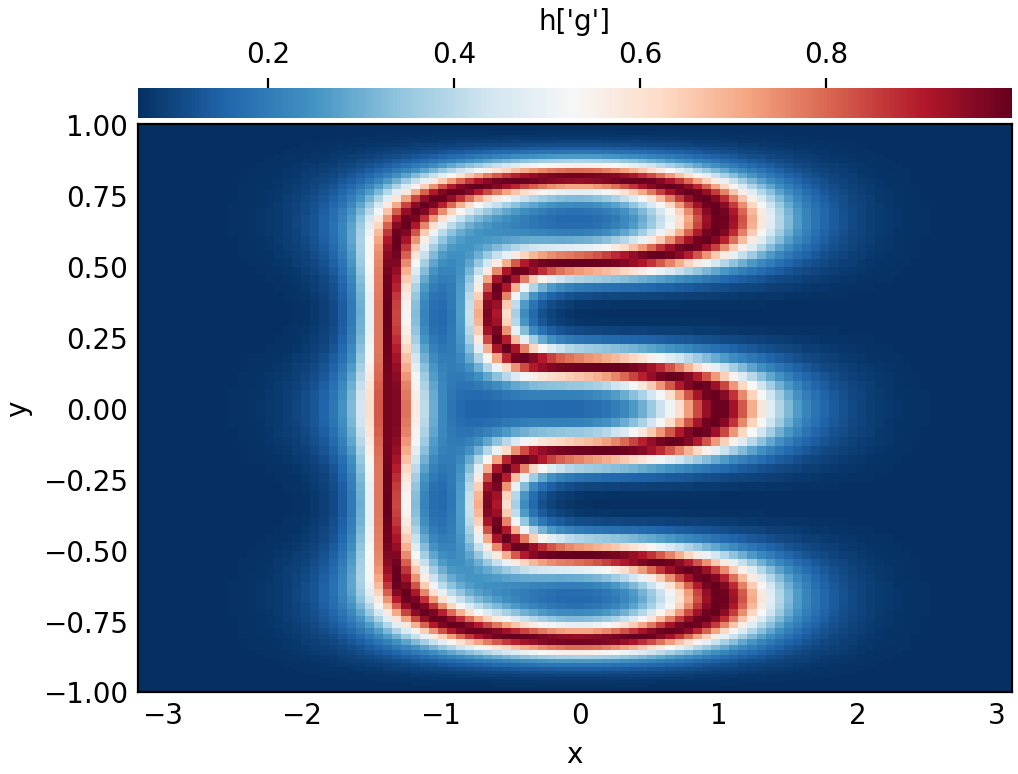
Deferred evaluation

A key point is that the operator objects symbolically represent an operation on the field arguments, and are evaluated using deferred evaluation. If we change the data of the field arguments and re-evaluate an operator, we get a new result.

[13]:
```
# Set scales back to 1 to build new grid data
f.set_scales(1)
f['g'] = 3*np.cos(1.5*np.pi*y)**2 * np.cos(x/2)**4 + 3*np.exp(-((2*x+2)**2 + (4*y+4/
↪3)**2)) + 3*np.exp(-((2*x+2)**2 + (4*y-4/3)**2))

# Plot grid values
f['g']
plot_bot_2d(f, title="f['g']", figkw=figkw);
```



[14]:
```
h = h_op.evaluate()

# Plot grid values
h['g']
plot_bot_2d(h, title="h['g']", figkw=figkw);
```

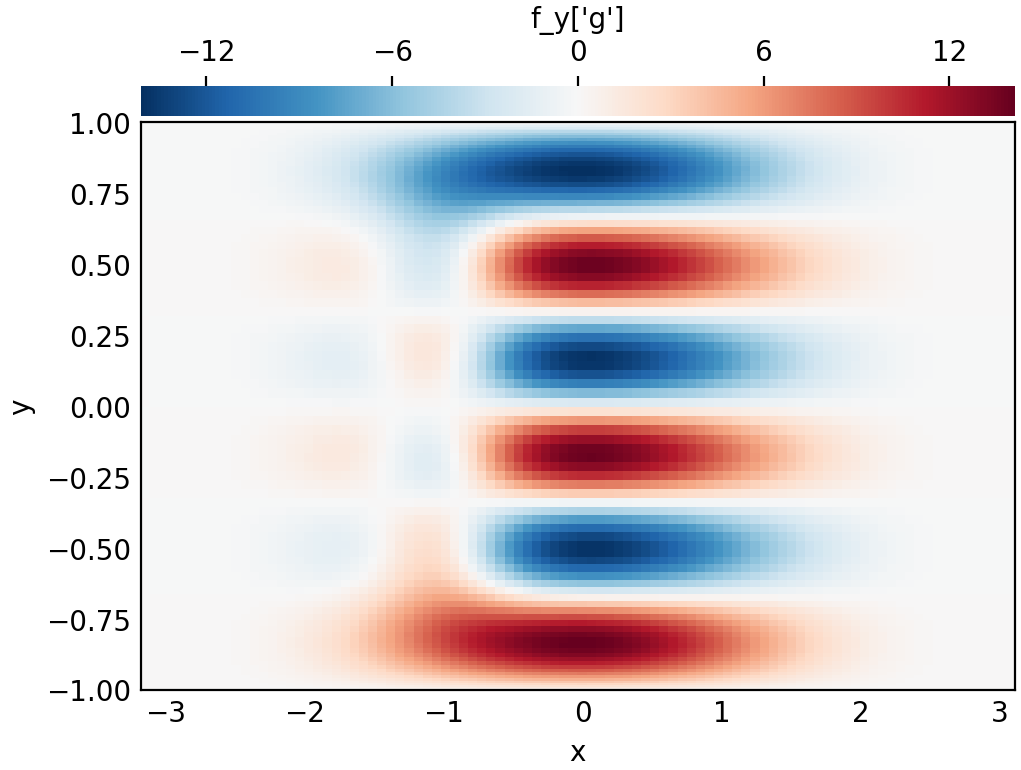


Differentiation, integration, interpolation

Operators are also used for differentiation, integration, and interpolation along each basis. There are several ways to apply these operators to a field or operator expression.

Basis operators: First, the operators for a given dimension can be accessed through the `Differentiate`, `Integrate`, and `Interpolate` methods of the corresponding basis object.

[15]:
```
# Basis operators
f_y_op = ybasis.Differentiate(f)
f_yint_op = ybasis.Integrate(f)
f_y0_op = ybasis.Interpolate(f, position=0)

# Plot f_y
f_y = f_y_op.evaluate()
f_y['g']
plot_bot_2d(f_y, title="f_y['g']", figkw=figkw);
```



Operator factories: Second, more general "factory" interfaces are available through the `operators` module. These factories provide a simple interface for composing the underlying basis operators in multiple dimensions. The `operators.differentiate` factory allows us to easily construct higher-order and mixed derivatives involving different bases. The `operators.integrate` and `operators.interpolate` factories allow us to integrate/interpolate along multiple axes, as well.

[16]:
```
# Operator factories
f_xxyy_op = de.operators.differentiate(f, x=2, y=2)
f_int_op = de.operators.integrate(f, 'x', 'y')
f_00_op = de.operators.interpolate(f, x=0, y=0)

# See structure of f_int_op
print('f_int_op:', f_int_op)
print()

# Print total integral
# The result is a constant field, so we can print its value at any grid point
f_int = f_int_op.evaluate()
print('Total integral of f:', f_int['g'][0,0])
```

```
f_int_op: integ_y(integ_x(f))

Total integral of f: 9.424586589055345
```

Field methods: Finally, the `differentiate`, `integrate`, and `interpolate` field methods provide short-cuts for building and evaluating these operations, directly returning the resulting fields.

[17]:
```
# Field methods
fxxyy = f.differentiate(x=2, y=2)
total_int_f = f.integrate('x', 'y')
f00 = f.interpolate(x=0, y=0)

# Print point interpolation
# The result is a constant field, so we can print its value at any grid point
print('f(x=0, y=0):', f00['g'][0,0])
```

```
f(x=0, y=0): 3.0185735210880122
```

Tutorial 3: Problems and Solvers

Overview: This tutorials covers the basics of setting up and solving problems using Dedalus. Dedalus solves symbolically specified initial value, boundary value, and eigenvalue problems.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from dedalus import public as de
```

[2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

3.1: Problems

Problem formulations

Dedalus standardizes the formulation of all initial value problems by taking systems of symbolically specified equations boundary conditions and manipulating them into the following generic form:

$$\mathcal{M} \cdot \partial_t \mathcal{X} + \mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X}, t)$$

where $\mathcal{M}$ and $\mathcal{L}$ are matrices of linear differential operators, $\mathcal{X}$ is a state vector of the unknown fields, $\mathcal{F}$ is a vector of general nonlinear expressions. This form encapsulates prognostic/evolution equations (equations containing time derivatives) and diagnostic/algebraic constraints (equations without time derivatives), both in the interior of the domain and on the boundary.

Importantly, the left-hand side (LHS) of the equations must be time-independent, first-order in temporal derivatives, and linear in the problem variables. The right-hand side (RHS) of the equations may contain nonlinear and time-dependent terms, but no temporal derivatives. In addition, the LHS must be first-order in non-separable (e.g. Chebyshev) derivatives, and may only contain non-constant coefficients that vary in the non-periodic dimensions (others must be placed on the RHS).

In addition to initial value problems, there are similar standard forms for generalized eigenvalue problems ($\sigma\mathcal{M} \cdot \mathcal{X} + \mathcal{L} \cdot \mathcal{X} = 0$), linear boundary value problems ($\mathcal{L} \cdot \mathcal{X} = \mathcal{G}$), and nonlinear boundary value problems ($\mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X})$). These four problem types are represented by the `IVP`, `EVP`, `LBVP`, and `NLBVP` problem classes, respectively.

To create a problem object, you must provide a domain object and the names of the variables that appear in the equations. Let's start setting up the complex Ginzburg-Landau equation (CGLE) for a variable $u(x, t)$ on a finite interval $x \in [0, 300]$ with homogeneous Dirichlet boundary conditions:

$$\partial_t u = u + (1 + ib)\partial_{xx} u - (1 + ic)|u|^2 u$$

$$u(x=0) = u(x=300) = 0$$

The CGLE only has one primative variable but is second-order in its spatial derivatives, so we'll have to introduce an extra variable to reduce the equation to first-order. We also pick a dealiasing factor of 2 to correctly dealias the cubic nonlinearity.

[3]:
```
# Build bases and domain
x_basis = de.Chebyshev('x', 1024, interval=(0, 300), dealias=2)
domain = de.Domain([x_basis], grid_dtype=np.complex128)

# Build problem
problem = de.IVP(domain, variables=['u', 'ux'])
```

Variable metadata

Metadata for the problem variables can be specified through the `meta` attribute of the problem object, and indexing by variable name, axis, and property, respectively.

The most common metadata to set here is the `parity` option when using the `SinCos` basis. The parity for each field and for each `SinCos` basis needs to be set to either `+1` or `-1` to indicate that the field has even (cosine) or odd (sine) parity around the interval endpoints for the corresponding dimension.

Here we aren't using a `SinCos` basis, but if we were, we could set the variable parities as:

[4]:
```
# problem.meta['u']['x']['parity'] = +1
# problem.meta['ux']['x']['parity'] = -1
```

Parameters and non-constant coefficients

Before adding equations to the problem, we first add any parameters, defined as fields or scalars used in the equations but not part of the state vector of problem variables, to the `problem.parameters` dictionary.

For constant/scalar parameters, like we have here, we simply add the desired numerical values to the parameters dictionary.

[5]:
```
problem.parameters['b'] = 0.5
problem.parameters['c'] = -1.76
```

For non-constant coefficients (NCCs), we pass a field object with the desired data. Dedalus only accepts NCCs that couple the non-periodic dimensions, i.e. are constant along all `Fourier` and `SinCos` dimensions, so that those dimensions remain linearly uncoupled. To inform the parser that a NCC will not couple these directions, you must explicitly add some metadata to the NCC fields indiciating that they are constant along any periodic directions.

We don't have NCCs or periodic dimensions here, but we'll sketch the process here anyways. Consider a 3D problem on a Fourier (x), SinCos (y), and Chebyshev (z) domain. Here's how we would add a simple non-constant coefficient in z to a problem:

[6]:
```
# ncc = Field(domain, name='c')
# ncc['g'] = z**2
# ncc.meta['x', 'y']['constant'] = True
# problem.parameters['c'] = ncc
```

Substitutions

To simplify equation entry, you can define substitutions which effectively act as string-replacement rules during the parsing process. Substitutions can be used to provide short aliases to quantities computed from the problem variables. They can also define shortcut functions similar to Python lambda functions, but with normal mathematical syntax.

Here we'll create a simple substitution for computing the squared magnitude of a variable:

[7]:
```
# Function-like substitution using dummy variables
problem.substitutions["mag_sq(A)"] = "A * conj(A)"
```

Equation entry

Equations and boundary conditions are then entered in plain text, optionally with condition strings specifying which separable modes (indexed by `nx` and `ny` for separable axes named `x` and `y`, etc.) that equation applies to.

The parsing namespace basically consists of:

- The variables, parameters, and substitutions defined in the problem
- The axis names representing the individual basis grids, e.g. `'x'`
- The derivative operators for each basis, named as e.g. `'dx'`
- The `differentiate`, `integrate`, and `interpolate` factories as `'d'`, `'integ'`, and `'interp'`
- `'left'` and `'right'` as aliases to interpolation at the Chebyshev endpoints, if present
- Time and temporal derivatives as `'t'` and `'dt'`
- Simple mathematical functions (logarithmic and trigonometric), e.g. `'sin'`, `'exp'`, …

Let's enter the CGLE (as two first-order equations) and the boundary conditions:

[8]:
```
# Add main equation, with linear terms on the LHS and nonlinear terms on the RHS
problem.add_equation("dt(u) - u - (1 + 1j*b)*dx(ux) = - (1 + 1j*c) * mag_sq(u) * u")

# Add auxiliary equation defining the first-order reduction
problem.add_equation("ux - dx(u) = 0")

# Add boundary conditions
problem.add_equation("left(u) = 0")
problem.add_equation("right(u) = 0")
```

3.2: Solvers

Building a solver

Each problem type (IVP, EVP, LBVP, and NLBVP) has a corresponding solver class that actually performs the solution steps for a corresponding problem. Solvers are simply built using the `problem.build_solver` method.

For IVPs, we select a timestepping method when building the solver. Several multistep and Runge-Kutta IMEX schemes are available in the timesteppers.py module, and can be selected by name.

[9]:
```
# Build solver
solver = problem.build_solver('RK222')
```

```
2020-11-28 23:48:12,636 pencil 0/1 INFO :: Building pencil matrix 1/1 (~100%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.4e+01/s
```

Setting stop criteria

For IVPs, stopping criteria for halting time evolution are specified by setting the `solver.stop_iteration`, `solver.stop_wall_time` (seconds since solver instantiation), and/or `solver.stop_sim_time` attributes.

Let's stop at $t = 500$ in simulation units:

[10]:
```
# Stopping criteria
solver.stop_sim_time = 500
solver.stop_wall_time = np.inf
solver.stop_iteration = np.inf
```

Setting initial conditions

The fields representing the problem variables can be accessed with a dictionary-like interface through the `solver.state` system. For IVPs and nonlinear BVPs, initial conditions are set by directly modifying the state variable data before running a simulation.

[11]:
```
# Reference local grid and state fields
x = domain.grid(0)
u = solver.state['u']
ux = solver.state['ux']

# Setup a sine wave
u.set_scales(1)
u['g'] = 1e-3 * np.sin(5 * np.pi * x / 300)
u.differentiate('x', out=ux);
```

Solving/iterating a problem

IVPs are iterated using the `solver.step` method with a provided timestep. EVPs are solved using the `solver.solve_dense` or `solver.solve_sparse` methods. LBVPs are solved using the `solver.solve` method. NLBVPs are iterated using the `solver.newton_iteration` method.

The logic controlling the main-loop of a Dedalus IVP simulation occurs explicitly in the simulation script. The `solver.proceed` property can change from `True` to `False` once any of the specified stopping criteria have been met. Let's timestep our problem until the halting condition is reached, copying the grid values of `u` every few iterations. This should take just a few seconds to run.

[12]:
```
# Setup storage
u.set_scales(1)
u_list = [np.copy(u['g'])]
t_list = [solver.sim_time]

# Main loop
dt = 0.05
while solver.ok:
```

(continues on next page)

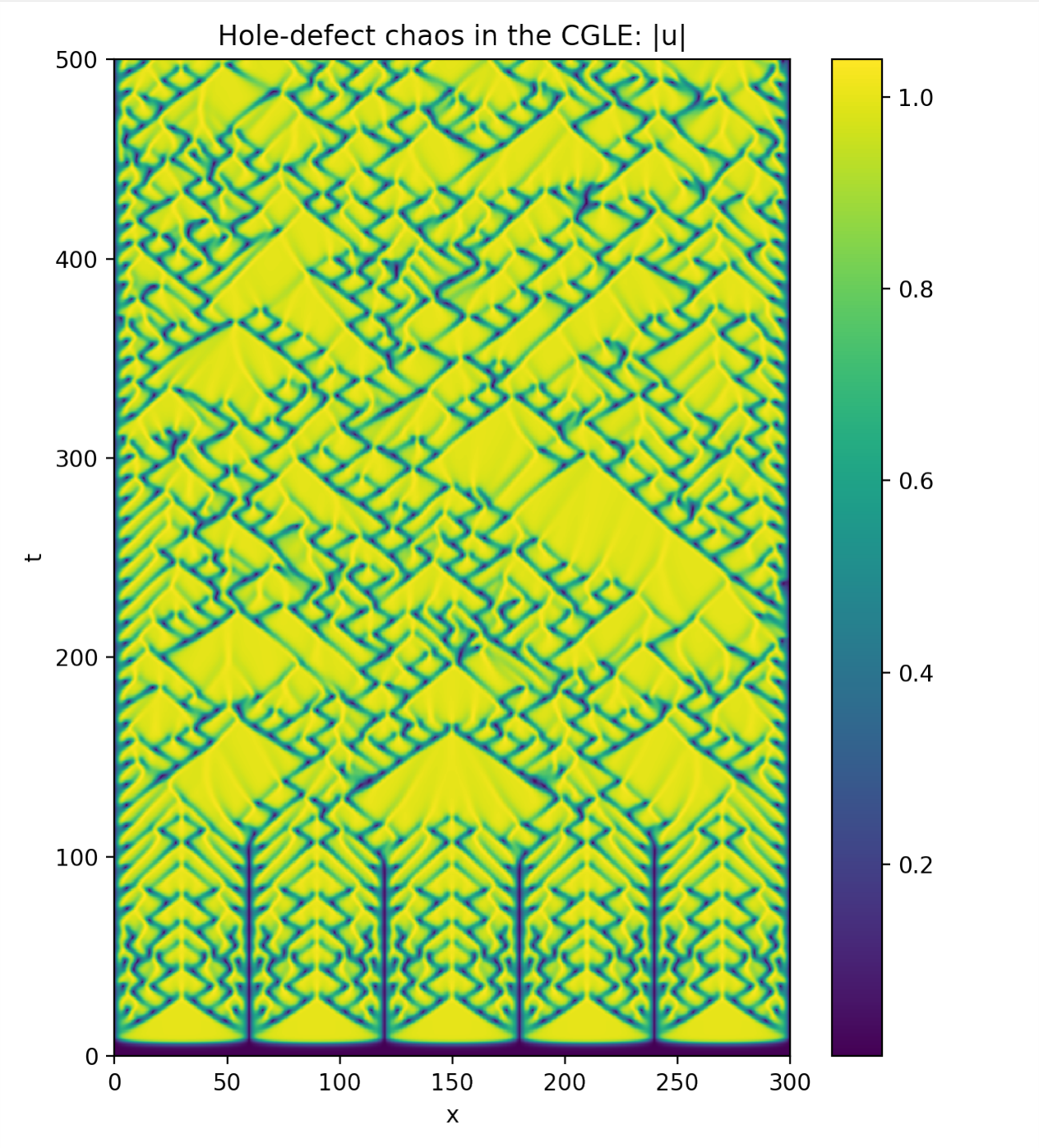
(continued from previous page)

```
    solver.step(dt)
    if solver.iteration % 10 == 0:
        u.set_scales(1)
        u_list.append(np.copy(u['g']))
        t_list.append(solver.sim_time)
    if solver.iteration % 1000 == 0:
        print('Completed iteration {}'.format(solver.iteration))

# Convert storage lists to arrays
u_array = np.array(u_list)
t_array = np.array(t_list)
```

```
Completed iteration 1000
Completed iteration 2000
Completed iteration 3000
Completed iteration 4000
Completed iteration 5000
Completed iteration 6000
Completed iteration 7000
Completed iteration 8000
Completed iteration 9000
Completed iteration 10000
2020-11-28 23:48:32,822 solvers 0/1 INFO :: Simulation stop time reached.
```

Now let's make a space-time plot of the magnitude of the solution:

[13]:
```
# Plot solution
plt.figure(figsize=(6, 7), dpi=100)
plt.pcolormesh(x, t_array, np.abs(u_array), shading='nearest')
plt.colorbar()
plt.xlabel('x')
plt.ylabel('t')
plt.title('Hole-defect chaos in the CGLE: |u|')
plt.tight_layout()
```


Hole-defect chaos in the CGLE: |u|
500
400
300
200
100
0
t
0
50
100
150
200
250
300
x
1.0
0.8
0.6
0.4
0.2

Tutorial 4: Analysis and Post-processing

Overview: This notebook covers the basics of data analysis and post-processing using Dedalus. Analysis tasks can be specified symbolically and are saved to distributed HDF5 files.

[1]:
```
import pathlib
import subprocess
import h5py
import numpy as np
import matplotlib.pyplot as plt
from dedalus import public as de
```

[2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

[3]:
```
# Clean up any old files
import shutil
shutil.rmtree('analysis', ignore_errors=True)
```

4.1: Analysis

Dedalus includes a framework for evaluating and saving arbitrary analysis tasks while an initial value problem is running. To get started, let's setup the complex Ginzburg-Landau problem from the previous tutorial.

[4]:
```
# Build bases and domain
x_basis = de.Chebyshev('x', 1024, interval=(0, 300), dealias=2)
domain = de.Domain([x_basis], grid_dtype=np.complex128)

# Build problem
problem = de.IVP(domain, variables=['u', 'ux'])
problem.parameters['b'] = 0.5
problem.parameters['c'] = -1.76
problem.substitutions["mag_sq(A)"] = "A * conj(A)"
problem.add_equation("dt(u) - u - (1 + 1j*b)*dx(ux) = - (1 + 1j*c) * mag_sq(u) * u")
problem.add_equation("ux - dx(u) = 0")
problem.add_equation("left(u) = 0")
problem.add_equation("right(u) = 0")

# Build solver
solver = problem.build_solver('RK222')
solver.stop_sim_time = 500
solver.stop_wall_time = np.inf
solver.stop_iteration = np.inf

# Reference local grid and state fields
x = domain.grid(0)
u = solver.state['u']
ux = solver.state['ux']

# Setup a sine wave
u.set_scales(1)
```

(continues on next page)

(continued from previous page)

```
u['g'] = 1e-3 * np.sin(5 * np.pi * x / 300)
u.differentiate('x', out=ux);
```

```
2020-11-28 23:49:10,234 pencil 0/1 INFO :: Building pencil matrix 1/1 (~100%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.8e+01/s
```

Analysis handlers

The explicit evaluation of analysis tasks during timestepping is controlled by the `solver.evaluator` object. Various handler objects can be attached to the evaluator, and control when the evaluator computes their own set of tasks and what happens to the resulting data.

For example, an internal `SystemHandler` object directs the evaluator to evaluate the RHS expressions on every iteration, and uses the data for the explicit part of the timestepping algorithm.

For simulation analysis, the most useful handler is the `FileHandler`, which regularly computes tasks and writes the data to HDF5 files. When setting up a file handler, you specify the name/path for the output directory/files, as well as the cadence at which you want the handler's tasks to be evaluated. This cadence can be in terms of any combination of

- simulation time, specified with `sim_dt`
- wall time, specified with `wall_dt`
- iteration number, specified with `iter`

To limit file sizes, the output from a file handler is split up into different "sets" over time, each containing some number of writes that can be limited with the `max_writes` keyword when the file handler is constructed.

Let's setup a file handler to be evaluated every few iterations.

[5]:
```
analysis = solver.evaluator.add_file_handler('analysis', iter=10, max_writes=200)
```

You can add an arbitrary number of file handlers to save different sets of tasks at different cadences and to different files.

Analysis tasks

Analysis tasks are added to a given handler using the `add_task` method. Tasks are entered in plain text, and parsed using the same namespace that is used for equation entry. For each task, you can additionally specify the output layout and scaling factors.

Let's add tasks for tracking the average magnitude of the solution.

[6]:
```
analysis.add_task("integ(sqrt(mag_sq(u)),'x')/300", layout='g', name='<|u|>')
```

For checkpointing, you can also simply specify that all of the state variables should be saved.

[7]:
```
analysis.add_system(solver.state, layout='g')
```

We can now run the simulation just as in the previous tutorial, but without needing to manually save any data during the main loop. The evaluator will automatically compute and save the specified analysis tasks at the proper cadence as the simulation is advanced.

[8]:
```
# Main loop
dt = 0.05
while solver.ok:
    solver.step(dt)
    if solver.iteration % 1000 == 0:
        print('Completed iteration {}'.format(solver.iteration))
```

```
Completed iteration 1000
Completed iteration 2000
Completed iteration 3000
Completed iteration 4000
Completed iteration 5000
Completed iteration 6000
Completed iteration 7000
Completed iteration 8000
Completed iteration 9000
Completed iteration 10000
2020-11-28 23:49:42,950 solvers 0/1 INFO :: Simulation stop time reached.
```

4.2: Post-processing

File arrangement

By default, the output files for each file handler are arranged as follows: 1. A base folder taking the name that was specified when the file handler was constructed, e.g. `analysis/`. 2. Within the base folder are subfolders for each set of outputs, with the same name plus a set number, e.g. `analysis_s0/`. 3. Within each set subfolder are HDF5 files for each process, with the same name plus a process number, e.g. `analysis_s0_p1.h5`.

Let's take a look at the output files from our example problem. We should see five sets (10000 total iterations, output every 10 iterations, 200 writes per file) and data from one process (indexed starting from 0).

[9]:
```
print(subprocess.check_output("find analysis", shell=True).decode())
```

```
analysis
analysis/analysis_s1
analysis/analysis_s1/analysis_s1_p0.h5
analysis/analysis_s5
analysis/analysis_s5/analysis_s5_p0.h5
analysis/analysis_s2
analysis/analysis_s2/analysis_s2_p0.h5
analysis/analysis_s3
analysis/analysis_s3/analysis_s3_p0.h5
analysis/analysis_s4
analysis/analysis_s4/analysis_s4_p0.h5
```

Merging output files

By default, each process writes its local portion of the analysis tasks to its own file, but often it is substantially easier to deal with the global dataset. The distributed process files can be easily merged into a global file for each set using the `merge_process_files` function from the `dedalus.tools.post` module.

Since we ran this problem serially, here this will essentially just perform a copy of the root process file, but we'll do the merge for illustrative purposes, anyways.

[10]:
```
from dedalus.tools import post
post.merge_process_files("analysis", cleanup=True)
```

```
2020-11-28 23:49:42,978 post 0/1 INFO :: Merging files from analysis
```

After the merge, we see that instead of a subfolder and process files for each output set, we have a single global set file for each output set.

[11]:
```
print(subprocess.check_output("find analysis", shell=True).decode())
```

```
analysis
analysis/analysis_s5.h5
analysis/analysis_s1.h5
analysis/analysis_s4.h5
analysis/analysis_s3.h5
analysis/analysis_s2.h5

```

For some types of analysis, it's additionally convenient to merge the output sets together into a single file that's global in space and time, which can be done with the `merge_sets` function.

Note: this can generate very large files, so it's not recommended for post-processing that just requires slices in time, e.g. plotting snapshots of an analysis task at different times. However, if you want to explicitly plot a quantity versus time, instead of slicing over time, it can be useful.

[12]:
```
set_paths = list(pathlib.Path("analysis").glob("analysis_s*.h5"))
post.merge_sets("analysis/analysis.h5", set_paths, cleanup=True)
```

```
2020-11-28 23:49:43,326 post 0/1 INFO :: Creating joint file analysis/analysis.h5
```

Now we see that the two sets have been merged into a single file.

[13]:
```
print(subprocess.check_output("find analysis", shell=True).decode())
```

```
analysis
analysis/analysis.h5

```

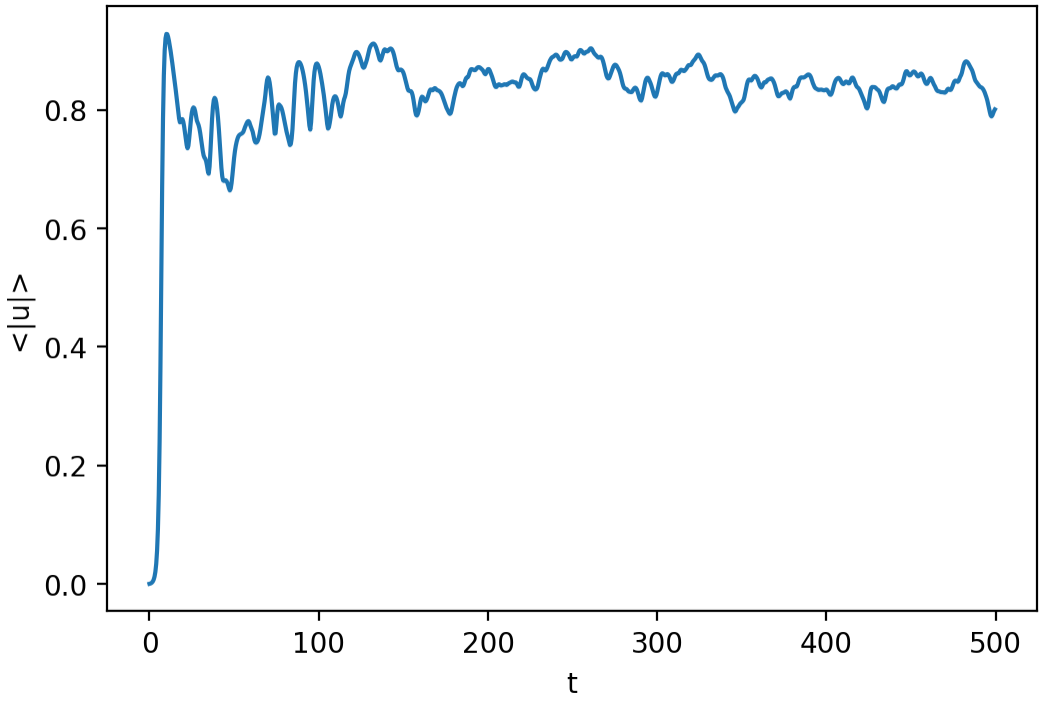
Handling data

Each HDF5 file contains a "tasks" group containing a dataset for each task assigned to the file handler. The first dimension of the dataset is time, and the subsequent dimensions are the spatial dimensions of the output field.

The HDF5 datasets are self-describing, with dimensional scales attached to each axis. For the first axis, these include the simulation time, wall time, iteration, and write number. For the spatial axes, the scales correspond to grid points or modes, based on the task layout. See the h5py docs for more details.

Let's open up the merged analysis file and plot time series of the average magnitude.

[14]:
```
with h5py.File("analysis/analysis.h5", mode='r') as file:
    # Load datasets
    mag_u = file['tasks']['<|u|>']
    t = mag_u.dims[0]['sim_time']
    # Plot data
    fig = plt.figure(figsize=(6, 4), dpi=100)
    plt.plot(t[:], mag_u[:].real)
    plt.xlabel('t')
    plt.ylabel('<|u|>')
```



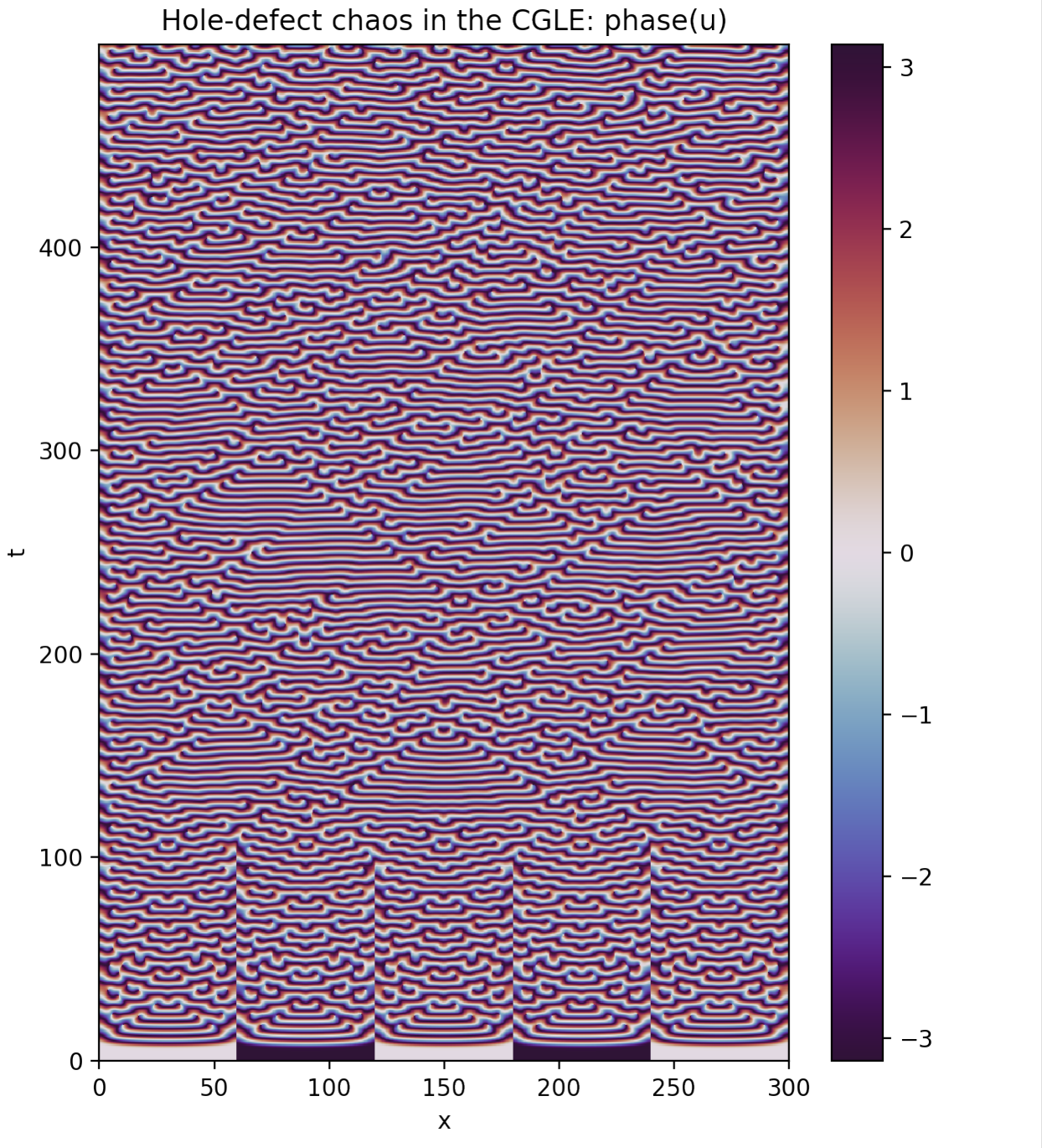
Now let's look at the saved solution over space and time. Let's plot the phase this time instead of the amplitude.

[15]:
```
with h5py.File("analysis/analysis.h5", mode='r') as file:
    # Load datasets
    u = file['tasks']['u']
    t = u.dims[0]['sim_time']
    x = u.dims[1][0]
    # Plot data
    u_phase = np.arctan2(u[:].imag, u[:].real)
    plt.figure(figsize=(6,7), dpi=100)
```

(continues on next page)

(continued from previous page)

```
    plt.pcolormesh(x[:], t[:], u_phase, shading='nearest', cmap='twilight_shifted')
    plt.colorbar()
    plt.xlabel('x')
    plt.ylabel('t')
    plt.title('Hole-defect chaos in the CGLE: phase(u)')
    plt.tight_layout()
```



1.2.2 Example Notebooks

Below are several notebooks that walk through the setup and execution of more complicated multidimensional example problems.

Kelvin-Helmholtz Instability

(image: Chuck Doswell)

We will simulate the incompressible Kelvin-Helmholtz problem. We non-dimensionalize the problem by taking the box height to be one and the jump in velocity to be one. Then the Reynolds number is given by

$$\mathrm{Re} = \frac{UH}{\nu} = \frac{1}{\nu}.$$

We use no slip boundary conditions, and a box with aspect ratio $L/H = 2$. The initial velocity profile is given by a hyperbolic tangent, and only a single mode is initially excited. We will also track a passive scalar which will help us visualize the instability.

First, we import the necessary modules.

[1]:
```
%matplotlib inline
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import time
from IPython import display

from dedalus import public as de
from dedalus.extras import flow_tools

import logging
logger = logging.getLogger(__name__)
```

To perform an initial value problem (IVP) in Dedalus, you need three things:

1. A domain to solve the problem on
2. Equations to solve
3. A timestepping scheme

Problem Domain

First, we will specify the domain. Domains are built by taking the direct product of bases. Here we are running a 2D simulation, so we will define x and y bases. From these, we build the domain.

[3]:
```
# Aspect ratio 2
Lx, Ly = (2., 1.)
nx, ny = (192, 96)

# Create bases and domain
```

(continues on next page)

(continued from previous page)

```
x_basis = de.Fourier('x', nx, interval=(0, Lx), dealias=3/2)
y_basis = de.Chebyshev('y',ny, interval=(-Ly/2, Ly/2), dealias=3/2)
domain = de.Domain([x_basis, y_basis], grid_dtype=np.float64)
```

The last basis (y direction) is represented in Chebyshev polynomials. This will allow us to apply interesting boundary conditions in the y direction. We call the other directions (in this case just x) the "horizontal" directions. The horizontal directions must be "easy" in the sense that taking derivatives cannot couple different horizontal modes. Right now, we have Fourier and Sin/Cos series implemented for the horizontal directions, and are working on implementing spherical harmonics.

Equations

Next we will define the equations that will be solved on this domain. The equations are

$$\partial_t u + \boldsymbol{u}\cdot\boldsymbol{\nabla} u + \partial_x p = \frac{1}{\mathrm{Re}}\nabla^2 u$$

$$\partial_t v + \boldsymbol{u}\cdot\boldsymbol{\nabla} v + \partial_y p = \frac{1}{\mathrm{Re}}\nabla^2 v$$

$$\boldsymbol{\nabla}\cdot\boldsymbol{u} = 0$$

$$\partial_t s + \boldsymbol{u}\cdot\boldsymbol{\nabla} s = \frac{1}{\mathrm{ReSc}}\nabla^2 s$$

The equations are written such that the left-hand side (LHS) is treated implicitly, and the right-hand side (RHS) is treated explicitly. The LHS is limited to only linear terms, though linear terms can also be placed on the RHS. Since y is our special direction in this example, we also restrict the LHS to be at most first order in derivatives with respect to y.

We also set the parameters, the Reynolds and Schmidt numbers.

[4]:
```
Reynolds = 1e4
Schmidt = 1.

problem = de.IVP(domain, variables=['p','u','v','uy','vy','S','Sy'])
problem.parameters['Re'] = Reynolds
problem.parameters['Sc'] = Schmidt
problem.add_equation("dt(u) + dx(p) - 1/Re*(dx(dx(u)) + dy(uy)) = - u*dx(u) - v*uy")
problem.add_equation("dt(v) + dy(p) - 1/Re*(dx(dx(v)) + dy(vy)) = - u*dx(v) - v*vy")
problem.add_equation("dx(u) + vy = 0")
problem.add_equation("dt(S) - 1/(Re*Sc)*(dx(dx(S)) + dy(Sy)) = - u*dx(S) - v*Sy")
problem.add_equation("uy - dy(u) = 0")
problem.add_equation("vy - dy(v) = 0")
problem.add_equation("Sy - dy(S) = 0")
```

Because we are using this first-order formalism, we define auxiliary variables `uy`, `vy`, and `Sy` to be the y-derivative of `u`, `v`, and `S` respectively.

Next, we set our boundary conditions. "Left" boundary conditions are applied at $y = -Ly/2$ and "right" boundary conditions are applied at $y = +Ly/2$.

[5]:
```
problem.add_bc("left(u) = 0.5")
problem.add_bc("right(u) = -0.5")
problem.add_bc("left(v) = 0")
```

(continues on next page)

(continued from previous page)

```
problem.add_bc("right(v) = 0", condition="(nx != 0)")
problem.add_bc("left(p) = 0", condition="(nx == 0)")
problem.add_bc("left(S) = 0")
problem.add_bc("right(S) = 1")
```

Note that we have a special boundary condition for the $k_x = 0$ mode (singled out by `condition="(nx==0)"`). This is because the continuity equation implies $\partial_y v = 0$ if $k_x = 0$; thus, $v = 0$ on the top and bottom are redundant boundary conditions. We replace one of these with a gauge choice for the pressure.

Timestepping

We have implemented a variety of multi-step and Runge-Kutta implicit-explicit timesteppers in Dedalus. The available options can be seen in the timesteppers.py module. For this problem, we will use a third-order, four-stage Runge-Kutta integrator. Changing the timestepping algorithm is as easy as changing one line of code.

[6]:
```
ts = de.timesteppers.RK443
```

Initial Value Problem

We now have the three ingredients necessary to set up our IVP:

[7]:
```
solver =  problem.build_solver(ts)
```

```
2020-10-30 21:55:41,722 pencil 0/1 INFO :: Building pencil matrix 1/96 (~1%) Elapsed: 0s,
↪ Remaining: 5s, Rate: 1.8e+01/s
2020-10-30 21:55:42,027 pencil 0/1 INFO :: Building pencil matrix 10/96 (~10%) Elapsed:␣
↪0s, Remaining: 3s, Rate: 2.8e+01/s
2020-10-30 21:55:42,271 pencil 0/1 INFO :: Building pencil matrix 20/96 (~21%) Elapsed:␣
↪1s, Remaining: 2s, Rate: 3.3e+01/s
2020-10-30 21:55:42,489 pencil 0/1 INFO :: Building pencil matrix 30/96 (~31%) Elapsed:␣
↪1s, Remaining: 2s, Rate: 3.6e+01/s
2020-10-30 21:55:42,654 pencil 0/1 INFO :: Building pencil matrix 40/96 (~42%) Elapsed:␣
↪1s, Remaining: 1s, Rate: 4.0e+01/s
2020-10-30 21:55:42,842 pencil 0/1 INFO :: Building pencil matrix 50/96 (~52%) Elapsed:␣
↪1s, Remaining: 1s, Rate: 4.3e+01/s
2020-10-30 21:55:43,020 pencil 0/1 INFO :: Building pencil matrix 60/96 (~62%) Elapsed:␣
↪1s, Remaining: 1s, Rate: 4.4e+01/s
2020-10-30 21:55:43,187 pencil 0/1 INFO :: Building pencil matrix 70/96 (~73%) Elapsed:␣
↪2s, Remaining: 1s, Rate: 4.6e+01/s
2020-10-30 21:55:43,397 pencil 0/1 INFO :: Building pencil matrix 80/96 (~83%) Elapsed:␣
↪2s, Remaining: 0s, Rate: 4.6e+01/s
2020-10-30 21:55:43,565 pencil 0/1 INFO :: Building pencil matrix 90/96 (~94%) Elapsed:␣
↪2s, Remaining: 0s, Rate: 4.7e+01/s
2020-10-30 21:55:43,670 pencil 0/1 INFO :: Building pencil matrix 96/96 (~100%) Elapsed:␣
↪2s, Remaining: 0s, Rate: 4.8e+01/s
```

Now we set our initial conditions. We set the horizontal velocity and scalar field to tanh profiles, and using a single-mode initial perturbation in v.

[8]:
```
x = domain.grid(0)
y = domain.grid(1)
```

(continues on next page)

(continued from previous page)

```
u = solver.state['u']
uy = solver.state['uy']
v = solver.state['v']
vy = solver.state['vy']
S = solver.state['S']
Sy = solver.state['Sy']

a = 0.05
sigma = 0.2
flow = -0.5
amp = -0.2
u['g'] = flow*np.tanh(y/a)
v['g'] = amp*np.sin(2.0*np.pi*x/Lx)*np.exp(-(y*y)/(sigma*sigma))
S['g'] = 0.5*(1+np.tanh(y/a))
u.differentiate('y',out=uy)
v.differentiate('y',out=vy)
S.differentiate('y',out=Sy)
```

[8]:
```
<Field 140447541869648>
```

Now we set integration parameters and the CFL.

[9]:
```
solver.stop_sim_time = 2.01
solver.stop_wall_time = np.inf
solver.stop_iteration = np.inf

initial_dt = 0.2*Lx/nx
cfl = flow_tools.CFL(solver,initial_dt,safety=0.8)
cfl.add_velocities(('u','v'))
```

Analysis

We have a sophisticated analysis framework in which the user specifies analysis tasks as strings. Users can output full data cubes, slices, volume averages, and more. Here we will only output a few 2D slices, and a 1D profile of the horizontally averaged concentration field. Data is output in the hdf5 file format.

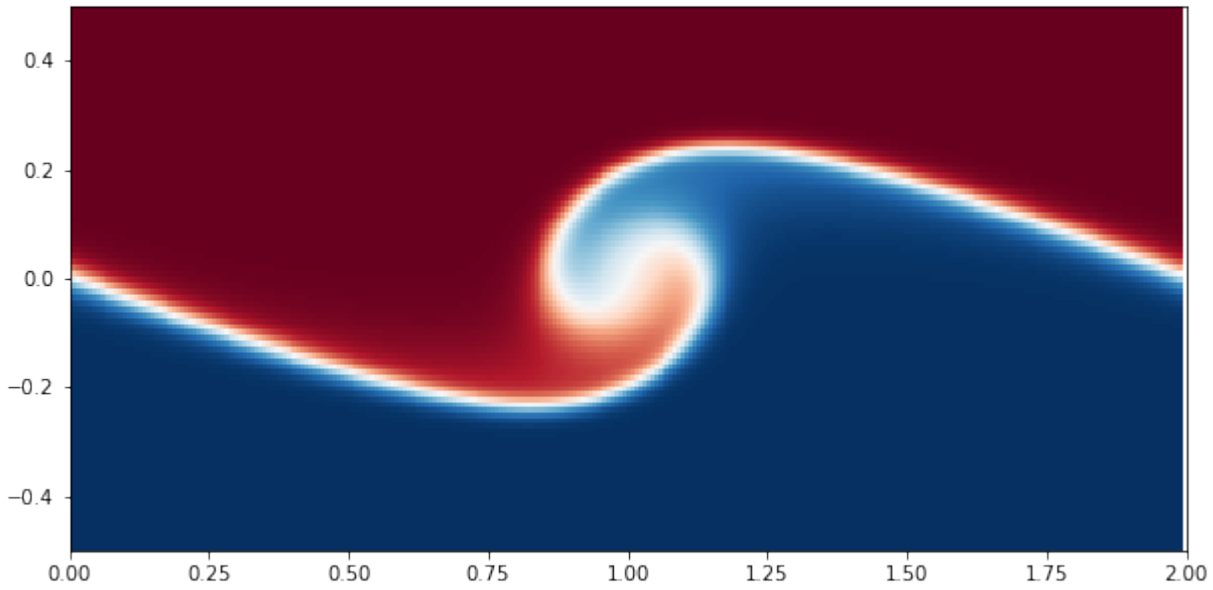
[10]:
```
analysis = solver.evaluator.add_file_handler('analysis_tasks', sim_dt=0.1, max_writes=50)
analysis.add_task('S')
analysis.add_task('u')
solver.evaluator.vars['Lx'] = Lx
analysis.add_task("integ(S,'x')/Lx", name='S profile')
```

Main Loop

We now have everything set up for our simulation. In Dedalus, the user writes their own main loop.

[11]:
```
# Make plot of scalar field
x = domain.grid(0,scales=domain.dealias)
y = domain.grid(1,scales=domain.dealias)
xm, ym = np.meshgrid(x,y)
fig, axis = plt.subplots(figsize=(10,5))
p = axis.pcolormesh(xm, ym, S['g'].T, cmap='RdBu_r');
axis.set_xlim([0,2.])
axis.set_ylim([-0.5,0.5])

logger.info('Starting loop')
start_time = time.time()
while solver.ok:
    dt = cfl.compute_dt()
    solver.step(dt)
    if solver.iteration % 10 == 0:
        # Update plot of scalar field
        p.set_array(np.ravel(S['g'][:-1,:-1].T))
        display.clear_output()
        display.display(plt.gcf())
        logger.info('Iteration: %i, Time: %e, dt: %e' %(solver.iteration, solver.sim_
↪time, dt))

end_time = time.time()

p.set_array(np.ravel(S['g'][:-1,:-1].T))
display.clear_output()
# Print statistics
logger.info('Run time: %f' %(end_time-start_time))
logger.info('Iterations: %i' %solver.iteration)
```

```
2020-10-30 21:58:16,584 __main__ 0/1 INFO :: Run time: 152.722414
2020-10-30 21:58:16,586 __main__ 0/1 INFO :: Iterations: 269
```

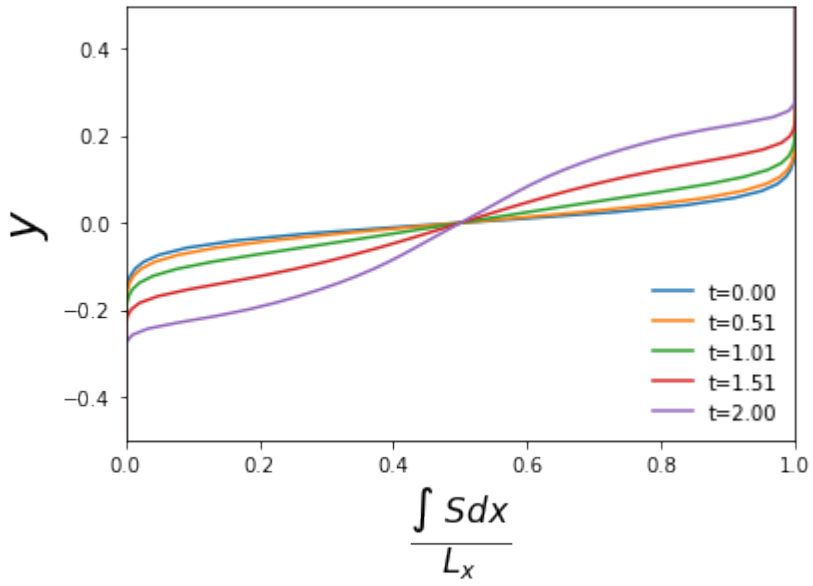


Analysis

As an example of doing some analysis, we will load in the horizontally averaged profiles of the scalar field s and plot them.

[12]:
```
# Read in the data
f = h5py.File('analysis_tasks/analysis_tasks_s1/analysis_tasks_s1_p0.h5','r')
y = f['/scales/y/1.0'][:]
t = f['scales']['sim_time'][:]
S_ave = f['tasks']['S profile'][:]
f.close()

S_ave = S_ave[:,0,:] # remove length-one x dimension
```

[13]:
```
for i in range(0,21,5):
  plt.plot(S_ave[i,:],y,label='t=%4.2f' %t[i])

plt.ylim([-0.5,0.5])
plt.xlim([0,1])
plt.xlabel(r'$\frac{\int \ S dx}{L_x}$',fontsize=24)
plt.ylabel(r'$y$',fontsize=24)
plt.legend(loc='lower right').draw_frame(False)
```



[ ]:

Axisymmetric Taylor-Couette flow in Dedalus

(image: wikipedia)

Taylor-Couette flow is characterized by three dimensionless numbers:

$\eta = R1/R2$, the ratio of the inner cylidner radius R_1 to the outer cylinder radius R_2

$\mu = \Omega_2/\Omega_1$, the ratio of the OUTER cylinder rotation rate Ω_2 to the inner rate Ω_1

$\mathrm{Re} = \Omega_1 R_1 \delta/\nu$, the Reynolds numbers, where $\delta = R_2 - R_1$, the gap width between the cylinders

We non dimensionalize the flow in terms of

$[\mathrm{L}] = \delta = R_2 - R_1$

$[\mathrm{V}] = R_1\Omega_1$

$[\mathrm{M}] = \rho\delta^3$

And choose $\delta = 1$, $R_1\Omega_1 = 1$, and $\rho = 1$.

[1]:
```
%pylab inline
```

[2]:
```
import numpy as np
import time
import h5py

import dedalus.public as de
from dedalus.extras import flow_tools

import logging
logger = logging.getLogger(__name__)
```

Input parameters

These parameters are taken from Barenghi (1991) J. Comp. Phys. We'll compute the growth rate and compare it to the value $\gamma_{analytic} = 0.430108693$

[3]:
```
# Input parameters from Barenghi (1991)
eta = 1./1.444 # R1/R2
alpha = 3.13   # vertical wavenumber
Re = 80.       # in units of R1*Omega1*delta/nu
mu = 0.        # Omega2/Omega1

# Computed quantitites
omega_in = 1.
omega_out = mu * omega_in
r_in = eta/(1. - eta)
r_out = 1./(1. - eta)
height = 2.*np.pi/alpha
v_l = 1. # by default, we set v_l to 1.
v_r = omega_out*r_out
```

Problem Domain

Every PDE takes place somewhere, so we define a *domain*, which in this case is the z and r directions. Because the r direction has walls, we use a `Chebyshev` basis, but the z direction is periodic, so we use a `Fourier` basis. The `Domain` object combines these.

[4]:
```
# Create bases
r_basis = de.Chebyshev('r', 32, interval=(r_in, r_out), dealias=3/2)
z_basis = de.Fourier('z', 32, interval=(0., height), dealias=3/2)
domain = de.Domain([z_basis, r_basis], grid_dtype=np.float64)
```

Equations

We use the `IVP` object, which can parse a set of equations in plain text and combine them into an initial value problem.

Here, we code up the equations for the "primative" variables, $\mathbf{v} = u\hat{\mathbf{r}} + v\hat{\theta} + w\hat{\mathbf{z}}$ and p, along with their first derivatives.

The equations are the incompressible, axisymmetric Navier-Stokes equations in cylindrical coordinates

The axes will be called z and r, and we will expand the non-constant r^2 terms, to a cutoff precision of 10^{-8}. These non-constant coefficients (called "NCC" in Dedalus) are geometric here, but they could be background states in convection, position dependent diffusion coefficients, etc.

We also add the parameters to the object, so we can use their names in the equations below.

[5]:
```
TC = de.IVP(domain, variables=['p', 'u', 'v', 'w', 'ur', 'vr', 'wr'], ncc_cutoff=1e-8)
TC.parameters['nu'] = 1./Re
TC.parameters['v_l'] = v_l
TC.parameters['v_r'] = v_r
mu = TC.parameters['v_r']/TC.parameters['v_l'] * eta
```

The equations are multiplied through by r^2, so that there are no $1/r$ terms, which require more coefficients in the expansion.

[6]:
```
TC.add_equation("r*ur + u + r*dz(w) = 0")
TC.add_equation("r*r*dt(u) - r*r*nu*dr(ur) - r*nu*ur - r*r*nu*dz(dz(u)) + nu*u +␣
↪r*r*dr(p) = -r*r*u*ur - r*r*w*dz(u) + r*v*v")
TC.add_equation("r*r*dt(v) - r*r*nu*dr(vr) - r*nu*vr - r*r*nu*dz(dz(v)) + nu*v  = -
↪r*r*u*vr - r*r*w*dz(v) - r*u*v")
TC.add_equation("r*dt(w) - r*nu*dr(wr) - nu*wr - r*nu*dz(dz(w)) + r*dz(p) = -r*u*wr -␣
↪r*w*dz(w)")
TC.add_equation("ur - dr(u) = 0")
TC.add_equation("vr - dr(v) = 0")
TC.add_equation("wr - dr(w) = 0")
```

Initial and Boundary Conditions

First we create some aliases to the r and z grids, so we can quickly compute the analytic Couette flow solution for unperturbed, unstable axisymmetric flow.

[7]:
```
r = domain.grid(1, scales=domain.dealias)
z = domain.grid(0, scales=domain.dealias)

p_analytic = (eta/(1-eta**2))**2 * (-1./(2*r**2*(1-eta)**2) -2*np.log(r) +0.5*r**2 * (1.-
↪eta)**2)
v_analytic = eta/(1-eta**2) * ((1. - mu)/(r*(1-eta)) - r * (1.-eta) * (1 - mu/eta**2))
```

And now we add boundary conditions, simply by typing them in plain text, just like the equations.

[8]:
```
# boundary conditions
TC.add_bc("left(u) = 0")
TC.add_bc("left(v) = v_l")
TC.add_bc("left(w) = 0")
TC.add_bc("right(u) = 0", condition="nz != 0")
TC.add_bc("right(v) = v_r")
TC.add_bc("right(w) = 0")
TC.add_bc("left(p) = 0", condition="nz == 0")
```

We can now set the parameters of the problem, ν, v_l, and v_r, and have the code log μ to the output (which can be `stdout`, a file, or both).

Timestepping

We have implemented a variety of multi-step and Runge-Kutta implicit-explicit timesteppers in Dedalus. The available options can be seen in the timesteppers.py module. Here we pick `RK443`, an IMEX Runga-Kutta scheme. We set our maximum timestep `max_dt`, and choose the various stopping parameters.

Finally, we've got our full initial value problem (represented by the `IVP`) object: a timestepper, a domain, and a `ParsedProblem` (or equation set).

[9]:
```
dt = max_dt = 1.
omega1 = TC.parameters['v_l']/r_in
period = 2*np.pi/omega1

ts = de.timesteppers.RK443
```

(continues on next page)

(continued from previous page)

```
IVP = TC.build_solver(ts)
IVP.stop_sim_time = 15.*period
IVP.stop_wall_time = np.inf
IVP.stop_iteration = 10000000
```

```
2020-10-30 21:57:04,858 pencil 0/1 INFO :: Building pencil matrix 1/16 (~6%) Elapsed: 0s,
↪ Remaining: 0s, Rate: 4.1e+01/s
2020-10-30 21:57:04,908 pencil 0/1 INFO :: Building pencil matrix 2/16 (~12%) Elapsed:␣
↪0s, Remaining: 1s, Rate: 2.7e+01/s
2020-10-30 21:57:04,978 pencil 0/1 INFO :: Building pencil matrix 4/16 (~25%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.8e+01/s
2020-10-30 21:57:05,068 pencil 0/1 INFO :: Building pencil matrix 6/16 (~38%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.6e+01/s
2020-10-30 21:57:05,148 pencil 0/1 INFO :: Building pencil matrix 8/16 (~50%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.5e+01/s
2020-10-30 21:57:05,237 pencil 0/1 INFO :: Building pencil matrix 10/16 (~62%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.5e+01/s
2020-10-30 21:57:05,311 pencil 0/1 INFO :: Building pencil matrix 12/16 (~75%) Elapsed:␣
↪0s, Remaining: 0s, Rate: 2.5e+01/s
2020-10-30 21:57:05,387 pencil 0/1 INFO :: Building pencil matrix 14/16 (~88%) Elapsed:␣
↪1s, Remaining: 0s, Rate: 2.5e+01/s
2020-10-30 21:57:05,455 pencil 0/1 INFO :: Building pencil matrix 16/16 (~100%) Elapsed:␣
↪1s, Remaining: 0s, Rate: 2.6e+01/s
```

We initialize the state vector, given by `IVP.state`. To make life a little easier, we set some aliases first:

[10]:
```
p = IVP.state['p']
u = IVP.state['u']
v = IVP.state['v']
w = IVP.state['w']
ur = IVP.state['ur']
vr = IVP.state['vr']
wr = IVP.state['wr']
```

Next, we create a new field, ϕ, defined on the `domain`, which we'll use below to compute incompressible, random velocity perturbations.

[11]:
```
phi = domain.new_field(name='phi')
```

Here, we set all the fields and states to their dealiased domains. Dedalus allows us to set the "scale" of our data: this allows us to interpolate our data to a grid of any size at spectral accuracy. Of course, this isn't CSI: Fluid Dynamics, so you won't get any more detail than your highest spectral coefficient.

[12]:
```
for f in [phi,p,u,v,w,ur,vr,wr]:
    f.set_scales(domain.dealias, keep_data=False)
```

Now we set the state vector with our previously computed analytic solution and compute the first derivatives (to make our system first order).

[13]:
```
v['g'] = v_analytic
# p['g'] = p_analytic

v.differentiate('r',out=vr)
```

[13]:
```
<Field 140589376849848>
```

And finally, we set some small, incompressible perturbations to the velocity field so we can kick off our linear instability.

First, we initialize ϕ (which we created above) to Gaussian noise and then mask it to only appear in the center of the domain, so we don't violate the boundary conditions. We then take its curl to get the velocity perturbations.

Unfortunately, Gaussian noise on the grid is generally a bad idea: zone-to-zone variations (that is, the highest frequency components) will be amplified arbitrarily by any differentiation. So, let's filter out those high frequency components using this handy little function:

[14]:
```
def filter_field(field,frac=0.5):
    dom = field.domain
    local_slice = dom.dist.coeff_layout.slices(scales=dom.dealias)
    coeff = []
    for i in range(dom.dim)[::-1]:
        coeff.append(np.linspace(0,1,dom.global_coeff_shape[i],endpoint=False))
    cc = np.meshgrid(*coeff)

    field_filter = np.zeros(dom.local_coeff_shape,dtype='bool')
    for i in range(dom.dim):
        field_filter = field_filter | (cc[i][local_slice] > frac)
    field['c'][field_filter] = 0j
```

This is not the best filter: it assumes that cutting off above a certain Chebyshev mode n and Fourier mode n will be OK, though this may produce anisotropies in the data (I haven't checked). If you're worrying about the anisotropy of the initial noise of your ICs, you can always replace this filter with something better.

[15]:
```
# incompressible perturbation, arbitrary vorticity
# u = -dz(phi)
# w = dr(phi) + phi/r

phi['g'] = 1e-3* np.random.randn(*v['g'].shape)*np.sin(np.pi*(r - r_in))
filter_field(phi)
phi.differentiate('z',out=u)
u['g'] *= -1
phi.differentiate('r',out=w)
w['g'] += phi['g']/r

u.differentiate('r',out=ur)
w.differentiate('r',out=wr)
```

[15]:
```
<Field 140589376849904>
```

Now we check that incompressibility is indeed satisfied, first by computing $\nabla \cdot u$,

[16]:
```
divu0 = domain.new_field(name='divu0')
u.differentiate('r',out=divu0)
divu0['g'] += u['g']/r + w.differentiate('z')['g']
```

and then by plotting it to make sure it's nowhere greater than $\sim 10^{-15}$.

[17]:
```
figsize(12,8)
pcolormesh((r[0]*ones_like(z)).T,(z*ones_like(r)).T,divu0['g'].T,cmap='PuOr')
colorbar()
```

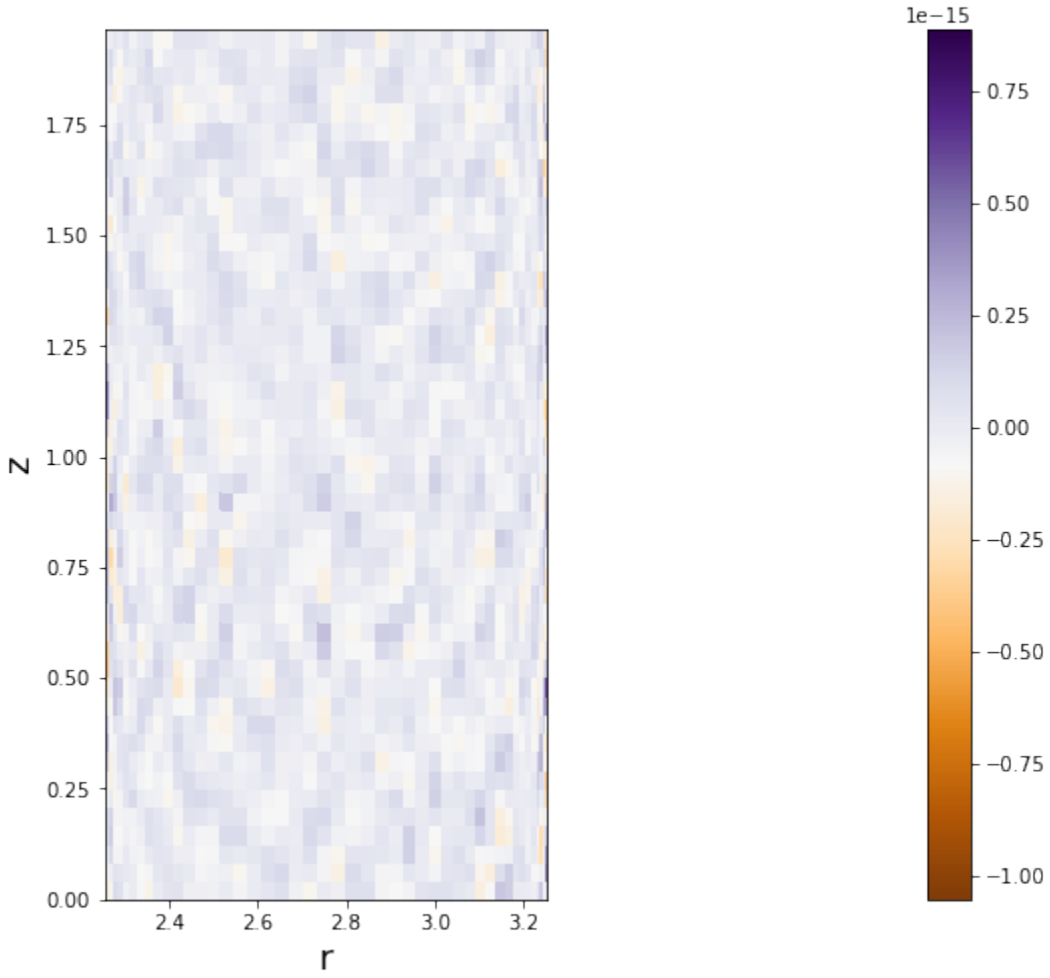
(continues on next page)

(continued from previous page)

```
axis('image')
xlabel('r', fontsize=18)
ylabel('z', fontsize=18)
```

```
/Users/kburns/Software/miniconda3/envs/dedalus/lib/python3.6/site-packages/ipykernel_
↪launcher.py:2: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same␣
↪dimensions as C is deprecated since 3.3.  Either specify the corners of the␣
↪quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set␣
↪rcParams['pcolor.shading'].  This will become an error two minor releases later.
```

[17]:
```
Text(0, 0.5, 'z')
```



Time step size and the CFL condition

Here, we use the nice CFL calculator provided by the `flow_tools` package in the `extras` module.

[18]:
```
CFL = flow_tools.CFL(IVP, initial_dt=1e-3, cadence=5, safety=0.3,
                     max_change=1.5, min_change=0.5)
CFL.add_velocities(('u', 'w'))
```

Analysis

Dedalus has a very powerful inline analysis system, and here we set up a few.

[19]:
```
# Integrated energy every 10 iterations.
analysis1 = IVP.evaluator.add_file_handler("scalar_data", iter=10)
analysis1.add_task("integ(0.5 * (u*u + v*v + w*w))", name="total kinetic energy")
analysis1.add_task("integ(0.5 * (u*u + w*w))", name="meridional kinetic energy")
analysis1.add_task("integ((u*u)**0.5)", name='u_rms')
analysis1.add_task("integ((w*w)**0.5)", name='w_rms')

# Snapshots every half an inner rotation period.
analysis2 = IVP.evaluator.add_file_handler('snapshots',sim_dt=0.5*period, max_size=2**30)
analysis2.add_system(IVP.state, layout='g')

# Radial profiles every 100 timesteps.
analysis3 = IVP.evaluator.add_file_handler("radial_profiles", iter=100)
analysis3.add_task("integ(r*v, 'z')", name='Angular Momentum')
```

The Main Loop

And here we actually run the simulation!

[20]:
```
dt = CFL.compute_dt()
# Main loop
start_time = time.time()

while IVP.ok:
    IVP.step(dt)
    if IVP.iteration % 10 == 0:
        logger.info('Iteration: %i, Time: %e, dt: %e' %(IVP.iteration, IVP.sim_time, dt))
    dt = CFL.compute_dt()


end_time = time.time()

# Print statistics
logger.info('Total time: %f' %(end_time-start_time))
logger.info('Iterations: %i' %IVP.iteration)
logger.info('Average timestep: %f' %(IVP.sim_time/IVP.iteration))
```

```
2020-10-30 21:57:06,891 __main__ 0/1 INFO :: Iteration: 10, Time: 1.200000e-02, dt: 1.
↪500000e-03
2020-10-30 21:57:07,550 __main__ 0/1 INFO :: Iteration: 20, Time: 3.825000e-02, dt: 3.
↪375000e-03
2020-10-30 21:57:07,965 __main__ 0/1 INFO :: Iteration: 30, Time: 9.731250e-02, dt: 7.
↪593750e-03
2020-10-30 21:57:08,373 __main__ 0/1 INFO :: Iteration: 40, Time: 2.302031e-01, dt: 1.
↪708594e-02
2020-10-30 21:57:09,029 __main__ 0/1 INFO :: Iteration: 50, Time: 5.292070e-01, dt: 3.
↪844336e-02
2020-10-30 21:57:09,662 __main__ 0/1 INFO :: Iteration: 60, Time: 1.201966e+00, dt: 8.
↪649756e-02
2020-10-30 21:57:10,245 __main__ 0/1 INFO :: Iteration: 70, Time: 2.715673e+00, dt: 1.
↪946195e-01
2020-10-30 21:57:10,643 __main__ 0/1 INFO :: Iteration: 80, Time: 6.121514e+00, dt: 4.
↪378939e-01
2020-10-30 21:57:11,072 __main__ 0/1 INFO :: Iteration: 90, Time: 1.378466e+01, dt: 9.
↪852613e-01
2020-10-30 21:57:11,596 __main__ 0/1 INFO :: Iteration: 100, Time: 3.102673e+01, dt: 2.
↪216838e+00
2020-10-30 21:57:12,092 __main__ 0/1 INFO :: Iteration: 110, Time: 6.982139e+01, dt: 4.
↪987885e+00
2020-10-30 21:57:12,705 __main__ 0/1 INFO :: Iteration: 120, Time: 1.571094e+02, dt: 1.
↪122274e+01
2020-10-30 21:57:12,985 solvers 0/1 INFO :: Simulation stop time reached.
2020-10-30 21:57:12,994 __main__ 0/1 INFO :: Total time: 6.586734
2020-10-30 21:57:12,997 __main__ 0/1 INFO :: Iterations: 124
2020-10-30 21:57:13,017 __main__ 0/1 INFO :: Average timestep: 1.760728
```

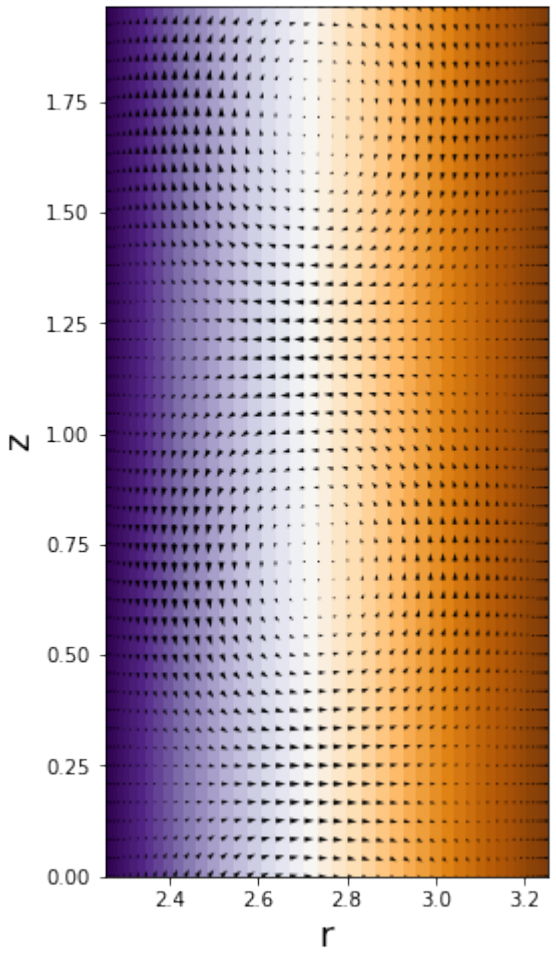
Analysis

First, let's look at our last time snapshot, plotting the background $v\hat{\theta}$ velocity with arrows representing the meridional flow:

[21]:
```
figsize(12,8)
pcolormesh((r[0]*ones_like(z)).T,(z*ones_like(r)).T,v['g'].T,cmap='PuOr')
quiver((r[0]*ones_like(z)).T,(z*ones_like(r)).T,u['g'].T,w['g'].T,width=0.005)
axis('image')
xlabel('r', fontsize=18)
ylabel('z', fontsize=18)
```

```
/Users/kburns/Software/miniconda3/envs/dedalus/lib/python3.6/site-packages/ipykernel_
↪launcher.py:2: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same␣
↪dimensions as C is deprecated since 3.3.  Either specify the corners of the␣
↪quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set␣
↪rcParams['pcolor.shading'].  This will become an error two minor releases later.
```

[21]:
```
Text(0, 0.5, 'z')
```



But we really want some quantitative comparison with the growth rate $\gamma_{analytic}$ from Barenghi (1991). First we construct a small helper function to read our timeseries data, and then we load it out of the self-documented HDF5 file.

[22]:
```
def get_timeseries(data, field):
    data_1d = []
    time = data['scales/sim_time'][:]
    data_out = data['tasks/%s'%field][:,0,0]
    return time, data_out
```

[23]:
```
data = h5py.File("scalar_data/scalar_data_s1/scalar_data_s1_p0.h5", "r")
t, ke = get_timeseries(data, 'total kinetic energy')
t, kem = get_timeseries(data, 'meridional kinetic energy')
t, urms = get_timeseries(data, 'u_rms')
t, wrms = get_timeseries(data, 'w_rms')
```

In order to compare to Barenghi (1991), we scale our results by the Reynolds number, because we have non-dimensionalized slightly differently than he did.

[24]:
```
t_window = (t/period > 2) & (t/period < 14)

gamma_w, log_w0 = np.polyfit(t[t_window], np.log(wrms[t_window]),1)
```

(continues on next page)

(continued from previous page)

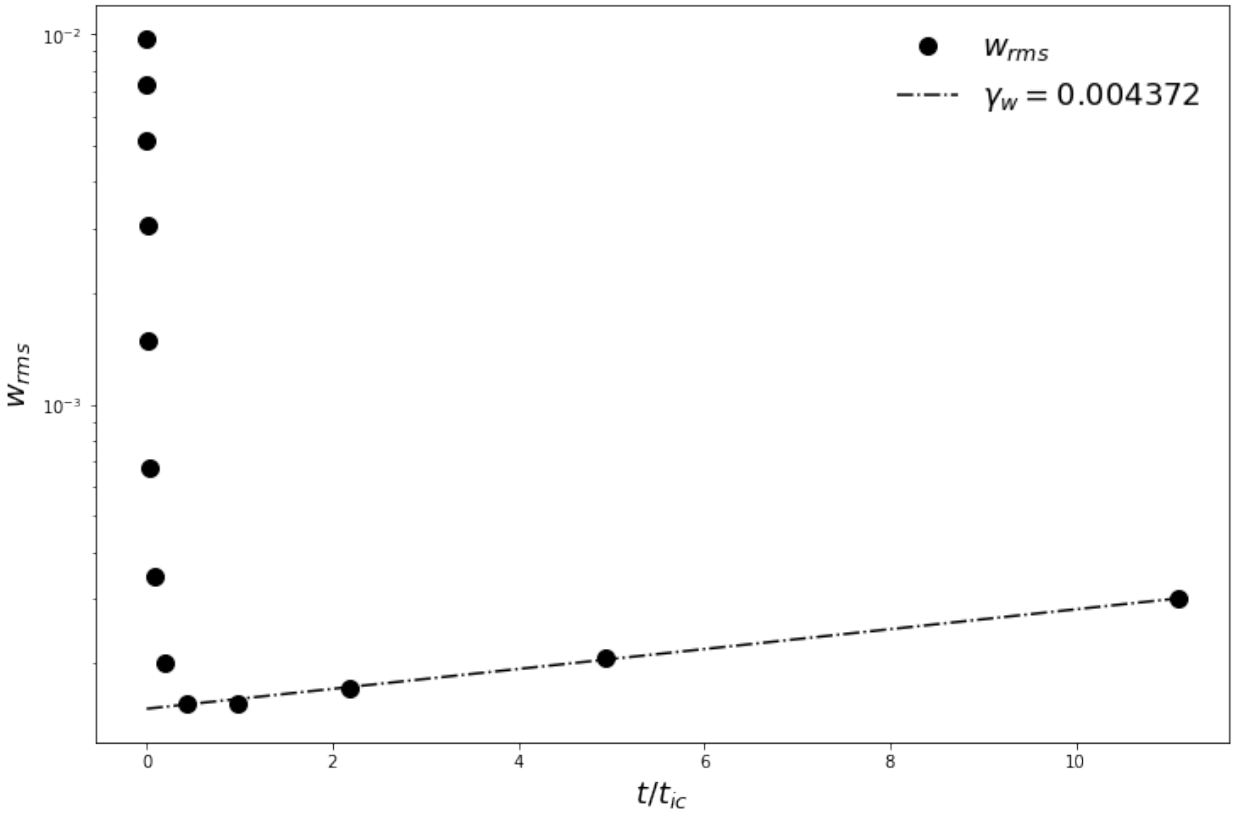
```
gamma_w_scaled = gamma_w*Re
gamma_barenghi = 0.430108693
rel_error_barenghi = (gamma_barenghi - gamma_w_scaled)/gamma_barenghi

print("gamma_w = %10.8f" % gamma_w_scaled)
print("relative error = %10.8e" % rel_error_barenghi)
```

```
gamma_w = 0.34979841
relative error = 1.86720895e-01
```

This looks like a rather high error (order 10% or so), but we know from Barenghi (1991) that the error is dominated by the timestep. Here, we've used a very loose timestep, but if you fix dt (rather than using the CFL calculator), you can get much lower errors at the cost of a longer run.

Finally, we plot the RMS w velocity, and compare it with the fitted exponential solution

[25]:
```
fig = figure()
ax = fig.add_subplot(111)
ax.semilogy(t/period, wrms, 'ko', label=r'$w_{rms}$',ms=10)
ax.semilogy(t/period, np.exp(log_w0)*np.exp(gamma_w*t), 'k-.', label='$\gamma_w = %f$' %␣
↪gamma_w)
ax.legend(loc='upper right',fontsize=18).draw_frame(False)
ax.set_xlabel(r"$t/t_{ic}$",fontsize=18)
ax.set_ylabel(r"$w_{rms}$",fontsize=18)

fig.savefig('growth_rates.png')
```



1.2.3 Example Scripts

A wider range of examples are available under the `examples` subdirectory of the main code repository, which you can browse here. These example scripts can be copied to any working directory with the command `python3 -m dedalus get_examples`. These examples cover a wider range of use cases, including larger multidimensional problems designed for parallel execution. Basic post-processing and plotting scripts are also provided with many problems. These simulation and processing scripts may be useful as a starting point for implementing different problems and equation sets.

1.3 User Guide & How-To's

This is a work-in-progress user guide and how-to section for Dedalus. Contributions and suggestions for additions to this section are very welcome!

General user guide:

1.3.1 Configuring Dedalus

Various aspects of the underlying numerics, logging behavior, and output behavior of Dedalus can be modified through a configuration interface using Python's standard-library *ConfigParser* structure.

A `dedalus.cfg` file with the default configuration and descriptions of each option is included in the package. This file can be copied to any working directory with the command `python3 -m dedalus get_config`. The configuration settings can be modified by, in order of increasing precedence:

1. Modifying the package's `dedalus.cfg` file (not recommended).
2. Creating a user-based config file at `~/.dedalus/dedalus.cfg`.
3. Creating a local config file called `dedalus.cfg` in the directory where you execute a Dedalus script.
4. Modifying configuration values at the top of a Dedalus script like:

```
from dedalus.tools.config import config
config['logging']['stdout_level'] = 'debug'
```

1.3.2 Performance Tips

Stack Configuration

Disable multithreading

Dedalus does not fully implement hybrid parallelism, so the best performance is typically seen when there is one MPI process for each available core. Some underlying stack components (e.g. Numpy or Scipy or the libraries they wrap) may still attempt to use multiple threads behind the scenes, though, and this can substantially degrade performance. We therefore recommend explicitly disabling threading by setting environment variables such as `OMP_NUM_THREADS` to 1 before running Dedalus.

Domain Specification

Resolutions for faster transforms

The transforms for the `Fourier`, `SinCos`, and `Chebyshev` bases are computed using fast Fourier transforms (FFTs). The underlying FFT algorithms are most efficient when the transform sizes are products of small primes. We recommend choosing basis resolutions that are powers of two, or powers of two multiplied by other small factors.

Process meshes for better load balancing

Dedalus uses multidimensional block distributions when decomposing domains in parallel. By default, problems are parallelized over a 1D process mesh of all available MPI processes. Multidimensional parallelization is easily enabled by specifing a mesh shape using the `mesh` keyword when instantiating a `Domain` object. The specified mesh shape should be a tuple with a length one less than the problem dimension.

Ideal load balancing occurs when the size of a distributed dimension is evenly divisible by the corresponding mesh size. We recommend choosing mesh sizes that are powers of two, or powers of two multiplied by other small factors. An "isotropic" mesh with the same number of processes in each mesh dimension, e.g. `(8, 8)`, will theoretically be the most efficient for a given number of processes.

Avoid empty cores

Note that it is possible to end up with empty cores in certain layouts if the mesh shape is chosen improperly or if a mesh is not specified for a 3D problem ran over many cores. For example, consider a problem with global shape `N = (64, 64, 64)` distributed on `P = 256` cores. Keeping the default 1D mesh or choosing a mesh of shape `(128, 2)` or `(2, 128)` will result in many empty cores – a better choice would be a mesh of shape `(16, 16)`.

Problem Formulation

Minimize the number of problem variables

The number of variables used in a problem can have a large impact on the overall simulation performance. We recommend formulating problems to use as few variables as possible, within the constraints posed by Dedalus itself, which require that PDEs are written as first-order systems in terms of temporal and non-Fourier spatial derivatives. Often the `problem.substitutions` interface can be used to simplify the entry of complex equations without introducing new variables into a problem.

Formulate boundary conditions as Dirichlet conditions

Maintaining a high degree of bandedness in the LHS matrices can greatly improve the performance of Dedalus. This requires using Dirichlet rather than integral, Neumann, or other mixed boundary conditions. Note that the first-order formulation required by Dedalus often means that such boundary conditions can be posed as Dirichlet conditions on the first-order variables, e.g. the Neumann condition `dz(u) = 0` for an elliptic or parabolic problem can be written as the Dirichlet condition `uz = 0`, assuming the first-order reduction definition `"uz - dz(u) = 0"` is part of the PDE system.

Avoid non-smooth or rational-function NCCs

High bandedness also requires that non-constant coefficients (NCCs) appearing on the LHS are spectrally smooth. An amplitude cutoff and a limit to the number of terms used when expanding the LHS NCCs can be specified with the `ncc_cutoff` and `max_ncc_terms` keywords when instantiating a problem. These settings can have a large performance impact for problems with NCCs that are not low-degree polynomials. Problems with NCCs that are rational functions (such as `1/r**2` terms that appear in problems in curvilinear coordinates) should usually be multiplied through to clear the polynomial denominators, resulting in purely polynomial and hence well-banded NCC operators.

Timestepping

Avoid changing the simulation timestep unless necessary

Changing the simulation timestep requires refactorizing the LHS matrices, so the timestep should be changed as infrequently as possible for maximum performance. If you are using the built-in CFL tools to calculate the simulation timestep, be sure to specify a `threshold` parameter greater than zero to prevent the timestep from changing due to small variations in the flow.

1.3.3 Troubleshooting

Singular matrix errors

If you come across an error in the linear solver stating that a matrix/factor is singular, that means that the linear LHS portion of the PDE system is not solvable. This error indicates that some degrees of freedom of the solution are unconstrained and some of the specified equations are redundant (these are equivalent since the LHS matrices must be square).

These errors are often due to imposing boundary conditions that are redundant for some set of modes and/or failing to constrain a gauge freedom in the solution. For instance, when simulating incompressibly hydrodynamics in a channel, specifying the mean wall-normal velocity on both walls is redundant with the incompressiblity constraint (the mean wall-normal velocity at one wall must be the same as at the other wall for the flow to be divergence-free). Instead, the wall-normal velocity boundary condition at one wall should be removed and replaced with a gauge condition setting the constant portion of the pressure. This is why you'll see these types of boundary conditions for the wall-normal velocity (e.g. `w`) in Dedalus scripts for incompressible flow:

```
problem.add_bc("left(w) = 0")
problem.add_bc("right(w) = 0", condition="(nx != 0)")
problem.add_bc("right(p) = 0", condition="(nx == 0)")
```

Out of memory errors

Spectral simulations with implicit timestepping can require a large amount of memory to store the LHS matrices and their factorizations. The best way to minimize the required memory is to minimize the LHS matrix size by using as few variables as possible and to minimize the LHS matrix bandwidth (see the *Performance Tips* page). Beyond this, several of the Dedalus configuration options can be changed the minimize the simulation's memory footprint, potentially at the cost of reduced performance (see the *Configuring Dedalus* page).

Reducing memory consumption in Dedalus is an ongoing effort. Any assistance with memory profiling and contributions reducing the code's memory footprint would be greatly appreciated!

Maintaining Hermitian symmetry with real variables

In certain problems with real variables, numerical instabilities may arise due to the loss of Hermitian symmetry in the Fourier components of the solution. When the grid datatype is set to `float` or `np.float64` in Dedalus, that means that the first Fourier transform (say in the x direction) will be computed with a real-to-complex FFT where the negative half of the spectrum (kx < 0) is discarded. Subsequent transforms are then complex-to-complex.

This strategy maintains the proper Hermitian symmetry for the majority of the modes, since only the non-negative kx half of the spectrum is evolved and the negative kx modes are computed from the conjugates of the positive kx modes whenever they are needed (i.e. for interpolation along x). However, modes with kx = 0 should have coefficients that are real after the x-transform, and should have Hermitian symmetry in e.g. ky if there's a subsequent Fourier transform in y. If these conditions are not enforced and the PDE has a linear instability for modes with kx = 0, then the imaginary part of the solution can potentially grow without bound (the nonlinear terms are computed in grid space, so they only affect the real portion of the solution and can't saturate a growing imaginary part).

A simple solution is to periodically transform all the state variables to grid space, which will project away any imaginary component that may have been building up during timestepping. This is done in Dedalus every 100 timesteps by default. This cadence can be modified via the `enforce_real_cadence` keyword when instantiating an IVP solver, and may need to be decreased in simulations with strong linear instabilities.

Specific how-to's:

1.3.4 General Functions

The `GeneralFunction` class enables users to simply define new explicit operators for the right-hand side and analysis tasks of their simulations. Such operators can be used to apply arbitrary user-defined functions to the grid values or coefficients of some set of input fields, or even do things like introduce random data or read data from an external source.

A `GeneralFunction` object is instantiated with a Dedalus domain, a layout object or descriptor (e.g. `'g'` or `'c'` for grid or coefficient space), a function, a list of arguments, and a dictionary of keywords. The resulting object is a Dedalus operator that can be evaluated and composed like other Dedalus operators. It operates by first ensuring that any arguments that are Dedalus field objects are in the specified layout, then calling the function with the specified arguments and keywords, and finally setting the result as the output data in the specified layout.

Here's an example how you can use this class to apply a nonlinear function to the grid data of a single Dedalus field. First, we define the underlying function we want to apply to the field data – say the error function from scipy:

```
from scipy import special

def erf_func(field):
    # Call scipy erf function on the field's data
    return special.erf(field.data)
```

Second, we make a wrapper that returns a `GeneralFunction` instance that applies `erf_func` to a provided field in grid space. This function produces a Dedalus operator, so it's what we want to use on the RHS or in analysis tasks:

```
import dedalus.public as de

def erf_operator(field):
    # Return GeneralFunction instance that applies erf_func in grid space
    return de.operators.GeneralFunction(
        field.domain,
        layout = 'g',
        func = erf_func,
```

(continues on next page)

(continued from previous page)

```
        args = (field,)
    )
```

Finally, we add this wrapper to the parsing namespace to make it available in string-specified equations and analysis tasks:

```
de.operators.parseables['erf'] = erf_operator
```

1.3.5 Parallel Data Interaction

Documentation coming soon.

1.3.6 Output Formatting

Documentation coming soon.

1.3.7 Restarting Simulations

Documentation coming soon.

1.4 Methodology

1.4.1 Dedalus Methods Paper

This is a work-in-progress section on the mathematical and computational methodologies underlying Dedalus. Until this section is completed, please consult the Dedalus methods paper:

> K J Burns, G M Vasil, J S Oishi, D Lecoanet, B P Brown, "Dedalus: A Flexible Framework for Numerical Simulations with Spectral Methods," Physical Review Research, vol. 2, no. 2, p. 838, Apr. 2020. [doi] [arxiv] [bibtex] [examples]

1.5 dedalus

1.5.1 Subpackages

dedalus.core

Submodules

dedalus.core.basis

Abstract and built-in classes for spectral bases.

Module Contents

`logger`

`DEFAULT_LIBRARY`

`FFTW_RIGOR`

`DIRICHLET_PRECONDITIONING`

`class Basis`

Base class for spectral bases.

These classes define methods for transforming, differentiating, and integrating corresponding series represented by their spectral coefficients.

Parameters

- **base_grid_size** (*int*) – Number of grid points
- **interval** (*tuple of floats*) – Spatial domain of basis
- **dealias** (*float, optional*) – Fraction of modes to keep after dealiasing (default: 1.)

Variables

- **grid_dtype** (*dtype*) – Grid data type
- **coeff_size** (*int*) – Number of spectral coefficients
- **coeff_embed** (*int*) – Padded number of spectral coefficients for transform
- **coeff_dtype** (*dtype*) – Coefficient data type

`abstract set_dtype(self, grid_dtype)`

Set transforms based on grid data type.

`abstract forward(self, gdata, cdata, axis, meta, scale)`

Grid-to-coefficient transform.

`abstract backward(self, cdata, gdata, axis, meta, scale)`

Coefficient-to-grid transform.

`abstract differentiate(self, cdata, cderiv, axis)`

Differentiate using coefficients.

`abstract integrate(self, cdata, cint, axis)`

Integrate over interval using coefficients.

`abstract interpolate(self, cdata, cint, position, axis)`

Interpolate in interval using coefficients.

`property library(self)`

`grid_size(self, scale)`

Compute scaled grid size.

`grid_spacing(self, scale=1.0)`

Compute grid spacings.

`grid_array_object(self, domain, axis)`

Grid array object.

`grid_spacing_object(self, domain, axis)`

`check_arrays(self, cdata, gdata, axis, scale)`

Verify provided arrays sizes and dtypes are correct. Build compliant arrays if not provided.

class TransverseBasis
Base class for bases supporting transverse differentiation.

class ImplicitBasis
Base class for bases supporting implicit methods.

These bases define the following matrices encoding the respective linear functions acting on a series represented by its spectral coefficients:

Linear operators (square matrices): Pre : preconditioning (default: identity) Diff : differentiation Mult(p) : multiplication by p-th basis element

Linear functionals (vectors): left_vector : left-endpoint evaluation right_vector : right-endpoint evaluation integ_vector : integration over interval interp_vector : interpolation in interval

abstract Multiply(*self*, *p*, *ncc_basis_meta*, *arg_basis_meta*)
p-element multiplication matrix.

abstract Precondition(*self*)
Preconditioning matrix.

DropTau(*self*, *n_tau*)
Matrix dropping tau+match rows.

DropNonfirst(*self*)
Matrix dropping non-first rows.

DropNonconstant(*self*)
Matrix dropping non-constant rows.

DropMatch(*self*)
Matrix dropping match rows.

PreconditionDropTau(*self*, *n_tau*)
Preconditioning with tau+match filtering.

PreconditionDropMatch(*self*)
Preconditioning with match filtering.

NCC(*self*, *ncc_basis_meta*, *arg_basis_meta*, *coeffs*, *cutoff*, *max_terms*)
Build NCC multiplication matrix.

class Chebyshev(*name*, *base_grid_size*, *interval=(- 1, 1)*, *dealias=1*, *tau_after_pre=True*)
Chebyshev polynomial basis on the roots grid.

element_label = T

separable = False

coupled = True

default_meta(*self*)

grid(*self*, *scale=1.0*)
Build Chebyshev roots grid.

grid_spacing(*self*, *scale=1.0*)
Build Chebyshev roots grid spacing.

set_dtype(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

Integrate(*self*)
Build integration class.

Interpolate(*self*)
Buld interpolation class.

Differentiate(*self*)
Build differentiation class.

Precondition(*self*)
Preconditioning matrix.

T_n = (U_n - U_(n-2)) / 2 U_(-n) = -U_(n-2)

Dirichlet(*self*)
Dirichlet recombination matrix.

D[0] = T[0] D[1] = T[1] D[n] = T[n] - T[n-2]

<T[i]|D[j]> = <T[i]|T[j]> - <T[i]|T[j-2]> = (i,j) - (i,j-2)

Multiply(*self*, *p*, *ncc_basis_meta*, *arg_basis_meta*)
p-element multiplication matrix.

class Legendre(*name*, *base_grid_size*, *interval=(- 1, 1)*, *dealias=1*, *tau_after_pre=True*)
Legendre polynomial basis on the roots grid.

element_label = P

separable = False

coupled = True

default_meta(*self*)

grid(*self*, *scale=1.0*)
Build Legendre polynomial roots grid.

set_dtype(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

Integrate(*self*)
Build integration class.

Interpolate(*self*)
Buld interpolation class.

Differentiate(*self*)
Build differentiation class.

Precondition(*self*)

Preconditioning matrix. 2 * (2*n + 1) * P[n] = (n + 2) * J11[n] - n * J11[n-2]

Dirichlet(*self*)
Dirichlet recombination matrix.

D[0] = P[0] D[1] = P[1] D[n] = P[n] - P[n-2]

<P[i]|D[j]> = <P[i]|T[j]> - <P[i]|T[j-2]> = (i,j) - (i,j-2)

Multiply(*self*, *p*, *ncc_basis_meta*, *arg_basis_meta*)
p-element multiplication matrix.

class Hermite(*name*, *base_grid_size*, *center=0.0*, *stretch=1.0*, *dealias=1*, *tau_after_pre=False*)
Hermite function/polynomial basis.

Interval: The functions live on (-inf, inf) but are centered and scaled by the affine transformation from (-1, 1) to the specified interval.

Hermite functions (with envelope, default): hf[n](xp) = exp(-xn^2/2) H[n](xn) / N[n] N[n]**2 = ^(1/2) 2^n n! integ hf[n] hf[m] dxn = [n,m]

Hermite polynomials (without envelope): hp[n](xp) = H[n](xn) integ hp[n] hp[m] exp(-xn**2) dx = [n,m] N[n]**2

`element_label = h`

`separable = False`

`coupled = True`

`default_meta`(*self*)

`grid`(*self*, *scale=1.0*)
Build Hermite polynomial roots grid.

`grid_array_object`(*self*, *domain*, *axis*)
Grid array object.

`set_dtype`(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

`Integrate`(*self*)
Build integration class.

`Interpolate`(*self*)
Buld interpolation class.

`Differentiate`(*self*)
Build differentiation class.

`Precondition`(*self*)
Preconditioning matrix.

`Dirichlet`(*self*)
Dirichlet recombination matrix.

`Multiply`(*self*, *p*, *ncc_basis_meta*, *arg_basis_meta*)
Hermite multiplication matrix.

`class Laguerre`(*name*, *base_grid_size*, *edge=0.0*, *stretch=1.0*, *dealias=1*, *tau_after_pre=False*)
Laguerre function/polynomial basis.

Interval: The functions live on (0, inf) but are centered and scaled by the affine transformation from (0, 1) to the specified interval.

Laguerre functions (with envelope, default): gn[n](xp) = exp(-x/2) L[n](xn) integ gn[n] gn[m] dxn = [n,m]

Laguerre polynomials (without envelope): gp[n](xp) = L[n](xn) integ gp[n] gp[m] exp(-xn) dx = [n,m]

`element_label = g`

`separable = False`

`coupled = True`

`default_meta`(*self*)

`grid`(*self*, *scale=1.0*)
Build Laguerre polynomial roots grid.

`grid_array_object`(*self*, *domain*, *axis*)
Grid array object.

`set_dtype`(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

`Integrate`(*self*)
Build integration class.

`Interpolate`(*self*)
Buld interpolation class.

`Differentiate`(*self*)
Build differentiation class.

`Precondition`(*self*)

Preconditioning matrix. L[n;1] = sum_{i=0}^{n} L[n;0] L[n;0] = L[n;1] - L[n-1;1]

`Dirichlet`(*self*)
Dirichlet recombination matrix.

G[0] = g[0] G[n] = g[n] - g[n-1]

<g[i]|G[j]> = <g[i]|g[j]> - <g[i]|g[j-1]> = (i,j) - (i,j-1)

`Multiply`(*self*, *p*, *ncc_basis_meta*, *arg_basis_meta*)
Laguerre multiplication matrix.

`class Fourier`(*name*, *base_grid_size*, *interval=(0, 2 * pi)*, *dealias=1*)
Fourier complex exponential basis.

`separable = True`

`coupled = False`

`element_label = k`

`default_meta`(*self*)

`grid`(*self*, *scale=1.0*)
Build evenly spaced Fourier grid.

`grid_spacing`(*self*, *scale=1.0*)
Build Fourier grid spacing.

`set_dtype`(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

`Integrate`(*self*)
Build integration class.

`Interpolate`(*self*)
Build interpolation class.

`Differentiate`(*self*)
Build differentiation class.

`HilbertTransform`(*self*)
Build Hilbert transform class.

`class SinCos`(*name*, *base_grid_size*, *interval=(0, pi)*, *dealias=1*)
Sin/Cos series basis.

`element_label = k`

separable = True

coupled = False

default_meta(*self*)

grid(*self*, *scale=1.0*)
Evenly spaced interior grid: cos(Nx) = 0

grid_spacing(*self*, *scale=1.0*)
Build cosine grid spacing.

grid_array_object(*self*, *domain*, *axis*)
Grid array object.

set_dtype(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

Integrate(*self*)
Build integration class.

Interpolate(*self*)
Build interpolation class.

Differentiate(*self*)
Build differentiation class.

HilbertTransform(*self*)
Build Hilbert transform class.

class Compound(*name*, *subbases*, *dealias=1*)
Compound basis joining adjascent subbases.

separable = False

coupled = True

default_meta(*self*)

property library(*self*)

grid(*self*, *scale=1.0*)
Build compound grid.

grid_spacing(*self*, *scale=1.0*)
Build compound grid spacing.

grid_array_object(*self*, *domain*, *axis*)
Grid array object.

set_dtype(*self*, *grid_dtype*)
Determine coefficient properties from grid dtype.

coeff_start(*self*, *index*)

grid_start(*self*, *index*, *scale*)

sub_gdata(*self*, *gdata*, *index*, *axis*)
Retreive gdata corresponding to one subbasis.

sub_cdata(*self*, *cdata*, *index*, *axis*)
Retrieve cdata corresponding to one subbasis.

forward(*self*, *gdata*, *cdata*, *axis*, *meta*, *scale*)
Forward transforms.

backward(*self*, *cdata*, *gdata*, *axis*, *meta*, *scale*)
Backward transforms.

Integrate(*self*)
Build integration class.

Interpolate(*self*)
Buld interpolation class.

Differentiate(*self*)
Build differentiation class.

Precondition(*self*)
Preconditioning matrix.

Dirichlet(*self*)

Multiply(*self*, *subindex*, *p*, *ncc_basis_meta*, *arg_basis_meta*)
p-element multiplication matrix.

NCC(*self*, *ncc_basis_meta*, *arg_basis_meta*, *coeffs*, *cutoff*, *max_terms*)
Build NCC multiplication matrix.

DropTau(*self*, *n_tau*)
Matrix dropping tau+match rows.

DropNonfirst(*self*)
Matrix dropping non-first rows.

DropNonconstant(*self*)
Matrix dropping non-constant rows.

DropMatch(*self*)
Matrix dropping last row from each subbasis except the last.

PreconditionDropTau(*self*, *n_tau*)
Preconditioning and tau+match filtering.

PreconditionDropMatch(*self*)
Preconditioning and match filtering.

MatchRows(*self*)
Matrix matching subbases.

dedalus.core.distributor

Classes for available data layouts and the paths between them.

Module Contents

logger

GROUP_TRANSFORMS

TRANSPOSE_LIBRARY

GROUP_TRANSPOSES

SYNC_TRANSPOSES

ALLTOALLV

class Distributor(*domain*, *comm=None*, *mesh=None*)

Directs parallelized distribution and transformation of fields over a domain.

Variables

- **comm** (*MPI communicator*) – Global MPI communicator
- **rank** (*int*) – Internal MPI process number
- **size** (*int*) – Number of MPI processes
- **mesh** (*tuple of ints, optional*) – Process mesh for parallelization (default: 1-D mesh of available processes)
- **comm_cart** (*MPI communicator*) – Cartesian MPI communicator over mesh
- **coords** (*array of ints*) – Coordinates in cartesian communicator (None if outside mesh)
- **layouts** (*list of layout objects*) – Available layouts for domain
- **paths** (*list of path objects*) – Transforms and transposes between layouts

Notes

Computations are parallelized by splitting D-dimensional data fields over an R-dimensional mesh of MPI processes, where R < D. In coefficient space, we take the first R dimensions of the data to be distributed over the mesh, leaving the last (D-R) dimensions local. To transform such a data cube to grid space, we loop backwards over the D dimensions, performing each transform if the corresponding dimension is local, and performing an MPI transpose with the next dimension otherwise. This effectively bubbles the first local dimension up from the (D-R)-th to the first dimension, transforming to grid space along the way. In grid space, then, the first dimensional is local, followed by R dimensions distributed over the mesh, and the last (D-R-1) dimensions local.

The distributor object for a given domain constructs layout objects describing each of the (D+R+1) layouts (sets of transform/distribution states) and the paths between them (D transforms and R transposes).

get_layout_object(*self*, *input*)

Dereference layout identifiers.

buffer_size(*self*, *scales*)

Compute necessary buffer size (bytes) for all layouts.

class Layout(*domain*, *mesh*, *coords*, *local*, *grid_space*, *dtype*)

Object describing the data distribution for a given transform and distribution state.

Variables

- **local** (*array of bools*) – Axis locality flags (True/False for local/distributed)
- **grid_space** (*array of bools*) – Axis grid-space flags (True/False for grid/coeff space)
- **dtype** (*numeric type*) – Data type
- **scales.** (*All methods require a tuple of the current transform*) –

global_shape(*self*, *scales*)

Compute global data shape.

blocks(*self*, *scales*)

Compute block sizes for data distribution.

start(*self*, *scales*)

Compute starting coordinates for local data.

local_shape(*self*, *scales*)
Compute local data shape.

slices(*self*, *scales*)
Compute slices for selecting local portion of global data.

buffer_size(*self*, *scales*)
Compute necessary buffer size (bytes).

class Transform(*layout0*, *layout1*, *axis*, *basis*)
Directs transforms between two layouts.

group_data(*self*, *nfields*, *scales*)

increment_group(*self*, *fields*)

decrement_group(*self*, *fields*)

increment_single(*self*, *field*)
Backward transform.

decrement_single(*self*, *field*)
Forward transform.

increment(*self*, *fields*)
Backward transform.

decrement(*self*, *fields*)
Forward transform.

class Transpose(*layout0*, *layout1*, *axis*, *comm_cart*)
Directs transposes between two layouts.

increment(*self*, *fields*)
Transpose from layout0 to layout1.

decrement(*self*, *fields*)
Transpose from layout1 to layout0.

increment_single(*self*, *field*)
Transpose field from layout0 to layout1.

decrement_single(*self*, *field*)
Transpose field from layout1 to layout0.

increment_group(*self*, **fields*)
Transpose group from layout0 to layout1.

decrement_group(*self*, **fields*)
Transpose group from layout1 to layout0.

dedalus.core.domain

Class for problem domain.

Module Contents

`logger`

`class Domain`(*bases*, *grid_dtype=np.complex128*, *comm=None*, *mesh=None*)
Problem domain composed of orthogonal bases.

Parameters

- **bases** (*list of basis objects*) – Bases composing the domain
- **grid_dtype** (*dtype*) – Grid data type
- **mesh** (*tuple of ints, optional*) – Process mesh for parallelization (default: 1-D mesh of available processes)

Variables

- **dim** (*int*) – Dimension of domain, equal to length of bases list
- **distributor** (*distributor object*) – Data distribution controller

`global_grid_shape`(*self*, *scales=None*)

`local_grid_shape`(*self*, *scales=None*)

`get_basis_object`(*self*, *basis_like*)
Return basis from a related object.

`grids`(*self*, *scales=None*)

`grid`(*self*, *axis*, *scales=None*)
Return local grid along one axis.

`all_grids`(*self*, *scales=None*)
Return list of local grids along each axis.

`elements`(*self*, *axis*)
Return local elements along one axis.

`all_elements`(*self*)
Return list of local elements along each axis.

`grid_spacing`(*self*, *axis*, *scales=None*)
Return local grid spacings along one axis.

`all_grid_spacings`(*self*, *scales=None*)
Return list of local grid spacings along each axis.

`new_data`(*self*, *type*, ***kw*)

`new_field`(*self*, ***kw*)

`new_fields`(*self*, *nfields*, ***kw*)

`remedy_scales`(*self*, *scales*)

`class EmptyDomain`(*grid_dtype=np.complex128*)

`get_basis_object`(*self*, *basis_like*)
Return basis from a related object.

`new_data`(*self*, *type*, ***kw*)

`combine_domains`(**domains*)

dedalus.core.evaluator

Class for centralized evaluation of expression trees.

Module Contents

FILEHANDLER_MODE_DEFAULT

FILEHANDLER_PARALLEL_DEFAULT

FILEHANDLER_TOUCH_TMPFILE

logger

class Evaluator(*domain*, *vars*)
Coordinates evaluation of operator trees through various handlers.

Parameters

- **domain** (*domain object*) – Problem domain
- **vars** (*dict*) – Variables for parsing task expression strings

add_dictionary_handler(*self*, ***kw*)
Create a dictionary handler and add to evaluator.

add_system_handler(*self*, ***kw*)
Create a system handler and add to evaluator.

add_file_handler(*self*, *filename*, ***kw*)
Create a file handler and add to evaluator.

add_handler(*self*, *handler*)
Add a handler to evaluator.

evaluate_group(*self*, *group*, ***kw*)
Evaluate all handlers in a group.

evaluate_scheduled(*self*, *wall_time*, *sim_time*, *iteration*, ***kw*)
Evaluate all scheduled handlers.

evaluate_handlers(*self*, *handlers*, *id=None*, ***kw*)
Evaluate a collection of handlers.

require_coeff_space(*self*, *fields*)
Move all fields to coefficient layout.

static get_fields(*tasks*)
Get field set for a collection of tasks.

static attempt_tasks(*tasks*, ***kw*)
Attempt tasks and return the unfinished ones.

class Handler(*domain*, *vars*, *group=None*, *wall_dt=np.inf*, *sim_dt=np.inf*, *iter=np.inf*)
Group of tasks with associated scheduling data.

Parameters

- **domain** (*domain object*) – Problem domain
- **vars** (*dict*) – Variables for parsing task expression strings
- **group** (*str, optional*) – Group name for forcing selected handelrs (default: None)

- **wall_dt** (*float, optional*) – Wall time cadence for evaluating tasks (default: infinite)
- **sim_dt** (*float, optional*) – Simulation time cadence for evaluating tasks (default: infinite)
- **iter** (*int, optional*) – Iteration cadence for evaluating tasks (default: infinite)

add_task(*self*, *task*, *layout='g'*, *name=None*, *scales=None*)
Add task to handler.

add_tasks(*self*, *tasks*, ***kw*)
Add multiple tasks.

add_system(*self*, *system*, ***kw*)
Add fields from a FieldSystem.

class DictionaryHandler(**args*, ***kw*)
Handler that stores outputs in a dictionary.

process(*self*, ***kw*)
Reference fields from dictionary.

class SystemHandler(*domain*, *vars*, *group=None*, *wall_dt=np.inf*, *sim_dt=np.inf*, *iter=np.inf*)
Handler that sets fields in a FieldSystem.

build_system(*self*)
Build FieldSystem and set task outputs.

process(*self*, ***kw*)
Gather fields into system.

class FileHandler(*base_path*, **args*, *max_writes=np.inf*, *max_size=2 ** 30*, *parallel=None*, *mode=None*, ***kw*)
Handler that writes tasks to an HDF5 file.

Parameters

- **base_path** (*str*) – Base path for analyis output folder
- **max_writes** (*int, optional*) – Maximum number of writes per set (default: infinite)
- **max_size** (*int, optional*) – Maximum file size to write to, in bytes (default: 2**30 = 1 GB). (Note: files may be larger after final write.)
- **parallel** (*bool, optional*) – Perform parallel writes from each process to single file (True), or separately write to individual process files (False). Default behavior set by config option.
- **mode** (*str, optional*) – 'overwrite' to delete any present analysis output with the same base path. 'append' to begin with set number incremented past any present analysis output. Default behavior set by config option.

check_file_limits(*self*)
Check if write or size limits have been reached.

get_file(*self*)
Return current HDF5 file, creating if necessary.

property current_path(*self*)

create_current_file(*self*)
Generate new HDF5 file in current_path.

setup_file(*self*, *file*)

process(*self*, *world_time*, *wall_time*, *sim_time*, *timestep*, *iteration*, ***kw*)
Save task outputs to HDF5 file.

get_write_stats(*self*, *layout*, *scales*, *constant*, *index*)
Determine write parameters for nonconstant subspace of a field.

get_hdf5_spaces(*self*, *layout*, *scales*, *constant*, *index*)
Create HDF5 space objects for writing nonconstant subspace of a field.

dedalus.core.field

Class for data fields.

Module Contents

logger

class Operand

diff(*self*, **args*, ***kw*)
Create Differentiation operator with this operand.

integ(*self*, **args*, ***kw*)
Create Integration operator with this operand.

interp(*self*, **args*, ***kw*)
Create Interpolation operator with this operand.

static parse(*string*, *namespace*, *domain*)
Build operand from a string expression.

static cast(*x*, *domain=None*)

static raw_cast(*x*)

class Data

set_scales(*self*, *scales*, *keep_data=True*)
Set new transform scales.

atoms(*self*, **types*, ***kw*)

has(*self*, **atoms*)

expand(*self*, **vars*)
Return self.

canonical_linear_form(*self*, **vars*)
Return self.

split(*self*, **vars*)

replace(*self*, *old*, *new*)
Replace an object in the expression tree.

comp_order(*self*, *ops*, *vars*)

mul_order(*self*, *vars*)

operator_dict(*self*, *index*, *vars*, ***kw*)

sym_diff(*self*, *var*)
Symbolically differentiate with respect to var.

class Scalar(*value=0*, *name=None*, *domain=None*)

class ScalarMeta(*scalar=None*)
Shortcut class to return scalar metadata for any axis.

as_ncc_operator(*self*, *frozen_arg_basis_meta*, *cutoff*, *max_terms*, *cacheid=None*)
Return self.value.

class Array(*domain*, *name=None*)

property scales(*self*)

from_global_vector(*self*, *data*, *axis*)

from_local_vector(*self*, *data*, *axis*)

as_ncc_operator(*self*, *frozen_arg_basis_meta*, *cutoff*, *max_terms*, *cacheid=None*)
Cast to field and convert to NCC operator.

class Field(*domain*, *name=None*, *scales=None*)
Scalar field over a domain.

Parameters

- **domain** (*domain object*) – Problem domain
- **name** (*str, optional*) – Field name (default: Python object id)

Variables

- **layout** (*layout object*) – Current layout of field
- **data** (*ndarray*) – View of internal buffer in current layout

property layout(*self*)

property scales(*self*)

create_buffer(*self*, *scales*)
Create buffer for Field data.

set_scales(*self*, *scales*, *keep_data=True*)
Set new transform scales.

require_layout(*self*, *layout*)
Change to specified layout.

towards_grid_space(*self*)
Change to next layout towards grid space.

towards_coeff_space(*self*)
Change to next layout towards coefficient space.

require_grid_space(*self*, *axis=None*)
Require one axis (default: all axes) to be in grid space.

require_coeff_space(*self*, *axis=None*)
Require one axis (default: all axes) to be in coefficient space.

require_local(*self*, *axis*)
Require an axis to be local.

differentiate(*self*, **args*, ***kw*)
Differentiate field.

integrate(*self*, **args*, ***kw*)
Integrate field.

interpolate(*self*, **args*, ***kw*)
Interpolate field.

antidifferentiate(*self*, *basis*, *bc*, *out=None*)
Antidifferentiate field by setting up a simple linear BVP.

Parameters

- **basis** (*basis-like*) – Basis to antidifferentiate along
- **bc** (*(str, object) tuple*) – Boundary conditions as (functional, value) tuple. *functional* is a string, e.g. "left", "right", "int" *value* is a field or scalar
- **out** (*field, optional*) – Output field

copy(*self*)

static cast(*input*, *domain*)

as_ncc_operator(*self*, *frozen_arg_basis_meta*, *cutoff*, *max_terms*, *cacheid=None*)
Convert to operator form representing multiplication as a NCC.

dedalus.core.future

Classes for future evaluation.

Module Contents

logger

PREALLOCATE_OUTPUTS

class Future(**args*, *domain=None*, *out=None*)
Base class for deferred operations on data.

Parameters

- ***args** (*Operands*) – Operands. Number must match class attribute *arity*, if present.
- **out** (*data, optional*) – Output data object. If not specified, a new object will be used.

Notes

Operators are stacked (i.e. provided as arguments to other operators) to construct trees that represent compound expressions. Nodes are evaluated by first recursively evaluating their subtrees, and then calling the *operate* method.

arity

store_last = False

reset(*self*)
Restore original arguments.

atoms(*self*, **types*, *include_out=False*)

has(*self*, **atoms*)

replace(*self*, *old*, *new*)
Replace an object in the expression tree.

evaluate(*self*, *id=None*, *force=True*)
Recursively evaluate operation.

attempt(*self*, *id=None*)
Recursively attempt to evaluate operation.

meta(*self*)

meta_dirichlet(*self*, *axis*)

abstract meta_constant(*self*, *axis*)

abstract meta_parity(*self*, *axis*)

abstract meta_envelope(*self*, *axis*)

abstract check_conditions(*self*)
Check that all argument fields are in proper layouts.

abstract operate(*self*, *out*)
Perform operation.

as_ncc_operator(*self*, *frozen_arg_basis_meta*, *cutoff*, *max_terms*, *cacheid=None*)

class FutureScalar(**args*, *domain=None*, *out=None*)
Class for deferred operations producing a Scalar.

future_type

meta

class FutureArray(**args*, *domain=None*, *out=None*)
Class for deferred operations producing an Array.

future_type

class FutureField(**args*, *domain=None*, *out=None*)
Class for deferred operations producing a Field.

future_type

static parse(*string*, *namespace*, *domain*)
Build FutureField from a string expression.

static cast(*input*, *domain*)
Cast an object to a FutureField.

unique_domain(**args*)
Return unique domain from a set of fields.

dedalus.core.metadata

Module Contents

class AliasDict(**args*, ***kw*)
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

class MultiDict(*args, **kw)

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

class DictGroup(*dicts)

class Metadata(*domain*)

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

freeze_dict(*dict*)

Freeze dict as tuple of items.

freeze_meta(*meta*)

Freeze metadata as tuple of frozen dicts.

dedalus.core.operators

Abstract and built-in classes defining deferred operations on fields.

Module Contents

parseables

parseable(*op*)

addname(*name*)

is_integer(*x*)

class Operator

property base(*self*)

comp_order(*self*, *ops*, *vars*)

mul_order(*self*, *vars*)

class FieldCopy(*arg*, ***kw*)
Operator making a new field copy of data.

name = FieldCopy

check_conditions(*self*)

meta_constant(*self*, *axis*)

meta_parity(*self*, *axis*)

meta_envelope(*self*, *axis*)

property base(*self*)

sym_diff(*self*, *var*)
Symbolically differentiate with respect to var.

split(*self*, **vars*)

class FieldCopyScalar(*arg*, ***kw*)
Operator making a new field copy of data.

argtypes

operate(*self*, *out*)

class FieldCopyArray(*arg*, ***kw*)
Operator making a new field copy of data.

argtypes

operate(*self*, *out*)

class FieldCopyField(*arg*, ***kw*)
Operator making a new field copy of data.

argtypes

operate(*self*, *out*)

class NonlinearOperator

expand(*self*, **vars*)
Return self.

canonical_linear_form(*self*, **vars*)
Raise if arguments contain specified variables (default: None)

split(*self*, **vars*)

class GeneralFunction(*domain*, *layout*, *func*, *args=[]*, *kw={}*, *out=None*)
Operator wrapping a general python function.

Parameters

- **domain** (*domain object*) – Domain
- **layout** (*layout object or identifier*) – Layout of function output
- **func** (*function*) – Function producing field data

- **args** (*list*) – Arguments to pass to func
- **kw** (*dict*) – Keywords to pass to func
- **out** (*field, optional*) – Output field (default: new field)

Notes

On evaluation, this wrapper evaluates the provided function with the given arguments and keywords, and takes the output to be data in the specified layout, i.e.

    out[layout] = func(*args, **kw)

meta_constant(*self*, *axis*)

check_conditions(*self*)

operate(*self*, *out*)

class UnaryGridFunction(*func*, *arg*, ***kw*)

arity = 1

supported

aliased

property base(*self*)

meta_constant(*self*, *axis*)

meta_parity(*self*, *axis*)

meta_envelope(*self*, *axis*)

sym_diff(*self*, *var*)
Symbolically differentiate with respect to var.

class UnaryGridFunctionScalar(*func*, *arg*, ***kw*)

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class UnaryGridFunctionArray(*func*, *arg*, ***kw*)

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class UnaryGridFunctionField(*func*, *arg*, ***kw*)

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class Arithmetic

arity = 2

comp_order(*self*, *ops*, *vars*)

class Add

name = Add

str_op = +

property base(*self*)

meta_constant(*self*, *axis*)

meta_parity(*self*, *axis*)

meta_envelope(*self*, *axis*)

expand(*self*, **vars*)
Expand arguments containing specified variables (default: all).

canonical_linear_form(*self*, **vars*)
Ensure arguments have same dependency on specified variables.

split(*self*, **vars*)

operator_dict(*self*, *index*, *vars*, ***kw*)
Produce matrix-operator dictionary over specified variables.

sym_diff(*self*, *var*)
Symbolically differentiate with respect to var.

mul_order(*self*, *vars*)

class AddScalarScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddArrayArray

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddFieldField

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddScalarArray

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddArrayScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddScalarField

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddFieldScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddArrayField

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class AddFieldArray

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class Multiply

name = Mul

str_op = *

property base(*self*)

meta_constant(*self*, *axis*)

meta_parity(*self*, *axis*)

meta_envelope(*self*, *axis*)

expand(*self*, **vars*)

Distribute over sums containing specified variables (default: all).

canonical_linear_form(*self*, **vars*)
Eliminate nonlinear multiplications and float specified variables right.

split(*self*, **vars*)

operator_dict(*self*, *index*, *vars*, ***kw*)
Produce matrix-operator dictionary over specified variables.

sym_diff(*self*, *var*)
Symbolically differentiate with respect to var.

mul_order(*self*, *vars*)

class MultiplyScalarScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyArrayArray

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyFieldField

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyScalarArray

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyArrayScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyScalarField

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyFieldScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyArrayField

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class MultiplyFieldArray

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class Power

name = Pow

str_op = **

property base(*self*)

mul_order(*self*, *vars*)

class PowerDataScalar

argtypes

meta_constant(*self*, *axis*)

meta_parity(*self*, *axis*)

meta_envelope(*self*, *axis*)

sym_diff(*self*, *var*)

Symbolically differentiate with respect to var.

class PowerScalarScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class PowerArrayScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class PowerFieldScalar

argtypes

check_conditions(*self*)

operate(*self*, *out*)

class LinearOperator

kw

expand(*self*, **vars*)
Distribute over sums containing specified variables (default: all).

canonical_linear_form(*self*, **vars*)
Change argument to canonical linear form.

split(*self*, **vars*)

operator_dict(*self*, *index*, *vars*, ***kw*)
Produce matrix-operator dictionary over specified variables.

abstract operator_form(*self*, *index*)

sym_diff(*self*, *var*)
Symbolically differentiate with respect to var.

class TimeDerivative

name = dt

property base(*self*)

meta_constant(*self*, *axis*)

meta_parity(*self*, *axis*)

meta_envelope(*self*, *axis*)

operator_form(*self*, *index*)

operate(*self*, *out*)

class LinearBasisOperator

meta(*self*)

class Separable

operator_form(*self*, *index*)

check_conditions(*self*)

operate(*self*, *out*)

apply_vector_form(*self*, *out*)

abstract explicit_form(*self*, *input*, *output*, *axis*)

abstract vector_form(*self*)

class Coupled

operator_form(*self*, *index*)

check_conditions(*self*)

operate(*self*, *out*)

apply_matrix_form(*self*, *out*)

abstract explicit_form(*self*, *input*, *output*, *axis*)

abstract matrix_form(*self*)

operator_dict(*self*, *index*, *vars*, ***kw*)
Produce matrix-operator dictionary over specified variables.

class Integrate(*arg0*, ***kw*)

name = integ

meta_constant(*self*, *axis*)

integrate(*arg0*, **bases*, *out=None*)

class Interpolate(*arg0*, *position*, *out=None*)

name = interp

distribute(*self*)

meta_constant(*self*, *axis*)

interpolate(*arg0*, *out=None*, ***basis_kw*)

left(*arg0*, *out=None*)
Shortcut for left interpolation along last axis.

right(*arg0*, *out=None*)
Shortcut for right interpolation along last axis.

class Differentiate(*arg0*, ***kw*)

name = d

meta_constant(*self*, *axis*)

expand(*self*, **vars*)
Distribute over sums and apply the product rule to arguments containing specified variables (default: all).

differentiate(*arg0*, **bases*, *out=None*, ***basis_kw*)

class HilbertTransform(*arg0*, ***kw*)

name = Hilbert

meta_constant(*self*, *axis*)

hilberttransform(*arg0*, **bases*, *out=None*, ***basis_kw*)

dedalus.core.pencil

Classes for manipulating pencils.

Module Contents

logger

build_pencils(*domain*)
Create the set of pencils over a domain.

Parameters **domain** (*domain object*) – Problem domain

Returns **pencils** – Pencil objects

Return type list

build_matrices(*pencils*, *problem*, *matrices*)
Build pencil matrices.

class Pencil(*domain*, *local_index*, *global_index*)
Object holding problem matrices for a given transverse wavevector.

Parameters **index** (*tuple of ints*) – Transverse indeces for retrieving pencil from system data

fast_bmat(*blocks*)
Build sparse matrix from sparse COO blocks.

sparse_perm(*perm*, *M*)
Build sparse permutation matrix from permutation vector.

simple_reorder(*N0*, *N1*)

left_permutation(*zbasis*, *n_vars*, *eqs*, *bc_top*, *interleave_subbases*)
Left permutation acting on equations. bc_top determines if constant equations are placed at the top or bottom of the matrix.

Input ordering: Equations > Subbases > Modes

Output ordering with interleave_subbases=True: Modes > Subbases > Equations

Output ordering with interleave_subbases=False: Subbases > Modes > Equations

right_permutation(*zbasis*, *problem*, *interleave_subbases*)
Right permutation acting on variables.

Input ordering: Variables > Subbases > Modes

Output ordering with interleave_subbases=True: Modes > Subbases > Variables

Output ordering with interleave_subbases=False: Subbases > Modes > Variables

dedalus.core.problems

Classes for representing systems of equations.

Module Contents

BC_TOP

INTERLEAVE_SUBBASES

STORE_EXPANDED_MATRICES

logger

class Namespace

Class ensuring a conflict-free namespace for parsing.

Initialize self. See help(type(self)) for accurate signature.

copy(*self*)

Copy entire namespace.

add_substitutions(*self*, *substitutions*)

Parse substitutions in current namespace before adding to self.

class ProblemBase(*domain*, *variables*, *ncc_cutoff=1e-06*, *max_ncc_terms=None*, *entry_cutoff=1e-12*)

Base class for problems consisting of a system of PDEs, constraints, and boundary conditions.

Parameters

- **domain** (*domain object*) – Problem domain
- **variables** (*list of str*) – List of variable names, e.g. [‘u’, ‘v’, ‘w’]
- **ncc_cutoff** (*float, optional*) – Mode amplitude cutoff for LHS NCC expansions (default: 1e-6)
- **max_ncc_terms** (*int, optional*) – Maximum terms to include in LHS NCC expansions (default: None (no limit))
- **entry_cutoff** (*float, optional*) – Matrix entry cutoff to avoid fill-in from cancellation errors (default: 1e-12)

Variables

- **parameters** (*OrderedDict*) – External parameters used in the equations, and held constant during integration.
- **substitutions** (*OrderedDict*) – String-substitutions to be used in parsing.

Notes

Equations are entered as strings of the form “LHS = RHS”, where the left-hand side contains terms that are linear in the dependent variables (and will be parsed into a sparse matrix system), and the right-hand side contains terms that are non-linear (and will be parsed into operator trees).

The specified axes (from domain), variables, parameters, and substitutions are recognized by the parser, along with the built-in operators, which include spatial derivatives (of the form “dx()” for an axis named “x”) and basic mathematical operators (trigonometric and logarithmic).

The LHS terms must be linear in the specified variables and first-order in coupled derivatives.

property nvars_const(*self*)

property nvars_nonconst(*self*)

add_equation(*self*, *equation*, *condition='True'*, *tau=None*)
Add equation to problem.

add_bc(*self*, **args*, ***kw*)
Add boundary condition to problem.

namespace(*self*)
Build namespace for problem parsing.

build_solver(*self*, **args*, ***kw*)
Build corresponding solver class.

class InitialValueProblem(*domain*, *variables*, *time='t'*, ***kw*)
Class for non-linear initial value problems.

Parameters

- **domain** (*domain object*) – Problem domain
- **variables** (*list of str*) – List of variable names, e.g. [‘u’, ‘v’, ‘w’]
- **time** (*str, optional*) – Time label, default: ‘t’

Notes

This class supports non-linear initial value problems. The LHS terms must be linear in the specified variables, first-order in coupled derivatives, first-order in time derivatives, and contain no explicit time dependence.

M.dt(X) + L.X = F(X, t)

solver_class

namespace_additions(*self*)
Build namespace for problem parsing.

class LinearBoundaryValueProblem(*domain*, *variables*, *ncc_cutoff=1e-06*, *max_ncc_terms=None*, *entry_cutoff=1e-12*)
Class for inhomogeneous, linear boundary value problems.

Parameters

- **domain** (*domain object*) – Problem domain
- **variables** (*list of str*) – List of variable names, e.g. [‘u’, ‘v’, ‘w’]

Notes

This class supports inhomogeneous, linear boundary value problems. The LHS terms must be linear in the specified variables and first-order in coupled derivatives, and the RHS must be independent of the specified variables.

L.X = F

solver_class

namespace_additions(*self*)

class NonlinearBoundaryValueProblem(*domain*, *variables*, *ncc_cutoff=1e-06*, *max_ncc_terms=None*, *entry_cutoff=1e-12*)

Class for nonlinear boundary value problems.

Parameters

- **domain** (*domain object*) – Problem domain
- **variables** (*list of str*) – List of variable names, e.g. [‘u’, ‘v’, ‘w’]

Notes

This class supports nonlinear boundary value problems. The LHS terms must be linear in the specified variables and first-order in coupled derivatives.

L.X = F(X)

The problem is reduced into a linear BVP for an update to the solution using the Newton-Kantorovich method and symbolically-computed Frechet derivatives of the RHS.

L.(X0 + X1) = F(X0) + dF(X0).X1 L.X1 - dF(X0).X1 = F(X0) - L.X0

solver_class

namespace_additions(*self*)

Build namespace for problem parsing.

class EigenvalueProblem(*domain*, *variables*, *eigenvalue*, *tolerance=1e-10*, ***kw*)

Class for linear eigenvalue problems.

Parameters

- **domain** (*domain object*) – Problem domain
- **variables** (*list of str*) – List of variable names, e.g. [‘u’, ‘v’, ‘w’]
- **eigenvalue** (*str*) – Eigenvalue label, e.g. ‘sigma’ WARNING: ‘lambda’ is a python reserved word. You *cannot* use it as your eigenvalue. Also, note that unicode symbols don’t work on all machines.
- **tolerance** (*float*) – A floating point number (>= 0) which helps ‘define’ zero for the RHS of the equation. If the RHS has nonzero NCCs which add to zero, dedalus will check to make sure that the max of the expression on the RHS normalized by the max of all NCCs going in that expression is smaller than this tolerance (see ProblemBase._check_if_zero())

Notes

This class supports linear eigenvalue problems. The LHS terms must be linear in the specified variables, first-order in coupled derivatives, and linear or independent of the specified eigenvalue. The RHS must be zero.

M.X + L.X = 0

solver_class

namespace_additions(*self*)

Build namespace for problem parsing.

IVP

LBVP

NLBVP

EVP

dedalus.core.solvers

Classes for solving differential equations.

Module Contents

logger

class EigenvalueSolver(*problem*, *matsolver=None*)

Solves linear eigenvalue problems for oscillation frequency omega, (d_t -> -i omega) for a given pencil, and stores the eigenvalues and eigenvectors. The set_state method can be used to set solver.state to the specified eigenmode.

Parameters **problem** (*problem object*) – Problem describing system of differential equations and constraints

Variables

- **state** (*system object*) – System containing solution fields (after solve method is called)
- **eigenvalues** (*numpy array*) – Contains a list of eigenvalues omega
- **eigenvectors** (*numpy array*) – Contains a list of eigenvectors. The eigenvector corresponding to the ith eigenvalue is in eigenvectors[:,i]
- **eigenvalue_pencil** (*pencil*) – The pencil for which the eigenvalue problem has been solved.

solve_dense(*self*, *pencil*, *rebuild_coeffs=False*, ***kw*)

Solve EVP for selected pencil.

Parameters

- **pencil** (*pencil object*) – Pencil for which to solve the EVP
- **rebuild_coeffs** (*bool, optional*) – Flag to rebuild cached coefficient matrices (default: False)
- **Other keyword options passed to scipy.linalg.eig.**

solve_sparse(*self*, *pencil*, *N*, *target*, *rebuild_coeffs=False*, ***kw*)

Perform targeted sparse eigenvalue search for selected pencil.

Parameters

- **pencil** (*pencil object*) – Pencil for which to solve the EVP
- **N** (*int*) – Number of eigenmodes to solver for. Note: the dense method may be more efficient for finding large numbers of eigenmodes.
- **target** (*complex*) – Target eigenvalue for search.
- **rebuild_coeffs** (*bool, optional*) – Flag to rebuild cached coefficient matrices (default: False)
- **Other keyword options passed to scipy.sparse.linalg.eigs.**

set_state(*self*, *index*)

Set state vector to the specified eigenmode.

Parameters **index** (*int*) – Index of desired eigenmode

`solve`(*self*, **args*, ***kw*)
Deprecated. Use solve_dense instead.

`class LinearBoundaryValueSolver`(*problem*, *matsolver=None*)
Linear boundary value problem solver.

Parameters

- **problem** (*problem object*) – Problem describing system of differential equations and constraints.
- **matsolver** (*matsolver class or name, optional*) – Matrix solver routine (default set by config file).

Variables `state` (*system object*) – System containing solution fields (after solve method is called).

`solve`(*self*, *rebuild_coeffs=False*)
Solve BVP.

`class NonlinearBoundaryValueSolver`(*problem*, *matsolver=None*)
Nonlinear boundary value problem solver.

Parameters

- **problem** (*problem object*) – Problem describing system of differential equations and constraints
- **matsolver** (*matsolver class or name, optional*) – Matrix solver routine (default set by config file).

Variables `state` (*system object*) – System containing solution fields (after solve method is called)

`newton_iteration`(*self*, *damping=1*)
Update solution with a Newton iteration.

`class InitialValueSolver`(*problem*, *timestepper*, *matsolver=None*, *enforce_real_cadence=100*)
Initial value problem solver.

Parameters

- **problem** (*problem object*) – Problem describing system of differential equations and constraints
- **timestepper** (*timestepper class or name*) – Timestepper to use in evolving initial conditions
- **matsolver** (*matsolver class or name, optional*) – Matrix solver routine (default set by config file).
- **enforce_real_cadence** (*int, optional*) – Iteration cadence for enforcing Hermitian symmetry on real variables (default: 100).

Variables

- `state` (*system object*) – System containing current solution fields
- `dt` (*float*) – Timestep
- `stop_sim_time` (*float*) – Simulation stop time, in simulation units
- `stop_wall_time` (*float*) – Wall stop time, in seconds from instantiation
- `stop_iteration` (*int*) – Stop iteration

- **time** (*float*) – Current simulation time
- **iteration** (*int*) – Current iteration

property sim_time(*self*)

get_world_time(*self*)

load_state(*self*, *path*, *index=-1*)
Load state from HDF5 file.

Parameters

- **path** (*str or pathlib.Path*) – Path to Dedalus HDF5 savefile
- **index** (*int, optional*) – Local write index (within file) to load (default: -1)

Returns

- **write** (*int*) – Global write number of loaded write
- **dt** (*float*) – Timestep at loaded write

property proceed(*self*)
Check that current time and iteration pass stop conditions.

property ok(*self*)
Deprecated. Use 'solver.proceed'.

sim_dt_cadences(*self*)
Build array of finite handler sim_dt cadences.

step(*self*, *dt*, *trim=False*)
Advance system by one iteration/timestep.

evolve(*self*, *timestep_function*)
Advance system until stopping criterion is reached.

evaluate_handlers_now(*self*, *dt*, *handlers=None*)
Evaluate all handlers right now. Useful for writing final outputs.

by default, all handlers are evaluated; if a list is given only those will be evaluated.

dedalus.core.system

Classes for systems of coefficients/fields.

Module Contents

class CoeffSystem(*pencil_length*, *domain*)
Representation of a collection of fields that don't need to be transformed, and are therefore stored as a contigous set of coefficient data, joined along the last axis, for efficient pencil and group manipulation.

Parameters

- **pencil_length** (*int*) – Number of coefficients in a single pencil
- **domain** (*domain object*) – Problem domain

Variables **data** (*ndarray*) – Contiguous buffer for field coefficients

get_pencil(*self*, *pencil*)
Return pencil view from system buffer.

set_pencil(*self*, *pencil*, *data*)
Set pencil data in system buffer.

class FieldSystem(*fields*)
Collection of fields alongside a CoeffSystem buffer for efficient pencil and group manipulation.

Parameters **fields** (*list of field objets*) – Fields to join into system

Variables

- **data** (*ndarray*) – Contiguous buffer for field coefficients
- **fields** (*list*) – Field objects
- **nfields** (*int*) – Number of fields in system
- **field_dict** (*dict*) – Dictionary of fields
- **slices** (*dict*) – Dictionary of last-axis slice objects connecting field and system data

gather(*self*)
Copy fields into system buffer.

scatter(*self*)
Extract fields from system buffer.

dedalus.core.timesteppers

ODE solvers for timestepping.

Module Contents

schemes

add_scheme(*scheme*)

class MultistepIMEX(*pencil_length*, *domain*)
Base class for implicit-explicit multistep methods.

Parameters

- **pencil_length** (*int*) – Number of coefficients in a single pencil
- **domain** (*domain object*) – Problem domain

Notes

These timesteppers discretize the system M.dt(X) + L.X = F

into the general form aj M.X(n-j) + bj L.X(n-j) = cj F(n-j)

where j runs from {0, 0, 1} to {amax, bmax, cmax}.

The system is then solved as (a0 M + b0 L).X(n) = cj F(n-j) - aj M.X(n-j) - bj L.X(n-j)

where j runs from {1, 1, 1} to {cmax, amax, bmax}.

References

D. Wang and S. J. Ruuth, Journal of Computational Mathematics 26, (2008).*

- **Our coefficients are related to those used by Wang as:** amax = bmax = cmax = s aj = (s-j) / k(n+s-1) bj = (s-j) cj = (s-j)

step(*self*, *solver*, *dt*)
Advance solver by one timestep.

class CNAB1(*pencil_length*, *domain*)
1st-order Crank-Nicolson Adams-Bashforth scheme [Wang 2008 eqn 2.5.3]

Implicit: 2nd-order Crank-Nicolson Explicit: 1st-order Adams-Bashforth (forward Euler)

amax = 1

bmax = 1

cmax = 1

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class SBDF1(*pencil_length*, *domain*)
1st-order semi-implicit BDF scheme [Wang 2008 eqn 2.6]

Implicit: 1st-order BDF (backward Euler) Explicit: 1st-order extrapolation (forward Euler)

amax = 1

bmax = 1

cmax = 1

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class CNAB2(*pencil_length*, *domain*)
2nd-order Crank-Nicolson Adams-Bashforth scheme [Wang 2008 eqn 2.9]

Implicit: 2nd-order Crank-Nicolson Explicit: 2nd-order Adams-Bashforth

amax = 2

bmax = 2

cmax = 2

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class MCNAB2(*pencil_length*, *domain*)
2nd-order modified Crank-Nicolson Adams-Bashforth scheme [Wang 2008 eqn 2.10]

Implicit: 2nd-order modified Crank-Nicolson Explicit: 2nd-order Adams-Bashforth

amax = 2

bmax = 2

cmax = 2

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class SBDF2(*pencil_length*, *domain*)
2nd-order semi-implicit BDF scheme [Wang 2008 eqn 2.8]

Implicit: 2nd-order BDF Explicit: 2nd-order extrapolation

amax = 2

bmax = 2

cmax = 2

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class CNLF2(*pencil_length*, *domain*)
2nd-order Crank-Nicolson leap-frog scheme [Wang 2008 eqn 2.11]

Implicit: ?-order wide Crank-Nicolson Explicit: 2nd-order leap-frog

amax = 2

bmax = 2

cmax = 2

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class SBDF3(*pencil_length*, *domain*)
3rd-order semi-implicit BDF scheme [Wang 2008 eqn 2.14]

Implicit: 3rd-order BDF Explicit: 3rd-order extrapolation

amax = 3

bmax = 3

cmax = 3

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class SBDF4(*pencil_length*, *domain*)
4th-order semi-implicit BDF scheme [Wang 2008 eqn 2.15]

Implicit: 4th-order BDF Explicit: 4th-order extrapolation

amax = 4

bmax = 4

cmax = 4

classmethod compute_coefficients(*self*, *timesteps*, *iteration*)

class RungeKuttaIMEX(*pencil_length*, *domain*)
Base class for implicit-explicit multistep methods.

Parameters

- **pencil_length** (*int*) – Number of fields in problem
- **domain** (*domain object*) – Problem domain

Notes

These timesteppers discretize the system M.dt(X) + L.X = F

by constructing s stages M.X(n,i) - M.X(n,0) + k Hij L.X(n,j) = k Aij F(n,j)

where j runs from {0, 0} to {i, i-1}, and F(n,i) is evaluated at time t(n,i) = t(n,0) + k ci

The s stages are solved as (M + k Hii L).X(n,i) = M.X(n,0) + k Aij F(n,j) - k Hij L.X(n,j)

where j runs from {0, 0} to {i-1, i-1}.

The final stage is used as the advanced solution*: X(n+1,0) = X(n,s) t(n+1,0) = t(n,s) = t(n,0) + k

- **Equivalently the Butcher tableaus must follow** b_im = H[s, :] b_ex = A[s, :] c[s] = 1

References

U. M. Ascher, S. J. Ruuth, and R. J. Spiteri, Applied Numerical Mathematics (1997).

step(*self*, *solver*, *dt*)
Advance solver by one timestep.

class RK111(*pencil_length*, *domain*)
1st-order 1-stage DIRK+ERK scheme [Ascher 1997 sec 2.1]

stages = 1

c

A

H

class RK222(*pencil_length*, *domain*)
2nd-order 2-stage DIRK+ERK scheme [Ascher 1997 sec 2.6]

stages = 2

c

A

H

class RK443(*pencil_length*, *domain*)
3rd-order 4-stage DIRK+ERK scheme [Ascher 1997 sec 2.8]

stages = 4

c

A

H

class RKSMR(*pencil_length*, *domain*)
(3-)-order 3rd-stage DIRK+ERK scheme [Spalart 1991 Appendix]

stages = 3

c

A

H

dedalus.extras

Submodules

dedalus.extras.atmospheres

Module Contents

class DedalusAtmosphere(*z_basis*, *num_coeffs=20*)

truncate_atmosphere(*self*, *key*)

get_coeffs(*self*, *key*)

get_values(*self*, *key*)

check_spectrum(*self*, *key*, *individual_plots=True*)

check_atmosphere(*self*, *key*, *individual_plots=True*)

class Polytrope(*gamma*, *polytropic_index*, *z0*, *z*, ***args*)

grad_ln_rho(*self*)

grad_S(*self*)

grad_ln_T(*self*)

rho(*self*)

T(*self*)

P(*self*)

S(*self*)

class ScaledPolytrope(*gamma*, *polytropic_index*, *z0*, *z*, ***args*)

grad_ln_rho(*self*)

grad_S(*self*)

grad_ln_T(*self*)

rho(*self*)

T(*self*)

P(*self*)

S(*self*)

Lz = 10

dedalus.extras.flow_tools

Extra tools that are useful in hydrodynamical problems.

Module Contents

logger

class GlobalArrayReducer(*comm*, *dtype=np.float64*)
Directs parallelized reduction of distributed array data.

Parameters

- **comm** (*MPI communicator*) – MPI communicator
- **dtype** (*data type, optional*) – Array data type (default: np.float64)

reduce_scalar(*self*, *local_scalar*, *mpi_reduce_op*)
Compute global reduction of a scalar from each process.

global_min(*self*, *data*, *empty=np.inf*)
Compute global min of all array data.

global_max(*self*, *data*, *empty=- np.inf*)
Compute global max of all array data.

global_mean(*self*, *data*)
Compute global mean of all array data.

class GlobalFlowProperty(*solver*, *cadence=1*)
Directs parallelized determination of a global flow property on the grid.

Parameters

- **solver** (*solver object*) – Problem solver
- **cadence** (*int, optional*) – Iteration cadence for property evaluation (default: 1)

Examples

```
>>> flow = GlobalFlowProperty(solver)
>>> flow.add_property('sqrt(u*u + w*w) * Lz / nu', name='Re')
...
>>> flow.max('Re')
1024.5
```

add_property(*self*, *property*, *name*, *precompute_integral=False*)
Add a property.

min(*self*, *name*)
Compute global min of a property on the grid.

max(*self*, *name*)
Compute global max of a property on the grid.

grid_average(*self*, *name*)
Compute global mean of a property on the grid.

volume_average(*self*, *name*)
Compute volume average of a property.

class CFL(*solver*, *initial_dt*, *cadence=1*, *safety=1.0*, *max_dt=np.inf*, *min_dt=0.0*, *max_change=np.inf*, *min_change=0.0*, *threshold=0.0*)

Computes CFL-limited timestep from a set of frequencies/velocities.

Parameters

- **solver** (*solver object*) – Problem solver
- **initial_dt** (*float*) – Initial timestep
- **cadence** (*int, optional*) – Iteration cadence for computing new timestep (default: 1)
- **safety** (*float, optional*) – Safety factor for scaling computed timestep (default: 1.)
- **max_dt** (*float, optional*) – Maximum allowable timestep (default: inf)
- **min_dt** (*float, optional*) – Minimum allowable timestep (default: 0.)
- **max_change** (*float, optional*) – Maximum fractional change between timesteps (default: inf)
- **min_change** (*float, optional*) – Minimum fractional change between timesteps (default: 0.)
- **threshold** (*float, optional*) – Fractional change threshold for changing timestep (default: 0.)

Notes

The new timestep is computed by summing across the provided frequencies for each grid point, and then reciprocating the maximum "total" frequency from the entire grid.

compute_dt(*self*)
Compute CFL-limited timestep.

add_frequency(*self*, *freq*)
Add an on-grid frequency.

add_velocity(*self*, *velocity*, *axis*)
Add grid-crossing frequency from a velocity along one axis.

add_velocities(*self*, *components*)
Add grid-crossing frequencies from a tuple of velocity components.

add_nonconservative_diffusivity(*self*, *diffusivity*)
Add grid-crossing frequencies from a diffusivity along all axes. This method treats the non-conservative form, e.g.

dt(C) = diff*di(di(C)) + …

The corresponding timescale for the i-th axis is therefore freq_i = diff / spacing_i**2

add_conservative_diffusivity(*self*, *diffusivity*)
Add grid-crossing frequencies from a diffusivity along all axes. This method treats the conservative form, e.g.

dt(C) = di(diff*di(C)) + …

Expanding the divergence gives advective terms and diffusive terms di(diff*di(C)) = di(diff)*di(C) + diff*di(di(C))

This results in advective and diffusive frequencies freq_adv_i = di(diff) / spacing_i freq_diff_i = diff / spacing_i**2

dedalus.extras.plot_tools

Module Contents

class FieldWrapper(*field*)
Class to mimic h5py dataset interface for Dedalus fields.

property shape(*self*)

class DimWrapper(*field*, *axis*)
Wrapper class to mimic h5py dimension scales.

property label(*self*)

plot_bot(*dset*, *image_axes*, *data_slices*, *image_scales=(0, 0)*, *clim=None*, *even_scale=False*, *cmap='RdBu_r'*, *axes=None*, *figkw={}*, *title=None*, *func=None*)
Plot a 2d slice of the grid data of a dset/field.

Parameters

- **dset** (*h5py dset or Dedalus Field object*) – Dataset to plot
- **image_axes** (*tuple of ints (xi, yi)*) – Data axes to use for image x and y axes
- **data_slices** (*tuple of slices, ints*) – Slices selecting image data from global data
- **image_scales** (*tuple of ints or strs (xs, ys)*) – Axis scales (default: (0,0))
- **clim** (*tuple of floats, optional*) – Colorbar limits (default: (data min, data max))
- **even_scale** (*bool, optional*) – Expand colorbar limits to be symmetric around 0 (default: False)
- **cmap** (*str, optional*) – Colormap name (default: 'RdBu_r')
- **axes** (*matplotlib.Axes object, optional*) – Axes to overplot. If None (default), a new figure and axes will be created.
- **figkw** (*dict, optional*) – Keyword arguments to pass to plt.figure (default: {})
- **title** (*str, optional*) – Title for plot (default: dataset name)
- **func** (*function(xmesh, ymesh, data), optional*) – Function to apply to selected meshes and data before plotting (default: None)

plot_bot_2d(*dset*, *transpose=False*, ***kw*)
Plot the grid data of a 2d field.

Parameters

- **field** (*field object*) – Field to plot
- **transpose** (*bool, optional*) – Flag for transposing plot (default: False)
- **Other keyword arguments are passed on to plot_bot.**

plot_bot_3d(*dset*, *normal_axis*, *normal_index*, *transpose=False*, ***kw*)
Plot a 2d slice of the grid data of a 3d field.

Parameters

- **field** (*field object*) – Field to plot
- **normal_axis** (*int or str*) – Index or name of normal axis
- **normal_index** (*int*) – Index along normal direction to plot

- **transpose** (*bool, optional*) – Flag for transposing plot (default: False)
- **Other keyword arguments are passed on to plot_bot.**

class MultiFigure(*nrows*, *ncols*, *image*, *pad*, *margin*, *scale=1.0*, ***kw*)
An array of generic images within a matplotlib figure.

Parameters

- **nrows, ncols** (*int*) – Number of image rows/columns.
- **image** (*Box instance*) – Box describing the image shape.
- **pad** (*Frame instance*) – Frame describing the padding around each image.
- **margin** (*Frame instance*) – Frame describing the margin around the array of images.
- **scale** (*float, optional*) – Scaling factor to convert from provided box/frame units to figsize. Margin will be automatically expanded so that fig dimensions are integers.
- **Other keywords passed to plt.figure.**

add_axes(*self*, *i*, *j*, *rect*, ***kw*)
Add axes to a subfigure.

Parameters

- **i, j** (*int*) – Image row/column
- **rect** (*tuple of floats*) – (left, bottom, width, height) in fractions of image width and height
- **Other keywords passed to Figure.add_axes.**

class Box(*x*, *y*)
2d-vector-like object for representing image sizes and offsets.

Parameters **x, y** (*float*) – Box width/height.

property xbox(*self*)

property ybox(*self*)

class Frame(*top*, *bottom*, *left*, *right*)
Object for representing a non-uniform frame around an image.

Parameters **top, bottom, left, right** (*float*) – Frame widths.

property bottom_left(*self*)

property top_right(*self*)

quad_mesh(*x*, *y*, *cut_x_edges=False*, *cut_y_edges=False*)
Construct quadrilateral mesh arrays from two grids. Intended for use with e.g. plt.pcolor.

Parameters

- **x** (*1d array*) – Grid for last axis of the mesh.
- **y** (*1d array*) – Grid for first axis of the mesh.
- **cut_x_edges, cut_y_edges** (*bool, optional*) – True to truncate edge quadrilaterals at x/y grid edges. False (default) to center edge quadrilaterals at x/y grid edges.

get_1d_vertices(*grid*, *cut_edges=False*)
Get vertices dividing a 1d grid.

Parameters

- **grid** (*1d array*) – Grid.

- **cut_edges** (*bool, optional*) – True to set edge vertices at grid edges. False (default) to center edge segments at grid edges.

pad_limits(*xgrid*, *ygrid*, *xpad=0.0*, *ypad=0.0*, *square=None*)
Compute padded image limits for x and y grids.

Parameters

- **xgrid** (*array*) – Grid for x axis of image.
- **ygrid** (*array*) – Grid for y axis of image.
- **xpad** (*float, optional*) – Padding fraction for x axis (default: 0.).
- **ypad** (*float, optional*) – Padding fraction for y axis (default: 0.).
- **square** (*axis object, optional*) – Extend limits to have a square aspect ratio within an axis.

get_plane(*dset*, *xaxis*, *yaxis*, *slices*, *xscale=0*, *yscale=0*, ***kw*)
Select plane from dataset. Intended for use with e.g. plt.pcolor.

Parameters

- **dset** (*h5py dataset*) – Dataset
- **xaxis, yaxis** (*int*) – Axes for plotting
- **xscale, yscale** (*int or str*) – Corresponding axis scales
- **slices** (*tuple of ints, slice objects*) – Selection object for dataset
- **Other keywords passed to quad_mesh**

dedalus.libraries

Subpackages

dedalus.libraries.fftw

Submodules

dedalus.libraries.matsolvers

Matrix solver wrappers.

Module Contents

matsolvers

add_solver(*solver*)

class SolverBase(*matrix*, *solver=None*)
Abstract base class for all solvers.

solve(*self*, *vector*)

class SparseSolver(*matrix*, *solver=None*)
Base class for sparse solvers.

sparse = True

banded = False

class BandedSolver(*matrix*, *solver=None*)
Base class for banded solvers.

sparse = False

banded = True

static sparse_to_banded(*matrix*, *u=None*, *l=None*)
Convert sparse matrix to banded format.

class DenseSolver(*matrix*, *solver=None*)
Base class for dense solvers.

sparse = False

banded = False

class UmfpackSpsolve(*matrix*, *solver=None*)
UMFPACK spsolve.

solve(*self*, *vector*)

class SuperluNaturalSpsolve(*matrix*, *solver=None*)
SuperLU+NATURAL spsolve.

solve(*self*, *vector*)

class SuperluColamdSpsolve(*matrix*, *solver=None*)
SuperLU+COLAMD spsolve.

solve(*self*, *vector*)

class UmfpackFactorized(*matrix*, *solver=None*)
UMFPACK LU factorized solve.

solve(*self*, *vector*)

class SuperluNaturalFactorized(*matrix*, *solver=None*)
SuperLU+NATURAL LU factorized solve.

solve(*self*, *vector*)

class SuperluNaturalFactorizedTranspose(*matrix*, *solver=None*)
SuperLU+NATURAL LU factorized solve.

solve(*self*, *vector*)

class SuperluColamdFactorized(*matrix*, *solver=None*)
SuperLU+COLAMD LU factorized solve.

solve(*self*, *vector*)

class ScipyBanded(*matrix*, *solver=None*)
Scipy banded solve.

solve(*self*, *vector*)

class SPQR_solve(*matrix*, *solver=None*)
SuiteSparse QR solve.

solve(*self*, *vector*)

class BandedQR(*matrix*, *solver=None*)
pybanded QR solve.

solve(*self*, *vector*)

class SparseInverse(*matrix*, *solver=None*)
Sparse inversion solve.

solve(*self*, *vector*)

class DenseInverse(*matrix*, *solver=None*)
Dense inversion solve.

solve(*self*, *vector*)

class BlockInverse(*matrix*, *solver*)
Block inversion solve.

dedalus.tests

Dedalus testing module.

Subpackages

dedalus.tests.special_functions

Submodules

dedalus.tests.special_functions.airy

Compute Airy functions by solving the Airy equation:

f_xx - (A + Bx) f = 0 f(-1) = C f(+1) = D

Module Contents

default_params

dedalus_domain(*N*)
Construct Dedalus domain for solving the Airy equation.

dedalus_solution(*A*, *B*, *C*, *D*, *N*)
Use Dedalus to solve the Airy equation.

exact_solution(*A*, *B*, *C*, *D*, *N*)
Use scipy to construct exact solution to the Airy equation.

test_airy(*params=default_params*)
Compare Dedalus and exact results.

dedalus.tests.special_functions.bessel

Compute Bessel function by solving the Bessel equation:

x**2 f_xx + x f_x + (x**2 - A**2) f = 0 f(0) = 0 f(30) = B

Module Contents

default_params

dedalus_domain(*N*)
Construct Dedalus domain for solving the Airy equation.

dedalus_solution(*A*, *B*, *N*)
Use Dedalus to solve the Airy equation.

exact_solution(*A*, *B*, *N*)
Use scipy to construct exact solution to the Airy equation.

test_bessel(*params=default_params*)
Compare Dedalus and exact results.

Submodules

dedalus.tests.test_bvp

Test 2D BVP with various bases and dtypes.

Module Contents

bench_wrapper(*test*)

test_poisson_2d_periodic(*benchmark*, *x_basis_class*, *y_basis_class*, *Nx*, *Ny*, *dtype*)

test_poisson_2d_periodic_firstorder(*benchmark*, *x_basis_class*, *y_basis_class*, *Nx*, *Ny*, *dtype*)

DoubleChebyshev(*name*, *N*, *interval=(- 1, 1)*, *dealias=1*)

DoubleLegendre(*name*, *N*, *interval=(- 1, 1)*, *dealias=1*)

test_poisson_2d_nonperiodic(*benchmark*, *x_basis_class*, *y_basis_class*, *Nx*, *Ny*, *dtype*)

DoubleLaguerre(*name*, *N*, *center=0.0*, *stretch=1.0*, *dealias=1*)

LCCL(*name*, *N*, *center=0.0*, *stretch=1.0*, *cwidth=1.0*, *dealias=1*)

test_gaussian_free(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

test_gaussian_forced(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

ChebLag(*name*, *N*, *edge=0.0*, *stretch=1.0*, *cwidth=1.0*, *dealias=1*)

test_exponential_free(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

test_exponential_forced(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

test_double_exponential_free(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

test_double_exponential_forced(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

dedalus.tests.test_evp

Test 1D EVP.

Module Contents

logger

bench_wrapper(*test*)

DoubleChebyshev(*name*, *N*, *interval=(- 1, 1)*, *dealias=1*)

DoubleLegendre(*name*, *N*, *interval=(- 1, 1)*, *dealias=1*)

test_wave_dense_evp(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

test_wave_sparse_evp(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

test_qho_dense_evp(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

DoubleLaguerre(*name*, *N*, *center=0.0*, *stretch=1.0*, *dealias=1*)

LCCL(*name*, *N*, *center=0.0*, *stretch=1.0*, *cwidth=1.0*, *dealias=1*)

test_qho_stretch_dense_evp(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

ChebLag(*name*, *N*, *edge=0.0*, *stretch=1.0*, *cwidth=1.0*, *dealias=1*)

test_half_qho_dense_evp(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

dedalus.tests.test_ivp

Test 1D IVP with various timesteppers.

Module Contents

bench_wrapper(*test*)

test_heat_1d_periodic(*benchmark*, *x_basis_class*, *Nx*, *timestepper*, *dtype*)

test_heat_1d_periodic_firstorder(*benchmark*, *x_basis_class*, *Nx*, *timestepper*, *dtype*)

test_heat_1d_nonperiodic(*benchmark*, *x_basis_class*, *Nx*, *timestepper*, *dtype*)

test_heat_ode_1d_nonperiodic(*benchmark*, *x_basis_class*, *Nx*, *timestepper*, *dtype*)

dedalus.tests.test_matsolvers

Test matrix solvers.

Module Contents

bench_wrapper(*test*)

diagonal_solver(*Nx*, *Ny*, *dtype*)

block_solver(*Nx*, *Ny*, *dtype*)

coupled_solver(*Nx*, *Ny*, *dtype*)

solvers

test_matsolver_setup_bench(*benchmark*, *solver*, *matsolver*)

test_matsolver_solve_bench(*benchmark*, *solver*, *matsolver*)

dedalus.tests.test_nlbvp

Test 1D NLBVP with various dtypes.

Module Contents

logger

bench_wrapper(*test*)

DoubleChebyshev(*name*, *N*, *interval=(- 1, 1)*, *dealias=1*)

DoubleLegendre(*name*, *N*, *interval=(- 1, 1)*, *dealias=1*)

test_sin_nlbvp(*benchmark*, *x_basis_class*, *Nx*, *dtype*)

dedalus.tests.test_output

Test 1D IVP with various timesteppers.

Module Contents

bench_wrapper(*test*)

test_1d_output(*x_basis_class*, *Nx*, *timestepper*, *dtype*)

Package Contents

file

root

test()
: Run tests.

bench()
: Run benchmarks.

cov()
: Print test coverage.

dedalus.tools

Submodules

dedalus.tools.array

Tools for array manipulations.

Module Contents

interleaved_view(*data*)
View n-dim complex array as (n+1)-dim real array, where the last axis separates real and imaginary parts.

reshape_vector(*data*, *dim=2*, *axis=- 1*)
Reshape 1-dim array as a multidimensional vector.

axslice(*axis*, *start*, *stop*, *step=None*)
Slice array along a specified axis.

zeros_with_pattern(**args*)
Create sparse matrix with the combined pattern of other sparse matrices.

expand_pattern(*input*, *pattern*)
Return copy of sparse matrix with extended pattern.

apply_matrix(*matrix*, *array*, *axis*, ***kw*)
Contract any direction of a multidimensional array with a matrix.

add_sparse(*A*, *B*)
Add sparse matrices, promoting scalars to multiples of the identity.

dedalus.tools.cache

Tools for caching computations.

Module Contents

class CachedAttribute(*method*)
Descriptor for building attributes during first access.

class CachedFunction(*function*, *max_size=None*)
Decorator for caching function outputs.

class CachedMethod(*function*, *max_size=None*)
Descriptor for caching method outputs.

class CachedClass(*cls*, **args*, ***kw*)
Metaclass for caching instantiation.

serialize_call(*args*, *kw*, *argnames*, *defaults*)
Serialize args/kw into cache key.

dedalus.tools.config

Configuration handling.

Module Contents

config

dedalus.tools.dispatch

Tools for emulating multiple dispatch.

Module Contents

exception SkipDispatchException(*output*)
Exceptions for shortcutting MultiClass dispatch.

Initialize self. See help(type(self)) for accurate signature.

class SkipDispatch
Metclass for skipping dispatch based on arguments.

class MultiClass
Metaclass for dispatching instantiation to subclasses.

class CachedMultiClass(*cls*, **args*, ***kw*)
Metaclass for dispatching and caching instantiantation to subclasses.

dedalus.tools.exceptions

Custom exception classes.

Module Contents

exception NonlinearOperatorError
Exceptions for nonlinear LHS terms.

Initialize self. See help(type(self)) for accurate signature.

exception SymbolicParsingError
Exceptions for syntactic and mathematical problems in equations.

Initialize self. See help(type(self)) for accurate signature.

exception UnsupportedEquationError
Exceptions for valid but unsupported equations.

Initialize self. See help(type(self)) for accurate signature.

exception UndefinedParityError
Exceptions for data/operations with undefined parity.

Initialize self. See help(type(self)) for accurate signature.

dedalus.tools.general

Extended built-ins, etc.

Module Contents

class OrderedSet(**args*)
Ordered set based on uniqueness of dictionary keys.

update(*self*, **args*)

add(*self*, *item*)

rev_enumerate(*sequence*)
Simple reversed enumerate.

natural_sort(*iterable*, *reverse=False*)
Sort alphanumeric strings naturally, i.e. with "1" before "10". Based on http://stackoverflow.com/a/4836734.

oscillate(*iterable*)
Oscillate forward and backward through an iterable.

unify(*objects*)
Check if all objects in a collection are equal. If so, return one of them. If not, raise.

dedalus.tools.logging

Logging setup.

Module Contents

MPI_RANK

MPI_SIZE

stdout_level

file_level

nonroot_level

filename

rootlogger

formatter

stdout_handler

file_path

dedalus.tools.parallel

Tools for running in parallel.

Module Contents

class Sync(*comm=MPI.COMM_WORLD*, *enter=True*, *exit=True*)
Context manager for synchronizing MPI processes.

Parameters

- **enter** (*boolean, optional*) – Apply MPI barrier on entering context. Default: True
- **exit** (*boolean, optional*) – Apply MPI barrier on exiting context. Default: True

sync_glob(*path*, *glob*, *comm=MPI.COMM_WORLD*)
Syncronized pathlib globbing for consistent results across processes.

Parameters

- **path** (*str or pathlib.Path*) – Base path for globbing.
- **pattern** (*str*) – Glob pattern.
- **comm** (*mpi4py communicator, optional*) – MPI communicator. Default: MPI.COMM_WORLD

dedalus.tools.parsing

Tools for equation parsing.

Module Contents

split_equation(*equation*)
Split equation string into LHS and RHS strings.

Examples

```
>>> split_equation('f(x, y=5) = x**2')
('f(x, y=5)', 'x**2')
```

split_call(*call*)
Convert math-style function definitions into head and arguments.

Examples

```
>>> split_call('f(x, y)')
('f', ('x', 'y'))
>>> split_call('f')
('f', ())
```

lambdify_functions(*call*, *result*)
Convert math-style function definitions into lambda expressions. Pass other statements without modification.

Examples

```
>>> lambdify_functions('f(x, y)', 'x*y')
('f', 'lambda x,y: x*y')
>>> lambdify_functions('f', 'a*b')
('f', 'a*b')
```

dedalus.tools.plot_op

Module Contents

class Node(*label*, *level*)

class Leaf(**args*)

count = 0.0

class Tree(*operator*)

build(*self*, *arg*, *level*)

set_position(*self*, *node*)

plot_operator(*operator*, *fontsize=8*, *figsize=8*, *opsize=0.3*, *saveas=None*)

pad(*xmin*, *xmax*, *ymin*, *ymax*, *pad=0.0*, *square=False*)

dedalus.tools.post

Post-processing helpers.

Module Contents

`logger`

`visit_writes`(*set_paths*, *function*, *comm=MPI.COMM_WORLD*, ***kw*)
Apply function to writes from a list of analysis sets.

Parameters

- **set_paths** (*list of str or pathlib.Path*) – List of set paths
- **function** (*function(set_path, start, count, **kw)*) – A function on an HDF5 file, start index, and count.
- **comm** (*mpi4py.MPI.Intracomm, optional*) – MPI communicator (default: COMM_WORLD)
- **Other keyword arguments are passed on to `function`**

Notes

This function is parallelized over writes, and so can be effectively parallelized up to the number of writes from all specified sets.

`get_assigned_writes`(*set_paths*, *comm=MPI.COMM_WORLD*)
Divide writes from a list of analysis sets between MPI processes.

Parameters

- **set_paths** (*list of str or pathlib.Path*) – List of set paths
- **comm** (*mpi4py.MPI.Intracomm, optional*) – MPI communicator (default: COMM_WORLD)

`get_all_writes`(*set_paths*)
Get write numbers from a list of analysis sets.

Parameters **set_paths** (*list of str or pathlib.Path*) – List of set paths

`get_assigned_sets`(*base_path*, *distributed=False*, *comm=MPI.COMM_WORLD*)
Divide analysis sets from a FileHandler between MPI processes.

Parameters

- **base_path** (*str or pathlib.Path*) – Base path of FileHandler output
- **distributed** (*bool, optional*) – Divide distributed sets instead of merged sets (default: False)
- **comm** (*mpi4py.MPI.Intracomm, optional*) – MPI communicator (default: COMM_WORLD)

`merge_process_files`(*base_path*, *cleanup=False*, *comm=MPI.COMM_WORLD*)
Merge process files from all distributed analysis sets in a folder.

Parameters

- **base_path** (*str or pathlib.Path*) – Base path of FileHandler output
- **cleanup** (*bool, optional*) – Delete distributed files after merging (default: False)
- **comm** (*mpi4py.MPI.Intracomm, optional*) – MPI communicator (default: COMM_WORLD)

Notes

This function is parallelized over sets, and so can be effectively parallelized up to the number of distributed sets.

`merge_process_files_single_set`(*set_path*, *cleanup=False*)
Merge process files from a single distributed analysis set.

Parameters

- **set_path** (*str of pathlib.Path*) – Path to distributed analysis set folder
- **cleanup** (*bool, optional*) – Delete distributed files after merging (default: False)

`merge_setup`(*joint_file*, *proc_path*)
Merge HDF5 setup from part of a distributed analysis set into a joint file.

Parameters

- **joint_file** (*HDF5 file*) – Joint file
- **proc_path** (*str or pathlib.Path*) – Path to part of a distributed analysis set

`merge_data`(*joint_file*, *proc_path*)
Merge data from part of a distributed analysis set into a joint file.

Parameters

- **joint_file** (*HDF5 file*) – Joint file
- **proc_path** (*str or pathlib.Path*) – Path to part of a distributed analysis set

`merge_sets`(*joint_path*, *set_paths*, *cleanup=False*, *comm=MPI.COMM_WORLD*)
Merge analysis sets.

Parameters

- **joint_path** (*string or pathlib.Path*) – Path for merged file.
- **set_paths** (*list of strings or pathlib.Path objects*) – Paths of all sets to be merged
- **cleanup** (*bool, optional*) – Delete set files after merging (default: False)
- **comm** (*mpi4py.MPI.Intracomm, optional*) – MPI communicator (default: COMM_WORLD)

`merge_analysis`

dedalus.tools.progress

Tools for tracking iteration progress.

Module Contents

`log_progress`(*iterable*, *logger*, *level*, ***kw*)
Log iteration progress.

`print_progress`(*iterable*, *stream=sys.stdout*, ***kw*)
Print iteration progress to a stream.

`track`(*iterable*, *write*, *desc='Iteration'*, *iter=1*, *frac=1.0*, *dt=np.inf*)
Track an iterator attaching messages at set cadences.

format_time(*total_sec*)
Format time strings.

dedalus.tools.sparse

Tools for working with sparse matrices.

Module Contents

scipy_sparse_eigs(*A*, *B*, *N*, *target*, *matsolver*, ***kw*)
Perform targeted eigenmode search using the scipy/ARPACK sparse solver for the reformulated generalized eigenvalue problem

A.x = B.x ==> (A - B)^I B.x = (1/(-)) x

for eigenvalues near the target .

Parameters

- **A, B** (*scipy sparse matrices*) – Sparse matrices for generalized eigenvalue problem
- **N** (*int*) – Number of eigenmodes to return
- **target** (*complex*) – Target for eigenvalue search
- **matsolver** (*matrix solver class*) – Class implementing solve method for solving sparse systems.
- **Other keyword options passed to scipy.sparse.linalg.eigs.**

same_dense_block_diag(*blocks*, *format=None*, *dtype=None*)
Build a block diagonal sparse matrix from identically shaped dense blocks.

Parameters

- **blocks** (*sequence of 2D ndarrays*) – Input matrix blocks.
- **format** (*str, optional*) – The sparse format of the result (e.g. "csr"). If not given, the matrix is returned in "coo" format.
- **dtype** (*dtype specifier, optional*) – The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

Returns res

Return type sparse matrix

fast_csr_matvec(*A_csr*, *x_vec*, *out_vec*)
Fast CSR matvec skipping type and shape checks. The result is added to the specificed output array, so the output should be manually zeroed prior to calling this routine, if necessary.

1.5.2 Submodules

dedalus.dev

Interface for accessing all submodules.

dedalus.public

Interface for tools typically accessed for solving a problem.

CHAPTER

TWO

LINKS

- Project homepage: http://dedalus-project.org
- Code repository: https://github.com/DedalusProject/dedalus
- Documentation: http://dedalus-project.readthedocs.org
- Mailing list: https://groups.google.com/forum/#!forum/dedalus-users

CHAPTER

THREE

DEVELOPERS

- Keaton Burns
- Geoff Vasil
- Jeff Oishi
- Daniel Lecoanet
- Ben Brown

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

Z