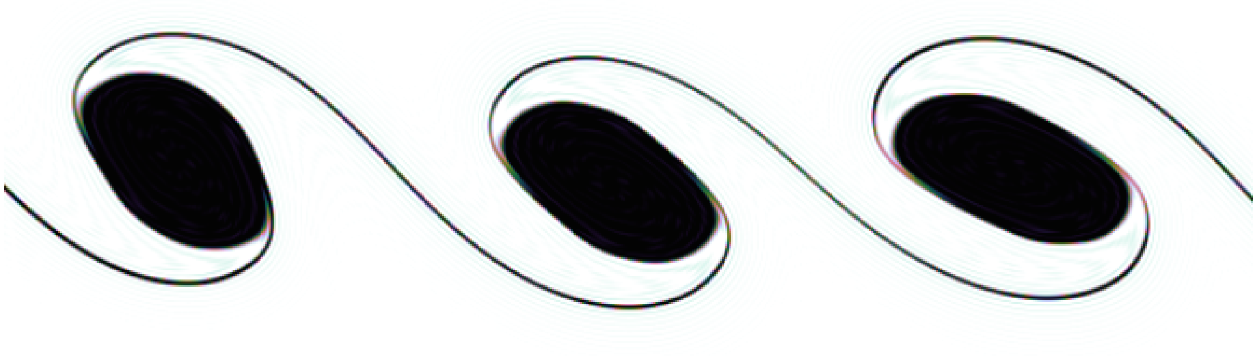
pyqg Documentation

Release 0.7.3.dev0

PyQG team

May 19, 2022

CONTENTS



pyqg is a python solver for quasigeostrophic systems. Quasigeostophic equations are an approximation to the full fluid equations of motion in the limit of strong rotation and stratification and are most applicable to geophysical fluid dynamics problems.

Students and researchers in ocean and atmospheric dynamics are the intended audience of pyqg. The model is simple enough to be used by students new to the field yet powerful enough for research. We strive for clear documentation and thorough testing.

pyqg supports a variety of different configurations using the same computational kernel. The different configurations are evolving and are described in detail in the documentation. The kernel, implement in cython, uses a pseudo-spectral method which is heavily dependent of the fast Fourier transform. For this reason, pyqg tries to use pyfftw and the FFTW Fourier Transform library. (If pyfftw is not available, it falls back on numpy.fft) With pyfftw, the kernel is multi-threaded but does not support mpi. Optimal performance will be achieved on a single system with many cores.

CHAPTER ONE

CONTENTS

1.1 Installation

1.1.1 Requirements

The only requirements are

- Python (3.6 or later)
- numpy (1.6 or later)
- Cython (0.2 or later)

Because pyqg is a pseudo-spectral code, it realies heavily on fast-Fourier transforms (FFTs), which are the main performance bottlneck. For this reason, we try to use fftw (a fast, multithreaded, open source C library) and pyfftw (a python wrapper around fftw). These packages are optional, but they are strongly recommended for anyone doing high-resolution, numerically demanding simulations.

- fftw (3.3 or later)
- pyfftw (0.9.2 or later)

If pyqg can't import pyfftw at compile time, it can fall back on numpy's fft routines. **Note that the numpy_ fallback requires a local install (see [below](#installing-pyqg)).**

PyQG can also conveniently store model output data as an xarray dataset. The feature (which is used in some of the examples in this documentation) requires xarray.

1.1.2 Instructions

The easiest and quickest way: installing pyqg with conda

We suggest that you install pyqg using conda. This will automatically install pyfftw as well, so then you will be done and can ignore the remaining instructions on this page. To install pyqg with conda,

```
$ conda install -c conda-forge pyqg
```

Alternatives

In our opinion, the best way to get python and numpy is to use a distribution such as Anaconda (recommended) or Canopy. These provide robust package management and come with many other useful packages for scientific computing. The pyqg developers are mostly using anaconda.

Note: If you don't want to use pyfftw and are content with numpy's slower performance, you can skip ahead to *Installing pyqg*.

Installing fftw and pyfftw can be slightly painful. Hopefully the instructions below are sufficient. If not, please send feedback.

Installing fftw and pyfftw

Once you have installed pyfftw via one of these paths, you can proceed to *Installing pyqg*.

The easy way: installing with conda

If you are using Anaconda, we have discovered that you can easily install pyffw using the `conda` command. Although pyfftw is not part of the main Anaconda distribution, it is distributed as a conda pacakge through several user channels.

There is a useful blog post describing how the pyfftw conda package was created. There are currently 13 pyfftw user packages hosted on anaconda.org. Each has different dependencies and platform support (e.g. linux, windows, mac.) The conda-forge version is the most popular and appears to have the broadest cross-platform support. To install it, open a terminal and run the command

```
$ conda install -c conda-forge pyfftw
```

The hard way: installing from source

This is the most difficult step for new users. You will probably have to build FFTW3 from source. However, if you are using Ubuntu linux, you can save yourself some trouble by installing fftw using the apt package manager

```
$ sudo apt-get install libfftw3-dev libfftw3-doc
```

Otherwise you have to build FFTW3 from source. Your main resource for the FFTW homepage. Below we summarize the steps

First download the source code.

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

Then run the configure command

```
$ ./configure --enable-threads --enable-shared
```

Note: If you don't have root privileges on your computer (e.g. on a shared cluster) the best approach is to ask your system administrator to install FFTW3 for you. If that doesn't work, you will have to install the FFTW3 libraries into a location in your home directory (e.g. `$HOME/fftw`) and add the flag `--prefix=$HOME/fftw` to the configure command above.

Then build the software

```
$ make
```

Then install the software

```
$ sudo make install
```

This will install the FFTW3 libraries into you system's library directory. If you don't have root privileges (see note above), remove the `sudo`. This will install the libraries into the `prefix` location you specified.

You are not done installing FFTW yet. pyfftw requires special versions of the FFTW library specialized to different data types (32-bit floats and double-long floars). You need to-configure and re-build FFTW two more times with extra flags.

```
$ ./configure --enable-threads --enable-shared --enable-float
$ make
$ sudo make install
$ ./configure --enable-threads --enable-shared --enable-long-double
$ make
$ sudo make install
```

At this point, you FFTW installation is complete. We now move on to pyfftw. pyfftw is a python wrapper around the FFTW libraries. The easiest way to install it is using `pip`:

```
$ pip install pyfftw
```

or if you don't have root privileges

```
$ pip install pyfftw --user
```

If this fails for some reason, you can manually download and install it according to the instructions on github. First clone the repository:

```
$ git clone https://github.com/hgomersall/pyFFTW.git
```

Then install it

```
$ cd pyFFTW
$ python setup.py install
```

or

```
$ python setup.py install --user
```

if you don't have root privileges. If you installed FFTW in a non-standard location (e.g. $HOME/fftw), you might have to do something tricky at this point to make sure pyfftw can find FFTW. (I figured this out once, but I can't remember how.)

Installing pyqg

Note: The pyqg kernel is written in Cython and uses OpenMP to parallelise some operations for a performance boost. If you are using Mac OSX Yosemite or later OpenMP support is not available out of the box. While pyqg will still run without OpenMP, it will not be as fast as it can be. See *Installing with OpenMP support on OSX* below for more information on installing on OSX with OpenMP support.

With pyfftw installed, you can now install pyqg. The easiest way is with pip:

```
$ pip install pyqg
```

You can also clone the pyqg git repository to use the latest development version.

```
$ git clone https://github.com/pyqg/pyqg.git
```

Then install pyqg locally on your system:

```
$ cd pyqg && pip install --editable .
```

This will also allow you to make and test changes to the library. pyqg is a work in progress, and we really encourage users to contribute to its *Development*

Note that due to Cython build considerations, this local install method is required if you do not wish to use pyfftw.

Installing with OpenMP support on OSX

There are two options for installing on OSX with OpenMP support. Both methods require using the Anaconda distribution of Python.

1. Using Homebrew

Install the GCC-5 compiler in `/usr/local` using Homebrew:

```
$ brew install gcc --without-multilib --with-fortran
```

Install Cython from the conda repository

```
$ conda install cython
```

Install pyqg using the homebrew `gcc` compiler

```
$ CC=/usr/local/bin/gcc-5 pip install pyqg
```

2. Using the HPC precompiled gcc binaries.

The HPC for Mac OSX sourceforge project has copies of the latest `gcc` precompiled for Mac OSX. Download the latest version of gcc from the HPC site and follow the installation instructions.

Install Cython from the conda repository

```
$ conda install cython
```

Install pyqg using the HPC `gcc` compiler

```
$ CC=/usr/local/bin/gcc pip install pyqg
```

1.2 Equations Solved

A detailed description of the equations solved by the various pyqg models

1.2.1 Equations For Two-Layer QG Model

The two-layer quasigeostrophic evolution equations are (1)

$$\partial_t\, q_1 + \mathsf{J}\left(\psi_1\,, q_1\right) + \beta\, {\psi_1}_x = \text{ssd}\,,$$

and (2)

$$\partial_t\, q_2 + \mathsf{J}\left(\psi_2\,, q_2\right) + \beta\, {\psi_2}_x = -r_{ek}\nabla^2\psi_2 + \text{ssd}\,,$$

where the horizontal Jacobian is $\mathsf{J}\left(A\,, B\right) = A_x B_y - A_y B_x$. Also in (1) and (2) ssd denotes small-scale dissipation (in turbulence regimes, ssd absorbs enstrophy that cascades towards small scales). The linear bottom drag in (2) dissipates large-scale energy.

The potential vorticities are (3)

$$q_1 = \nabla^2\psi_1 + F_1\left(\psi_2 - \psi_1\right)\,,$$

and (4)

$$q_2 = \nabla^2\psi_2 + F_2\left(\psi_1 - \psi_2\right)\,,$$

where

$$F_1 \equiv \frac{k_d^2}{1+\delta}\,, \qquad \text{and} \qquad F_2 \equiv \delta\, F_1\,,$$

with the deformation wavenumber

$$k_d^2 \equiv \frac{f_0^2}{g'}\frac{H_1+H_2}{H_1 H_2}\,,$$

where $H = H_1 + H_2$ is the total depth at rest.

Forced-dissipative equations

We are interested in flows driven by baroclinic instabilty of a base-state shear $U_1 - U_2$. In this case the evolution equations (1) and (2) become (5)

$$\partial_t\, q_1 + \mathsf{J}\left(\psi_1\,, q_1\right) + \beta_1\, {\psi_1}_x = \text{ssd}\,,$$

and (6)

$$\partial_t\, q_2 + \mathsf{J}\left(\psi_2\,, q_2\right) + \beta_2\, {\psi_2}_x = -r_{ek}\nabla^2\psi_2 + \text{ssd}\,,$$

where the mean potential vorticity gradients are (9,10)

$$\beta_1 = \beta + F_1\,\left(U_1 - U_2\right)\,, \qquad \text{and} \qquad \beta_2 = \beta - F_2\,\left(U_1 - U_2\right)\,.$$

Equations in Fourier space

We solve the two-layer QG system using a pseudo-spectral doubly-peridioc model. Fourier transforming the evolution equations (5) and (6) gives (7)

$$\partial_t \widehat{q}_1 = -\widehat{\mathsf{J}}\left(\psi_1\,, q_1\right) - \mathrm{i}\,k\,\beta_1\,\widehat{\psi}_1 + \widehat{\mathrm{ssd}}\,,$$

and

$$\partial_t \widehat{q}_2 = -\widehat{\mathsf{J}}\left(\psi_2\,, q_2\right) - \mathrm{i}\,k\,\beta_2\,\widehat{\psi}_2 + r_{ek}\,\kappa^2\;\widehat{\psi}_2 + \widehat{\mathrm{ssd}}\,,$$

where, in the pseudo-spectral spirit, $\widehat{\mathsf{J}}$ means the Fourier transform of the Jacobian i.e., we compute the products in physical space, and then transform to Fourier space.

In Fourier space the "inversion relation" (3)-(4) is

$$\underbrace{\begin{bmatrix} -(\kappa^2 + F_1) & F_1 \\ F_2 & -\left(\kappa^2 + F_2\right) \end{bmatrix}}_{\equiv \mathsf{M}_2} \begin{bmatrix} \widehat{\psi}_1 \\ \widehat{\psi}_2 \end{bmatrix} = \begin{bmatrix} \widehat{q}_1 \\ \widehat{q}_2 \end{bmatrix} ,$$

or equivalently

$$\begin{bmatrix} \widehat{\psi}_1 \\ \widehat{\psi}_2 \end{bmatrix} = \underbrace{\frac{1}{\det \mathrm{M}_2} \begin{bmatrix} -(\kappa^2 + F_2) & -F_1 \\ -F_2 & -\left(\kappa^2 + F_1\right) \end{bmatrix}}_{=\mathsf{M}_2{}^{-1}} \begin{bmatrix} \widehat{q}_1 \\ \widehat{q}_2 \end{bmatrix} ,$$

where

$$\det \mathsf{M}_2 = \kappa^2\left(\kappa^2 + F_1 + F_2\right)\,.$$

Marching forward

We use a third-order Adams-Bashford scheme

$$\widehat{q}_i^{n+1} = E_f \times \left[\widehat{q}_i^{n} + \frac{\Delta t}{12}\left(23\,\widehat{Q}_i^{n} - 16\widehat{Q}_i^{n-1} + 5\widehat{Q}_i^{n-2}\right)\right]\,,$$

where

$$\widehat{Q}_i^n \equiv -\widehat{\mathsf{J}}\left(\psi_i^n\,, q_i^n\right) - \mathrm{i}\,k\,\beta_i\,\widehat{\psi}_i^n, \qquad i = 1, 2\,.$$

The AB3 is initialized with a first-order AB (or forward Euler)

$$\widehat{q}_i^{1} = E_f \times \left[\widehat{q}_i^{0} + \Delta t \widehat{Q}_i^{0}\right]\,,$$

The second step uses a second-order AB scheme

$$\widehat{q}_i^{2} = E_f \times \left[\widehat{q}_i^{1} + \frac{\Delta t}{2}\left(3\,\widehat{Q}_i^{1} - \widehat{Q}_i^{0}\right)\right]\,.$$

The small-scale dissipation is achieve by a highly-selective exponential filter

$$E_f = \begin{cases} \mathrm{e}^{-23.6\,(\kappa^{\star} - \kappa_c)^4}: & \kappa \ge \kappa_c \\ 1: & \text{otherwise}\,. \end{cases}$$

where the non-dimensional wavenumber is

$$\kappa^{\star} \equiv \sqrt{(k\,\Delta x)^2 + (l\,\Delta y)^2}\,,$$

and κ_c is a (non-dimensional) wavenumber cutoff here taken as 65% of the Nyquist scale $\kappa^{\star}_{ny} = \pi$. The parameter -23.6 is obtained from the requirement that the energy at the largest wanumber ($\kappa^{\star} = \pi$) be zero whithin machine double precision:

$$\frac{\log 10^{-15}}{(0.35\,\pi)^4} \approx -23.5\,.$$

For experiments with $|\widehat{q_i}| \ll \mathcal{O}(1)$ one can use a smaller constant.

Diagnostics

The kinetic energy is

$$E = \tfrac{1}{H\,S}\int \tfrac{1}{2}H_1\,|\boldsymbol{\nabla}\psi_1|^2 + \tfrac{1}{2}H_2\,|\boldsymbol{\nabla}\psi_2|^2\,dS\,.$$

The potential enstrophy is

$$Z = \tfrac{1}{H\,S}\int \tfrac{1}{2}H_1\,q_1^2 + \tfrac{1}{2}H_2\,q_2^2\,dS\,.$$

We can use the enstrophy to estimate the eddy turn-over timescale

$$T_e \equiv \frac{2\,\pi}{\sqrt{Z}}\,.$$

1.2.2 Layered quasi-geostrophic model

Consider an N-layer quasi-geostrophic (QG) model with rigid lid and flat topography (for reference, see Eq. 5.85 in Vallis, 2017). The N-layer QG potential vorticity is

$$\begin{aligned}
q_1 &= \nabla^2\psi_1 + \frac{f_0^2}{H_1}\left(\frac{\psi_2-\psi_1}{g_1'}\right)\,, \\
q_n &= \nabla^2\psi_n + \frac{f_0^2}{H_n}\left(\frac{\psi_{n-1}-\psi_n}{g_{n-1}'} - \frac{\psi_n-\psi_{n+1}}{g_n'}\right)\,, \qquad\qquad n = 2,\dots,N-1\,, \\
q_N &= \nabla^2\psi_N + \frac{f_0^2}{H_N}\left(\frac{\psi_{N-1}-\psi_N}{g_{N-1}'}\right)\,,
\end{aligned}$$

where q_n is the n-th layer QG potential vorticity, and ψ_n is the streamfunction, f_0 is the inertial frequency, H_n is the layer depth. Also the n-th buoyancy jump (reduced gravity) is

$$g_n' \equiv g\frac{\rho_{n+1}-\rho_n}{\rho_n}\,,$$

where g is the acceleration due to gravity and ρ_n is the layer density.

The relationship between q_n and ψ_n can be conveniently written as

$$\mathbf{q} = (\mathbf{S} + \nabla^2\mathbf{I})\boldsymbol{\psi}$$

where $\mathbf{q} = (q_1, ..., q_N)^{\mathrm{T}}$, $\boldsymbol{\psi} = (\psi_1, ..., \psi_N)^{\mathrm{T}}$, $\mathbf{I}$ is the $N \times N$ identity matrix, and the stretching matrix $\mathbf{S}$ is

$$\mathbf{S} \equiv f_0^2 \begin{bmatrix} -\frac{1}{H_1 g_1'} & \frac{1}{H_1 g_1'} & 0 & 0 & \dots \\ & \vdots & & \vdots & \\ \dots & \frac{1}{H_n g_{n-1}'} & -\frac{1}{H_n}\left(\frac{1}{g_{n-1}'} + \frac{1}{g_n'}\right) & \frac{1}{H_n g_n'} & \dots \\ & \vdots & & \vdots & \\ \dots & 0 & 0 & \frac{1}{H_N g_{N-1}'} & -\frac{1}{H_N g_{N-1}'} \end{bmatrix}.$$

The dynamics of the system is given by the evolution of PV. In particular, we assume a background flow with background velocity $\overrightarrow{V} = (U, V)$ such that

$$u_n^{\text{tot}} = U_n - \psi_{ny},$$
$$v_n^{\text{tot}} = V_n + \psi_{nx},$$

and

$$q_n^{\text{tot}} = Q_n + q_n,$$

where Q_n is the n-th layer background PV. Q_n satisfies

$$\mathbf{Q} = \beta + \mathbf{SV}x - \mathbf{SU}y,$$

where $\mathbf{Q}$, $\mathbf{U}$, $\mathbf{V}$ are defined similarly to $\mathbf{q}$ and $\boldsymbol{\psi}$. We then obtain the evolution equations

$$q_{nt} + \mathsf{J}(\psi_n, q_n) + \overrightarrow{V}_n \cdot \nabla q_n + Q_{ny}\psi_{nx} - Q_{nx}\psi_{ny} = \text{ssd}_n - r_{ek}\delta_{n,N}\nabla^2\psi_n, \; n = 1, \dots, N,$$

where ssd stands for small-scale dissipation, which is achieved by an spectral exponential filter or hyperviscosity, and r_{ek} is the linear bottom drag coefficient. The Dirac delta, $\delta_{n,N}$, indicates that the drag is only applied to the bottom layer. The advection of the background PV by the background flow is neglected because in each layer, the contribution of this term is constant for all locations.

Equations in spectral space

The evolution equation in spectral space is

$$\begin{aligned} \hat{q}_{nt} + \hat{\mathsf{J}}(\psi_n, q_n) + (\mathrm{i}kU_n + \mathrm{i}lV_n)\hat{q}_n \\ + (\mathrm{i}k\,Q_{ny} - \mathrm{i}l\,Q_{nx})\hat{\psi}_n = \widehat{\text{ssd}}_n + r_{ek}\delta_{n,N}\kappa^2\hat{\psi}_n, \;\; n = 1, \dots, N, \end{aligned}$$

where $\kappa^2 = k^2 + l^2$. Also, in the pseudo-spectral spirit, we write the transform of the nonlinear terms and the non-constant coefficient linear term as the transform of the products, calculated in physical space, as opposed to double convolution sums. That is, $\hat{\mathsf{J}}$ is the Fourier transform of Jacobian computed in the physical space.

The inversion relationship between PV and streamfunction is

$$\hat{\mathbf{q}} = \left(\mathbf{S} - \kappa^2\mathbf{I}\right)\hat{\boldsymbol{\psi}}.$$

Energy spectrum

The equation for the energy spectrum is,

$$E(k,l) \equiv \frac{1}{2H}\sum_{n=1}^{\mathrm{N}} H_n \kappa^2 |\hat{\psi}_n|^2 + \frac{1}{2H}\sum_{n=1}^{\mathrm{N}-1} \frac{f_0^2}{g'_n}|\hat{\psi}_n - \hat{\psi}_{n+1}|^2,$$

To obtain the spectral flux of different components, we take the time derivative of the energy spectrum

$$\begin{aligned}\frac{\partial E(k,l)}{\partial t} &= \frac{1}{H}\mathbb{R}\left[\sum_{n=1}^{N} H_n \kappa^2 \frac{\partial \hat{\psi}_n}{\partial t}\hat{\psi}_n^* + \sum_{n=1}^{N-1}\frac{f_0^2}{g'_n}\left(\frac{\partial \hat{\psi}_n}{\partial t} - \frac{\partial \hat{\psi}_{n+1}}{\partial t}\right)(\hat{\psi}_n^* - \hat{\psi}_{n+1}^*)\right] \\ &= -\frac{1}{H}\mathbb{R}\left[\sum_{n=1}^{N} H_n \hat{\psi}_n^* \frac{\partial}{\partial t}\left(-\kappa^2\hat{\psi}_n + \frac{f_0^2}{H_n}\frac{\hat{\psi}_{n-1} - \hat{\psi}_n}{g'_{n-1}}\mathbb{1}_{n>1} - \frac{f_0^2}{H_n}\frac{\hat{\psi}_n - \hat{\psi}_{n+1}}{g'_n}\mathbb{1}_{n<N}\right)\right] \\ &= -\frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\hat{\psi}_n^* \hat{q}_{nt}\right],\end{aligned}$$

where $\mathbb{1}$ is the indicator function. This suggests that the energy tendency of the layered QG system is just the dot product of the layer-weighted streamfunction and the tendency of QG potential vorticty. Substituting the expression of q_{nt} from above, we have

$$\begin{aligned}\frac{\partial E(k,l)}{\partial t} &= \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}[\hat{\psi}_n^* \hat{\mathsf{J}}(\psi_n, \nabla^2\psi_n)] + \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}[\hat{\psi}_n^* \hat{\mathsf{J}}(\psi_n, (\mathbf{S}\boldsymbol{\psi})_n)] \\ &+ \frac{1}{H}\sum_{n=1}^{N} H_n (kU_n + lV_n)\,\mathbb{R}[i\,\hat{\psi}_n^* (\mathbf{S}\hat{\boldsymbol{\psi}})_n] - r_{ek}\frac{H_N}{H}\kappa^2|\hat{\psi}_N|^2 \\ &- \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}[\hat{\psi}_n^* \widehat{\mathrm{ssd}}_n],\end{aligned}$$

where $*$ stands for complex conjugation. We also used the fact that the terms involving background vorticity gradients does not make contribution to the real part of the right-hand-side. The right-hand-side terms represent, from left to right,

I: The spectral divergence of the kinetic energy flux;

II: The spectral divergence of the potential energy flux;

III: The spectrum of the potential energy generation;

IV: The spectrum of the energy dissipation by linear bottom drag;

V: The spectrum of energy loss due to small scale dissipation.

We assume that the fifth term is relatively small, and that, in statistical steady state, the budget above is dominated by I through IV.

Contribution from subgrid parameterization

Subgrid-scale parameterizations in terms of tendencies in q can be added to the dynamical equation, and thus has contribution to the energy spectrum. In spectral space, let the effect of parameterization be

$$\left(\frac{\partial \hat{q}_n}{\partial t}\right)^{\text{sub}} = \hat{q}_n^{\text{sub}}$$

From the derivations above, we have

$$\left(\frac{\partial E(k,l)}{\partial t}\right)^{\text{sub}} = -\frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\hat{\psi}_n^* \hat{q}_n^{\text{sub}}\right],$$

which is the spectrum of the energy contribution from parameterizations.

We can further expand the contribution of parameterization into its contribution to kinetic energy and potential energy. To see how, we consider again the time derivative of the total energy in matrix form:

$$\frac{\partial E(k,l)}{\partial t} = -\frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\hat{\psi}_n^* \left((-\kappa^2\mathbf{I} + \mathbf{S})\frac{\partial \hat{\boldsymbol{\psi}}}{\partial t}\right)_n\right],$$

where the first term on the right-hand side is the change in kinetic energy, and the second term is the change in potential energy. Considering the streamfunction tendency is from parameterizations, and letting $\mathbf{A}(\mathbf{k}) = (\mathbf{S} - \kappa^2\mathbf{I})^{-1}$ so that $\hat{\boldsymbol{\psi}} = \mathbf{A}(\mathbf{k})\hat{\mathbf{q}}$, we have

$$\left(\frac{\partial E(k,l)}{\partial t}\right)^{\text{sub}} = \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\kappa^2 \hat{\psi}_n^* \left(\mathbf{A}\hat{\mathbf{q}}^{\text{sub}}\right)_n\right] - \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\hat{\psi}_n^* \left(\mathbf{S}\mathbf{A}\hat{\mathbf{q}}^{\text{sub}}\right)_n\right] \tag{1.1}$$

where on the right-hand side, the first term is the parameterized contribution towards kinetic energy, and the second term is towards potential energy.

Enstrophy spectrum

Similarly, the evolution of the barotropic enstrophy spectrum,

$$Z(k,l) \equiv \frac{1}{2H}\sum_{n=1}^{\mathrm{N}} H_n |\hat{q}_n|^2,$$

is governed by

$$\begin{aligned}\frac{\partial Z(k,l)}{\partial t} &= \frac{1}{H}\sum_{n=1}^{\mathrm{N}} \mathbb{R}\left[\hat{q}_n^* \hat{\mathrm{J}}(\psi_n, q_n)\right] + \frac{1}{H}\sum_{n=1}^{\mathrm{N}} (lQ_{n_x} - kQ_{n_y})\mathbb{R}\left[i(\mathbf{S}\hat{\boldsymbol{\psi}}^*)_n \hat{\psi}_n\right] \\ &+ r_{ek}\frac{H_N}{H}\kappa^2 \mathbb{R}\left[\hat{q}_N^* \hat{\psi}_N\right] + \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}[\hat{q}_n^* \widehat{\mathrm{ssd}}_n],\end{aligned}$$

where the terms above on the right represent, from left to right,

I: The spectral divergence of barotropic potential enstrophy flux;

II: The spectrum of barotropic potential enstrophy generation;

III: The spectrum of barotropic potential enstrophy loss due to bottom friction;

IV: The spectrum of barotropic potential enstrophy loss due to small scale dissipation.

The enstrophy dissipation is concentrated at the smallest scales resolved in the model and, in statistical steady state, we expect the budget above to be dominated by the balance between I and II.

1.2.3 Special case: two-layer model

With $N = 2$ (see *Equations For Two-Layer QG Model*), an alternative notation for the perturbation of potential vorticities can be written as

$$q_1 = \nabla^2\psi_1 + F_1(\psi_2 - \psi_1)$$
$$q_2 = \nabla^2\psi_2 + F_2(\psi_1 - \psi_2)\,,$$

where we use the following definitions where

$$F_1 \equiv \frac{k_d^2}{1+\delta}\,, \qquad \text{and} \qquad F_2 \equiv \delta\, F_1\,,$$

with the deformation wavenumber

$$k_d^2 \equiv \frac{f_0^2}{g}\frac{H_1 + H_2}{H_1 H_2}\,.$$

With this notation, the stretching matrix is simply

$$\mathbf{S} = \begin{bmatrix} -F_1 & F_1 \\ F_2 & -F_2 \end{bmatrix}.$$

The inversion relationship in Fourier space is

$$\begin{pmatrix} \widehat{\psi}_1 \\ \widehat{\psi}_2 \end{pmatrix} = -\frac{1}{\kappa^2(\kappa^2 + F_1 + F_2)} \begin{bmatrix} \kappa^2 + F_2 & F_1 \\ F_2 & \kappa^2 + F_1 \end{bmatrix} \begin{pmatrix} \widehat{q}_1 \\ \widehat{q}_2 \end{pmatrix}.$$

Substituting the inversion relationship to the rate of change of the energy spectrum above, we have

$$\begin{aligned} \frac{\partial E(k,l)}{\partial t} =& \mathbb{R}\left[(\delta_1\hat{\psi}_1^*, \delta_2\hat{\psi}_2^*) \cdot \begin{pmatrix} \hat{J}(\psi_1, q_1) + ik\beta_1\hat{\psi}_1 + ikU_1\hat{q}_1 \\ \hat{J}(\psi_2, q_2) + ik\beta_2\hat{\psi}_2 + ikU_2\hat{q}_2 - r_{ek}\kappa^2\hat{\psi}_2 \end{pmatrix} \right] \\ =& \sum_{n=1}^{2} \delta_n \mathbb{R}\left[\hat{\psi}_n^* \hat{J}(\psi_n, \nabla^2\psi_n)\right] + \delta_1 F_1 \mathbb{R}\left[(\hat{\psi}_1^* - \hat{\psi}_2^*)\hat{J}(\psi_1, \psi_2)\right] \\ &+ \delta_1 F_1 k(U_1 - U_2)\mathbb{R}\left[i\hat{\psi}_1^*\hat{\psi}_2\right] - r_{ek}\delta_2\kappa^2|\hat{\psi}_2|^2, \end{aligned}$$

in which the right-hand-side terms are, from left to right, the spectral divergence of kinetic energy flux, the spectral divergence of potential energy flux, the spectrum of available potential energy generation, and the spectral contribution by bottom drag. Note that we neglected the contribution from eddy viscosity (spectral filter), but they have the same form as the multi-layer case above.

1.2.4 Vertical modes

Standard vertical modes, , $\mathsf{p}_n(z)$, are the eigenvectors of the "stretching matrix"

$$\mathsf{S}\,\mathsf{p}_n = -R_n^{-2}\,\mathsf{p}_n\,,$$

where the R_n is by definition the n'th deformation radius (e.g., Flierl 1978). These orthogonal modes p_n are normalized to have unitary $L2$-norm

$$\frac{1}{H}\int_{-H}^{0} \mathsf{p}_n \mathsf{p}_m \mathrm{d}z = \delta_{nm}\,,$$

where δ_{mn}.

1.2.5 Linear stability analysis

With $h_b = 0$, the linear eigenproblem is

$$\mathsf{A}\,\Phi = \omega\,\mathsf{B}\,\Phi\,,$$

where

$$\mathsf{A} \equiv \mathsf{B}(\mathsf{U}\,k + \mathsf{V}\,l) + \mathsf{I}\,(k\,\mathsf{Q}_y - l\,\mathsf{Q}_x) + \mathsf{I}\,\delta_{\mathsf{NN}}\,\mathrm{i}\,r_{ek}\,\kappa^2\,,$$

where $\delta_{\mathsf{NN}} = [0, 0, \dots, 0, 1]$, and

$$\mathsf{B} \equiv \mathsf{S} - \mathsf{I}\kappa^2\,.$$

The growth rate is $\mathrm{Im}\{\omega\}$.

1.2.6 Equations For Equivalent Barotropic QG Model

The equivalent barotropic quasigeostrophy evolution equations is

$$\partial_t\,q + \mathsf{J}\,(\psi\,, q) + \beta\,\psi_x = \mathrm{ssd}\,.$$

The potential vorticity anomaly is

$$q = \nabla^2\psi - \kappa_d^2\psi\,,$$

where κ_d^2 is the deformation wavenumber. With $\kappa_d = \beta = 0$ we recover the 2D vorticity equation.

The inversion relationship in Fourier space is

$$\widehat{q} = -\left(\kappa^2 + \kappa_d^2\right)\widehat{\psi}\,.$$

The system is marched forward in time similarly to the two-layer model.

1.2.7 Surface Quasi-geostrophic Model

Surface quasi-geostrophy (SQG) is a relatively simple model that describes surface intensified flows due to buoyancy. One of it's advantages is that it only has two spatial dimensions but describes a three-dimensional solution.

The evolution equation is

$$\partial_t b + \mathsf{J}(\psi, b) = 0\,, \qquad \text{at} \qquad z = 0\,,$$

where $b = \psi_z$ is the buoyancy.

The interior potential vorticity is zero. Hence

$$\frac{\partial}{\partial z}\left(\frac{f_0^2}{N^2}\frac{\partial\psi}{\partial z}\right) + \nabla^2\psi = 0\,,$$

where N is the buoyancy frequency and f_0 is the Coriolis parameter. In the SQG model both N and f_0 are constants. The boundary conditions for this elliptic problem in a semi-infinite vertical domain are

$$b = \psi_z\,, \qquad \text{and} \qquad z = 0\,,$$

and

$$\psi = 0, \qquad \text{at} \qquad z \to -\infty\,.$$

The solutions to the elliptic problem above*, in horizontal Fourier space, gives the inversion relationship between surface buoyancy and surface streamfunction

$$\widehat{\psi} = \frac{f_0}{N}\frac{1}{\kappa}\widehat{b}\,, \qquad \text{at} \qquad z = 0\,.$$

The SQG evolution equation is marched forward similarly to the two-layer model.

---

* Since understanding this step is key to making your own modifications to the model, in more detail:

$$\frac{\partial}{\partial z}\left(\frac{f_0^2}{N^2}\frac{\partial\psi(x,y,z)}{\partial z}\right) + \nabla^2\psi(x,y,z) = 0$$

Taking the Fourier transform in the x and y directions with $\kappa^2 = k^2 + l^2$ we get

$$\frac{f_0^2}{N^2}\frac{\partial}{\partial z}\left(\frac{\partial\hat{\psi}}{\partial z}\right) = \kappa^2\hat{\psi}\,,$$

which has solution

$$\hat{\psi} = Ae^{\frac{\kappa N}{f_0}z} + Be^{-\frac{\kappa N}{f_0}z}\,,.$$

Our decay at negative infinity immediately tells us that $B = 0$. Differentiating with respect to z and evaluating at the surface tells us $A = f_0\hat{b}/\kappa N$ so that we have:

$$\hat{\psi}(k,l,z) = \frac{f_0}{N}\frac{1}{\kappa}\hat{b}(k,l,z)e^{\frac{\kappa N}{f_0}z}\,,.$$

Evaluating at $z = 0$ gives the inversion relation given above.

1.2.8 Parameterizations

pyqg support parameterizations, which are functions that take a `pyqg.Model` and return an additional term to add to its potential vorticity tendency every timestep (or two terms to add to each velocity tendency, in which case we apply them to PV after taking their curl). Typically, parameterizations are used to account for the contribution of phenomena occuring at subgrid scales. This approach can be a computationally efficient way to improve the physical realism of simulations without needing to increase their spatial resolution (which can be very expensive).

Using predefined parameterizations

pyqg implements a number of predefined parameterizations (see *Parameterizations* for a full list). You can use these in a `pyqg.Model` as follows:

```
param = pyqg.BackscatterBiharmonic(smag_constant=0.1, back_constant=0.95)
model = pyqg.QGModel(parameterization=param)
```

Note that parameterizations either target the tendencies of potential vorticity q or the velocities u and v. If you have two parameterizations with the same target, you can add them together, even as a weighted sum. If they have different targets, you can still use both, but they must be passed in as separate `q_parameterization` and `uv_parameterization` arguments:

```
param1 = pyqg.Smagorinsky() # targets uv
param2 = pyqg.ZannaBolton2020() # also targets uv
good_model = pyqg.QGModel(parameterization=param1 + 0.25*param2) # this works!

param3 = pyqg.BackscatterBiharmonic() # targets q
bad_model = pyqg.QGModel(parameterization=param1 + param3) # this will error!

# do this instead to combine parameterizations of different types
good_model2 = pyqg.QGModel(uv_parameterization=param1, q_parameterization=param3)
```

Defining new parameterizations

To define a new parameterization, you have two options. The first is just to define a Python function which takes a single argument (the model) and returns either a single real array of size `(nz, ny, nz)` if it targets q or an iterable of two such arrays if it targets u and v. This can then be passed to the model using the type-specific arguments:

```
# These parameterizations just add random noise, but with the right shape
noisy_q_param = lambda model: np.random.normal(size=model.q.shape)
noisy_uv_param = lambda model: np.random.normal(size=(2, *model.u.shape))

model1 = pyqg.QGModel(q_parameterization=noisy_q_param)
model2 = pyqg.QGModel(uv_parameterization=noisy_uv_param)
```

The second (and usually better) option is to define a subclass of `pyqg.UVParameterization` or `pyqg.QParameterization` with a new definition of `__call__`:

```
class NoisyQParam(pyqg.QParameterization):
    def __init__(self, scale):
        self.scale = scale

    def __call__(self, model):
        return np.random.normal(size=model.q.shape) * self.scale
```

If you would like to make your parameterization available for others to test, please consider *Contributing your parameterization to pyqg*.

Parameterization diagnostics

Parameterizations of potential vorticity affect how energy is redistributed across scales according to the following formula:

$$\left(\frac{\partial E(k,l)}{\partial t}\right)^{\text{param}} = -\frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\hat{\psi}_n^* \hat{q}_n^{\text{param}}\right],$$

The contribution of velocity parameterizations is analogous, except with $\hat{q}^{\text{param}}$ replaced by the curl of the velocity tendency terms in spectral space. This term is made available in the diagnostics under `paramspec`.

In the case of a quasi-geostrophic model, the `paramspec` can be decomposed into two terms which represent its contribution to the kinetic and available potential energy tendencies:

$$\left(\frac{\partial \mathrm{KE}(k,l)}{\partial t}\right)^{\text{param}} = \frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\kappa^2 \hat{\psi}_n^* \left(\mathbf{A}\hat{\mathbf{q}}^{\text{param}}\right)_n\right] \tag{1.2}$$

$$\left(\frac{\partial \mathrm{APE}(k,l)}{\partial t}\right)^{\text{param}} = -\frac{1}{H}\sum_{n=1}^{N} H_n \mathbb{R}\left[\hat{\psi}_n^* \left(\mathbf{S}\mathbf{A}\hat{\mathbf{q}}^{\text{param}}\right)_n\right] \tag{1.3}$$

where $\mathbf{A}(\mathbf{k}) = (\mathbf{S} - \kappa^2\mathbf{I})^{-1}$ and $\mathbf{S}$ is the model's stretching matrix (more details *here*).

We make these terms available in the diagnostics under `paramspec_KEflux` and `paramspec_APEflux`, respectively. When comparing the KE and APE fluxes of parameterized and unparameterized models, it may make sense to do so after adding these terms to the raw `KEflux` and `APEflux` values.

Evaluating subgrid parameterizations

As many parameterizations attempt to account for missing physics due to low resolution, we provide several helper methods for evaluating them.

Assume we have run a high-resolution model and both parameterized and unparameterized low-resolution models. We provide helper methods to compare the root mean squared difference in their resulting diagnostics (properly adding, e.g., `KEflux` and `paramspec_KEflux`), and even compute similarity metrics describing how much closer each of the parameterized model's diagnostics are to those of the high-resolution model as compared to those of the low-resolution model:

```
from pyqg.diagnostic tools import diagnostic_differences, diagnostic_similarities

m_highres = pyqg.QGModel(nx=256)
m_lowres = pyqg.QGModel(nx=64)
m_param = pyqg.QGModel(nx=64, parameterization=pyqg.BackscatterBiharmonic())
[m.run() for m in [m_highres, m_lowres, m_param]]

highres_lowres_diffs = diagnostic_differences(m_highres, m_lowres)
highres_param_diffs = diagnostic_differences(m_highres, m_param)

param_similarity = diagnostic_similarities(m_param,
                                           target=m_highres,
                                           baseline=m_lowres)
```

The `target` does not need to be a high-resolution model, but regardless, similarity scores near 1 indicate that the parameterization's diagnostics are much closer to the `target` than the `baseline`, while scores below 0 indicate they are further from the `target` than the `baseline`.

Contributing your parameterization to pyqg

We encourage contributions of parameterizations to pyqg for others to test. To add yours, please:

1. Define it as a subclass of `pyqg.UVParameterization` or `pyqg.QParameterization` *as described above*.
2. Add the code either to `pyqg/parameterizations.py` or a new file imported in `pyqg/__init__.py`.
3. Write a test ensuring it can be evaluated for the appropriate model classes.
4. Create or update a notebook in `docs/examples` to illustrate its effects or compare it to other parameterizations (optional but encouraged).
5. Create a pull request following the *normal development workflow*.

1.3 Examples

1.3.1 Two-Layer QG Model Example

Here is a quick overview of how to use the two-layer model. See the :py:class:`pyqg.QGModel` api documentation for further details.

First import numpy, matplotlib, and pyqg:

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline

import pyqg
from pyqg import diagnostic_tools as tools
```

Initialize and Run the Model

Here we set up a model which will run for 10 years and start averaging after 5 years. There are lots of parameters that can be specified as keyword arguments but we are just using the defaults.

[2]:
```
year = 24*60*60*360.
m = pyqg.QGModel(tmax=10*year, twrite=10000, tavestart=5*year)
m.run()
```

```
INFO:  Logger initialized
INFO: Step: 10000, Time: 7.20e+07, KE: 4.14e-04, CFL: 0.090
INFO: Step: 20000, Time: 1.44e+08, KE: 4.58e-04, CFL: 0.084
INFO: Step: 30000, Time: 2.16e+08, KE: 4.35e-04, CFL: 0.109
INFO: Step: 40000, Time: 2.88e+08, KE: 4.85e-04, CFL: 0.080
```

Convert Model Outpt to an xarray Dataset

Model variables, coordinates, attributes, and metadata can be stored conveniently as an xarray Dataset. (Notice that this feature requires xarray to be installed on your machine. See here for installation instructions: http://xarray.pydata.org/en/stable/getting-started-guide/installing.html#instructions)

[3]:
```
m_ds = m.to_dataset().isel(time=-1)
m_ds
```

[3]:
```
<xarray.Dataset>
Dimensions:              (lev: 2, y: 64, x: 64, l: 64, k: 33, lev_mid: 1)
Coordinates:
    time                 float64 3.11e+08
  * lev                  (lev) int64 1 2
  * lev_mid              (lev_mid) float64 1.5
  * x                    (x) float64 7.812e+03 2.344e+04 ... 9.766e+05 9.922e+05
  * y                    (y) float64 7.812e+03 2.344e+04 ... 9.766e+05 9.922e+05
  * l                    (l) float64 0.0 6.283e-06 ... -1.257e-05 -6.283e-06
  * k                    (k) float64 0.0 6.283e-06 ... 0.0001948 0.0002011
Data variables: (12/32)
    q                    (lev, y, x) float64 -1.822e-06 -1.356e-06 ... -1.087e-06
```

(continues on next page)

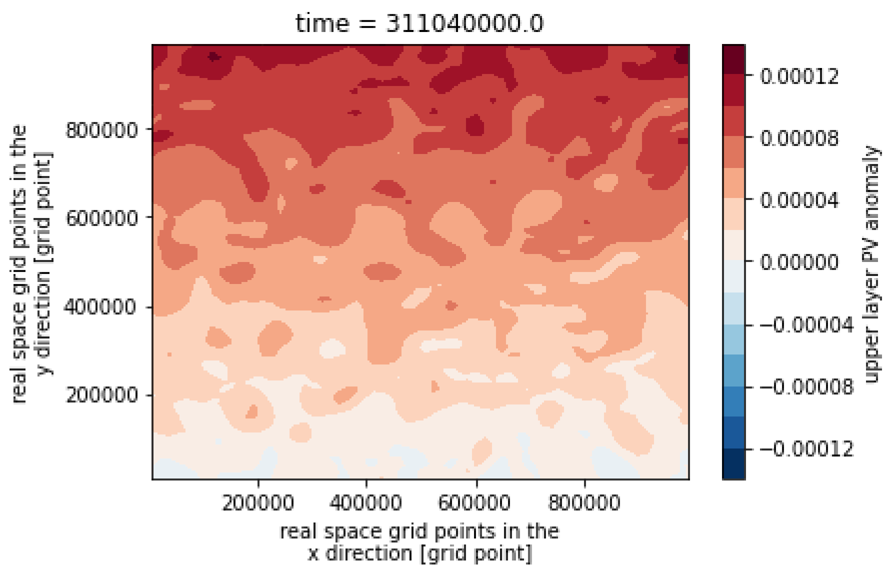
(continued from previous page)

```
    u                   (lev, y, x) float64 -0.05977 -0.04566 ... -0.001317
    v                   (lev, y, x) float64 0.04194 0.0462 ... 0.00118 0.009001
    ufull               (lev, y, x) float64 -0.03477 -0.02066 ... -0.001317
    vfull               (lev, y, x) float64 0.04194 0.0462 ... 0.00118 0.009001
    qh                  (lev, l, k) complex128 (0.002324483338567505+0j) ... (...
    ...                  ...
    ENSgenspec          (l, k) float64 0.0 -3.458e-24 ... 7.51e-52 -3.186e-61
    ENSfrictionspec     (l, k) float64 0.0 -7.479e-24 ... -2.395e-50 -7.94e-60
    APEgenspec          (l, k) float64 0.0 -7.781e-16 ... 1.69e-43 -7.168e-53
    APEflux             (l, k) float64 -0.0 -7.048e-16 ... 1.097e-28 2.951e-33
    KEflux              (l, k) float64 0.0 -4.226e-15 ... 5.188e-27 9.932e-32
    APEgen              float64 6.336e-11
Attributes: (12/23)
    pyqg:beta:        1.5e-11
    pyqg:delta:       0.25
    pyqg:del2:        0.8
    pyqg:dt:          7200.0
    pyqg:filterfac:   23.6
    pyqg:L:           1000000.0
    ...               ...
    pyqg:tc:          43200
    pyqg:tmax:        311040000.0
    pyqg:twrite:      10000
    pyqg:W:           1000000.0
    title:            pyqg: Python Quasigeostrophic Model
    reference:        https://pyqg.readthedocs.io/en/latest/index.html
```

Visualize Output

Let’s assign a new data variable, `q_upper`, as the **upper layer PV anomaly**. We access the PV values in the Dataset as `m_ds.q`, which has two levels and a corresponding background PV gradient, `m_ds.Qy`.

```
[4]: m_ds['q_upper'] = m_ds.q.isel(lev=0) + m_ds.Qy.isel(lev=0)*m_ds.y
     m_ds['q_upper'].attrs = {'long_name': 'upper layer PV anomaly'}
     m_ds.q_upper.plot.contourf(levels=18, cmap='RdBu_r');
```



Plot Diagnostics

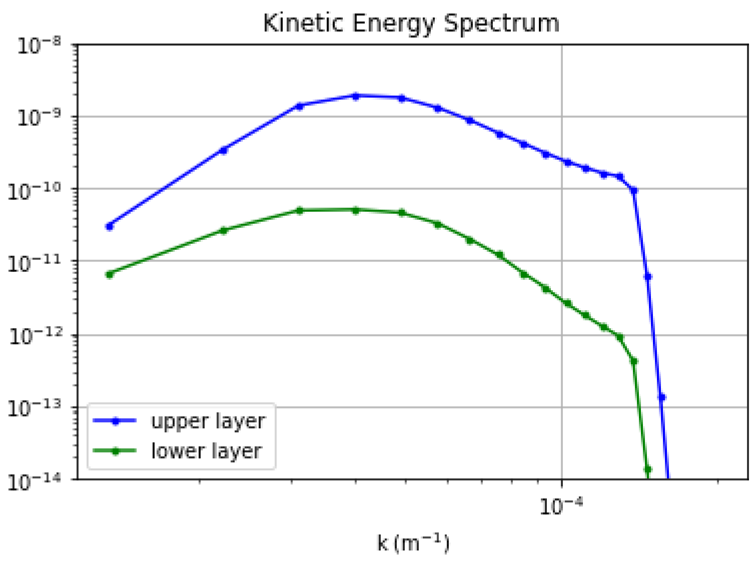
The model automatically accumulates averages of certain diagnostics. We can find out what diagnostics are available by calling

[5]:
```
m.describe_diagnostics()
```

```
NAME                | DESCRIPTION
--------------------------------------------------------------------------------
APEflux     | spectral flux of available potential energy
APEgen      | total available potential energy generation
APEgenspec  | the spectrum of the rate of generation of available potential energy
Dissspec    | Spectral contribution of filter dissipation to total energy
EKE         | mean eddy kinetic energy
EKEdiss     | total energy dissipation by bottom drag
ENSDissspec | Spectral contribution of filter dissipation to barotropic enstrophy
ENSflux     | barotropic enstrophy flux
ENSfrictionspec | the spectrum of the rate of dissipation of barotropic enstrophy due to␣
↪bottom friction
ENSgenspec  | the spectrum of the rate of generation of barotropic enstrophy
Ensspec     | enstrophy spectrum
KEflux      | spectral flux of kinetic energy
KEfrictionspec | total energy dissipation spectrum by bottom drag
KEspec      | kinetic energy spectrum
entspec     | barotropic enstrophy spectrum
paramspec   | Spectral contribution of subgrid parameterization (if present)
paramspec_APEflux | total additional APE flux due to subgrid parameterization
paramspec_KEflux | total additional KE flux due to subgrid parameterization
```

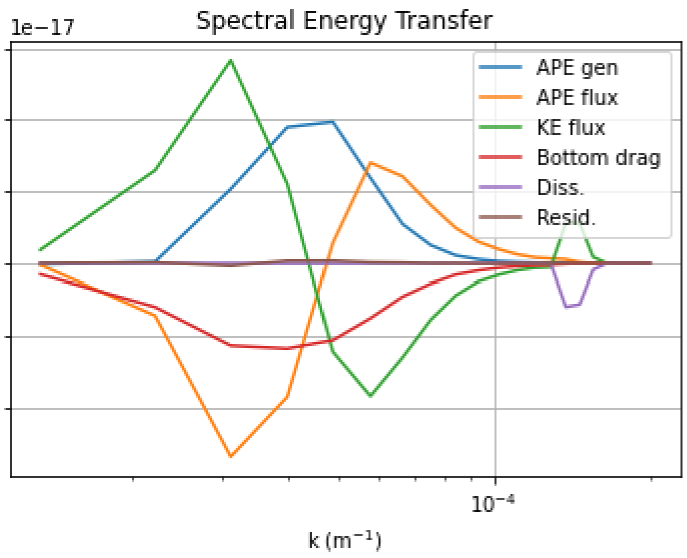
To look at the wavenumber energy spectrum, we plot the `KEspec` diagnostic. (Note that summing along the l-axis, as in this example, does not give us a true *isotropic* wavenumber spectrum.)

```
[6]: kr, kespec_upper = tools.calc_ispec(m, m_ds.KEspec.isel(lev=0).data)
     _, kespec_lower = tools.calc_ispec(m, m_ds.KEspec.isel(lev=1).data)

     plt.loglog(kr, kespec_upper, 'b.-', label='upper layer')
     plt.loglog(kr, kespec_lower, 'g.-', label='lower layer')
     plt.legend(loc='lower left')
     plt.ylim([1e-14,1e-8])
     plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
     plt.title('Kinetic Energy Spectrum');
```

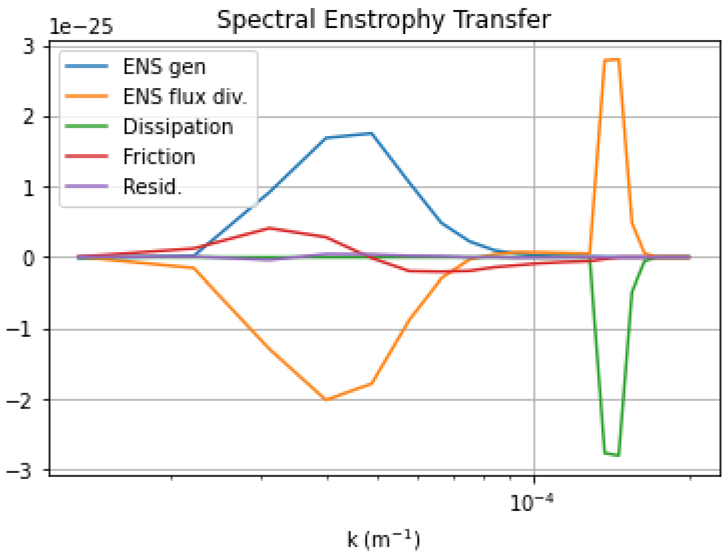


We can also plot the spectral fluxes of energy and enstrophy.

```
[8]: kr, APEgenspec = tools.calc_ispec(m, m_ds.APEgenspec.data)
     _, APEflux     = tools.calc_ispec(m, m_ds.APEflux.data)
     _, KEflux      = tools.calc_ispec(m, m_ds.KEflux.data)
     _, KEfrictionspec = tools.calc_ispec(m, m_ds.KEfrictionspec.data)
     _, Dissspec    = tools.calc_ispec(m, m_ds.Dissspec.data)

     ebud = [ APEgenspec,
              APEflux,
              KEflux,
              KEfrictionspec,
              Dissspec]
     ebud.append(-np.vstack(ebud).sum(axis=0))
     ebud_labels = ['APE gen','APE flux','KE flux','Bottom drag','Diss.','Resid.']
     [plt.semilogx(kr, term) for term in ebud]
     plt.legend(ebud_labels, loc='upper right')
     plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
     plt.title('Spectral Energy Transfer');
```



```
[9]: _, ENSflux     = tools.calc_ispec(m, m_ds.ENSflux.data.squeeze())
_, ENSgenspec = tools.calc_ispec(m, m_ds.ENSgenspec.data.squeeze())
_, ENSfrictionspec = tools.calc_ispec(m, m_ds.ENSfrictionspec.data.squeeze())
_, ENSDissspec = tools.calc_ispec(m, m_ds.ENSDissspec.data.squeeze())

ebud = [ ENSgenspec,
         ENSflux,
         ENSDissspec,
         ENSfrictionspec]
ebud.append(-np.vstack(ebud).sum(axis=0))
ebud_labels = ['ENS gen','ENS flux div.','Dissipation','Friction','Resid.']
[plt.semilogx(kr, term) for term in ebud]
plt.legend(ebud_labels, loc='best')
plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
plt.title('Spectral Enstrophy Transfer');
```



1.3.2 Fully developed baroclinic instability of a 3-layer flow

[1]:
```
import numpy as np
from numpy import pi
from matplotlib import pyplot as plt

import pyqg
from pyqg import diagnostic_tools as tools
```

Set up

[2]:
```
L =  1000.e3     # length scale of box    [m]
Ld = 15.e3       # deformation scale      [m]
kd = 1./Ld       # deformation wavenumber [m^-1]
Nx = 64          # number of grid points

H1 = 500.        # layer 1 thickness  [m]
H2 = 1750.       # layer 2
H3 = 1750.       # layer 3

U1 = 0.05          # layer 1 zonal velocity [m/s]
U2 = 0.025         # layer 2
U3 = 0.00          # layer 3

rho1 = 1025.
rho2 = 1025.275
rho3 = 1025.640

rek = 1.e-7       # linear bottom drag coeff.  [s^-1]
f0  = 0.0001236812857687059 # coriolis param [s^-1]
beta = 1.2130692965249345e-11 # planetary vorticity gradient [m^-1 s^-1]

Ti = Ld/(abs(U1))  # estimate of most unstable e-folding time scale [s]
dt = Ti/200.   # time-step [s]
tmax = 500*Ti      # simulation time [s]
```

[3]:
```
m = pyqg.LayeredModel(nx=Nx, nz=3, U = [U1,U2,U3],V = [0.,0.,0.],L=L,f=f0,beta=beta,
                         H = [H1,H2,H3], rho=[rho1,rho2,rho3],rek=rek,
                        dt=dt,tmax=tmax, twrite=10000, tavestart=Ti*200)
```

```
INFO:  Logger initialized
```

Initial condition

```
[4]: sig = 1.e-7
     qi = sig*np.vstack([np.random.randn(m.nx,m.ny)[np.newaxis,],
                         np.random.randn(m.nx,m.ny)[np.newaxis,],
                         np.random.randn(m.nx,m.ny)[np.newaxis,]])
     m.set_q(qi)
```

Run the model

```
[5]: m.run()
```

```
INFO: Step: 10000, Time: 1.50e+07, KE: 2.38e-04, CFL: 0.009
INFO: Step: 20000, Time: 3.00e+07, KE: 3.59e-02, CFL: 0.126
INFO: Step: 30000, Time: 4.50e+07, KE: 1.70e-01, CFL: 0.191
INFO: Step: 40000, Time: 6.00e+07, KE: 3.59e-01, CFL: 0.213
INFO: Step: 50000, Time: 7.50e+07, KE: 1.91e-01, CFL: 0.192
INFO: Step: 60000, Time: 9.00e+07, KE: 2.24e-01, CFL: 0.207
INFO: Step: 70000, Time: 1.05e+08, KE: 5.46e-01, CFL: 0.307
INFO: Step: 80000, Time: 1.20e+08, KE: 4.94e-01, CFL: 0.252
INFO: Step: 90000, Time: 1.35e+08, KE: 5.62e-01, CFL: 0.210
INFO: Step: 100000, Time: 1.50e+08, KE: 2.70e-01, CFL: 0.218
```

Xarray Dataset

Notice that the conversion to an xarray dataset requires xarray to be installed on your machine. See here for installation instructions: http://xarray.pydata.org/en/stable/getting-started-guide/installing.html#instructions

```
[6]: ds = m.to_dataset()
     ds
```

```
[6]: <xarray.Dataset>
     Dimensions:           (time: 1, lev: 3, y: 64, x: 64, l: 64, k: 33, lev_mid: 2)
     Coordinates:
       * time              (time) float64 1.5e+08
       * lev               (lev) int64 1 2 3
       * lev_mid           (lev_mid) float64 1.5 2.5
       * x                 (x) float64 7.812e+03 2.344e+04 ... 9.766e+05 9.922e+05
       * y                 (y) float64 7.812e+03 2.344e+04 ... 9.766e+05 9.922e+05
       * l                 (l) float64 0.0 6.283e-06 ... -1.257e-05 -6.283e-06
       * k                 (k) float64 0.0 6.283e-06 ... 0.0001948 0.0002011
     Data variables: (12/34)
         q                 (time, lev, y, x) float64 0.0001383 ... -8.462e-06
         u                 (time, lev, y, x) float64 0.0413 -0.05882 ... -0.1616
         v                 (time, lev, y, x) float64 -0.357 -0.3027 ... -0.1369
         ufull             (time, lev, y, x) float64 0.0913 -0.008824 ... -0.1616
         vfull             (time, lev, y, x) float64 -0.357 -0.3027 ... -0.1369
         qh                (time, lev, l, k) complex128 (9.956089311189431e-06+0j...
         ...                ...
         APEgenspec        (time, l, k) float64 0.0 1.724e-08 ... 5.211e-50
         KEspec_modal      (time, lev, l, k) float64 0.0 0.1616 ... 2.567e-41
```

(continues on next page)

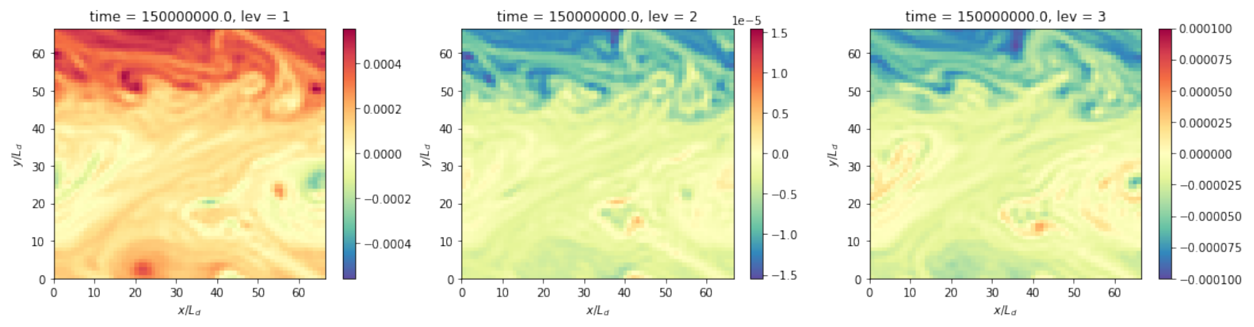
(continued from previous page)

```
    PEspec_modal        (time, lev_mid, l, k) float64 0.0 0.02742 ... 9.974e-42
    APEspec             (time, l, k) float64 0.0 0.04085 ... 2.995e-32 1.117e-41
    KEflux_div          (time, l, k) float64 0.0 6.588e-09 ... 1.817e-24 3.27e-29
    APEflux_div         (time, l, k) float64 0.0 -1.748e-08 ... 1.176e-29
Attributes: (12/24)
    pyqg:beta:        1.2130692965249345e-11
    pyqg:delta:       None
    pyqg:dt:          1500.0
    pyqg:filterfac:   23.6
    pyqg:L:           1000000.0
    pyqg:M:           4096
    ...               ...
    pyqg:tc:          100000
    pyqg:tmax:        150000000.0
    pyqg:twrite:      10000
    pyqg:W:           1000000.0
    title:            pyqg: Python Quasigeostrophic Model
    reference:        https://pyqg.readthedocs.io/en/latest/index.html
```

Snapshots

[7]:
```
PV = ds.q + ds.Qy * ds.y
PV['x'] = ds.x/ds.attrs['pyqg:rd']; PV.x.attrs = {'long_name': r'$x/L_d$'}
PV['y'] = ds.y/ds.attrs['pyqg:rd']; PV.y.attrs = {'long_name': r'$y/L_d$'}
```

[8]:
```
plt.figure(figsize=(18,4))

plt.subplot(131)
PV.sel(lev=1).plot(cmap='Spectral_r')

plt.subplot(132)
PV.sel(lev=2).plot(cmap='Spectral_r')

plt.subplot(133)
PV.sel(lev=3).plot(cmap='Spectral_r');
```



pyqg has a built-in method that computes the vertical modes. It is stored as an attribute in the Dataset.

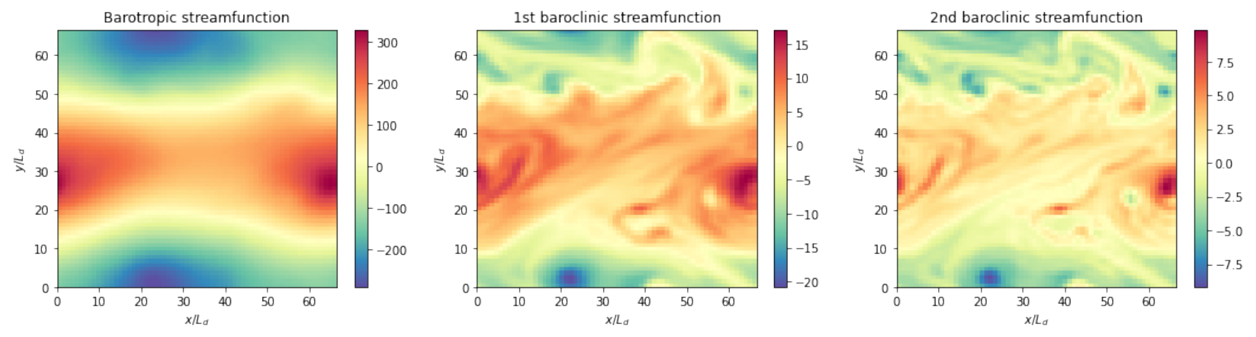
[9]:
```
print(f"The first baroclinic deformation radius is {ds.attrs['pyqg:radii'][1]/1.e3} km")
print(f"The second baroclinic deformation radius is {ds.attrs['pyqg:radii'][2]/1.e3} km")
```

```
The first baroclinic deformation radius is 15.375382785987185 km
The second baroclinic deformation radius is 7.975516271996243 km
```

We can project the solution onto the modes

```
[10]: pn = m.modal_projection(m.p)
```

```
[11]: plt.figure(figsize=(18,4))

plt.subplot(131)
plt.pcolormesh(m.x/m.rd, m.y/m.rd, pn[0]/(U1*Ld), cmap='Spectral_r', shading='auto')
plt.xlabel(r'$x/L_d$')
plt.ylabel(r'$y/L_d$')
plt.colorbar()
plt.title('Barotropic streamfunction')

plt.subplot(132)
plt.pcolormesh(m.x/m.rd, m.y/m.rd, pn[1]/(U1*Ld), cmap='Spectral_r', shading='auto')
plt.xlabel(r'$x/L_d$')
plt.ylabel(r'$y/L_d$')
plt.colorbar()
plt.title('1st baroclinic streamfunction')

plt.subplot(133)
plt.pcolormesh(m.x/m.rd, m.y/m.rd, pn[2]/(U1*Ld), cmap='Spectral_r', shading='auto')
plt.xlabel(r'$x/L_d$')
plt.ylabel(r'$y/L_d$')
plt.colorbar()
plt.title('2nd baroclinic streamfunction');
```



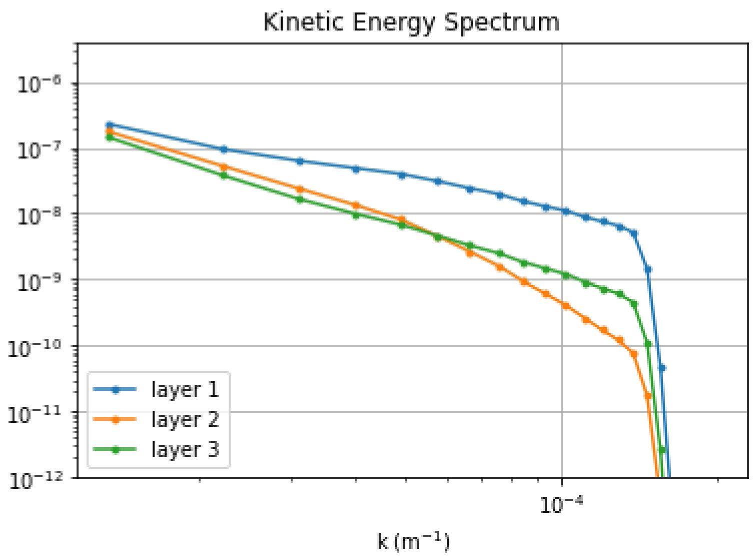
Diagnostics

```
[12]: kr, kespec_1 = tools.calc_ispec(m, ds.KEspec.sel(lev=1).data.squeeze())
_ , kespec_2 = tools.calc_ispec(m, ds.KEspec.sel(lev=2).data.squeeze())
_ , kespec_3 = tools.calc_ispec(m, ds.KEspec.sel(lev=3).data.squeeze())

plt.loglog(kr, kespec_1, '.-' )
plt.loglog(kr, kespec_2, '.-' )
plt.loglog(kr, kespec_3, '.-' )
```

(continues on next page)

(continued from previous page)

```
plt.legend(['layer 1','layer 2', 'layer 3'], loc='lower left')
plt.ylim([1e-12,4e-6]);
plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
plt.title('Kinetic Energy Spectrum');
```



By default the modal KE and PE spectra are also calculated

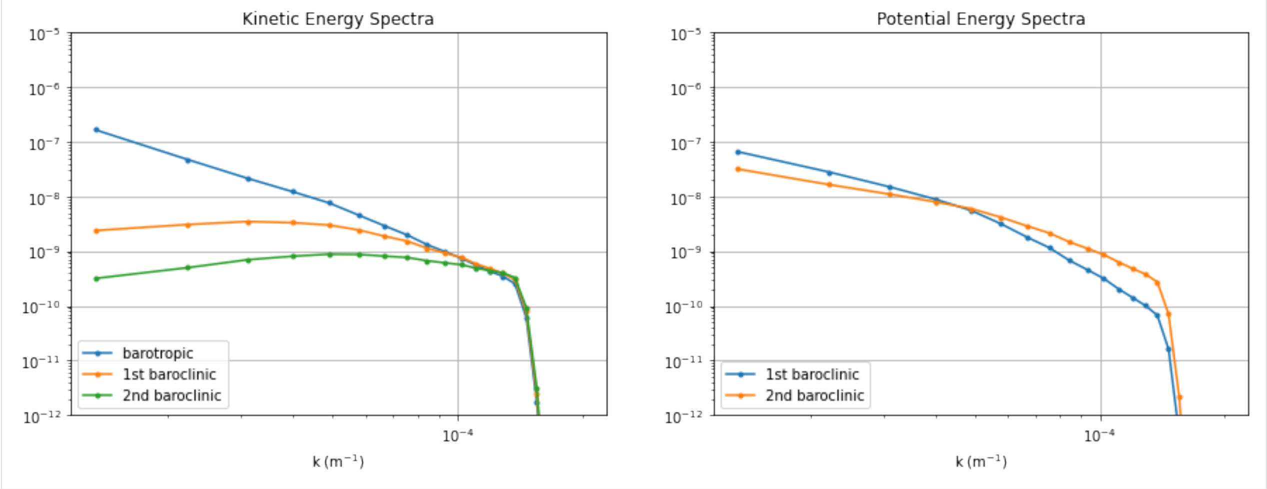
```
[13]: kr, modal_kespec_1 = tools.calc_ispec(m, ds.KEspec_modal.sel(lev=1).data.squeeze())
      _,  modal_kespec_2 = tools.calc_ispec(m, ds.KEspec_modal.sel(lev=2).data.squeeze())
      _,  modal_kespec_3 = tools.calc_ispec(m, ds.KEspec_modal.sel(lev=3).data.squeeze())

      _,  modal_pespec_2 = tools.calc_ispec(m, ds.PEspec_modal.sel(lev_mid=1.5).data.squeeze())
      _,  modal_pespec_3 = tools.calc_ispec(m, ds.PEspec_modal.sel(lev_mid=2.5).data.squeeze())
```

```
[14]: plt.figure(figsize=(15,5))

      plt.subplot(121)
      plt.loglog(kr, modal_kespec_1, '.-')
      plt.loglog(kr, modal_kespec_2, '.-')
      plt.loglog(kr, modal_kespec_3, '.-')

      plt.legend(['barotropic ','1st baroclinic', '2nd baroclinic'], loc='lower left')
      plt.ylim([1e-12,1e-5]);
      plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
      plt.title('Kinetic Energy Spectra');


      plt.subplot(122)
      plt.loglog(kr, modal_pespec_2, '.-')
      plt.loglog(kr, modal_pespec_3, '.-')

      plt.legend(['1st baroclinic', '2nd baroclinic'], loc='lower left')
      plt.ylim([1e-12,1e-5]);
```

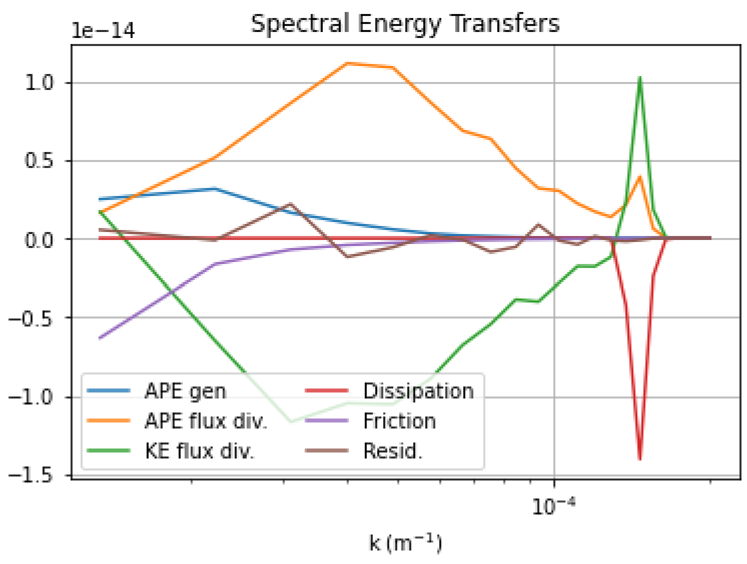
(continues on next page)

(continued from previous page)

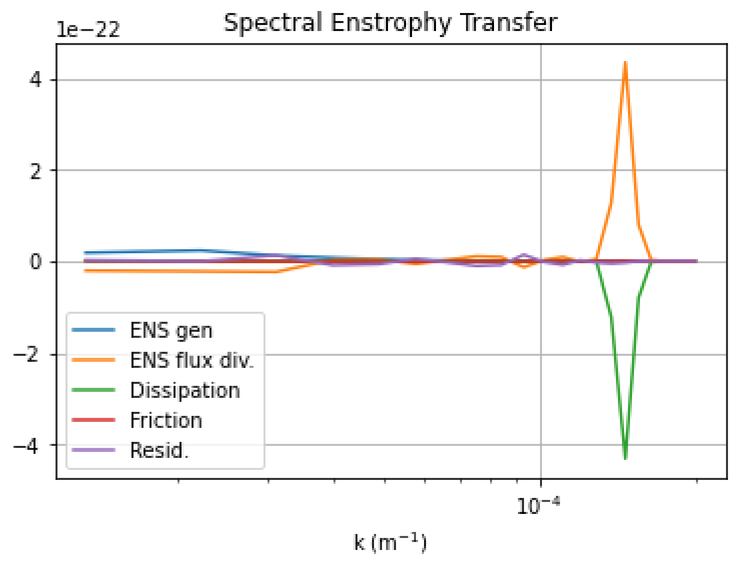
```
plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
plt.title('Potential Energy Spectra');
```



```
[15]: _, APEgenspec = tools.calc_ispec(m, ds.APEgenspec.data.squeeze())
_, APEflux    = tools.calc_ispec(m, ds.APEflux_div.data.squeeze())
_, KEflux     = tools.calc_ispec(m, ds.KEflux_div.data.squeeze())
_, Dissspec   = tools.calc_ispec(m, ds.Dissspec.squeeze().data)
_, KEfrictionspec = tools.calc_ispec(m, ds.KEfrictionspec.squeeze().data)

ebud = [ APEgenspec,
         APEflux,
         KEflux,
         Dissspec,
         KEfrictionspec]
ebud.append(-np.vstack(ebud).sum(axis=0))
ebud_labels = ['APE gen','APE flux div.','KE flux div.','Dissipation','Friction','Resid.
↪']
[plt.semilogx(kr, term) for term in ebud]
plt.legend(ebud_labels, loc='lower left', ncol=2)
plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
plt.title('Spectral Energy Transfers');
```



[16]:
```
_, ENSflux     = tools.calc_ispec(m, ds.ENSflux.data.squeeze())
_, ENSgenspec = tools.calc_ispec(m, ds.ENSgenspec.data.squeeze())
_, ENSfrictionspec = tools.calc_ispec(m, ds.ENSfrictionspec.data.squeeze())
_, ENSDissspec = tools.calc_ispec(m, ds.ENSDissspec.data.squeeze())

ebud = [ ENSgenspec,
         ENSflux,
         ENSDissspec,
         ENSfrictionspec]
ebud.append(-np.vstack(ebud).sum(axis=0))
ebud_labels = ['ENS gen','ENS flux div.','Dissipation','Friction','Resid.']
[plt.semilogx(kr, term) for term in ebud]
plt.legend(ebud_labels, loc='lower left')
plt.xlabel(r'k (m$^{-1}$)'); plt.grid()
plt.title('Spectral Enstrophy Transfer');
```



The dynamics here is similar to the reference experiment of Larichev & Held (1995). The APE generated through baroclinic instability is fluxed towards deformation length scales, where it is converted into KE. The KE the experiments and inverse tranfer, cascading up to the scale of the domain. The mechanical bottom drag essentially removes the large scale KE.

1.3.3 Barotropic Model

Here will will use pyqg to reproduce the results of the paper: J. C. Mcwilliams (1984). The emergence of isolated coherent vortices in turbulent flow. Journal of Fluid Mechanics, 146, pp 21-43 doi:10.1017/S0022112084001750

[23]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pyqg
```

McWilliams performed freely-evolving 2D turbulence ($R_d = \infty$, $\beta = 0$) experiments on a $2\pi \times 2\pi$ periodic box.

[24]:
```
# create the model object
m = pyqg.BTModel(L=2.*np.pi, nx=256,
                 beta=0., H=1., rek=0., rd=None,
                 tmax=40, dt=0.001, taveint=1,
                 ntd=4)
# in this example we used ntd=4, four threads
# if your machine has more (or fewer) cores available, you could try changing it
```

Initial condition

The initial condition is random, with a prescribed spectrum

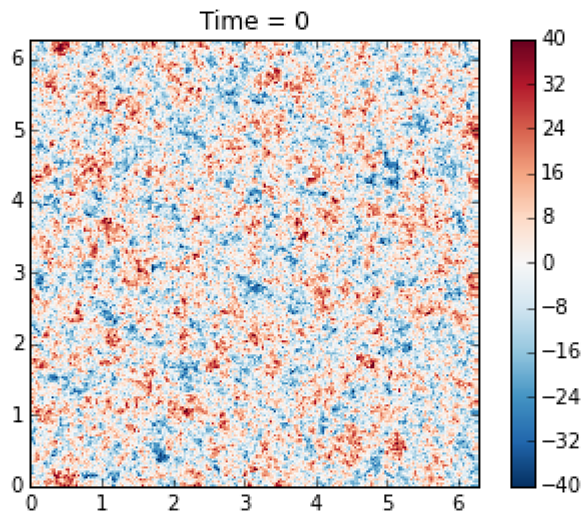
$$|\hat{\psi}|^2 = A\,\kappa^{-1}\left[1 + \left(\frac{\kappa}{6}\right)^4\right]^{-1},$$

where κ is the wavenumber magnitude. The constant A is determined so that the initial energy is $KE = 0.5$.

[3]:
```
# generate McWilliams 84 IC condition

fk = m.wv != 0
ckappa = np.zeros_like(m.wv2)
ckappa[fk] = np.sqrt( m.wv2[fk]*(1. + (m.wv2[fk]/36.)**2) )**-1

nhx,nhy = m.wv2.shape

Pi_hat = np.random.randn(nhx,nhy)*ckappa +1j*np.random.randn(nhx,nhy)*ckappa

Pi = m.ifft( Pi_hat[np.newaxis,:,:] )
Pi = Pi - Pi.mean()
Pi_hat = m.fft( Pi )
KEaux = m.spec_var( m.wv*Pi_hat )

pih = ( Pi_hat/np.sqrt(KEaux) )
qih = -m.wv2*pih
qi = m.ifft(qih)
```

[4]:
```
# initialize the model with that initial condition
m.set_q(qi)
```

[5]:
```
# define a quick function for plotting and visualize the initial condition
def plot_q(m, qmax=40):
    fig, ax = plt.subplots()
    pc = ax.pcolormesh(m.x,m.y,m.q.squeeze(), cmap='RdBu_r')
    pc.set_clim([-qmax, qmax])
    ax.set_xlim([0, 2*np.pi])
    ax.set_ylim([0, 2*np.pi]);
    ax.set_aspect(1)
    plt.colorbar(pc)
    plt.title('Time = %g' % m.t)
    plt.show()

plot_q(m)
```

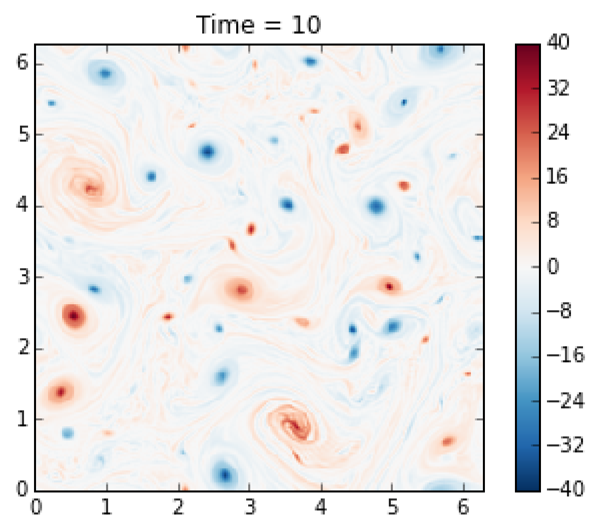


Runing the model

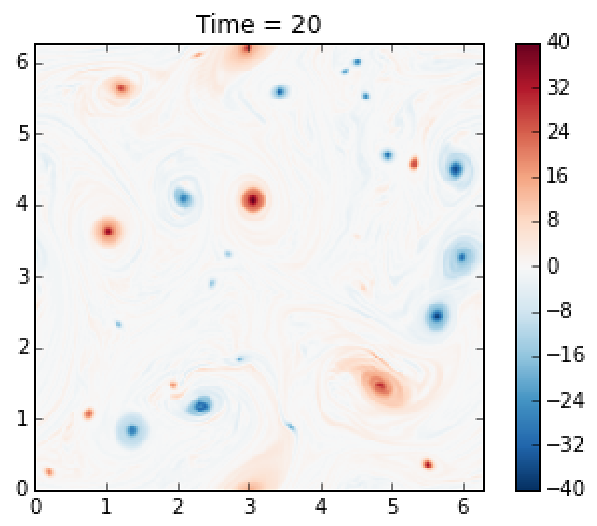
Here we demonstrate how to use the `run_with_snapshots` feature to periodically stop the model and perform some action (in this case, visualization).

[6]:
```
for _ in m.run_with_snapshots(tsnapstart=0, tsnapint=10):
    plot_q(m)
```

```
t=                 1, tc=      1000: cfl=0.104428, ke=0.496432737
t=                 1, tc=      2000: cfl=0.110651, ke=0.495084591
t=                 2, tc=      3000: cfl=0.101385, ke=0.494349348
t=                 3, tc=      4000: cfl=0.113319, ke=0.493862801
t=                 5, tc=      5000: cfl=0.112978, ke=0.493521035
t=                 6, tc=      6000: cfl=0.101435, ke=0.493292057
t=                 7, tc=      7000: cfl=0.092574, ke=0.493114415
t=                 8, tc=      8000: cfl=0.096229, ke=0.492987232
t=                 9, tc=      9000: cfl=0.097924, ke=0.492899499
```



```
t=               9, tc=     10000: cfl=0.103278, ke=0.492830631
t=              10, tc=     11000: cfl=0.102686, ke=0.492775849
t=              11, tc=     12000: cfl=0.099865, ke=0.492726644
t=              12, tc=     13000: cfl=0.110933, ke=0.492679673
t=              13, tc=     14000: cfl=0.102899, ke=0.492648562
t=              14, tc=     15000: cfl=0.102052, ke=0.492622263
t=              15, tc=     16000: cfl=0.106399, ke=0.492595449
t=              16, tc=     17000: cfl=0.122508, ke=0.492569708
t=              17, tc=     18000: cfl=0.120618, ke=0.492507272
t=              19, tc=     19000: cfl=0.103734, ke=0.492474633
```

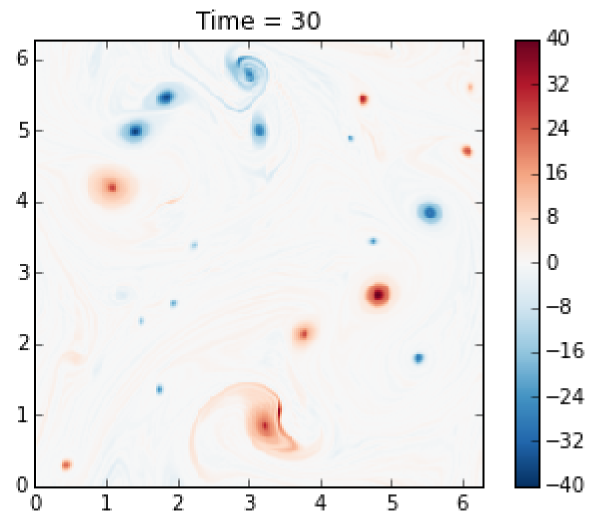


```
t=              20, tc=     20000: cfl=0.113210, ke=0.492452605
t=              21, tc=     21000: cfl=0.095246, ke=0.492439588
t=              22, tc=     22000: cfl=0.092449, ke=0.492429553
t=              23, tc=     23000: cfl=0.115412, ke=0.492419773
t=              24, tc=     24000: cfl=0.125958, ke=0.492407434
t=              25, tc=     25000: cfl=0.098588, ke=0.492396021
t=              26, tc=     26000: cfl=0.103689, ke=0.492387002
```

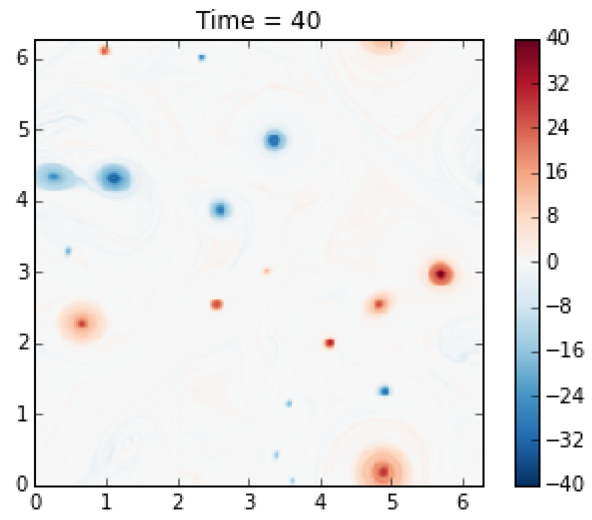
(continues on next page)

(continued from previous page)

```
t=                27, tc=     27000: cfl=0.103893, ke=0.492379606
t=                28, tc=     28000: cfl=0.108417, ke=0.492371082
t=                29, tc=     29000: cfl=0.112969, ke=0.492361675
```



```
t=                30, tc=     30000: cfl=0.127132, ke=0.492352666
t=                31, tc=     31000: cfl=0.122900, ke=0.492331664
t=                32, tc=     32000: cfl=0.110486, ke=0.492317502
t=                33, tc=     33000: cfl=0.101901, ke=0.492302225
t=                34, tc=     34000: cfl=0.099996, ke=0.492294952
t=                35, tc=     35000: cfl=0.106513, ke=0.492290743
t=                36, tc=     36000: cfl=0.121426, ke=0.492286228
t=                37, tc=     37000: cfl=0.125573, ke=0.492283246
t=                38, tc=     38000: cfl=0.108975, ke=0.492280378
t=                38, tc=     39000: cfl=0.110105, ke=0.492278000
```



```
t=                39, tc=     40000: cfl=0.104794, ke=0.492275760
```

The genius of McWilliams (1984) was that he showed that the initial random vorticity field organizes itself into strong coherent vortices. This is true in significant part of the parameter space. This was previously suspected but unproven, mainly because people did not have computer resources to run the simulation long enough. Thirty years later we can perform such simulations in a couple of minutes on a laptop!

Also, note that the energy is nearly conserved, as it should be, and this is a nice test of the model.

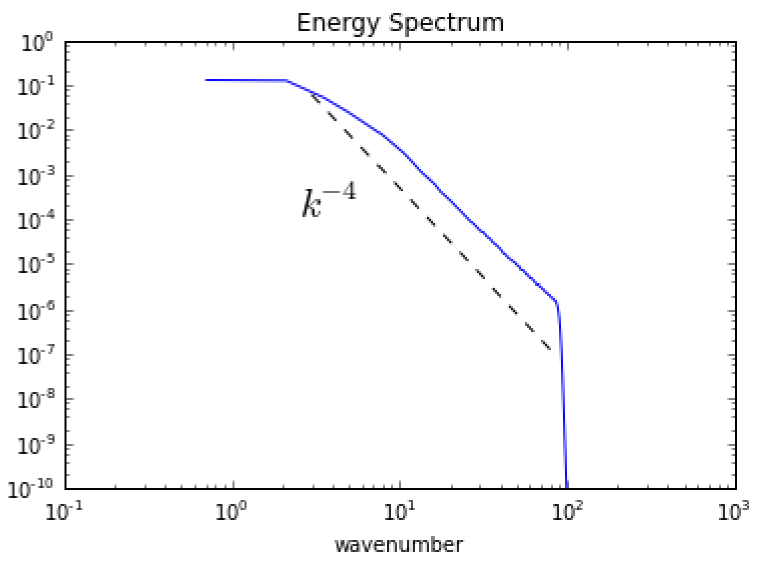
Plotting spectra

```
[7]: energy = m.get_diagnostic('KEspec')
     enstrophy = m.get_diagnostic('Ensspec')
```

```
[11]: # this makes it easy to calculate an isotropic spectrum
      from pyqg import diagnostic_tools as tools
      kr, energy_iso = tools.calc_ispec(m,energy.squeeze())
      _, enstrophy_iso = tools.calc_ispec(m,enstrophy.squeeze())
```

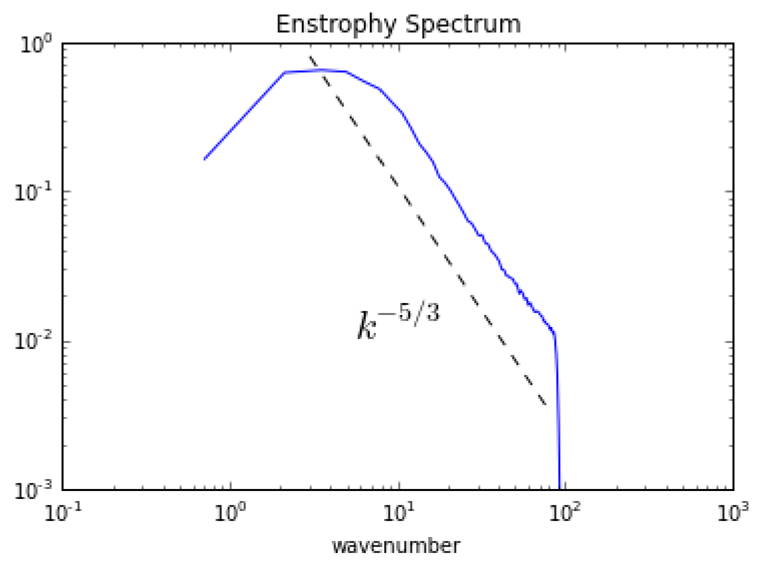
```
[17]: ks = np.array([3.,80])
      es = 5*ks**-4
      plt.loglog(kr,energy_iso)
      plt.loglog(ks,es,'k--')
      plt.text(2.5,.0001,r'$k^{-4}$',fontsize=20)
      plt.ylim(1.e-10,1.e0)
      plt.xlabel('wavenumber')
      plt.title('Energy Spectrum')
```

```
[17]: <matplotlib.text.Text at 0x10c1b1a90>
```



```
[20]: ks = np.array([3.,80])
      es = 5*ks**(-5./3)
      plt.loglog(kr,enstrophy_iso)
      plt.loglog(ks,es,'k--')
      plt.text(5.5,.01,r'$k^{-5/3}$',fontsize=20)
      plt.ylim(1.e-3,1.e0)
      plt.xlabel('wavenumber')
      plt.title('Enstrophy Spectrum')
```

[20]:
```
<matplotlib.text.Text at 0x10b5d2f50>
```



[ ]:

1.3.4 Surface Quasi-Geostrophic (SQG) Model

Here will will use pyqg to reproduce the results of the paper: I. M. Held, R. T. Pierrehumbert, S. T. Garner and K. L. Swanson (1985). Surface quasi-geostrophic dynamics. Journal of Fluid Mechanics, 282, pp 1-20 [doi:: http://dx.doi.org/10.1017/S0022112095000012)

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import pi
%matplotlib inline
from pyqg import sqg_model
```

```
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 21 days
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 21 days
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 21 days
```

Surface quasi-geostrophy (SQG) is a relatively simple model that describes surface intensified flows due to buoyancy. One of it's advantages is that it only has two spatial dimensions but describes a three-dimensional solution.

If we define b to be the buoyancy, then the evolution equation for buoyancy at each the top and bottom surface is

$$\partial_t b + J(\psi, b) = 0.$$

The invertibility relation between the streamfunction, ψ, and the buoyancy, b, is hydrostatic balance

$$b = \partial_z \psi.$$

Using the fact that the Potential Vorticity is exactly zero in the interior of the domain and that the domain is semi-infinite, yields that the inversion in Fourier space is,

$$\hat{b} = \frac{\kappa N}{f_0} \hat{\psi}.$$

Held et al. studied several different cases, the first of which was the nonlinear evolution of an elliptical vortex. There are several other cases that they studied and people are welcome to adapt the code to study those as well. But here we focus on this first example for pedagogical reasons.

```
# create the model object
year = 1.
m = sqg_model.SQGModel(L=2.*pi,nx=512, tmax = 26.005,
        beta = 0., Nb = 1., H = 1., f_0 = 0., dt = 0.005,
                     taveint=1, twrite=400, ntd=4)
# in this example we used ntd=4, four threads
# if your machine has more (or fewer) cores available, you could try changing it
```

```
INFO:  Logger initialized
INFO:  Kernel initialized
```

Initial condition

The initial condition is an elliptical vortex,

$$b = 0.01 \exp(-(x^2 + (4y)^2)/(L/y)^2$$

where L is the length scale of the vortex in the x direction. The amplitude is 0.01, which sets the strength and speed of the vortex. The aspect ratio in this example is 4 and gives rise to an instability. If you reduce this ratio sufficiently you will find that it is stable. Why don't you try it and see for yourself?

```
# Choose ICs from Held et al. (1995)
# case i) Elliptical vortex
x = np.linspace(m.dx/2,2*np.pi,m.nx) - np.pi
y = np.linspace(m.dy/2,2*np.pi,m.ny) - np.pi
x,y = np.meshgrid(x,y)

qi = -np.exp(-(x**2 + (4.0*y)**2)/(m.L/6.0)**2)
```

```
# initialize the model with that initial condition
m.set_q(qi[np.newaxis,:,:])
```

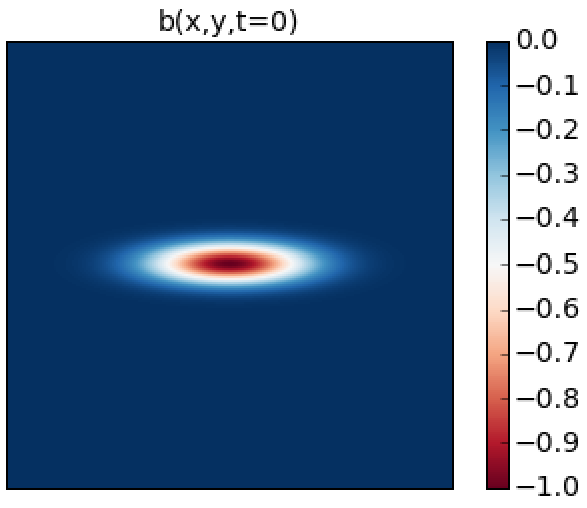
```
# Plot the ICs
plt.rcParams['image.cmap'] = 'RdBu'
plt.clf()
p1 = plt.imshow(m.q.squeeze() + m.beta * m.y)
plt.title('b(x,y,t=0)')
plt.colorbar()
plt.clim([-1, 0])
```

(continues on next page)

(continued from previous page)

```
plt.xticks([])
plt.yticks([])
plt.show()
```

```
/Users/crocha/anaconda/lib/python2.7/site-packages/matplotlib/collections.py:590:␣
↪FutureWarning: elementwise comparison failed; returning scalar instead, but in the␣
↪future will perform elementwise comparison
  if self._edgecolors == str('face'):
```

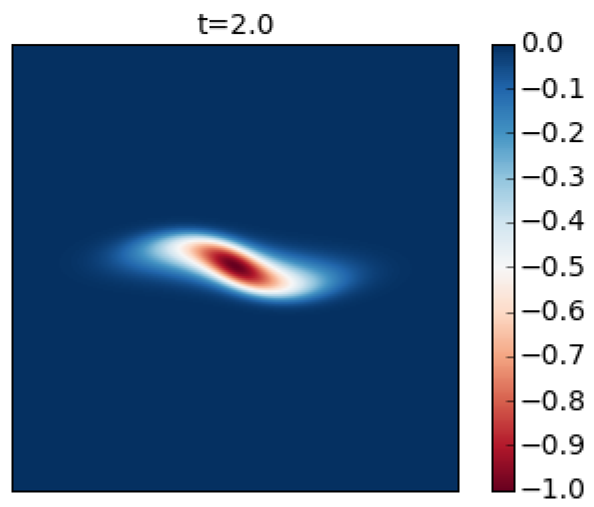


Runing the model

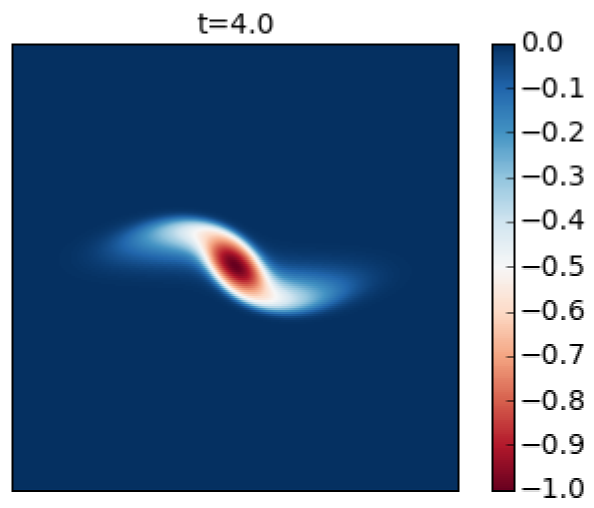
Here we demonstrate how to use the `run_with_snapshots` feature to periodically stop the model and perform some action (in this case, visualization).

```
for snapshot in m.run_with_snapshots(tsnapstart=0., tsnapint=400*m.dt):
    plt.clf()
    p1 = plt.imshow(m.q.squeeze() + m.beta * m.y)
    #plt.clim([-30., 30.])
    plt.title('t='+str(m.t))
    plt.colorbar()
    plt.clim([-1, 0])
    plt.xticks([])
    plt.yticks([])
    plt.show()
```

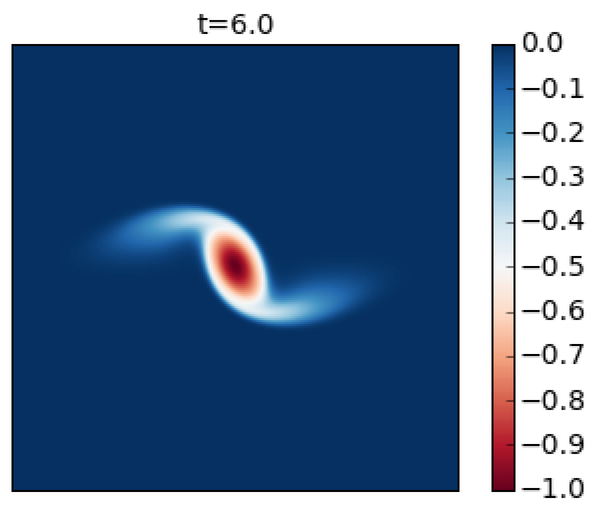
```
INFO: Step: 400, Time: 2.00e+00, KE: 5.21e-03, CFL: 0.245
```



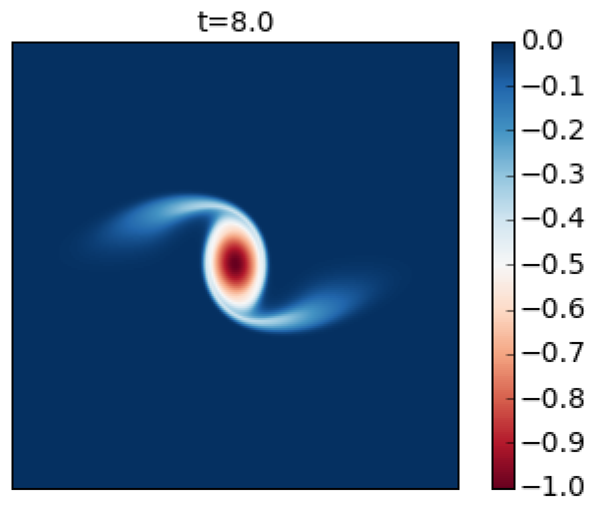
```
INFO: Step: 800, Time: 4.00e+00, KE: 5.21e-03, CFL: 0.239
```



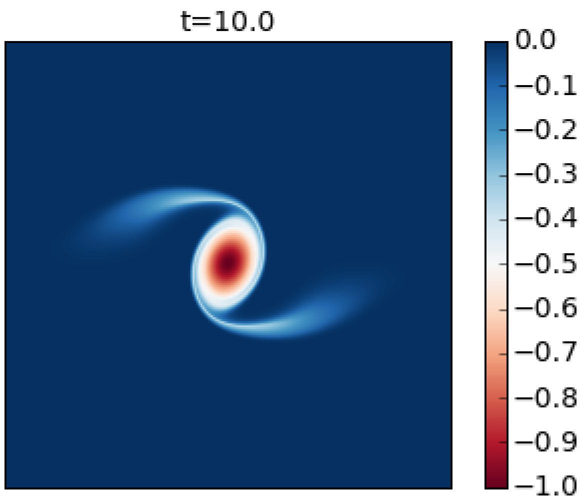
```
INFO: Step: 1200, Time: 6.00e+00, KE: 5.21e-03, CFL: 0.261
```



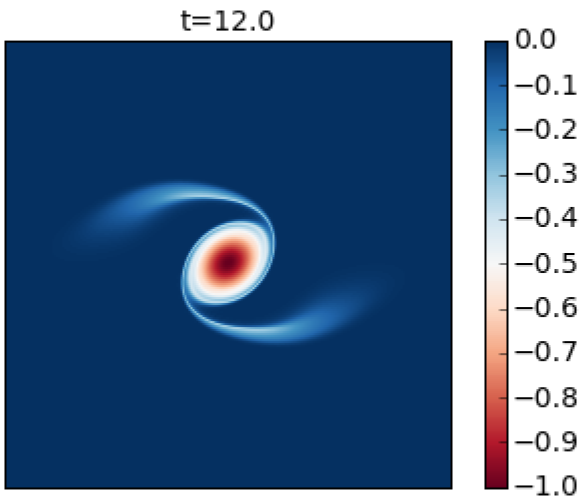
```
INFO: Step: 1600, Time: 8.00e+00, KE: 5.21e-03, CFL: 0.273
```



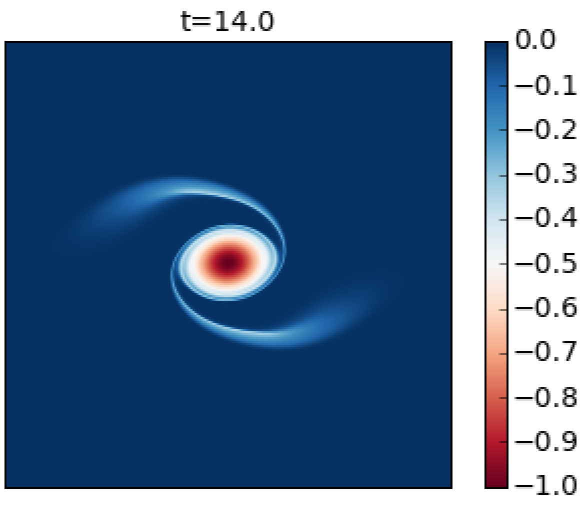
```
INFO: Step: 2000, Time: 1.00e+01, KE: 5.21e-03, CFL: 0.267
```



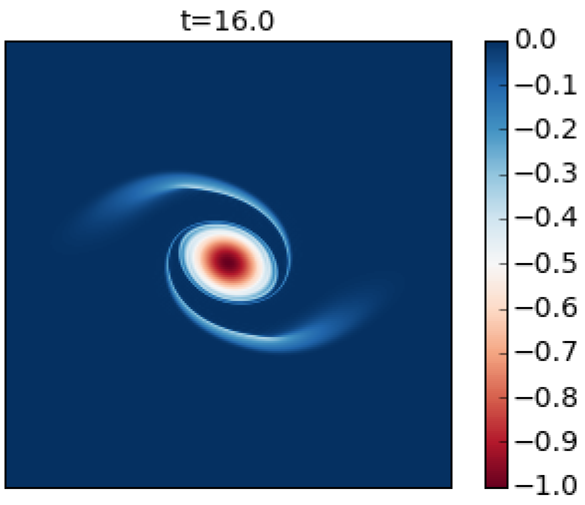
```
INFO: Step: 2400, Time: 1.20e+01, KE: 5.20e-03, CFL: 0.247
```



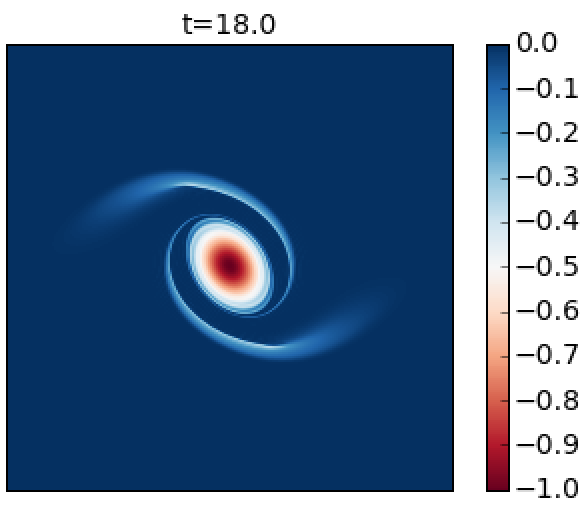
```
INFO: Step: 2800, Time: 1.40e+01, KE: 5.20e-03, CFL: 0.254
```



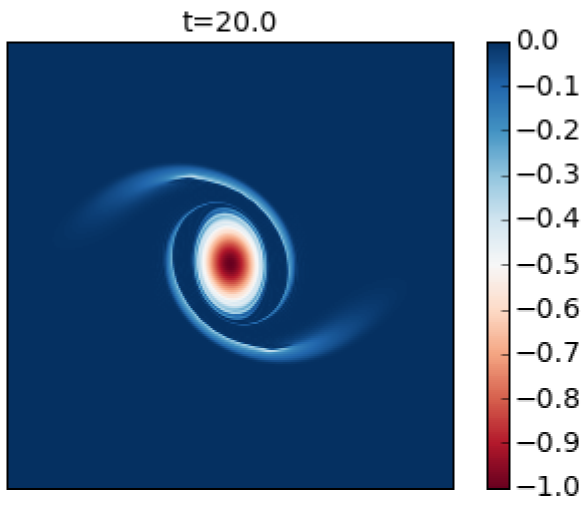
```
INFO: Step: 3200, Time: 1.60e+01, KE: 5.20e-03, CFL: 0.259
```



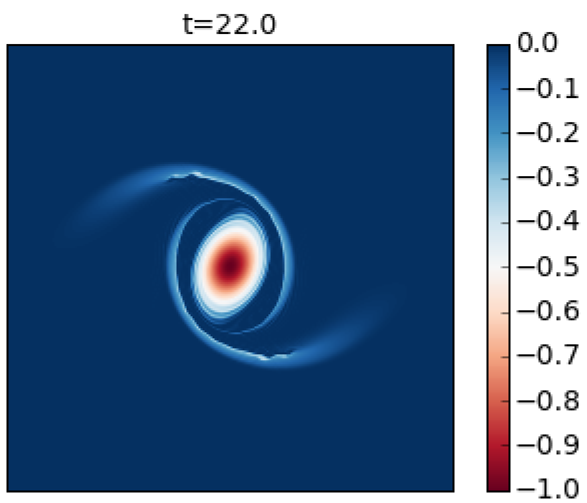
```
INFO: Step: 3600, Time: 1.80e+01, KE: 5.19e-03, CFL: 0.256
```



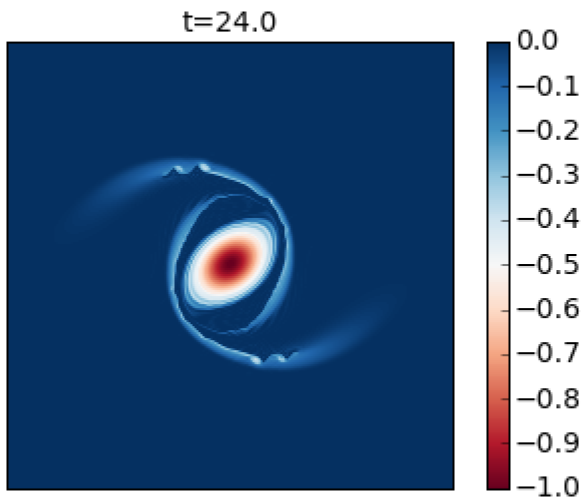
```
INFO: Step: 4000, Time: 2.00e+01, KE: 5.19e-03, CFL: 0.259
```



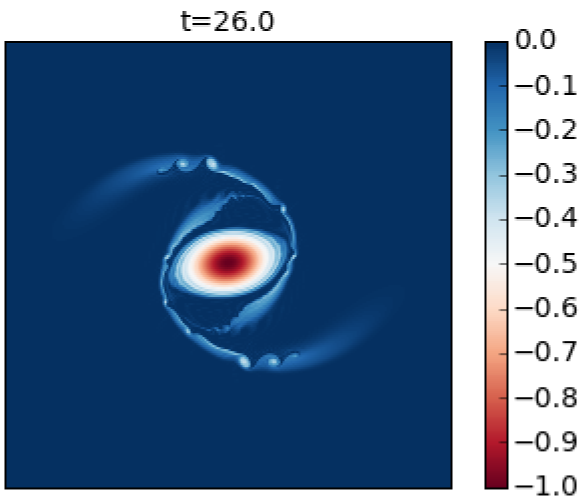
```
INFO: Step: 4400, Time: 2.20e+01, KE: 5.19e-03, CFL: 0.259
```



```
INFO: Step: 4800, Time: 2.40e+01, KE: 5.18e-03, CFL: 0.242
```



```
INFO: Step: 5200, Time: 2.60e+01, KE: 5.17e-03, CFL: 0.263
```



Compare these results with Figure 2 of the paper. In this simulation you see that as the cyclone rotates it develops thin arms that spread outwards and become unstable because of their strong shear. This is an excellent example of how smaller scale vortices can be generated from a mesoscale vortex.

You can modify this to run it for longer time to generate the analogue of their Figure 3.

1.3.5 Built-in linear stability analysis

[11]:
```
import numpy as np
from numpy import pi
import matplotlib.pyplot as plt
%matplotlib inline

import pyqg
```

[12]:
```
m = pyqg.LayeredModel(nx=256, nz = 2, U = [.01, -.01], V = [0., 0.], H = [1., 1.],
                              L=2*pi,beta=1.5, rd=1./20., rek=0.05, f=1.,delta=1.)
```

```
INFO:  Logger initialized
INFO:  Kernel initialized
```

To perform linear stability analysis, we simply call pyqg's built-in method `stability_analysis`:

[3]:
```
evals,evecs = m.stability_analysis()
```

The eigenvalues are stored in `omg`, and the eigenctors in `evec`. For plotting purposes, we use fftshift to reorder the entries

[4]:
```
evals = np.fft.fftshift(evals.imag,axes=(0,))

k,l = m.k*m.radii[1], np.fft.fftshift(m.l,axes=(0,))*m.radii[1]
```

It is also useful to analyze the fasted-growing mode:

[5]:
```
argmax = evals[m.Ny/2,:].argmax()
evec = np.fft.fftshift(evecs,axes=(1))[:,m.Ny/2,argmax]
```

(continues on next page)

(continued from previous page)

```
kmax = k[m.Ny/2,argmax]

x = np.linspace(0,4.*pi/kmax,100)
mag, phase = np.abs(evec), np.arctan2(evec.imag,evec.real)
```

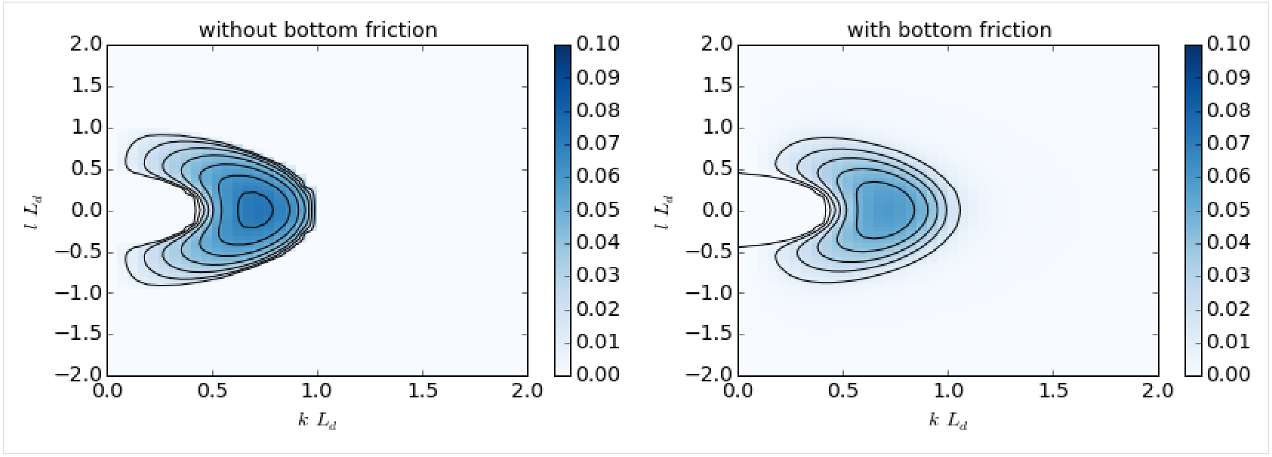
By default, the stability analysis above is performed without bottom friction, but the stability method also supports bottom friction:

[6]:
```
evals_fric, evecs_fric = m.stability_analysis(bottom_friction=True)
evals_fric = np.fft.fftshift(evals_fric.imag,axes=(0,))

argmax = evals_fric[m.Ny/2,:].argmax()
evec_fric = np.fft.fftshift(evecs_fric,axes=(1))[:,m.Ny/2,argmax]
kmax_fric = k[m.Ny/2,argmax]

mag_fric, phase_fric = np.abs(evec_fric), np.arctan2(evec_fric.imag,evec_fric.real)
```

Plotting growth rates

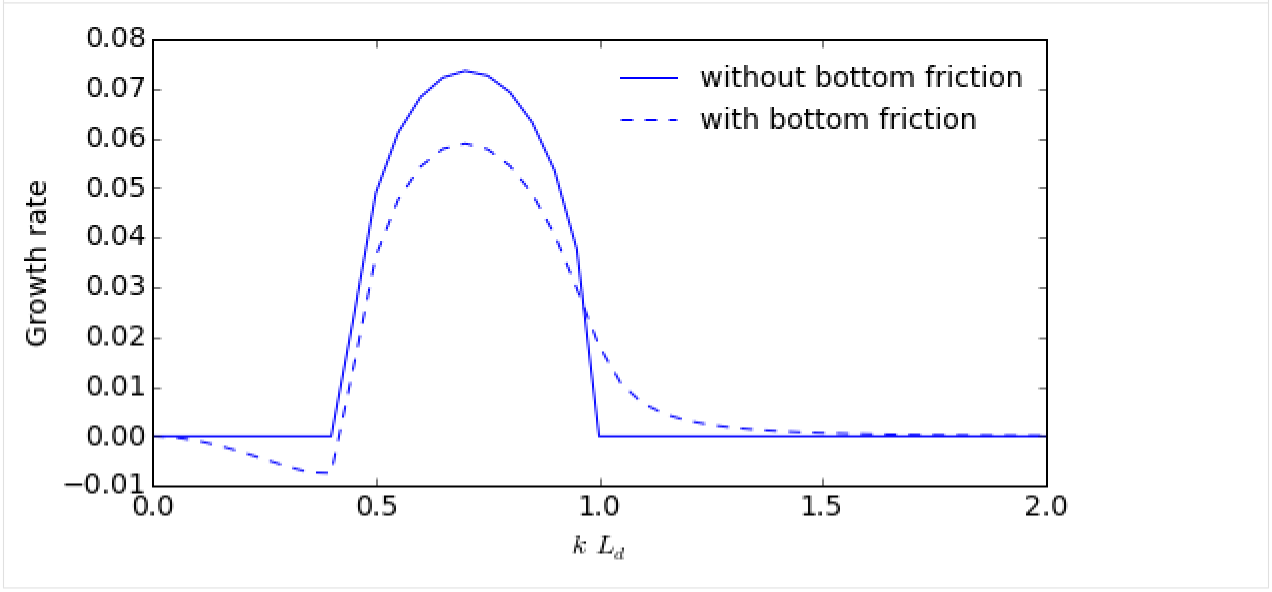
[10]:
```
plt.figure(figsize=(14,4))
plt.subplot(121)
plt.contour(k,l,evals,colors='k')
plt.pcolormesh(k,l,evals,cmap='Blues')
plt.colorbar()
plt.xlim(0,2.); plt.ylim(-2.,2.)
plt.clim([0.,.1])
plt.xlabel(r'$k \, L_d$'); plt.ylabel(r'$l \, L_d$')
plt.title('without bottom friction')

plt.subplot(122)
plt.contour(k,l,evals_fric,colors='k')
plt.pcolormesh(k,l,evals_fric,cmap='Blues')
plt.colorbar()
plt.xlim(0,2.); plt.ylim(-2.,2.)
plt.clim([0.,.1])
plt.xlabel(r'$k \, L_d$'); plt.ylabel(r'$l \, L_d$')
plt.title('with bottom friction')
```

[10]:
```
<matplotlib.text.Text at 0x11539e290>
```



```
[8]: plt.figure(figsize=(8,4))
     plt.plot(k[m.Ny/2,:],evals[m.Ny/2,:],'b',label='without bottom friction')
     plt.plot(k[m.Ny/2,:],evals_fric[m.Ny/2,:],'b--',label='with bottom friction')
     plt.xlim(0.,2.)
     plt.legend()
     plt.xlabel(r'$k\,L_d$')
     plt.ylabel(r'Growth rate')
```

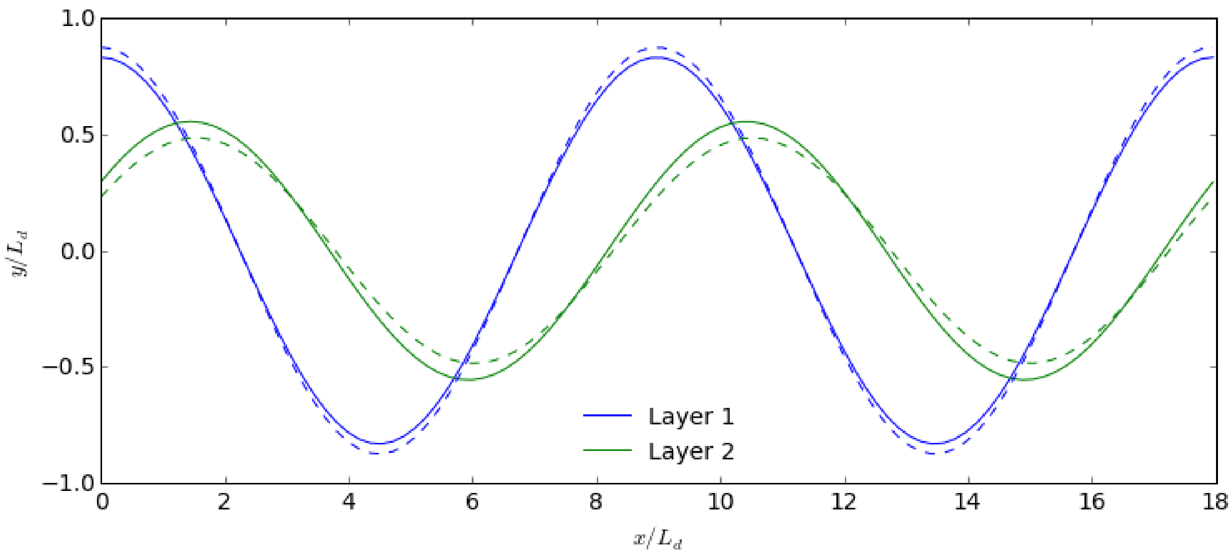
```
[8]: <matplotlib.text.Text at 0x10f9731d0>
```



Plotting the wavestructure of the most unstable modes

[9]:
```
plt.figure(figsize=(12,5))
plt.plot(x,mag[0]*np.cos(kmax*x + phase[0]),'b',label='Layer 1')
plt.plot(x,mag[1]*np.cos(kmax*x + phase[1]),'g',label='Layer 2')
plt.plot(x,mag_fric[0]*np.cos(kmax_fric*x + phase_fric[0]),'b--')
plt.plot(x,mag_fric[1]*np.cos(kmax_fric*x + phase_fric[1]),'g--')
plt.legend(loc=8)
plt.xlabel(r'$x/L_d$'); plt.ylabel(r'$y/L_d$')
```

[9]:
```
<matplotlib.text.Text at 0x114c6d410>
```



This calculation shows the classic phase-tilting of baroclinic unstable waves (e.g. Vallis 2006).

[ ]:

1.3.6 Parameterizations

In this notebook, we'll review tools for defining, running, and comparing subgrid parameterizations.

[1]:
```
import numpy as np
import pandas as pd
pd.set_option('display.max_columns', 10)
import pyqg
import pyqg.diagnostic_tools
import matplotlib.pyplot as plt
%matplotlib inline
```

Run baseline high- and low-resolution models

To illustrate the effect of parameterizations, we'll run two baseline models:

- a low-resolution model without parameterizations at `nx=64` resolution (where Δx is larger than the deformation radius r_d, preventing the model from fully resolving eddies),
- a high-resolution model at `nx=256` resolution (where Δx is ~4x finer than the deformation radius, so eddies can be almost fully resolved).

[2]:
```
%%time
year = 24*60*60*360.
base_kwargs = dict(dt=3600., tmax=5*year, tavestart=2.5*year, twrite=25000)

low_res = pyqg.QGModel(nx=64, **base_kwargs)
low_res.run()
```

```
INFO:  Logger initialized
INFO: Step: 25000, Time: 9.00e+07, KE: 4.14e-04, CFL: 0.042
```

```
CPU times: user 14.3 s, sys: 23.6 s, total: 38 s
Wall time: 10.8 s
```

[3]:
```
%%time
high_res = pyqg.QGModel(nx=256, **base_kwargs)
high_res.run()
```

```
INFO:  Logger initialized
INFO: Step: 25000, Time: 9.00e+07, KE: 4.62e-04, CFL: 0.217
```

```
CPU times: user 3min 47s, sys: 5min 43s, total: 9min 30s
Wall time: 2min 55s
```

Run Smagorinsky and backscatter parameterizations

Now we'll run two types of parameterization: one from Smagorinsky 1963 which models an effective eddy viscosity from subgrid stress, and one adapted from Jansen and Held 2014 and Jansen et al. 2015, which reinjects a fraction of the energy dissipated by Smagorinsky back into larger scales:

[4]:
```
def run_parameterized_model(p):
    model = pyqg.QGModel(nx=64, parameterization=p, **base_kwargs)
    model.run()
    return model
```

[5]:
```
%%time
smagorinsky = run_parameterized_model(
    pyqg.parameterizations.Smagorinsky(constant=0.08))
```

```
INFO:  Logger initialized
INFO: Step: 25000, Time: 9.00e+07, KE: 3.37e-04, CFL: 0.043
```

```
CPU times: user 39.3 s, sys: 1min, total: 1min 39s
Wall time: 30.3 s
```

[8]:
```
%%time
backscatter = run_parameterized_model(
    pyqg.parameterizations.BackscatterBiharmonic(smag_constant=0.08, back_constant=1.1))
```

```
INFO:  Logger initialized
INFO: Step: 25000, Time: 9.00e+07, KE: 5.35e-04, CFL: 0.048
```

```
CPU times: user 35.9 s, sys: 57.8 s, total: 1min 33s
Wall time: 27.5 s
```

Note how these are slightly slower than the baseline low-resolution model, but much faster than the high-resolution model.

See the parameterizations API section and code for examples of how these parameterizations are defined!

Compute similarity metrics between parameterized and high-resolution simulations

To assist with evaluating the effects of parameterizations, we include helpers for computing similarity metrics between model diagnostics. Similarity metrics quantify the percentage closer a diagnostic is to high resolution than low resolution; values greater than 0 indicate improvement over low resolution (with 1 being the maximum), while values below 0 indicate worsening. We can compute these for all diagnostics for all four simulations:

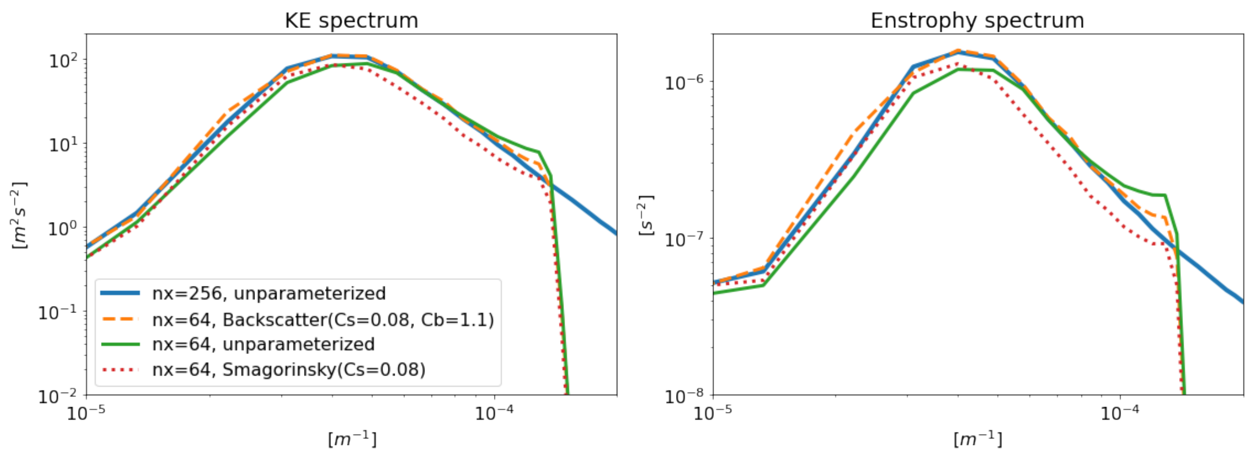
[9]:
```
def label_for(sim):
    return f"nx={sim.nx}, {sim.parameterization or 'unparameterized'}"

sims = [high_res, backscatter, low_res, smagorinsky]

pd.DataFrame.from_dict([
    dict(Simulation=label_for(sim),
        **pyqg.diagnostic_tools.diagnostic_similarities(sim, high_res, low_res))
    for sim in sims])
```

[9]:
```
                                        Simulation  Ensspec1  Ensspec2  \
0                          nx=256, unparameterized  1.000000  1.000000
1  nx=64, BackscatterBiharmonic(Cs=0.08, Cb=1.1)  0.617871  0.595677
2                           nx=64, unparameterized  0.000000  0.000000
3                       nx=64, Smagorinsky(Cs=0.08) -0.049632  0.476905

    KEspec1   KEspec2  ...  ENSfrictionspec  APEgenspec   APEflux    KEflux  \
0  1.000000  1.000000  ...         1.000000    1.000000  1.000000  1.000000
1  0.719966  0.776835  ...         0.444769    0.222930  0.571082  0.448418
2  0.000000  0.000000  ...         0.000000    0.000000  0.000000  0.000000
3 -0.242329 -0.269631  ...        -0.161524    0.069705 -0.171567 -0.296460

     APEgen
0  1.000000
1  0.015557
2  0.000000
3  0.206204

[4 rows x 19 columns]
```

Note that the high-resolution and low-resolution models themselves have similarity scores of 1 and 0 by definition. In this case, the backscatter parameterization is consistently closer to high-resolution than low-resolution, while the Smagorinsky is consistently further.

Let's plot some of the actual curves underlying these metrics to get a better sense:

[19]:
```
def plot_kwargs_for(sim):
    kw = dict(label=label_for(sim).replace('Biharmonic',''))
    kw['ls'] = (':' if sim.uv_parameterization else ('--' if sim.q_parameterization else
↪'-'))
    kw['lw'] = (4 if sim.nx==256 else 3)
    return kw

plt.figure(figsize=(16,6))
plt.rcParams.update({'font.size': 16})

plt.subplot(121, title="KE spectrum")
for sim in sims:
    plt.loglog(
        *pyqg.diagnostic_tools.calc_ispec(sim, sim.get_diagnostic('KEspec').sum(0)),
        **plot_kwargs_for(sim))
plt.ylabel("[$m^2 s^{-2}$]")
plt.xlabel("[$m^{-1}$]")
plt.ylim(1e-2,2e2)
plt.xlim(1e-5, 2e-4)
plt.legend(loc='lower left')

plt.subplot(122, title="Enstrophy spectrum")
for sim in sims:
    plt.loglog(
        *pyqg.diagnostic_tools.calc_ispec(sim, sim.get_diagnostic('Ensspec').sum(0)),
        **plot_kwargs_for(sim))
plt.ylabel("[$s^{-2}$]")
plt.xlabel("[$m^{-1}$]")
plt.ylim(1e-8,2e-6)
plt.xlim(1e-5, 2e-4)
plt.tight_layout()
```



The backscatter model, though low-resolution, has energy and enstrophy spectra that more closely resemble those of the high-resolution model.

1.4 API

1.4.1 Base Model Class

This is the base class from which all other models inherit. All of these initialization arguments are available to all of the other model types. This class is not called directly.

`class pyqg.Model`(*nz=1*, *nx=64*, *ny=None*, *L=1000000.0*, *W=None*, *dt=7200.0*, *twrite=1000.0*, *tmax=1576800000.0*, *tavestart=315360000.0*, *taveint=86400.0*, *useAB2=False*, *rek=5.787e-07*, *filterfac=23.6*, *f=None*, *g=9.81*, *q_parameterization=None*, *uv_parameterization=None*, *parameterization=None*, *diagnostics_list='all'*, *ntd=1*, *log_level=1*, *logfile=None*)

A generic pseudo-spectral inversion model.

Attributes

nx, ny [int] Number of real space grid points in the *x*, *y* directions (cython)

nk, nl [int] Number of spectral space grid points in the *k*, *l* directions (cython)

nz [int] Number of vertical levels (cython)

kk, ll [real array] Zonal and meridional wavenumbers (*nk*) (cython)

a [real array] inversion matrix (*nk*, *nk*, *nl*, *nk*) (cython)

q [real array] Potential vorticity in real space (*nz*, *ny*, *nx*) (cython)

qh [complex array] Potential vorticity in spectral space (*nk*, *nl*, *nk*) (cython)

ph [complex array] Streamfunction in spectral space (*nk*, *nl*, *nk*) (cython)

u, v [real array] Zonal and meridional velocity anomalies in real space (*nz*, *ny*, *nx*) (cython)

Ubg [real array] Background zonal velocity (*nk*) (cython)

Qy [real array] Background potential vorticity gradient (*nk*) (cython)

ufull, vfull [real arrays] Zonal and meridional full velocities in real space (*nz*, *ny*, *nx*) (cython)

uh, vh [complex arrays] Velocity anomaly components in spectral space (*nk*, *nl*, *nk*) (cython)

rek [float] Linear drag in lower layer (cython)

t [float] Model time (cython)

tc [int] Model timestep (cython)

dt [float] Numerical timestep (cython)

L, W [float] Domain length in x and y directions

filterfac [float] Amplitdue of the spectral spherical filter

twrite [int] Interval for cfl writeout (units: number of timesteps)

tmax [float] Total time of integration (units: model time)

tavestart [float] Start time for averaging (units: model time)

tsnapstart [float] Start time for snapshot writeout (units: model time)

taveint [float] Time interval for accumulation of diagnostic averages. (units: model time)

tsnapint [float] Time interval for snapshots (units: model time)

ntd [int] Number of threads to use. Should not exceed the number of cores on your machine.

pmodes [real array] Vertical pressure modes (unitless)

radii [real array] Deformation radii (units: model length)

q_parameterization [function or pyqg.Parameterization] Optional `Parameterization` object or function which takes the model as input and returns a `numpy` array of shape `(nz, ny, nx)` to be added to $\partial_t q$ before stepping forward. This can be used to implement subgrid forcing parameterizations.

uv_parameterization [function or pyqg.Parameterization] Optional `Parameterization` object or function which takes the model as input and returns a tuple of two `numpy` arrays, each of shape `(nz, ny, nx)`, to be added to the zonal and meridional velocity derivatives (respectively) at each timestep (by adding their curl to $\partial_t q$). This can also be used to implemented subgrid forcing parameterizations, but expressed in terms of velocity rather than potential vorticity.

Note: All of the test cases use `nx==ny`. Expect bugs if you choose these parameters to be different.

Note: All time intervals will be rounded to nearest *dt* interval.

Parameters

nx [int] Number of grid points in the x direction.

ny [int] Number of grid points in the y direction (default: nx).

L [number] Domain length in x direction. Units: meters.

W Domain width in y direction. Units: meters (default: L).

rek [number] linear drag in lower layer. Units: seconds $^{-1}$.

filterfac [number] amplitdue of the spectral spherical filter (originally 18.4, later changed to 23.6).

dt [number] Numerical timstep. Units: seconds.

twrite [int] Interval for cfl writeout. Units: number of timesteps.

tmax [number] Total time of integration. Units: seconds.

tavestart [number] Start time for averaging. Units: seconds.

tsnapstart [number] Start time for snapshot writeout. Units: seconds.

taveint [number] Time interval for accumulation of diagnostic averages. Units: seconds. (For performance purposes, averaging does not have to occur every timestep)

tsnapint [number] Time interval for snapshots. Units: seconds.

ntd [int] Number of threads to use. Should not exceed the number of cores on your machine.

q_parameterization [function or pyqg.Parameterization] Optional `Parameterization` object or function which takes the model as input and returns a `numpy` array of shape `(nz, ny, nx)` to be added to $\partial_t q$ before stepping forward. This can be used to implement subgrid forcing parameterizations.

uv_parameterization [function or pyqg.Parameterization] Optional `Parameterization` object or function which takes the model as input and returns a tuple of two `numpy` arrays, each of shape `(nz, ny, nx)`, to be added to the zonal and meridional velocity derivatives (respectively) at each timestep (by adding their curl to $\partial_t q$). This can also be used to implemented subgrid forcing parameterizations, but expressed in terms of velocity rather than potential vorticity.

parameterization [pyqg.Parameterization] An explicit `Parameterization` object representing either a velocity or potential vorticity parameterization, whose type will be inferred.

`describe_diagnostics()`
Print a human-readable summary of the available diagnostics.

`modal_projection(`*p*, *forward=True*`)`
Performs a field p into modal amplitudes pn using the basis [pmodes]. The inverse transform calculates p from pn

`property parameterization`
Return the model's parameterization if present (either in terms of PV or velocity, warning if there are both).

Returns

parameterization [pyqg.Parameterization or function]

`run()`
Run the model forward without stopping until the end.

`run_with_snapshots(`*tsnapstart=0.0*, *tsnapint=432000.0*`)`
Run the model forward, yielding to user code at specified intervals.

Parameters

tsnapstart [int] The timestep at which to begin yielding.

tstapint [int] The interval at which to yield.

`spec_var(`*ph*`)`
compute variance of p from Fourier coefficients ph

`stability_analysis(`*bottom_friction=False*`)`

Performs the baroclinic linear instability analysis given given the base state velocity :math: *(U, V)* and the stretching matrix :math: *S*:

$$A\Phi = \omega B\Phi,$$

where

$$A = B(Uk + Vl) + I(kQ_y - lQ_x) + 1j\delta_{NN} r_{ek} I\kappa^2$$

where $\delta_{NN} = [0, 0, \dots, 0, 1]$,

and

$$B = S - I\kappa^2.$$

The eigenstructure is

$$\Phi$$

and the eigenvalue is

$$`\omega`$$

The growth rate is $\mathrm{Im}\{\omega\}$.

Parameters

bottom_friction: optional inclusion linear bottom drag in the linear stability calculation (default is False, as if :math: *r_{ek} = 0*)

Returns

omega: complex array The eigenvalues with largest complex part (units: inverse model time)

phi: complex array The eigenvectors associated associated with omega (unitless)

`to_dataset()`

Convert outputs from model to an xarray dataset

Returns

ds [xarray.Dataset]

`vertical_modes()`

Calculate standard vertical modes. Simply the eigenvectors of the stretching matrix S

1.4.2 Specific Model Types

These are the actual models which are run.

`class pyqg.QGModel(`*beta=1.5e-11*, *rd=15000.0*, *delta=0.25*, *H1=500*, *U1=0.025*, *U2=0.0*, ***kwargs*`)`

Two layer quasigeostrophic model.

This model is meant to representflows driven by baroclinic instabilty of a base-state shear $U_1 - U_2$. The upper and lower layer potential vorticity anomalies q_1 and q_2 are

$$\begin{aligned} q_1 &= \nabla^2\psi_1 + F_1(\psi_2 - \psi_1) \\ q_2 &= \nabla^2\psi_2 + F_2(\psi_1 - \psi_2) \end{aligned}$$

with

$$\begin{aligned} F_1 &\equiv \frac{k_d^2}{1+\delta^2} \\ F_2 &\equiv \delta F_1 \; . \end{aligned}$$

The layer depth ratio is given by $\delta = H_1/H_2$. The total depth is $H = H_1 + H_2$.

The background potential vorticity gradients are

$$\begin{aligned} \beta_1 &= \beta + F_1(U_1 - U_2) \\ \beta_2 &= \beta - F_2(U_1 - U_2) \; . \end{aligned}$$

The evolution equations for q_1 and q_2 are

$$\begin{aligned} \partial_t\, q_1 + J(\psi_1\,, q_1) + \beta_1\, {\psi_1}_x &= \text{ssd} \\ \partial_t\, q_2 + J(\psi_2\,, q_2) + \beta_2\, {\psi_2}_x &= -r_{ek}\nabla^2\psi_2 + \text{ssd}\,. \end{aligned}$$

where *ssd* represents small-scale dissipation and r_{ek} is the Ekman friction parameter.

Parameters

beta [number] Gradient of coriolis parameter. Units: meters $^{-1}$ seconds $^{-1}$

rek [number] Linear drag in lower layer. Units: seconds $^{-1}$

rd [number] Deformation radius. Units: meters.

delta [number] Layer thickness ratio (H1/H2)

U1 [number] Upper layer flow. Units: meters seconds $^{-1}$

U2 [number] Lower layer flow. Units: meters seconds $^{-1}$

`set_U1U2`(*U1*, *U2*)
Set background zonal flow.

Parameters

U1 [number] Upper layer flow. Units: meters seconds $^{-1}$

U2 [number] Lower layer flow. Units: meters seconds $^{-1}$

`set_q1q2`(*q1*, *q2*, *check=False*)
Set upper and lower layer PV anomalies.

Parameters

q1 [array-like] Upper layer PV anomaly in spatial coordinates.

q1 [array-like] Lower layer PV anomaly in spatial coordinates.

`class pyqg.LayeredModel`(*beta=1.5e-11*, *nz=3*, *rd=15000.0*, *f=0.0001236812857687059*, *H=None*, *U=None*, *V=None*, *rho=None*, *delta=None*, ***kwargs*)
Layered quasigeostrophic model.

This model is meant to represent flows driven by baroclinic instabilty of a base-state shear. The potential vorticity anomalies qi are related to the streamfunction psii through

$$q_i = \nabla^2\psi_i + \frac{f_0^2}{H_i}\left(\frac{\psi_{i-1}-\psi_i}{g'_{i-1}} - \frac{\psi_i-\psi_{i+1}}{g'_i}\right), \qquad i = 2, \mathsf{N}-1\,,$$
$$q_1 = \nabla^2\psi_1 + \frac{f_0^2}{H_1}\left(\frac{\psi_2-\psi_1}{g'_1}\right), \qquad i = 1\,,$$
$$q_\mathsf{N} = \nabla^2\psi_\mathsf{N} + \frac{f_0^2}{H_\mathsf{N}}\left(\frac{\psi_{\mathsf{N}-1}-\psi_\mathsf{N}}{g'_\mathsf{N}}\right) + \frac{f_0}{H_\mathsf{N}}h_b\,, \qquad i = \mathsf{N}\,,$$

where the reduced gravity, or buoyancy jump, is

$$g'_i \equiv g\frac{\rho_{i+1}-\rho_i}{\rho_i}\,.$$

The evolution equations are

$$q_{i\,t} + \mathsf{J}\left(\psi_i\,, q_i\right) + \mathsf{Q}_y\psi_{i\,x} - \mathsf{Q}_x\psi_{i\,y} = \text{ssd} - r_{ek}\delta_{i\mathsf{N}}\nabla^2\psi_i\,, \qquad i = 1, \mathsf{N}\,,$$

where the mean potential vorticy gradients are

$$\mathsf{Q}_x = \mathsf{S}\mathsf{V}\,,$$

and

$$\mathsf{Q}_y = \beta\,\mathsf{I} - \mathsf{S}\mathsf{U}\,,$$

where S is the stretching matrix, I is the identity matrix, and the background velocity is

$\vec{\mathsf{V}}(z) = (\mathsf{U}, \mathsf{V})$.

Parameters

nz [integer number] Number of layers (> 1)

beta [number] Gradient of coriolis parameter. Units: meters $^{-1}$ seconds $^{-1}$

rd [number] Deformation radius. Units: meters. Only necessary for the two-layer (nz=2) case.

delta [number] Layer thickness ratio (H1/H2). Only necessary for the two-layer (nz=2) case. Unitless.

U [list of size nz] Base state zonal velocity. Units: meters seconds $^{-1}$

V [array of size nz] Base state meridional velocity. Units: meters seconds $^{-1}$

H [array of size nz] Layer thickness. Units: meters

rho: array of size nz. Layer density. Units: kilograms meters $^{-3}$

`class pyqg.BTModel(beta=0.0, rd=0.0, H=1.0, U=0.0, **kwargs)`

Single-layer (barotropic) quasigeostrophic model. This class can represent both pure two-dimensional flow and also single reduced-gravity layers with deformation radius `rd`.

The equivalent-barotropic quasigeostrophic evolution equations is

$$\partial_t q + J(\psi, q) + \beta \psi_x = \text{ssd}$$

The potential vorticity anomaly is

$$q = \nabla^2 \psi - \kappa_d^2 \psi$$

Parameters

beta [number, optional] Gradient of coriolis parameter. Units: meters $^{-1}$ seconds $^{-1}$

rd [number, optional] Deformation radius. Units: meters.

U [number, optional] Upper layer flow. Units: meters seconds $^{-1}$.

`set_U(U)`

Set background zonal flow.

Parameters

U [number] Upper layer flow. Units: meters seconds $^{-1}$.

`class pyqg.SQGModel(beta=0.0, Nb=1.0, f_0=1.0, H=1.0, U=0.0, **kwargs)`

Surface quasigeostrophic model.

Parameters

beta [number] Gradient of coriolis parameter. Units: meters $^{-1}$ seconds $^{-1}$

Nb [number] Buoyancy frequency. Units: seconds $^{-1}$.

U [number] Background zonal flow. Units: meters seconds $^{-1}$.

`set_U(U)`

Set background zonal flow

1.4.3 Lagrangian Particles

`class pyqg.LagrangianParticleArray2D`(*x0*, *y0*, *periodic_in_x=False*, *periodic_in_y=False*, *xmin=- inf*, *xmax=inf*, *ymin=- inf*, *ymax=inf*, *particle_dtype='f8'*)

A class for keeping track of a set of lagrangian particles in two-dimensional space. Tries to be fast.

Parameters

x0, y0 [array-like] Two arrays (same size) representing the particle initial positions.

periodic_in_x [bool] Whether the domain wraps in the x direction.

periodic_in_y [bool] Whether the domain 'wraps' in the y direction.

xmin, xmax [numbers] Maximum and minimum values of x coordinate

ymin, ymax [numbers] Maximum and minimum values of y coordinate

particle_dtype [dtype] Data type to use for particles

`step_forward_with_function`(*uv0fun*, *uv1fun*, *dt*)

Advance particles using a function to determine u and v.

Parameters

uv0fun [function] Called like `uv0fun(x,y)`. Should return the velocity field u, v at time t.

uv1fun(x,y) [function] Called like `uv1fun(x,y)`. Should return the velocity field u, v at time t + dt.

dt [number] Timestep.

`class pyqg.GriddedLagrangianParticleArray2D`(*x0*, *y0*, *Nx*, *Ny*, *grid_type='A'*, ***kwargs*)

Lagrangian particles with velocities given on a regular cartesian grid.

Parameters

x0, y0 [array-like] Two arrays (same size) representing the particle initial positions.

Nx, Ny: int Number of grid points in the x and y directions

grid_type: {'A'} Arakawa grid type specifying velocity positions.

`interpolate_gridded_scalar`(*x*, *y*, *c*, *order=1*, *pad=1*, *offset=0*)

Interpolate gridded scalar C to points x,y.

Parameters

x, y [array-like] Points at which to interpolate

c [array-like] The scalar, assumed to be defined on the grid.

order [int] Order of interpolation

pad [int] Number of pad cells added

offset [int] ???

Returns

ci [array-like] The interpolated scalar

`step_forward_with_gridded_uv`(*U0*, *V0*, *U1*, *V1*, *dt*, *order=1*)

Advance particles using a gridded velocity field. Because of the Runga-Kutta timestepping, we need two velocity fields at different times.

Parameters

U0, V0 [array-like] Gridded velocity fields at time t - dt.

U1, V1 [array-like] Gridded velocity fields at time t.

dt [number] Timestep.

order [int] Order of interpolation.

1.4.4 Diagnostic Tools

Utility functions for pyqg model data.

`pyqg.diagnostic_tools.`**`calc_ispec`**(*model*, *_var_dens*, *averaging=True*, *truncate=True*, *nd_wavenumber=False*, *nfactor=1*)

Compute isotropic spectrum *phr* from 2D spectrum of variable *signal2d* such that *signal2d.var() = phr.sum() * (kr[1] - kr[0])*.

Parameters

model [pyqg.Model instance] The model object from which *var_dens* originates

var_dens [squared modulus of fourier coefficients like this:] *np.abs(signal2d_fft)**2/m.M**2*

averaging: If True, spectral density is estimated with averaging over circles, otherwise summation is used and Parseval identity holds

truncate: If True, maximum wavenumber corresponds to inner circle in Fourier space, otherwise - outer circle

nd_wavenumber: If True, wavenumber is nondimensional: minimum wavenumber is 1 and corresponds to domain length/width, otherwise - wavenumber is dimensional [m^-1]

nfactor: width of the bin in sqrt(dk^2+dl^2) units

Returns

kr [array] isotropic wavenumber

phr [array] isotropic spectrum

`pyqg.diagnostic_tools.`**`diagnostic_differences`**(*m1*, *m2*, *reduction='rmse'*, *instantaneous=False*)

Compute a dictionary of differences in the diagnostics of two models at possibly different resolutions (e.g. for quantifying the effects of parameterizations). Applies normalization/isotropization to certain diagnostics before comparing them and skips others. Also computes differences for each vertical layer separately.

Parameters

m1 [pyqg.Model instance] The first model to compare

m2 [pyqg.Model instance] The second model to compare

reduction [string or function] A function that takes two arrays of diagnostics and computes a distance metric. Defaults to the root mean squared difference ('rmse').

instantaneous [boolean] If true, compute difference metrics for the instantaneous values of a diagnostic, rather than its time average. Defaults to false.

Returns

diffs [dict] A dictionary of diagnostic name => distance. If the diagnostic is defined over multiple layers, separate keys are included with an appended z index.

`pyqg.diagnostic_tools.`**`diagnostic_similarities`**`(`*model*, *target*, *baseline*, ***kw*`)`

Like *diagnostic_differences*, but returning a dictionary of similarity scores between negative infinity and 1 which quantify how much closer the diagnostics of a given *model* are to a *target* with respect to a *baseline*. Scores approach 1 when the distance between the model and the target is small compared to the baseline and are negative when that distance is greater.

Parameters

model [pyqg.Model instance] The model for which we want to compute similiarity scores (e.g. a parameterized low resolution model)

target [pyqg.Model instance] The target model (e.g. a high resolution model)

baseline [pyqg.Model instance] The baseline against which we check for improvement or degradation (e.g. an unparameterized low resolution model)

Returns

sims [dict] A dictionary of diagnostic name => similarity. If the diagnostic is defined over multiple layers, separate keys are included with an appended z index.

`pyqg.diagnostic_tools.`**`spec_sum`**`(`*ph2*`)`

Compute total spectral sum of the real spectral quantity``ph^2``.

Parameters

model [pyqg.Model instance] The model object from which *ph* originates

ph2 [real array] The field on which to compute the sum

Returns

var_dens [float] The sum of *ph2*

`pyqg.diagnostic_tools.`**`spec_var`**`(`*model*, *ph*`)`

Compute variance of `p` from Fourier coefficients `ph`.

Parameters

model [pyqg.Model instance] The model object from which *ph* originates

ph [complex array] The field on which to compute the variance

Returns

var_dens [float] The variance of *ph*

1.4.5 Parameterizations

`class pyqg.`**`Parameterization`**

A generic class representing a subgrid parameterization. Inherit from this class, $UVParameterization$, or $QParameterization$ to define a new parameterization.

`abstract __call__``(`*m*`)`

Call the parameterization given a pyqg.Model. Override this function in the subclass when defining a new parameterization.

Parameters

m [Model] The model for which we are evaluating the parameterization.

Returns

forcing [real array or tuple] The forcing associated with the model. If the model has been initialized with this parameterization as its `q_parameterization`, this should be an array of shape `(nz, ny, nx)`. For `uv_parameterization`, this should be a tuple of two such arrays or a single array of shape `(2, nz, ny, nx)`.

`abstract property parameterization_type`

Whether the parameterization applies to velocity (in which case this property should return `"uv_parameterization"`) or potential vorticity (in which case this property should return `"q_parameterization"`). If you inherit from `UVParameterization` or `QParameterization`, this will be defined automatically.

Returns

parameterization_type [string] Either `"uv_parameterization"` or `"q_parameterization"`, depending on how the output should be interpreted.

`__add__`(*other*)

Add two parameterizations (returning a new object).

Parameters

other [Parameterization] The parameterization to add to this one.

Returns

sum [Parameterization] The sum of the two parameterizations.

`__mul__`(*constant*)

Multiply a parameterization by a constant (returning a new object).

Parameters

constant [number] Multiplicative factor for scaling the parameterization.

Returns

product [Parameterization] The parameterization times the constant.

`class pyqg.parameterizations.UVParameterization`

A generic class representing a subgrid parameterization in terms of velocity. Inherit from this to define a new velocity parameterization.

`class pyqg.parameterizations.QParameterization`

A generic class representing a subgrid parameterization in terms of potential vorticity. Inherit from this to define a new potential vorticity parameterization.

`class pyqg.parameterizations.Smagorinsky`(*constant=0.1*)

Velocity parameterization from Smagorinsky 1963.

This parameterization assumes that due to subgrid stress, there is an effective eddy viscosity

$$\nu = (C_S \Delta)^2 \sqrt{2(S_{x,x}^2 + S_{y,y}^2 + 2S_{x,y}^2)}$$

which leads to updated velocity tendencies $\Pi_i, i \in \{1, 2\}$ corresponding to x and y respectively (equation is the same in each layer):

$$\Pi_i = 2\partial_i(\nu S_{i,i}) + \partial_{2-i}(\nu S_{i,2-i})$$

where C_S is a tunable Smagorinsky constant, Δ is the grid spacing, and

$$S_{i,j} = \frac{1}{2}(\partial_i \mathbf{u}_j + \partial_j \mathbf{u}_i)$$

Parameters

constant [number] Smagorinsky constant C_S. Defaults to 0.1.

`class pyqg.parameterizations.`**`BackscatterBiharmonic`**`(`*smag_constant=0.08*, *back_constant=0.99*, *eps=1e-32*`)`

PV parameterization based on Jansen and Held 2014 and Jansen et al. 2015 (adapted by Pavel Perezhogin). Assumes that a configurable fraction of Smagorinsky dissipation is scattered back to larger scales in an energetically consistent way.

Parameters

smag_constant [number] Smagorinsky constant C_S for the dissipative model. Defaults to 0.08.

back_constant [number] Backscatter constant C_B describing the fraction of Smagorinsky-dissipated energy which should be scattered back to larger scales. Defaults to 0.99. Normally should be less than 1, but larger values may still be stable, e.g. due to additional dissipation in the model from numerical filtering.

eps [number] Small constant to add to the denominator of the backscatter formula to prevent division by zero errors. Defaults to 1e-32.

`class pyqg.parameterizations.`**`ZannaBolton2020`**`(`*constant=- 46761284*`)`

Velocity parameterization derived from equation discovery by Zanna and Bolton 2020 (Eq. 6).

Parameters

constant [number] Scaling constant κ_{BC}. Units: meters $^{-2}$. Defaults to $\approx -4.68 \times 10^7$, a value obtained by empirically minimizing squared error with respect to the subgrid forcing that results from applying the filtering method of Guan et al. 2022 to a two-layer QGModel with default parameters.

1.5 Development

1.5.1 Team

- Malte Jansen, University of Chicago
- Ryan Abernathey, Columbia University / LDEO
- Cesar Rocha, Woods Hole Oceanographic Institution
- Francis Poulin, University of Waterloo

1.5.2 History

The numerical approach of pyqg was originally inspired by a MATLAB code by Glenn Flierl of MIT, who was a teacher and mentor to Ryan and Malte. It would be hard to find anyone in the world who knows more about this sort of model than Glenn. Malte implemented a python version of the two-layer model while at GFDL. In the summer of 2014, while both were at the WHOI GFD Summer School, Ryan worked with Malte refactor the code into a proper python package. Cesar got involved and brought pyfftw into the project. Ryan implemented a cython kernel. Cesar and Francis implemented the barotropic and sqg models.

1.5.3 Future

By adopting open-source best practices, we hope pyqg will grow into a widely used, community-based project. We know that many other research groups have their own "in house" QG models. You can get involved by trying out the model, filing issues if you find problems, and making pull requests if you make improvements.

1.5.4 Develpment Workflow

Anyone interested in helping to develop pyqg needs to create their own fork of our *git repository*. (Follow the github forking instructions. You will need a github account.)

Clone your fork on your local machine.

```
$ git clone git@github.com:USERNAME/pyqg
```

(In the above, replace USERNAME with your github user name.)

Then set your fork to track the upstream pyqg repo.

```
$ cd pyqg
$ git remote add upstream git://github.com/pyqg/pyqg.git
```

You will want to periodically sync your master branch with the upstream master.

```
$ git fetch upstream
$ git rebase upstream/master
```

Never make any commits on your local master branch. Instead open a feature branch for every new development task.

```
$ git checkout -b cool_new_feature
```

(Replace *cool_new_feature* with an appropriate description of your feature.) At this point you work on your new feature, using *git add* to add your changes. When your feature is complete and well tested, commit your changes

```
$ git commit -m 'did a bunch of great work'
```

and push your branch to github.

```
$ git push origin cool_new_feature
```

At this point, you go find your fork on github.com and create a pull request. Clearly describe what you have done in the comments. If your pull request fixes an issue or adds a useful new feature, the team will gladly merge it.

After your pull request is merged, you can switch back to the master branch, rebase, and delete your feature branch. You will find your new feature incorporated into pyqg.

```
$ git checkout master
$ git fetch upstream
$ git rebase upstream/master
$ git branch -d cool_new_feature
```

1.5.5 Virtual Environment

This is how to create a virtual environment into which to test-install pyqg, install it, check the version, and tear down the virtual environment.

```
$ conda create --yes -n test_env python=3.9 pip nose numpy cython scipy nose
$ conda install --yes -n test_env -c conda-forge pyfftw
$ source activate test_env
$ pip install pyqg
$ python -c 'import pyqg; print(pyqg.__version__);'
$ conda deactivate
$ conda env remove --yes -n test_env
```

1.5.6 Release Procedure

Once we are ready for a new release, someone needs to make a pull request which updates *docs/whats-new.rst* in preparation for the new version. Then, you can simply create a new release in Github, adding a new tag for the new version (following semver) and clicking “Auto-generate release notes” to summarize changes since the last release (with further elaboration if necessary).

After the release is created, a new version should be published to pypi automatically.

However, before creating the release, it’s worth checking testpypi to ensure the new version works. You can do that by:

1. Verifying the most recent test publish succeeded (and is for the most recent commit)
2. Finding the corresponding pre-release version in pyqg’s TestPyPI history (should look like *X.Y.Z.devN*)
3. Installing that version locally as follows:

```
# Create a temporary directory with a fresh conda environment
$ mkdir ~/tmp
$ cd ~/tmp
$ conda create --yes -n test_env python=3.9 pip nose numpy cython scipy nose setuptools␣
↪setuptools_scm
$ source activate test_env
$ pip install pyfftw # or install with conda-forge

# Install the latest pre-release version of pyqg
$ pip install -i https://test.pypi.org/simple/ --extra-index-url https://pypi.org/simple/
↪ --no-cache-dir pyqg==X.Y.Z.devN

# Ensure this imports successfully and prints out the pre-release version (X.Y.Z.devN)
$ python -c 'import pyqg; print(pyqg.__version__);'

# Clean up and remove the test environment
$ conda deactivate
$ conda env remove --yes -n test_env
```

If this all works, then you should be ready to create the Github release.

1.6 What's New

1.6.1 v0.7.2 (19 May 2022)

- Temporarily removes subgrid forcing method

1.6.2 v0.7.1 (17 May 2022)

- Fixes packaging bug

1.6.3 v0.7.0 (16 May 2022)

- Allow parameterizations as first-class objects
- Add an initial library of parameterizations
- Add tools for comparing diagnostics
- Add a method for computing subgrid forcing

1.6.4 v0.6.0 (16 May 2022)

- Generalize definition of parameterization spectrum diagnostic
- Add enstrophy budget diagnostics
- Normalize and unitize all diagnostics
- Fix issues with calculating isotropic spectra
- Other refactors and bug fixes

1.6.5 v0.5.0 (23 Mar. 2022)

- Added support for online parameterizations
- Dropped support for Python 2.7
- Improvements to the development and release process
- Miscellaneous bug fixes

1.6.6 v0.4.0 (15 Sep. 2021)

- Refactored diagnostics
- Added xarray support
- Improvements to documentation and build process
- Miscellaneous bug fixes

1.6.7 v0.3.0 (23 Nov. 2019)

- Revived development after long hiatus
- Reverted some experimental changes
- Several small bug fixes and documentation corrections
- Updated CI and doc build environments
- Adopted to versioneer for package versioning

1.6.8 v0.2.0 (27 April 2016)

Added compatibility with python 3.

Implemented linear baroclinic stability analysis method.

Implemented vertical mode methods and modal KE and PE spectra diagnostics.

Implemented multi-layer subclass.

Added new logger that leverages on built-in python logging.

Changed license to MIT.

1.6.9 v0.1.4 (22 Oct 2015)

Fixed bug related to the sign of advection terms (GH86).

Fixed bug in _calc_diagnostics (GH75). Now diagnostics start being averaged at tavestart.

1.6.10 v0.1.3 (4 Sept 2015)

Fixed bug in setup.py that caused openmp check to not work.

1.6.11 v0.1.2 (2 Sept 2015)

Package was not building properly through pip/pypi. Made some tiny changes to setup script. pypi forces you to increment the version number.

1.6.12 v0.1.1 (2 Sept 2015)

A bug-fix release with no api or feature changes. The kernel has been modified to support numpy fft routines.

- Removed pyfftw depenency (GH53)
- Cleaning of examples

1.6.13 v0.1 (1 Sept 2015)

Initial release.

PYTHON MODULE INDEX

INDEX