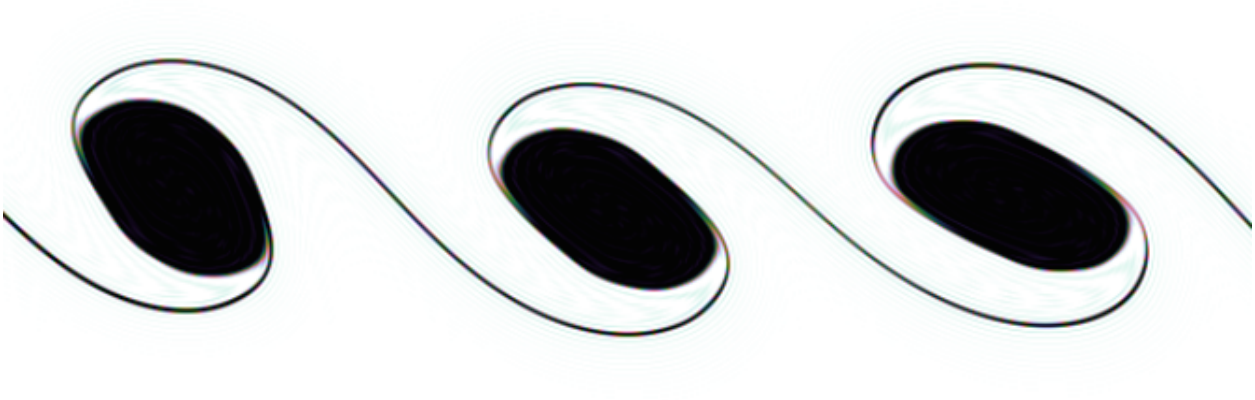
pyqg Documentation

Release 0.1

PyQG team

October 23, 2015

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 stratitifcation 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 1

Contents

1.1 Installation

1.1.1 Requirements

The only requirements are

- Python 2.7. (Python 3 support is in the works)
- numpy (1.6 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 will fall back on numpy's fft routines.

1.1.2 Instructions

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 mforbes channel version was selected for this documentation because its pyfftw package is compatible with the latest version of numpy (1.9.2) and both linux and mac platforms. We don't know who mforbes is, but we are greatful to him/her.

To install pyfftw from the mforbes channel, open a terminal and run the command

```
$ conda install -c mforbes pyfftw
```

If this doesn't work for you, or if it asks you to upgrade / downgrade more of your core pacakges (e.g. numpy) than you would like, you can easily try replacing `mforbes` with one of the other channels.

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

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 on your system:

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

(The `--user` flag is optional–use it if you don't have root privileges.)

If you want to make changes in the code, set up the development mode:

```
$ python setup.py develop
```

pyqg is a work in progress, and we really encourage users to contribute to its Development

1.2 Equations Solved

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

1.2.1 Layered quasigeostrophic model

$$q_{it} + \mathsf{J}\left(\psi_i\,, q_i\right) + U_i q_{ix} + V_i q_{iy} + Q_{iy}\psi_{ix} - Q_{ix}\psi_{iy} = \text{ssd} - r_{ek}\delta_{i\mathsf{N}}\nabla^2\psi_i\,, \qquad i = 1, \mathsf{N}\,,$$

where

$$q_i = \nabla^2\psi_i + \frac{f_0^2}{H_i}\left(\frac{\psi_{i-1} - \psi_i}{g'_{i-1}} - \frac{\psi_i - \psi_{i+1}}{g'_i}\right)\,, \qquad i = 2, \mathsf{N}-1\,,$$

$$and$$

$$q_1 = \nabla^2\psi_1 + \frac{f_0^2}{H_1}\left(\frac{\psi_2 - \psi_1}{g'_1}\right)\,, \qquad i = 1\,,$$

$$q_{\mathsf{N}} = \nabla^2\psi_{\mathsf{N}} + \frac{f_0^2}{H_{\mathsf{N}}}\left(\frac{\psi_{\mathsf{N}-1} - \psi_{\mathsf{N}}}{g'_{\mathsf{N}}}\right) + \frac{f_0}{H_{\mathsf{N}}}h_b\,, \qquad i = \mathsf{N}\,,$$

$$where the reduced gravity, or buoyancy jump, is$$

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

The inversion relationship in spectral space is

$$\hat{q}_i = \underbrace{\left(\mathsf{S} - \kappa^2\mathsf{I}\right)}_{\equiv\mathsf{A}}\hat{\psi}_i\,,$$

where the "stretching matrix" is

$$\mathsf{S} \equiv f_0^2\begin{bmatrix} -\frac{1}{g'_1 H_1} & \frac{1}{g'_1 H_1} & 0 & \cdots & \\ 0 & & & & \\ \vdots & \ddots & \ddots & \ddots & \\ & \frac{1}{g'_{i-1}H_i} & -\left(\frac{1}{g'_{i-1}H_i} + \frac{1}{g'_i H_i}\right) & \frac{1}{g'_i H_i} & \\ & \ddots & \ddots & \ddots & \\ & \cdots & 0 & \frac{1}{g'_{\mathsf{N}-1}H_{\mathsf{N}}} & -\frac{1}{g'_{\mathsf{N}-1}H_{\mathsf{N}}} \end{bmatrix}$$

The forced-dissipative equations in Fourier space are

$$\hat{q}_{it} + ik\,\hat{\psi}_i Q_y - il\,\hat{\psi}_i Q_x + (ikU_i + ilV_i)\hat{q}_i + \hat{\mathsf{J}}\left(\psi_i\,, q_i + \delta_{i\mathsf{N}}\tfrac{f_0}{H_{\mathsf{N}}}h_b\right) = \text{ssd}\,, \qquad i = 1, \mathsf{N}\,,$$

where the mean potential vorticy gradients are

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

$$where the background velocity is$$

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

1.2.2 Energy balance

The equation for the energy spectrum,

$$E(k,l) \equiv \underbrace{\frac{1}{2H}\sum_{i=1}^{\mathsf{N}} H_i \kappa^2 |\hat{\psi}_i|^2}_{\text{kinetic energy}} \quad + \quad \underbrace{\frac{1}{2H}\sum_{i=1}^{\mathsf{N}-1} \frac{f_0^2}{g_i'}|\hat{\psi}_i - \hat{\psi}_{i+1}|^2}_{\text{potential energy}} \ ,$$

is

$$\frac{d}{dt}E(k,l) = \underbrace{\frac{1}{H}\sum_{i=1}^{\mathsf{N}} H_i \mathrm{Re}[\hat{\psi}_i^\star \hat{\mathsf{J}}(\psi_i, \nabla^2\psi_i)]}_{I} + \underbrace{\frac{1}{H}\sum_{i=1}^{\mathsf{N}} H_i \mathrm{Re}[\hat{\psi}_i^\star \hat{\mathsf{J}}(\psi_i, (\mathsf{S}\psi)_i)]}_{II} + \underbrace{\frac{1}{H}\sum_{i=1}^{\mathsf{N}} H_i (kU_i + lV_i)\,\mathrm{Re}[i\,\hat{\psi}_i^\star (\mathsf{S}\hat{\psi}_i)]}_{III} \quad \underbrace{-r_{ek}\frac{H_{\mathsf{N}}}{H}\kappa^2|\hat{\psi}_{\mathsf{N}}|^2}_{IV}$$

where $\kappa^2 = k^2 + l^2$ and the terms above represent

I: Spectral divergence of the kinetic energy flux

II: Spectral divergence of the potential energy flux

III: The rate of potential energy generation

IV: The rate of energy dissipation through bottom friction

Using the notation of the two-layer model, the particular case $\mathsf{N} = 2$ is

$$\frac{d}{dt}E(k,l) = \underbrace{\frac{1}{H}\mathrm{Re}[H_1\hat{\psi}_1^\star\hat{\mathsf{J}}(\psi_1,\nabla^2\psi_1) + H_2\hat{\psi}_2^\star\hat{\mathsf{J}}(\psi_2,\nabla^2\psi_2)]}_{I} + \underbrace{\frac{H_1H_2}{H^2}\mathrm{Re}[(\hat{\psi}_1-\hat{\psi}_2)^\star\hat{\mathsf{J}}(\psi_1,\hat{\psi}_2])}_{II}$$

$$+ \underbrace{\frac{H_1H_2}{H^2}\left[(U_1-U_2)\,\mathrm{Re}[ik\,(\hat{\psi}_1^\star+\hat{\psi}_2^\star)(\hat{\psi}_2-\hat{\psi}_1)\right] + (V_1-V_2)\,\mathrm{Re}[il\,(\hat{\psi}_1^\star+\hat{\psi}_2^\star)(\hat{\psi}_2-\hat{\psi}_1)}_{III} \quad \underbrace{-r_{ek}\frac{H_1}{H}\kappa^2|\hat{\psi}_{\mathsf{N}}|^2}_{IV} \ .$$

1.2.3 Vertical modes

Standard vertical modes are the eigenvectors, $\phi_n(z)$, of the "stretching matrix"

$$\mathsf{S}\,\phi_n = -m_n^2\,\phi_n \ ,$$

where the n'th deformation radius is

$$R_n \equiv m_n^{-1} \ .$$

1.2.4 Linear stability analysis

With $h_b = 0$, the linear eigenproblem is

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

where

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

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

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

$$The growth rate is Im \ : math : `\{\omega\}`.$$

1.2.5 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 cascates 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

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

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)

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

and

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

where, in the pseudo-spectral spirit, $\hat{\mathsf{J}}$ 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

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

or equivalently

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

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

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

where

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

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

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

The second step uses a second-order AB scheme

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

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

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

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 $|\hat{q}_i| << \mathcal{O}(1)$ one can use a smaller constant.

Diagnostics

The kinetic energy is

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

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

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

1.2.6 Equations For Equivalent Barotropic QG Model

The equivalent barotropic quasigeostrophy evolution equations is

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

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

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

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

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

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

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

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

1.3 Examples

1.3.1 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

```
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.

```
# 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

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

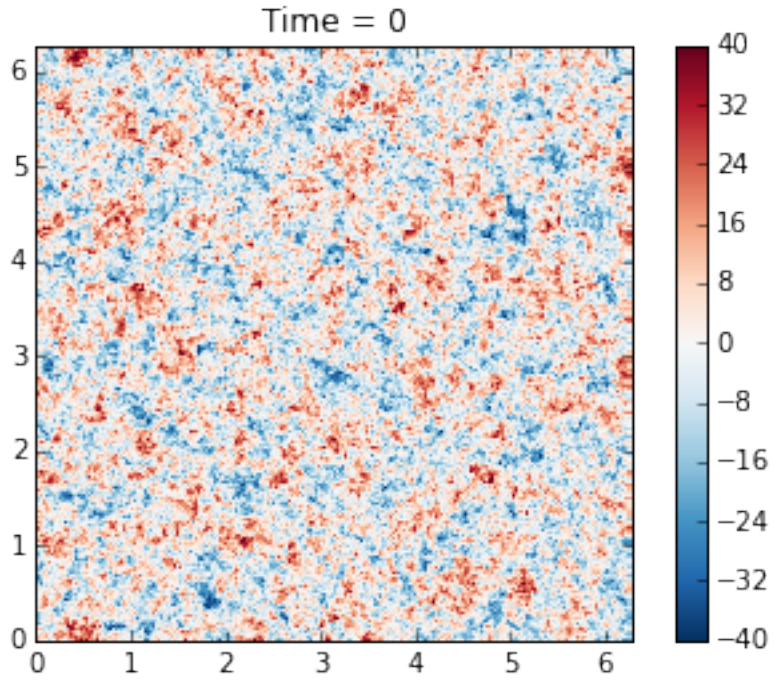
where κ is the wavenumber magnitude. The constant A is determined so that the initial energy is $KE = 0.5$.

```
# 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)
```

```
# initialize the model with that initial condition
m.set_q(qi)
```

```
# 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)
```

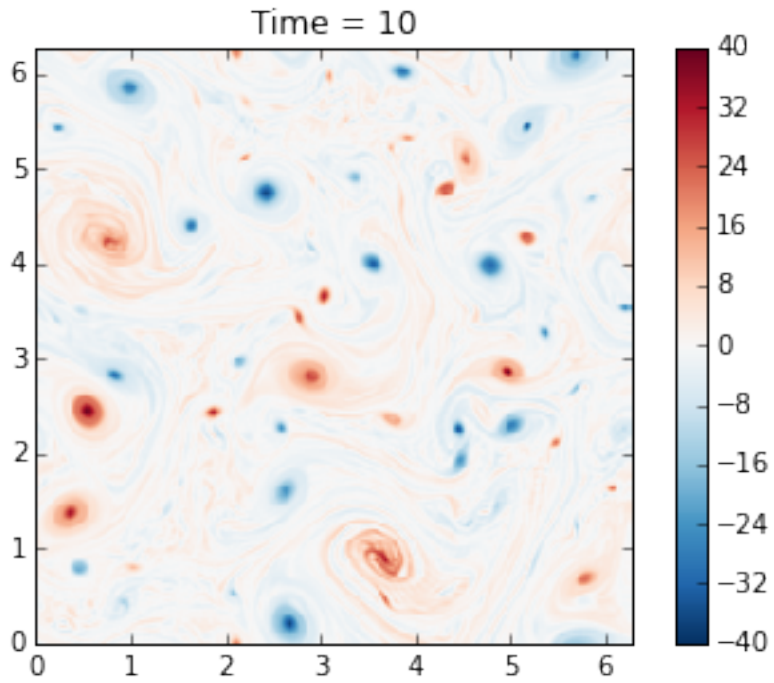


Runing the model

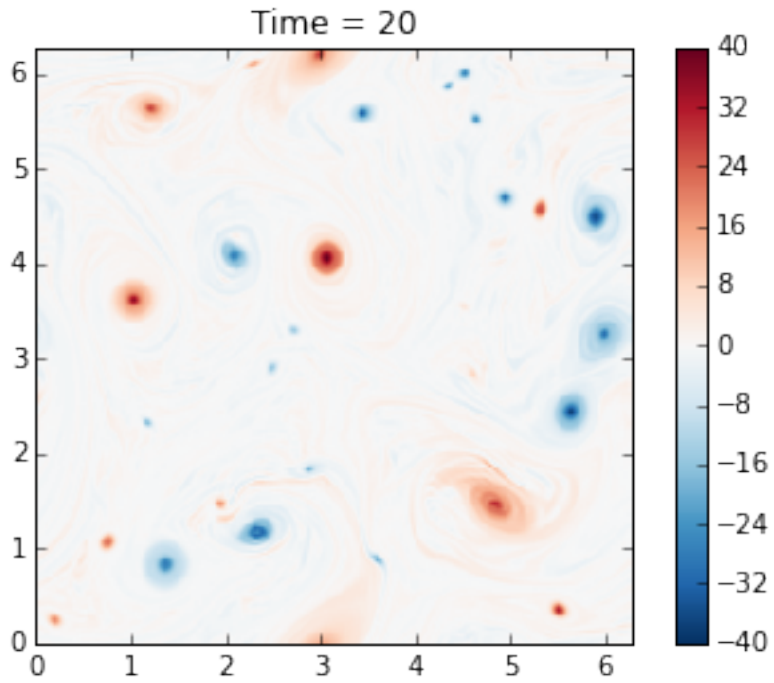
Here we demonstrate how to use the `run_with_snapshots` feature to periodically stop the model and perform some action (in this case, visualization).

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

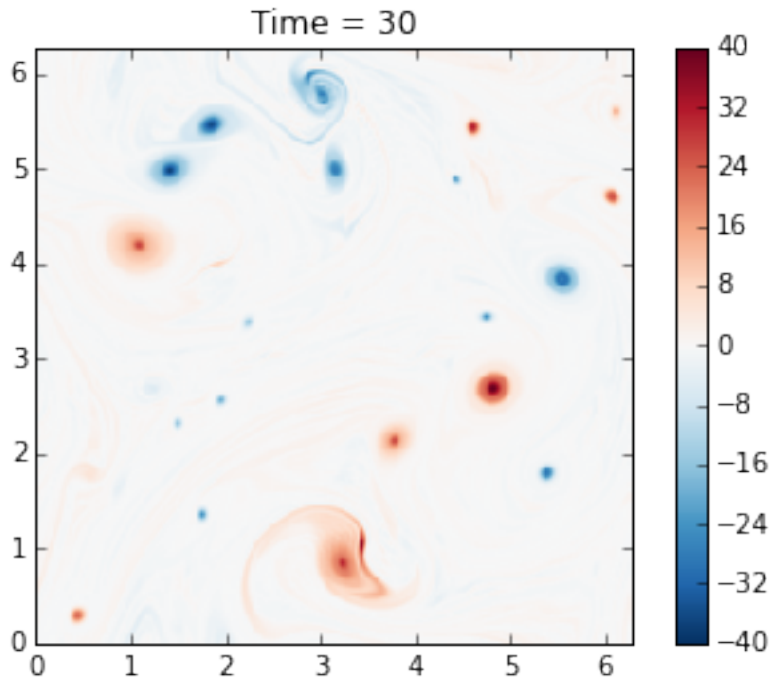
```
t=                1, tc=       1000: cfl=0.104428, ke=0.496432737
t=                1, tc=       2000: cfl=0.110651, ke=0.495084591
t=                2, tc=       3000: cfl=0.101385, ke=0.494349348
t=                3, tc=       4000: cfl=0.113319, ke=0.493862801
t=                5, tc=       5000: 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
```



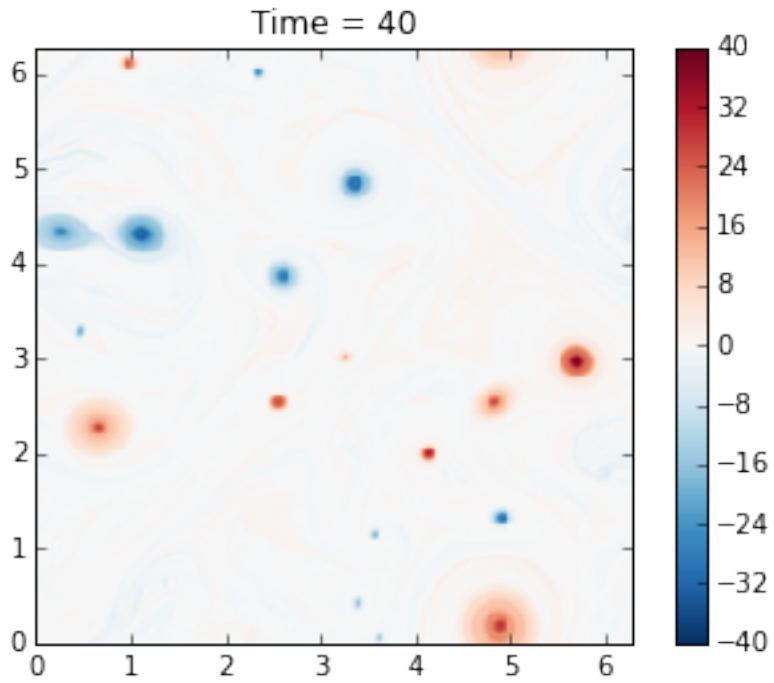

```
t=                9, tc=     10000: cfl=0.103278, ke=0.492830631
t=               10, tc=     11000: cfl=0.102686, ke=0.492775849
t=               11, tc=     12000: cfl=0.099865, ke=0.492726644
t=               12, tc=     13000: cfl=0.110933, ke=0.492679673
t=               13, tc=     14000: cfl=0.102899, 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
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.

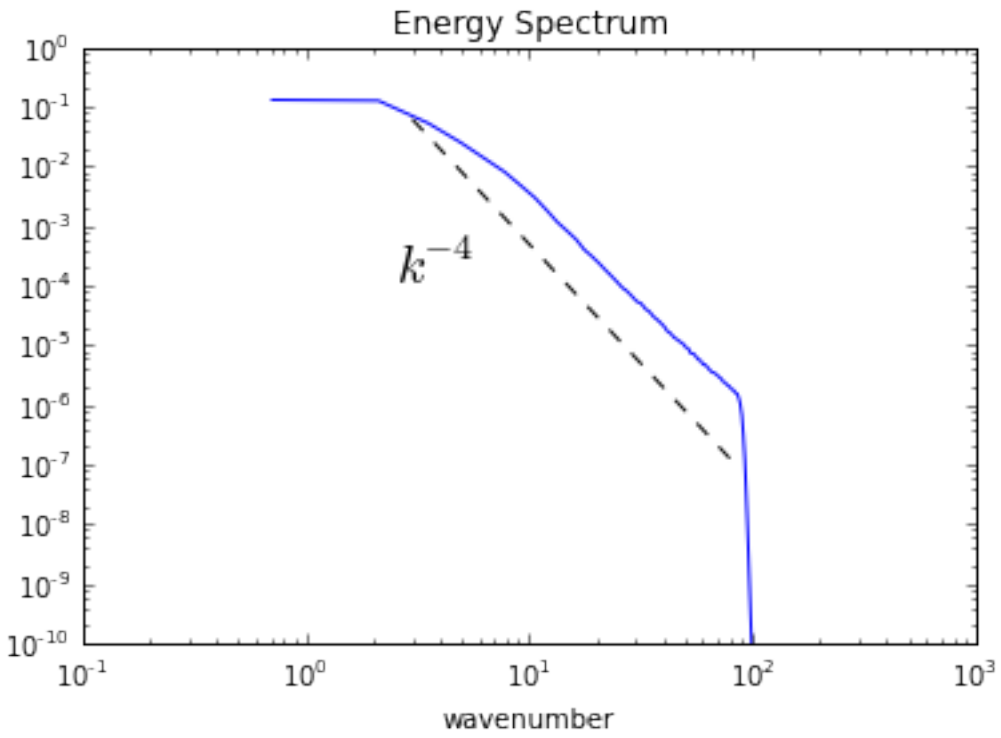
Plotting spectra

```
energy = m.get_diagnostic('KEspec')
enstrophy = m.get_diagnostic('Ensspec')
```

```
# 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())
```

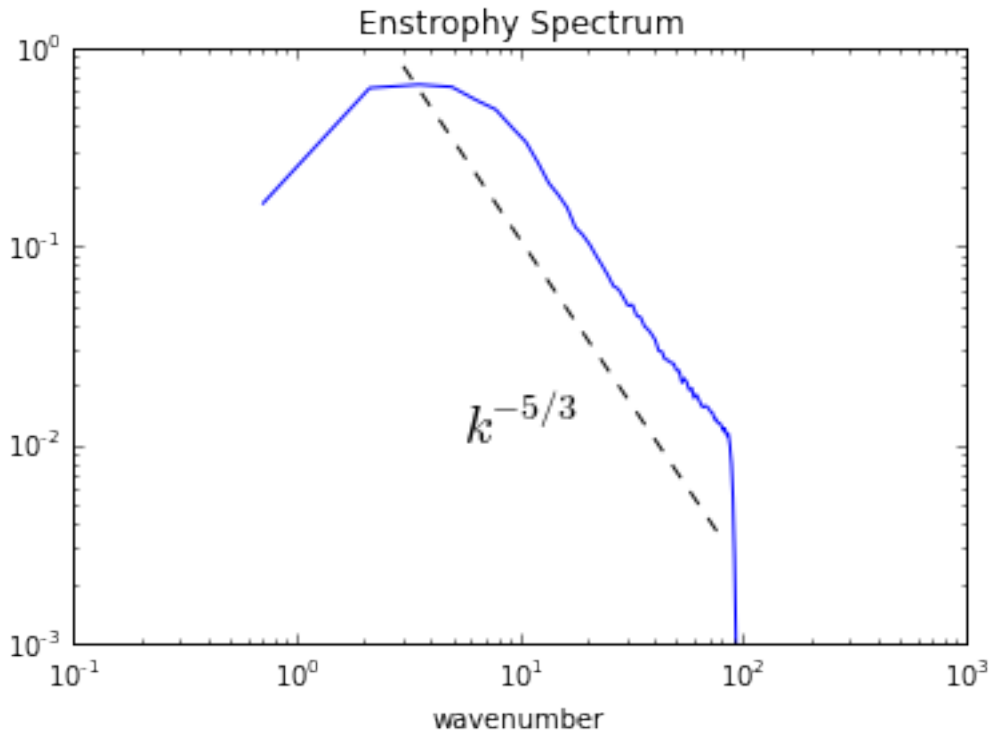
```
ks = np.array([3.,80])
es = 5*ks**-4
plt.loglog(kr,energy_iso)
plt.loglog(ks,es,'k--')
plt.text(2.5,.0001,r'$k^{-4}$',fontsize=20)
plt.ylim(1.e-10,1.e0)
plt.xlabel('wavenumber')
plt.title('Energy Spectrum')
```

```
<matplotlib.text.Text at 0x10c1b1a90>
```




```
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')
```

```
<matplotlib.text.Text at 0x10b5d2f50>
```



1.3.2 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
```

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,

$$\hat{b} = K\hat{\psi}.$$

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., rek = 0., rd = None, dt = 0.005,
                       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 an elliptical vortex,

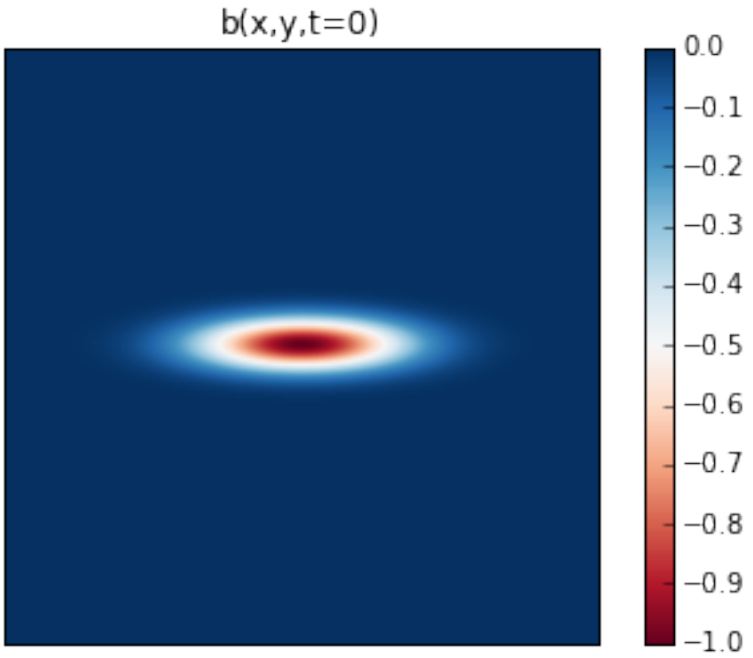
$$b = 0.01 \exp(-(x^2 + (4y)^2)/(L/y)^2$$

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,:,:])
```

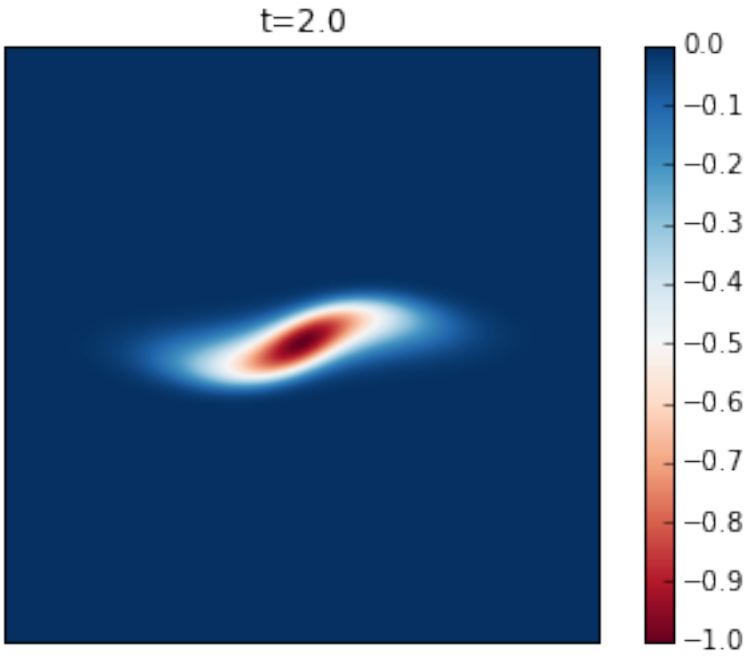
```
# Plot the ICs
plt.rcParams['image.cmap'] = 'RdBu'
plt.clf()
p1 = plt.imshow(m.q.squeeze() + m.beta * m.y)
plt.title('b(x,y,t=0)')
plt.colorbar()
plt.clim([-1, 0])
plt.xticks([])
plt.yticks([])
plt.show()
```

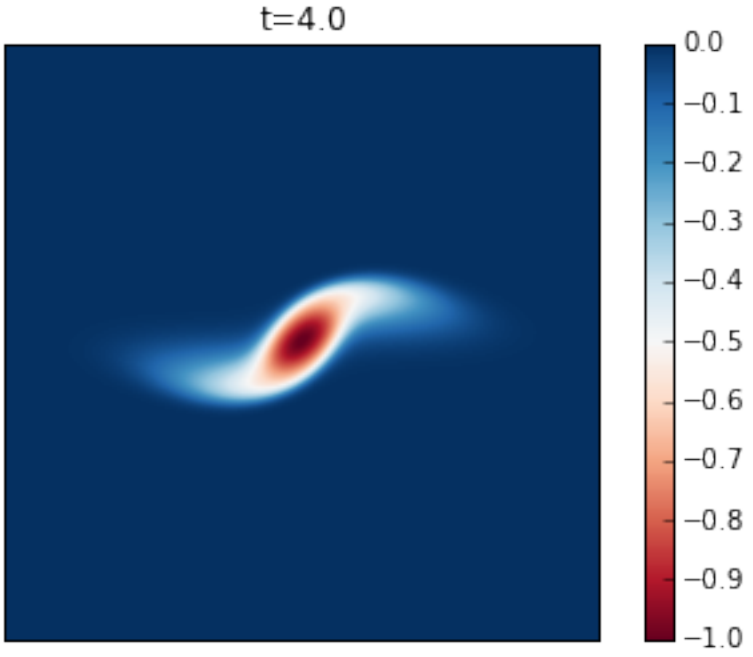


Runing the model

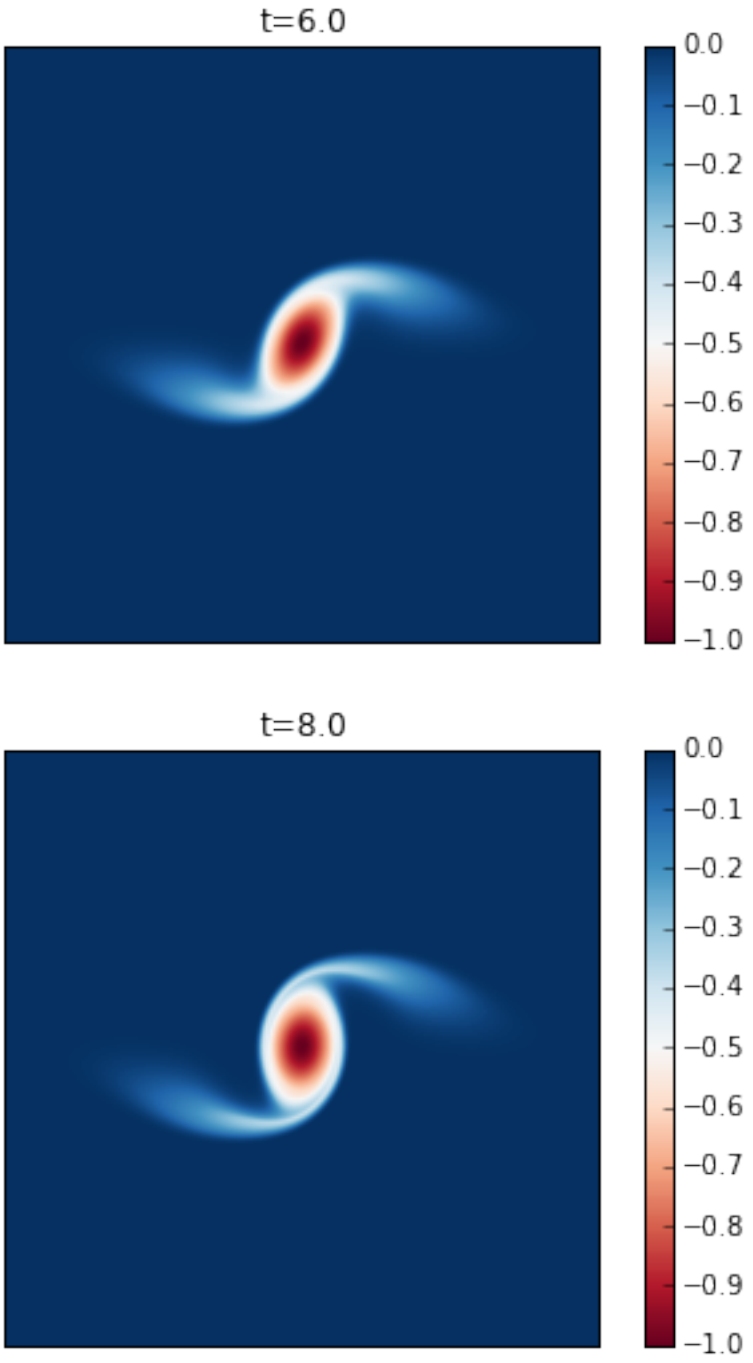
Here we demonstrate how to use the `run_with_snapshots` feature to periodically stop the model and perform some action (in this case, visualization).

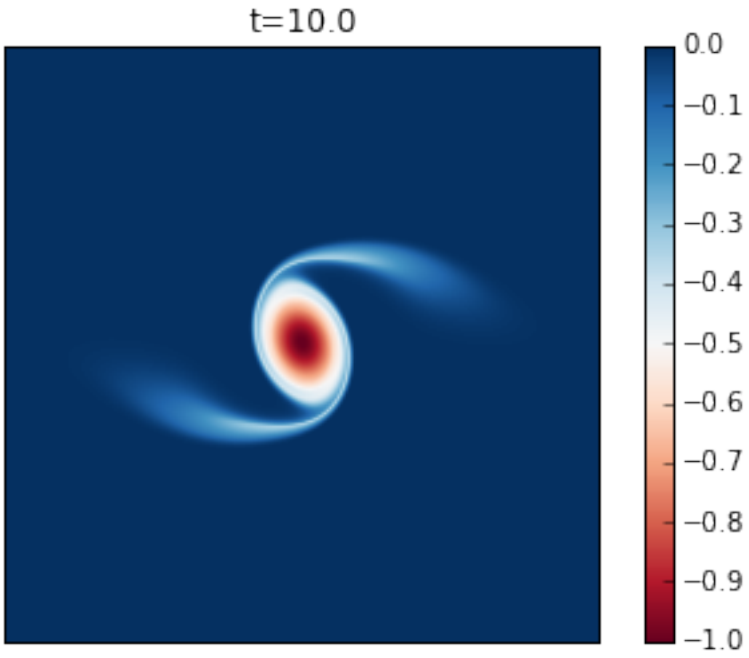
```
for snapshot in m.run_with_snapshots(tsnapstart=0, tsnapint=400*m.dt):
    plt.clf()
    p1 = plt.imshow(m.q.squeeze() + m.beta * m.y)
    #plt.clim([-30., 30.])
    plt.title('t='+str(m.t))
    plt.colorbar()
    plt.clim([-1, 0])
    plt.xticks([])
    plt.yticks([])
    plt.show()
```



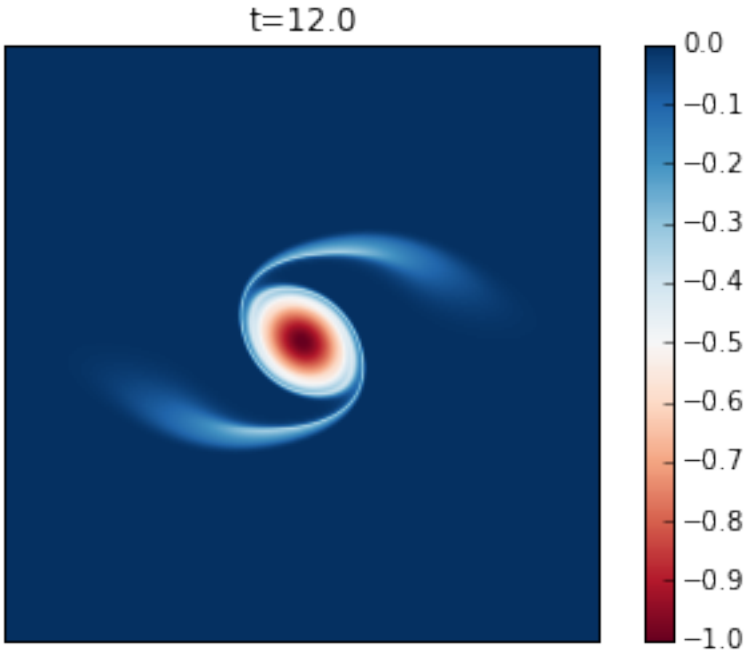


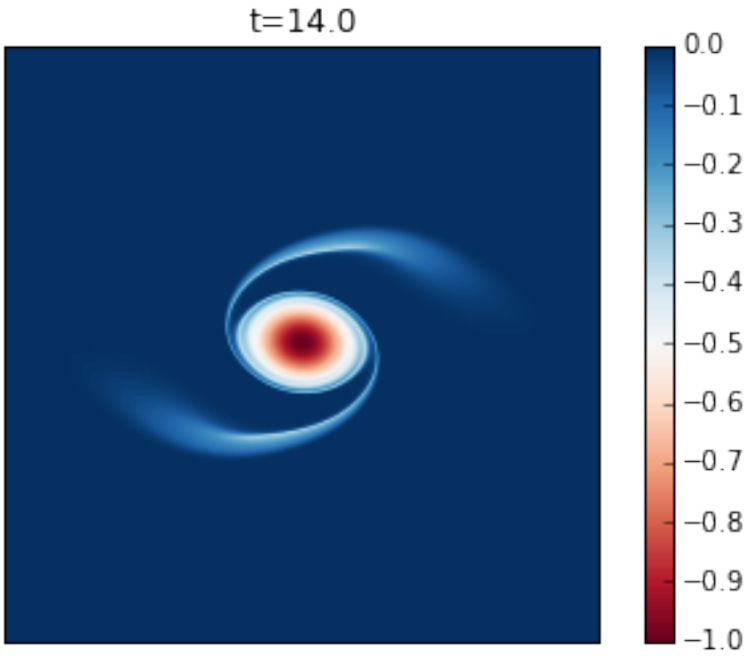

```
t=                4, tc=      1000: cfl=0.239869, ke=0.005206463
```


t=6.0
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.7
−0.8
−0.9
−1.0
t=8.0
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.7
−0.8
−0.9
−1.0

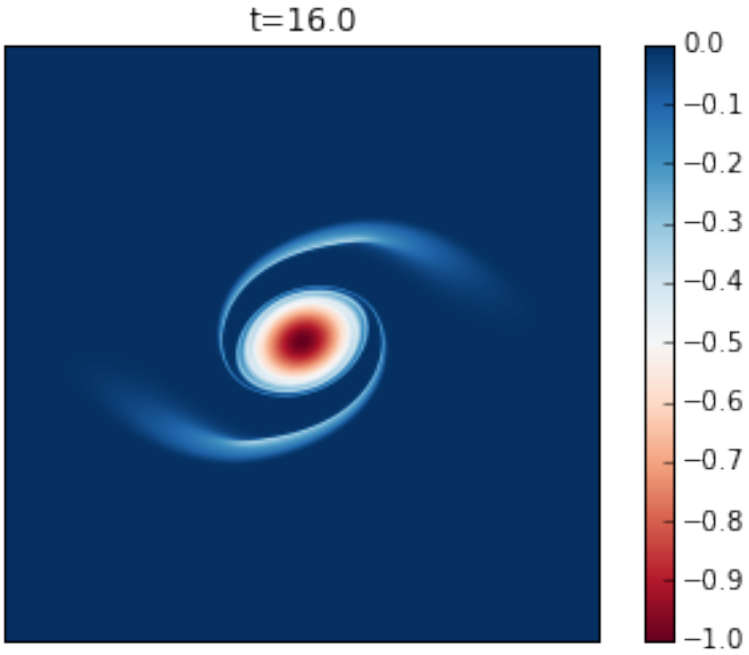


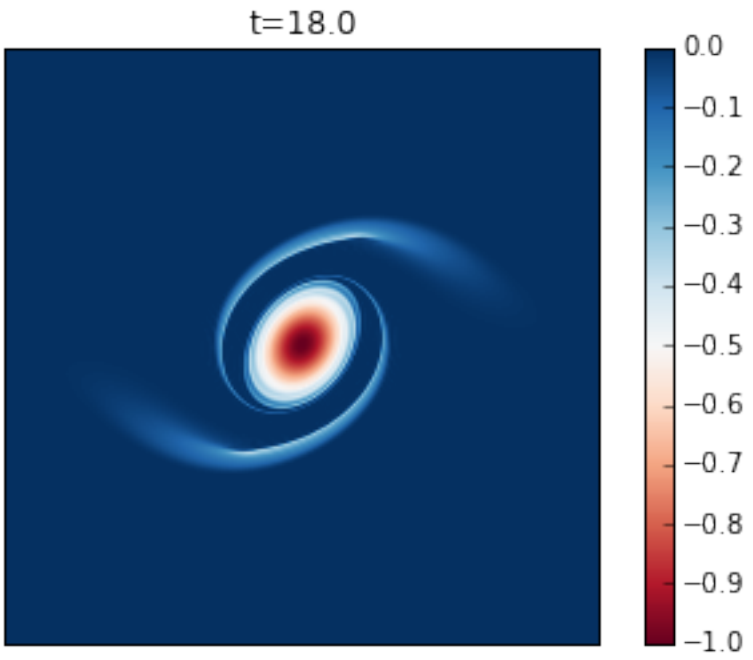

```
t=                10, tc=      2000: cfl=0.267023, ke=0.005206261
```

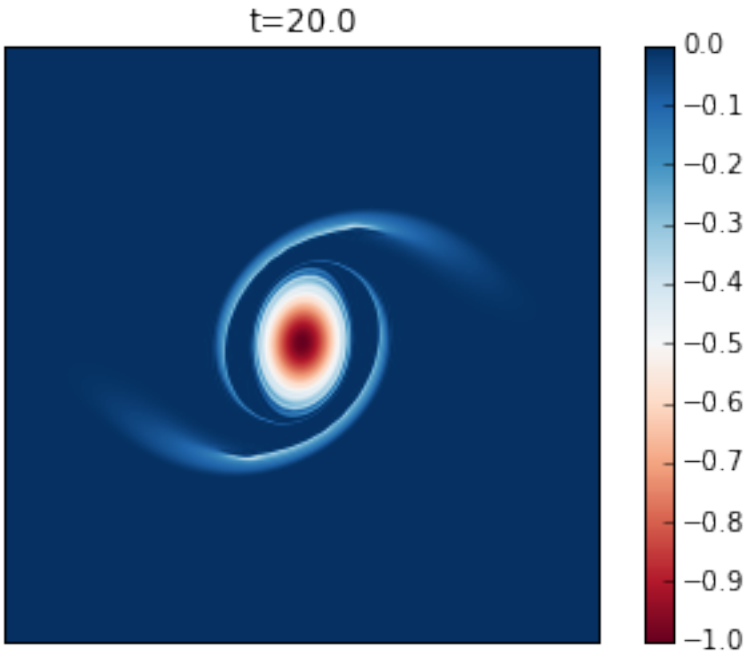




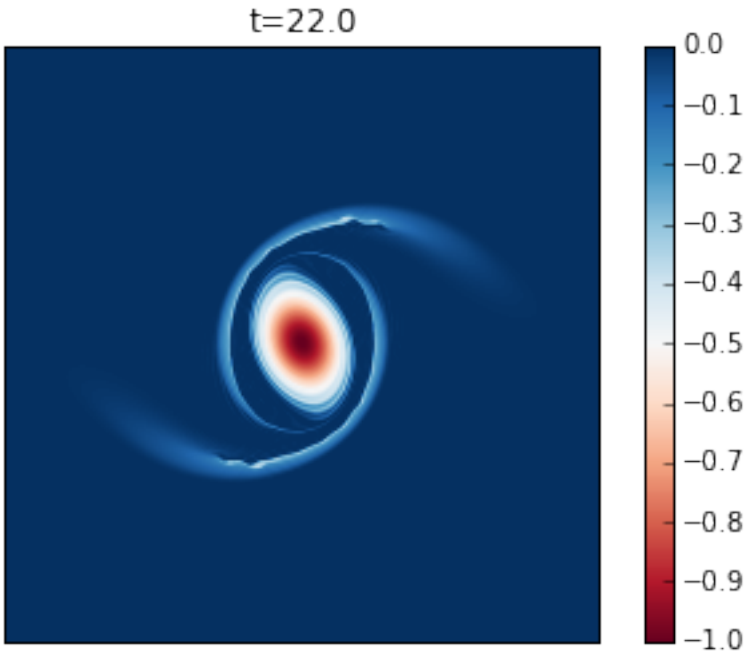

```
t=                15, tc=      3000: cfl=0.251901, ke=0.005199422
```

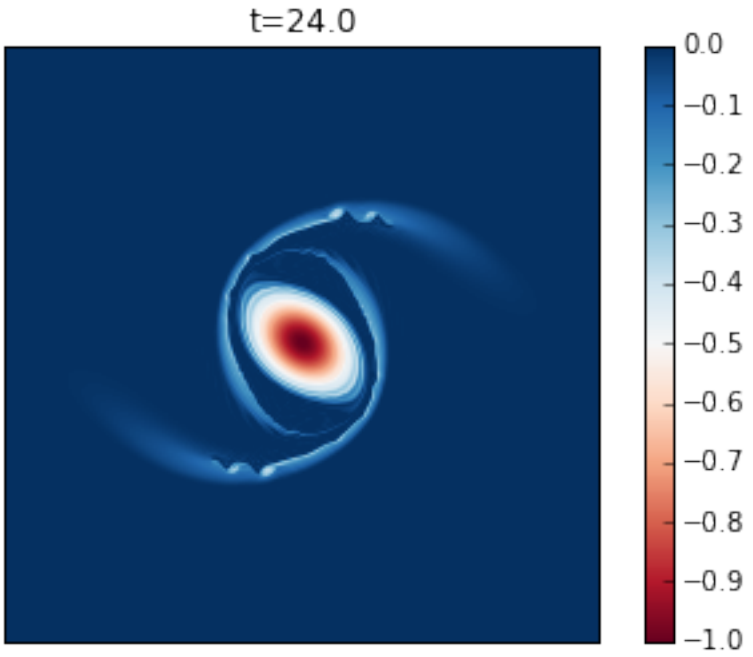




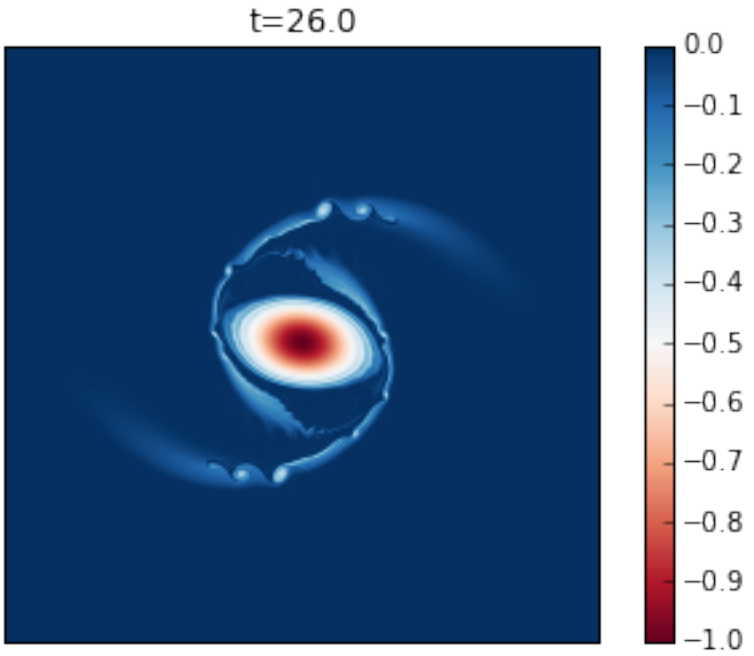



```
t=              20, tc=      4000: cfl=0.259413, ke=0.005189615
```






```
t=              24, tc=      5000: cfl=0.255257, ke=0.005176248
```



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.

1.4 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**(*nx=64*, *ny=None*, *L=1000000.0*, *W=None*, *dt=7200.0*, *twrite=1000.0*, *tmax=1576800000.0*, *tavestart=315360000.0*, *taveint=86400.0*, *f=10000.0*, *hb=None*, *useAB2=False*, *rek=5.787e-07*, *filterfac=23.6*, *diagnostics_list='all'*, *ntd=1*, *quiet=False*, *logfile=None*)

A generic pseudo-spectral inversion model.

Attributes

q	(real array) Potential vorticity in real space
qh	(complex array) Potential vorticity in spectral space
ph	(complex array) Streamfunction in spectral space
u, v	(real arrays) Velocity anomaly components in real space
ufull, vfull	(real arrays) Full velocity components in real space
uh, vh	(complex arrays) Velocity anomaly components in spectral space
nx, ny	(int) Number of grid points in the x and y directions
L, W	(float) Domain length in x and y directions
rek	(float) Linear drag in lower layer
filterfac	(float) Amplitdue of the spectral spherical filter
dt	(float) Numerical timstep
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.

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.

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

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*)
Baroclinic instability analysis

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*, *V1=0.0*, *V2=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

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

with

$$F_1 \equiv \frac{k_d^2}{1+\delta^2}$$
$$F_2 \equiv \delta F_1 \ .$$

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

$$\beta_1 = \beta + F_1(U_1 - U_2)$$
$$\beta_2 = \beta - F_2(U_1 - U_2) \ .$$

The evolution equations for q_1 and q_2 are

$$\partial_t \, q_1 + J(\psi_1 \, , q_1) + \beta_1 \, \psi_{1_x} = \text{ssd}$$
$$\partial_t \, q_2 + J(\psi_2 \, , q_2) + \beta_2 \, \psi_{2_x} = -r_{ek} \nabla^2 \psi_2 + \text{ssd} \, .$$

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: m/s

U2 : number

Lower layer flow. Units: m/s

layer2modal()
calculate modal streamfunction and PV

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

Parameters **U1** : number

Upper layer flow. Units: m/s

U2 : number

Lower layer flow. Units: m/s

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.**BTModel**(*beta=0.0*, *rd=0.0*, *H=1.0*, *U=0.0*, *V=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.

set_UV(*U*, *V*)

Set background zonal flow.

Parameters **U** : number

Upper layer flow. Units meters.

class pyqg.**SQGModel**(*beta=0.0*, *Nb=1.0*, *rd=0.0*, *H=1.0*, *U=0.0*, *V=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.

set_UV(*U*, *V*)

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*, *ph*)

Compute isotropic spectrum *phr* of *ph* from 2D spectrum.

Parameters **model** : pyqg.Model instance

The model object from which *ph* originates

ph : complex array

The field on which to compute the variance

Returns **kr** : array

isotropic wavenumber

phr : array

isotropic spectrum

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*

1.5 Development

1.5.1 Team

- Malte Jansen, University of Chicago

- Ryan Abernathey, Columbia University / LDEO
- Cesar Rocha, Scripps Institution of Oceanography / UCSD
- 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 implement the barotropic and sqg models.

1.5.3 Future

By adopting open-source best practices, we hope pyqg will grow into a widely used, communited-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=2.7 pip nose numpy cython scipy nose
$ conda install --yes -n test_env -c mforbes pyfftw
$ source activate test_env
$ pip install pyqg
$ python -c 'import pyqg; print(pyqg.__version__);'
$ source deactivate
$ conda env remove --yes -n test_env
```

1.6 What's New

1.6.1 v0.1.4 (? ? 2015)

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

Added new diagnostics. Those include time-averages of u, v, vq, and the spectral divergence of enstrophy flux.

Added topography.

Added new printout that leverages on standard python logger.

Added automated linear stability analysis.

Added multi layer model subclass.

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

1.6.2 v0.1.3 (4 Sept 2015)

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

1.6.3 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.

1.6.4 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

1.6.5 v0.1 (1 Sept 2015)

Initial release.

p

B

C

D

G

I

L

M

P

Q

R

S

V