qgsolver_doc Documentation

Release 0.0.1

Aurelien Ponte

Feb 05, 2018

Contents

CHAPTER 1

Equations of motion

See this *Equations of motions, grids*

CHAPTER 2

Install

We recommend conda for dependencies, see README on qgsolver github repository

CHAPTER 3

Tutorial

CHAPTER 4

PV inversion solver

We use PETSc in order to solver PV or omega equation inversion. The PETSc manual is very useful.

The petsc4py documentation and API may also be helpful.

In order to get details about the PV inversion solver, add the following options at run time:

```
mpirun -n 4 python analytical.py -mf -ksp_view -ksp_monitor -ksp_converged_reason
```

In order to profile with snakeviz you need first to generate a profile and then run snakeviz:

```
mpirun -n 4 python -m cProfile -o output.prof uniform.py
snakeviz output.prof
```

CHAPTER 5

Creating input files

input/ is the relevant folder.

For ROMS simulation outputs, you may be inspired to look at input/create_input_roms.py

For NEMO simulation outputs, you may be inspired to look at input/create_input_nemo.py

CHAPTER 6

API

6.1 Equations of motions, grids

6.1.1 Continuous form

Central state variables are the geostrophic streamfunction ψ and potential vorticity q which are related according to:

$$q(x,y,z) = f - f_0 + \Delta\psi + \partial_z\Big(\frac{f_0^2}{N^2}\partial_z\psi\Big)$$

where f_0 is the averaged Coriolis parameter and $N(z)$ is the buoyancy frequency.

Density anomalies and geostrophic currents are related to the streamfunction according to:

$$\partial_z\psi = -\frac{g\rho}{\rho_0 f_0}$$
$$(u,v) = (-\partial_y\psi, \partial_x\psi)$$

The evolution of the system is governed by the advection of potential vorticity and top and bottom densities by geostrophic currents:

$$\partial_t q + J(\psi,q) + J(\Psi,q) + J(\psi,Q) = 0$$
$$\partial_t\partial_z\psi + J(\psi,\partial_z\psi) + J(\Psi,\partial_z\psi) + J(\psi,\partial_z\Psi) = 0 \text{ at } z = 0, -h$$

where capitals represent the large scale - slowly evolving background.

Following Arakawa and Moorthi 1988, we solve for a generalized potential vorticity $\tilde{q}$:

$$\begin{aligned}\tilde{q}(x,y,z) &= f - f_0 + \Delta\psi + \partial_z\Big(\frac{f_0^2}{N^2}\partial_z\psi\Big) - \frac{f_0^2}{N^2}\partial_z\psi\delta(z=0) + \frac{f_0^2}{N^2}\partial_z\psi\delta(z=-h)\\ &= f - f_0 + \Delta\psi + \partial_z\Big(\frac{f_0^2}{N^2}\partial_z\psi\Big) + \frac{f_0}{N^2}\frac{g\rho}{\rho_0}\delta(z=0) - \frac{f_0}{N^2}\frac{g\rho}{\rho_0}\delta(z=-h)\end{aligned}$$

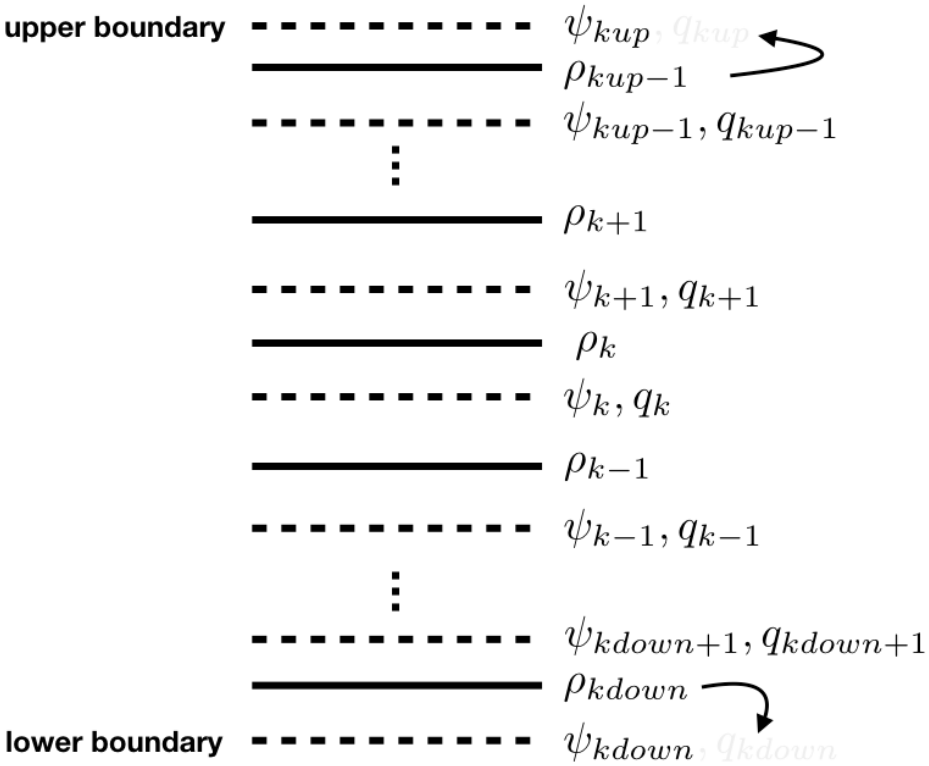
where $\delta(z=0) = 1/dz$ at $z=0$ (corresponds to ρ_{kup}, see description of the vertical grid) and $\delta(z=-h) = 1/dz$ at $z=-h$ (corresponds to ρ_{kdown})

The quasi-geostrophic evolution is then solely described by the advection of $\tilde{q}$:

$$\partial_t \tilde{q} + J(\psi, \tilde{q}) + J(\Psi, \tilde{q}) + J(\psi, \tilde{Q}) = 0$$

6.1.2 Vertical grid

The vertical grid is Charney-Phillips type, meaning streamfunction and potential vorticity are on identical vertical levels while density is at intermediate levels.



6.1.3 Horizontal grid

6.2 qgsolver package

6.2.1 Submodules

6.2.2 qgsolver.grid module

class qgsolver.grid.**grid**(*hgrid_in*, *vgrid_in*, *hdom_in*, *vdom_in*, *mask=False*, *verbose=1*)

Bases: object

Grid object

__init__(*hgrid_in*, *vgrid_in*, *hdom_in*, *vdom_in*, *mask=False*, *verbose=1*)
Builds a grid object

Parameters

- **hgrid_in** (*str, dict or None*) – horizontal grid file name or analytical grid if dict or None Example: hgrid = {'Lx':300.*1.e3, 'Ly':200.*1.e3, 'Nx':150, 'Ny':100}
- **vgrid_in** (*str, dict or None*) – vertical grid file name or analytical grid if dict or None Example: vgrid = {'H':4.e3, 'Nz':10}
- **hdom_in** (*dict*) – horizontal grid dimension description Example: hdom_in = {'Nx': 100, 'Ny': 200} hdom_in = {'Nx': 100, 'Ny': 200, 'i0': 10, 'j0': 20} i0 and j0 are start indices in grid input netcdf file missing parameters are deduced but one should have: Nx=iend-istart+1, Ny=jend-jstart+1
- **vdom_in** (*dict*) – vertical grid dimension description Example: vdom_in = {'Nz': 10, 'k0': 10} k0 is the start index in grid input netcdf file missing parameters are deduced but one should have: kup-kdown+1
- **mask** (*boolean, optional*) – activates the use of a mask, default is false
- **verbose** (*int, optional*) – degree of verbosity, 0 means no outputs, default is 1

load_metric_terms(*da*)
Load metric terms from self.hgrid_file

Parameters **da** (*petsc DMDA*) – holds the petsc grid

load_coriolis_parameter(*coriolis_file*, *da*)
Load the Coriolis parameter

Parameters

- **coriolis_file** (*str*) – netcdf file containing the Coriolis parameter
- **da** (*petsc DMDA*) – holds the petsc grid

load_mask(*mask_file*, *da*, *mask3D=False*)
Load reference mask from metrics file grid.D[grid._k_mask,:,:] will contain the mask

Parameters

- **mask_file** (*str*) – netcdf file containing the mask
- **da** (*petsc DMDA*) – holds the petsc grid
- **mask3D** (*boolean*) – flag for 3D masks, default is False

get_xyz()

get_xy()

get_z()

6.2.3 qgsolver.inout module

qgsolver.inout.**write_nc**(*V*, *vname*, *filename*, *da*, *grid*, *append=False*)
Write a variable to a netcdf file

Parameters

- **V** (*list of petsc vectors*) – (may contain None)
- **vname** (*list of str*) – corresponding variable names

- **filename** (*str*) – netcdf output filename qg object
- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **append** (*boolean*) – append data to an existing file if True, create new file otherwise default is False

qgsolver.inout.**read_nc_petsc**(*V*, *vname*, *filename*, *da*, *grid*, *fillmask=None*)

Read a variable from a netcdf file and stores it in a petsc Vector

Parameters

- **V** (*petsc Vec*) – one(!) petsc vector
- **vname** (*str*) – name of the variable in the netcdf file
- **filename** (*str*) – netcdf input filename
- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **fillmask** (*float, optional*) – value that will replace the default netCDF fill value for NaNs default is None

qgsolver.inout.**read_nc_petsc_2D**(*V*, *vname*, *filename*, *level*, *da*, *grid*)

Read a 2D variable from a netcdf file and stores it in a petsc 3D Vector at k=level

Parameters

- **V** (*petsc Vec*) – one(!) petsc vector
- **vname** (*str*) – name of the variable in the netcdf file
- **filename** (*str*) – netcdf input filename
- **level** (*int*) – vertical level that will be stored in the petsc vector (allways 3D)
- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder

qgsolver.inout.**read_nc**(*vnames*, *filename*, *grid*)

Read variables from a netcdf file Data is loaded on all MPI tiles.

Parameters

- **vnames** (*list of str*) – list of variables names name of the variable in the netcdf variable
- **filename** (*str*) – netcdf input filename
- **grid** (*qgsolver grid object*) – grid data holder

qgsolver.inout.**read_hgrid_dimensions**(*hgrid_file*)

Reads grid dimension from netcdf file Could put dimension names as optional inputs ...

Parameters **hgrid_file** (*str*) – grid filename

Returns

- **Nx** (*int*) – length of the 'x' dimension
- **Ny** (*int*) – length of the 'y' dimension

qgsolver.inout.**get_global**(*V*, *da*, *rank*)

Returns a copy of the global V array on process 0, otherwise returns None

Parameters

- **V** (*petsc Vec*) – petsc vector object
- **da** (*petsc DMDA*) – holds the petsc grid
- **rank** (*int*) – MPI rank

Returns **Vf** – Copyt of the global array

Return type ndarray

class qgsolver.inout.**input**(*variable*, *files*, *da*)

Bases: object

Hold data that will be used Interpolate data in time

__init__(*variable*, *files*, *da*)

init data input should test if variables are 2D

update(*time*)

interpolate input data at a given time

6.2.4 qgsolver.omegainv module

class qgsolver.omegainv.**omegainv**(*da*, *grid*, *bdy_type*, *f0*, *N2*, *verbose=0*, *solver='gmres'*, *pc=None*)

Bases: object

Omega equation solver

__init__(*da*, *grid*, *bdy_type*, *f0*, *N2*, *verbose=0*, *solver='gmres'*, *pc=None*)

Setup the Omega equation solver

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **bdy_type** (*dict*) –

  prescribe vertical and lateral boundary conditions. Examples bdy_type = {'bottom': 'D', 'top': 'D'} for Dirichlet bdy conditions bdy_type = {'bottom': 'N', 'top': 'N'} for Neumann bdy conditions bdy_type = {'bottom': 'N', 'top': 'N'} for Neumann bdy conditions using PSI instead of RHO bdy_type = {'periodic': None} for horizontal periodicity
- **f0** (*float*) – averaged Coriolis frequency, used in operator
- **N2** (*ndarray*) – buoyancy frequency, used in operator
- **verbose** (*int, optional*) – degree of verbosity, 0 means no outputs
- **solver** (*str, optional*) – petsc solver: 'gmres' (default), 'bicg', 'cg'
- **pc** (*str, optional*) – what is default? preconditionner: 'icc', 'bjacobi', 'asm', 'mg', 'none'

solve(*da*, *grid*, *state*, *W=None*, *PSI=None*, *U=None*, *V=None*, *RHO=None*)

Compute the omega equation inversion The result of the inversion is held in state.W

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid

- **grid** (*qgsolver grid object*) – grid data holder
- **state** (*state object*) – ocean state
- **W** (*petsc Vec, None, optional*) – input vertical velocity that will be used for boundary conditions and masked areas
- **PSI** (*petsc Vec, None, optional*) – streamfunction, use state.PSI if None
- **V, RHO** (*U,*) – vectors used for computations of the Q vector

set_rhs(*da*, *grid*, *W*, *PSI*, *U*, *V*, *RHO*)

Compute the RHS of the omega equation i.e: 2*f0*nabla.Q with Q=-J(nabla.psi,dpsidz)

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **W** (*petsc Vec*) – vertical velocity
- **PSI** (*petsc Vec, None, optional*) – streamfunction used to compute U, V, RHO if not provided
- **U** (*petsc Vec, None, optional*) – zonal velocity
- **V** (*petsc Vec, None, optional*) – meridional velocity
- **RHO** (*petsc Vec, None, optional*) – density

set_uv_from_psi(*da*, *grid*, *PSI*)

Compute U & V from Psi: U = -dPSIdy V = dPSIdx

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec*) – streamfunction

set_rho_from_psi(*da*, *grid*, *PSI*)

Compute RHO from Psi rho=-rho0*f0/g dPSIdz

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec*) – streamfunction

set_Q(*da*, *grid*, *U=None*, *V=None*, *RHO=None*)

Compute Q vector qxu = g/f0/rho0 * (dudx*drhodx + dvdx*drhody) at u point qyv = g/f0/rho0 * (dudy*drhodx + dvdy*drhody) at v point

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **U** (*petsc Vec, None, optional*) – zonal velocity

- **V** (*petsc Vec, None, optional*) – meridional velocity
- **RHO** (*petsc Vec, None, optional*) – density

compute_divQ(*da*, *grid*)
Compute Q vector divergence

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder

6.2.5 qgsolver.pvinv module

class qgsolver.pvinv.**pvinversion**(*da*, *grid*, *bdy_type*, *sparam*, *verbose=0*, *solver='gmres'*, *pc=None*)
Bases: object

PV inversion solver

__init__(*da*, *grid*, *bdy_type*, *sparam*, *verbose=0*, *solver='gmres'*, *pc=None*)
Setup the PV inversion solver

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **bdy_type** (*dict*) –

  prescribe vertical and lateral boundary conditions. Examples bdy_type = {'bottom': 'D', 'top': 'D'} for Dirichlet bdy conditions bdy_type = {'bottom': 'N_RHO', 'top': 'N_RHO'} for Neumann bdy conditions with RHO bdy_type = {'bottom': 'N_PSI', 'top': 'N_PSI'} for Neumann bdy conditions using PSI instead of RHO bdy_type = {'periodic': None} for horizontal periodicity
- **sparam** (*ndarray*) – numpy array containing f^2/N^2
- **verbose** (*int, optional*) – degree of verbosity, 0 means no outputs
- **solver** (*str, optional*) – petsc solver: 'gmres' (default), 'bicg', 'cg'
- **pc** (*str, optional*) – what is default? preconditionner: 'icc', 'bjacobi', 'asm', 'mg', 'none'

solve(*da*, *grid*, *state*, *Q=None*, *PSI=None*, *RHO=None*, *bstate=None*, *addback_bstate=True*, *top-down_rho=False*, *numit=False*)
Compute the PV inversion Uses prioritarily optional Q, PSI, RHO for RHS and bdy conditions

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **state** (*state object*) – ocean state
- **Q** (*petsc Vec, None, optional*) – potential vorticity, use state.Q if None
- **PSI** (*petsc Vec, None, optional*) – streamfunction, use state.PSI if None
- **RHO** (*petsc Vec, None, optional*) – density, use state.RHO if None

- **bstate** (*state object, None, optional*) – background state that will be added in advective terms
- **addback_bstate** (*boolean*) – if True, add background state back to output variables ()
- **topdown_rho** (*boolean*) – if True, indicates that RHO used for top down boundary conditions is contained in state.Q at indices kdown and kup
- **numit** (*boolean*) – if True, returns the number of iterations

Returns Put PV inversion result in state.PSI

Return type state.PSI

q_from_psi(*Q*, *PSI*)
Compute PV from a streamfunction

Parameters

- **Q** (*petsc Vec*) – output vector where data is stored
- **PSI** (*petsc Vec*) – input streamfunction used for the computation of PV

set_rhs_bdy(*da*, *grid*, *state*, *PSI*, *RHO*, *topdown_rho*)
Set South/North, East/West, Bottom/Top boundary conditions Set RHS along boundaries for inversion, may be an issue for time stepping

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **state** (*state object*) – ocean state
- **PSI** (*petsc Vec, None, optional*) – streamfunction, use state.PSI if None
- **RHO** (*petsc Vec, None, optional*) – density, use state.RHO if None
- **topdown_rho** (*boolean*) – if True, indicates that RHO used for top down boundary conditions is contained in state.Q at indices kdown and kup

set_rhs_bdy_bottom(*da*, *grid*, *state*, *PSI*, *RHO*, *topdown_rho*)
Set bottom boundary condition

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **state** (*state object*) – ocean state
- **PSI** (*petsc Vec, None, optional*) – streamfunction, use state.PSI if None
- **RHO** (*petsc Vec, None, optional*) – density, use state.RHO if None
- **topdown_rho** (*boolean*) – if True, indicates that RHO used for top down boundary conditions is contained in state.Q at indices kdown and kup

set_rhs_bdy_top(*da*, *grid*, *state*, *PSI*, *RHO*, *topdown_rho*)
Set top boundary condition

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid

- **grid** (*qgsolver grid object*) – grid data holder
- **state** (*state object*) – ocean state
- **PSI** (*petsc Vec, None, optional*) – streamfunction, use state.PSI if None
- **RHO** (*petsc Vec, None, optional*) – density, use state.RHO if None
- **topdown_rho** (*boolean*) – if True, indicates that RHO used for top down boundary conditions is contained in state.Q at indices kdown and kup

set_rhs_bdy_lat(*da*, *grid*, *PSI*)
Set lateral boundary condition

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec, None, optional*) – streamfunction, use state.PSI if None

set_rhs_mask(*da*, *grid*, *PSI*)
Set mask on rhs: where mask=0 (land) rhs=psi

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec*) – streamfunction used over masked areas

6.2.6 qgsolver.qg module

class qgsolver.qg.**qg_model**(*ncores_x=None*, *ncores_y=None*, *hgrid=None*, *vgrid=None*, *vdom={}*, *hdom={}*, *mask=False*, *boundary_types={}*, *N2=0.001*, *f0=7e-05*, *f0N2_file=None*, *dt=None*, *K=100.0*, *verbose=1*, *flag_pvinv=True*, *flag_omega=False*, ***kwargs*)
Bases: object

QG model

__init__(*ncores_x=None*, *ncores_y=None*, *hgrid=None*, *vgrid=None*, *vdom={}*, *hdom={}*, *mask=False*, *boundary_types={}*, *N2=0.001*, *f0=7e-05*, *f0N2_file=None*, *dt=None*, *K=100.0*, *verbose=1*, *flag_pvinv=True*, *flag_omega=False*, ***kwargs*)
QG model initializer

Parameters

- **ncores_x** (*int*) – number of MPI tilings in x direction
- **ncores_y** (*int*) – number of MPI tilings in y direction
- **hgrid** (*dict or str*) – defines horizontal grid choice
- **vgrid** (*dict or str*) – defines vertical grid choice
- **boundary_types** (*dict*) – may be used to turn on periodic boundary conditions {'periodic'}
- **N2** (*float, optional*) – Brunt Vaisala frequency, default=1.e-3
- **f0** (*float, optional*) – Coriolis frequency, default=7e-5
- **f0N2_file** (*str*) – netcdf file containing N2 and f0

- **dt** (*float, optional*) – time step
- **K** (*float, optional*) – dissipation coefficient, default = 1.e2
- **verbose** (*int, optional*) – degree of verbosity, 0 means no outputs
- **flag_pvinv** (*boolean, optional*) – turn on setup of PV inversion solver, default is True
- **flag_omega** (*boolean, optional*) – turn on setup of omega equation inversion solver, default is False

set_psi(***kwargs*)
Set psi, wrapper around state.set_psi

set_q(***kwargs*)
Set q, wrapper around state.set_q

set_rho(***kwargs*)
Set rho, wrapper around state.set_rho

set_bstate(***kwargs*)
Set background state

set_w(***kwargs*)
Set w, wrapper around state.set_w

invert_pv(*bstate=None*, *addback_bstate=True*)
wrapper around pv inversion solver pvinv.solve

invert_omega()
wrapper around solver solve method omegainv.solve

tstep(*nt=1*, *rho_sb=True*, *bstate=None*)
Time step wrapper tstepper.go

write_state(*v=['PSI', 'Q'], vname=['psi', 'q'], filename='output.nc', append=False*)
Outputs state to a netcdf file

Parameters

- **v** (*list of str*) – List of variables to output (must be contained in state object)
- **vname** (*list of str*) – list of the names used in netcdf files
- **filename** (*str*) – netcdf output filename
- **create** (*boolean, optional*) – if true creates a new file, append otherwise (default is True)

compute_CFL(*PSI=None*)
Compute CFL = max (u*dt/dx)

Parameters **PSI** (*petsc Vec, optional*) – PSI vector used for velocity computation

Returns **CFL** – CFL number

Return type float

compute_KE(*PSI=None*)
Compute the domain averaged kinetic energy, wrapper around state.compute_KE

Parameters **PSI** (*petsc Vec, optional*) – PSI vector used for velocity computation

Returns **KE** – Kinetic energy in m/s

Return type float

6.2.7 qgsolver.state module

class qgsolver.state.**state**(*da*, *grid*, *N2=0.001*, *f0=7e-05*, *f0N2_file=None*, *verbose=0*)
Bases: object

Ocean state variable holder

__init__(*da*, *grid*, *N2=0.001*, *f0=7e-05*, *f0N2_file=None*, *verbose=0*)
Declare Petsc vectors

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **N2** (*float, optional*) – Brunt Vaisala frequency, default=1.e-3
- **f0** (*float, optional,*) – Coriolis frequency, default=7.e-5
- **f0N2_file** (*str, optional*) – netcdf file containing N2 and f0, default is None
- **verbose** (*int, optional*) – degree of verbosity, 0 means no outputs

set_psi(*da*, *grid*, *analytical_psi=True*, *psi0=0.0*, *file=None*, ***kwargs*)
Set psi (streamfunction)

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **analytical_psi** (*boolean, optional*) – True set psi analytically, default is True
- **file** (*str, optional*) – filename where psi can be found

set_psi_analytically(*da*, *psi0*)
Set psi analytically

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **psi0** (*float*) – amplitude used to set the streamfunction

set_q(*da*, *grid*, *analytical_q=True*, *q0=1e-05*, *beta=0.0*, *file=None*, ***kwargs*)
Set q (PV)

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **analytical_psi** (*boolean, optional*) – True set psi analytically, default is True
- **file** (*str, optional*) – filename where q can be found
- **q0** (*float, optional*) – amplitude of the PV anomaly, default is 1.e-5
- **beta** (*float, optional*) – meridional variations of planetary vorticity, default is 0

set_q_analytically(*da*, *grid*, *q0*, *beta*)
Set q analytically

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid

- **grid** (*qgsolver grid object*) – grid data holder
- **q0** (*float*) – amplitude of the PV anomaly
- **beta** (*float*) – meridional variations of planetary vorticity

set_rho(*da*, *grid*, *analytical_rho=True*, *rhoana=0.0*, *file=None*, ***kwargs*)
Set rho (density)

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **analytical_psi** (*boolean, optional*) – True set psi analytically, default is True
- **file** (*str, optional*) – filename where rho can be found

set_rho_analytically(*da*, *rhoana*)
Set rho analytically

Parameters **da** (*petsc DMDA*) – holds the petsc grid

set_w(*da*, *grid*, *analytical_w=True*, *file=None*, ***kwargs*)
Set w

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **analytical_psi** (*boolean, optional*) – True set psi analytically, default is True
- **file** (*str, optional*) – filename where w can be found

set_w_analytically(*da*)
Set w analytically

Parameters **da** (*petsc DMDA*) – holds the petsc grid

update_rho(*da*, *grid*, *PSI=None*, *RHO=None*)
update rho from psi

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec, optional*) – PSI vector used for computation density, use state.PSI if None
- **RHO** (*petsc Vec, optional*) – RHO vector where output is stored, store in state.RHO if None

get_uv(*da*, *grid*, *PSI=None*)
Compute horizontal velocities from Psi: state._U = -dPSIdy, state._V = dPSIdx

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec, optional*) – PSI vector used for velocity computation

compute_KE(*da*, *grid*, *PSI=None*)
Compute domain averaged kinetic energy = 0.5 * sum (u**2+v**2)

Parameters

- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **PSI** (*petsc Vec, optional*) – PSI vector used for velocity computation

Returns **KE** – Kinetic energy in m/s

Return type float

qgsolver.state.**add**(*state1*, *state2*, *da=None*, *a1=1.0*, *a2=1.0*)
add fields of two states: a1*state1 + a2*state2

Parameters

- **state1** (*qgsolver state*) –
- **state2** (*qgsolver state*) –
- **da** (*None or petsc DMDA*) – if None state1 is updated; otherwise a new state is created
- **a1** (*float, optional*) – default value = 1.
- **a2** (*float, optional*) – default value = 1.

6.2.8 qgsolver.timestepper module

class qgsolver.timestepper.**time_stepper**(*da*, *grid*, *dt*, *K*, *petscBoundaryType*, *verbose=0*, *t0=0.0*)
Bases: object

Time stepper, parallel with petsc4py 4 steps explicit RungeKutta

go(*nt*, *da*, *grid*, *state*, *pvinv*, *rho_sb*, *bstate=None*)
Carry out the time stepping

Parameters

- **nt** (*int*) – Number of time steps
- **da** (*petsc DMDA*) – holds the petsc grid
- **grid** (*qgsolver grid object*) – grid data holder
- **state** (*state object*) – ocean state that will be timestepped
- **pvinv** (*pv inversion object*) – PV inverser
- **rho_sb** (*boolean, optional*) – turn on advection of surface and bottom densities, default if false
- **bstate** (*state object, None, optional*) – background state that will be added in advective terms

6.2.9 qgsolver.utils module

class qgsolver.utils.**plt**
Bases: object

perform online basic plots

6.2.10 qgsolver.window module

class qgsolver.window.**window**(*hgrid=None*, *vgrid=None*, *K=1e-06*, *vdom={}*, *hdom={}*, *ncores_x=None*, *ncores_y=None*, *bdy_type_in={}*, *mask3D=False*, *verbose=1*)

Bases: object

Computes a window for spectral computations

__init__(*hgrid=None*, *vgrid=None*, *K=1e-06*, *vdom={}*, *hdom={}*, *ncores_x=None*, *ncores_y=None*, *bdy_type_in={}*, *mask3D=False*, *verbose=1*)

Window model creation Parameters:

set_q(*analytical_q=True*, *file_q=None*)

Set q to a given value

set_q_analytically()

Set q analytically

invert_win()

wrapper around solver solve method

class qgsolver.window.**wininversion**(*win*)

Bases: object

Window inversion, parallel

__init__(*win*)

Setup the PV inversion solver

solve(*win*)

Compute the PV inversion

set_rhs_bdy(*win*)

Set South/North, East/West, Bottom/Top boundary conditions Set RHS along boundaries for inversion, may be an issue for time stepping

Parameters

- **da** – abstract distributed memory object of the domain
- **win** – win_model instance

set_rhs_mask(*win*)

Set mask on rhs: where mask=0 (land) rhs=psi

Parameters

- **da** – abstract distributed memory object of the domain
- **win** – win_model instance

6.2.11 Module contents

CHAPTER 7

Indices and tables

- genindex
- search

Python Module Index

q

Index

Symbols

A

C

G

I

L

O

P

Q

R

S

T

U

W