SeisFlows Documentation

Release 0.1

Dec 05, 2020

CONTENTS

CHAPTER

ONE

OVERVIEW

SeisFlows is a Python waveform inversion package with a growing user base in academia and industry. So far, the package has been used for production runs with a billion or so model parameters and for research on oil and gas exploration, earthquake seismology, and general nonlinear optimization problems.

To provide flexibility, SeisFlows is very modular. Users are offered choices in each of the following categories:

- workflow
- system
- solver
- pre-processing
- post-processing
- nonlinear optimization

The thing that ties everything together is the workflow. Execution of a workflow is equivalent to stepping through the code contained in `workflow.main`. Users are free to customize the default 'inversion' and 'migration' workflows from the main package.

A number of options exist for system and solver. By isolating system and solver machinery, users can switch from one application to another with relative ease. For example, if the study area in an earthquake tomography project expands, users can trade a regional Cartesian solver for a global solver. If a PBS cluster goes offline and a SLURM cluster comes online to replace it, users can trade the PBS system interface for a SLURM system interface.

A selection of ready-to-go system and solver interfaces is provided in the main package. Through these interfaces, SeisFlows (or prototypes of it) have run on clusters managed by the Department of Defense, Chevron Corp., Total S.A., Princeton University and other universities and institutions.

Users can also choose from various pre-processing and post-processing options. In our terminology, pre-processing consists of signal processing operations performed on seismic traces prior to the gradient computation. Post-processing consists of regularization or image processing operations carried out after the gradient computation.

If desired functionality is missing from the main package, users can contribute their own classes or overload default ones.

CHAPTER

TWO

USAGE

2.1 Installation

To install Seisflows, first clone the repository:

```
git clone https://github.com/rmodrak/seisflows.git
```

Then set environment variables. Add the following lines to `.bashrc` (or modify accordingly if using a shell other than bash):

```
export PATH=$PATH:/path/to/seisflows/scripts
export PYTHONPATH=$PYTHONPATH:/path/to/seisflows
```

2.1.1 Software Prerequisites

SeisFlows requires Python 2.7, Numpy, Scipy and Obspy. Users will need to install these packages before being able to use SeisFlows.

Forward modeling software is also a prerequisite; see *Solver Configuration* for more information.

2.1.2 Hardware Prerequisites

Access to a computer cluster is required for most applications. Base classes are provided for several common cluster configurations, including PBS and SLURM. Nonstandard configurations can often be accommodated through modifications to one of the base classes; see *System Configuration* for details.

2.2 Job Submission

Each job must have it own *working directory* within which users must supply two input files, `paths.py` and `parameters.py`.

To begin executing a workflow, simply type `sfrun` within a working directory. If an `inversion` workflow and `serial` system configuration, for example, are specified in the parameters file, the inversion will begin executing immediately in serial. If a PBS, SLURM, or LSF system configuration is specified instead, execution may wait until required resources become available.

Once the workflow starts running, status information is displayed to the terminal or to the file `output.log`. By default, updated models and other inversion results are output to the working directory.

2.3 Solver Configuration

SeisFlows includes Python interfaces for SPECFEM2D, SPECFEM3D, and SPECFEM3D_GLOBE. While the Python interfaces are part of the SeisFlows package, the solver source code must be downloaded separately through GitHub.

After downloading the solver source code, users must configure and compile it, following the instructions in the solver user manual. Summarized briefly, the configuration and compilation procedure is:

Prior to compilation, users need to run the `configure` script and prepare input files such as

- parameter file
- source file
- stations file.

To successfully run the `configure`, you may need to install compilers, libraries, and other software in your environment.

The result of compilation is a set of binary executables, such as

- mesher
- solver
- smoothing utility
- summing utility.

After compilation, solver input files must be gathered together in one directory and solver executables in another. The absolute paths to the directories containing input files and executables must be given in `paths.py`.

2.3.1 Writing Custom Solver Interfaces

Besides SPECFEM2D, SPECFEM3D, and SPECFEM3D_GLOBE, SeisFlows can interface with other solvers capable of running forward and adjoint simulations. Users unaffiliated with the main SeisFlows developers have succeeded in interfacing with, for example, their own finite difference solvers. For information about writing custom solver interfaces, see *Developer Reference*.

2.3.2 Design Philosophy

Integration of the solver with the other workflow components can be challenging. Here we try to give an idea of the issues involved from both a developer and a user standpoint.

- Solver computations account for most of the cost of an inversion. As a result, the solver must be written in an efficient compiled language, and wrappers must be written to integrate the compiled code with other software components.
- There is currently no mechanism for automatically compiling executables for SPECFEM2D, SPECFEM3D, or SPECFEM3D_GLOBE. Users must prepare their own SPECFEM input files and then follow the compilation procedure in the SPECFEM documentation.
- As described above *Job Submission*, SeisFlows uses two input files, `paths.py` and `parameters.py`. Problems could arise if parameters from SeisFlows input files conflict with parameters from solver input file. Users must make sure that there are no conflicts between SeisFlows parameters and solver parameters.
- In the solver routines, it's natural to represent velocity models as dictionaries, with different keys corresponding to different material parameters. In the optimization routines, it's natural to represent velocity models as vectors. To convert back and forth between these two representations, a pair of utility functions–`split` and `merge`–are included in solver.base.

2.4 System Configuration

SeisFlows can run on SLURM, PBS, and LSF clusters, as well as, for very small problems, laptops or desktops. A list of available system interface classes follows. By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

PBS_SM - For small inversions on PBS clusters. All resources are allocated at the beginning and all simulations are run within a single job. Requires that individual wavefield simulations run each on a single core, making this option suitable for small 2D inversions only.

PBS_LG - For large inversions on PBS clusters. The work of the inversion is divided between multiple jobs that are coordinated by a single long-running master job. Resources are allocated on a per simulation basis. Suitable for small to medium 3D inversions in which individual wavefield simulation span several or more nodes.

SLURM_SM - For small inversions on SLURM clusters. All resources are allocated at the beginning and all simulations are run within a single job. Requires that each individual wavefield simulation runs only a single core, making this option suitable for small 2D inversions only.

SLURM_LG - For large inversions on SLURM clusters. The work of the inversion is divided between multiple jobs that are coordinated by a single long-running master job. Resources are allocated on a per simulation basis. Suitable for 3D inversions in which individual wavefield simulation span several or more nodes.

SERIAL - Tasks that are normally carried out in parallel are instead carried out one at a time. Useful for debugging, but not much else.

MULTITHREADED - On desktops or laptops with multiple cores, allows embarrassingly parallel tasks to be carried out several at a time, rather than one at a time. Can be used to run small 2D inversions on a laptop or desktop.

LSF_SM - Same as SLURM_SM and PBS_SM, except for LSF clusters.

LSF_LG - Same as SLURM_LG and PBS_LG, except for LSF clusters.

2.4.1 Writing Custom System Interfaces

If your needs are more specialized, please view `seisflows.system` source code to get a sense for how to write your own custom system interfaces. In our experience, system interfaces require no more than a few hundred lines of code, so writing your own is generally possible once you are familiar with the SeisFlows framework and your own cluster environment.

2.4.2 Design Philosophy

To make SeisFlows work across different environments, our approach is to wrap system commands with a thin Python layer. To handle job submission, for example, we wrap the PBS command `qsub` and the SLURM command `sbatch` with a python utility called *system.submit*. The result is a consistent python interface across different clusters.

Filesystem settings can be adjusted by modifying values in the `PATH` dictionary, which is populated from `paths.py`. Output files and temporary files, by default, are written to the working directory. If a value for `PATH.SCRATCH` is supplied, temporary files are written there instead. If each compute node has its own local filesystem, a value for `PATH.LOCAL` can be supplied so that temporary files required only for a local process need not be written to the global filesystem.

As the size of an inversion grows, scalability and fault tolerance become increasingly important. If a single forward simulation spans more than one node, users must select `pbs_lg` or `slurm_lg` system configurations in `parameters.py`. If a forward simulation fits onto a single node, users should select `pbs_sm` or `slurm_sm` instead.

In SeisFlows, the overall approach to solving system interface problems is to use lightweight Python wrappers. For complex cluster configurations, heavier-weight solutions may be required. Users are referred to SAGA or Pegasus projects for ideas.

2.5 Developer Reference

To allow classes to work with one another, each must conform to an established interface. This means certain classes must implement certain methods, with specified input and output. Required methods include

- `setup` methods are generic methods, called from the `main` workflow script and meant to provide users the flexibility to perform any required setup tasks.
- `check` methods are the default mechanism for parameter declaration and checking and are called just once, prior to a job being submitted through the scheduler.

Besides required methods, classes may include any number of private methods or utility functions.

CHAPTER

THREE

EXECUTABLE

Seisflows provides several executable Python scripts in `/path/to/seisflows/scripts`. As the PATH has already been added to environmental varibales, thereby can be executed in any directory.

Tests checking module importing, system, and optimization are also provided in `/path/to/seisflows/tests`. It's recommmended to past these tests prior to other projects.

3.1 Scripts

- sfrun (sfsubmit): Workflow submission scripts.
- plotgll: Plots GLL model read from SPECFEM2D Fortran binary file
- sfclean: Delete directories output, output.stat, output.optim and scratch in the current directory
- sfresume: Resume current submitted workflow.
- sfexample: Run seisflows example. **Currently not available**.

3.2 Tests

- run_test_system: Test parallelizaiton environment.
- run_test_import: Testing seisflows module importing.
- run_test_optimize:
- run_test_preprocess:
- run_test_tools:

CHAPTER

FOUR

DESIGNING PROJECT

Before submitting jobs to the local server, parameters defined in `parameters.py` and `path.py` require a careful check, avoiding any unnecessary workload.

4.1 Parameter Files: `parameters.py`

`parameters.py` contains a list of parameter names and values. Prior to a job being submitted, parameters are checked so that errors can be detected without loss of queue time or wall time. Parameters are stored in a dictionary that is accessible from anywhere in the Python code. By convention, all parameter names must be upper case. Parameter values can be floats, integers, strings or any other Python data type. Parameters can be listed in any order.

`General`

`TITLE` Project title.

`WORKFLOW` Workflow specified for seisflows. 'inversion' and 'migration' are currently supported.

`SOLVER` Time domain solver specified for seisflows. See *Solver Configuration* and *seisflows.solver package* for supported options.

`SYSTEM` System type supported for seisflows. See *System Configuration* and *seisflows.system package* for supported options..

`OPTIMIZE` Optimization method used for inversion. See *seisflows.optimize package* for supported options.

`PREPROCESS` Preprocessing workflow specified. See *seisflows.preprocess package* for supported options.

`POSTPROCESS` Postprocessing workflow specified. 'base' needs to be specified.

`MISFIT` Type of misfit for evaluation. See *seisflows.plugins.misfit module* for supported options.

`MATERIALS` Materials of simulation domain. 'Elastic' and 'Acoustic' are currently supported. See seisflows.solver.base.

`Workflow`

`BEGIN` First iteration index.

`END` Last iteration index.

`NREC` Number of receivers.

`NSRC` Number of sources. SEM source file needs to be stored with a six-digit index suffix.

`SAVEMODEL` Frequency of saving model. `1` by default.

`SAVEGRADIENT` Frequency of saving gradient. `1` by default.

SAVEKERNELS Frequency of saving kernels. `0` by default.

SAVETRACES Frequency of saving traces. `0` by default.

SAVERESIDUALS Frequency of saving residuals. `0` by default.

`Preprocessing`

FORMAT Data file format.

CHANNELS Data channels. Currenly, 'su', or 'SU' need to be specified.

NORMALIZE Apply normalization for traces. 'NormalizeEventsL1', 'NormalizeEventsL2', 'NormalizeTracesL1', 'NormalizeTracesL2' are currently supported. See seisflows.preprocess.

`Filter`

BANDPASS Boolean type bandpass switch for traces.

FILTER Type of filter used. See *seisflows.preprocess package* for supported options.

FREQMIN Low frequency corner.

FREQMAX High frequency corner.

`Mute`

MUTE List type switch for trace mute. *seisflows.preprocess package* for supported options.

`Postprocessing`

SMOOTH Smoothing radius. See `xsmooth_sem` for usage.

`Optimization`

PRECOND Preconditioner type. See *Path Files: paths.py* and *seisflows.plugins.preconds.diagonal module*.

STEPMAX Maximum trial steps

`Solver`

NT Number of time steps defined in Par_file.

DT Time step defined in Par_file.

F0 Dominant frequency defined in SOURCE.

`System`

NTASK Number of tasks submitted. Currently, **NTASK** must satisfy 1 <= NTASK <= NSRC.

NPROC Number of processors.

MPIEXEC MPI executable prefix, e.g., `mpirun -np 13`. Note for a space at the end of the string, as seisflows concatenates the prefix with SPECFEM executable command.

4.2 Path Files: `paths.py`

`paths.py` contains a list of path names and values. Prior to a job being submitted, paths are checked so that errors can be detected without loss of queue time or wall time. Paths are stored in a dictionary that is accessible from anywhere in the Python code. By convention, all names must be upper case, and all values must be absolute paths. Paths can be listed in any order.

DATA PATH contains seismic data if field data is used for inversion. Data of difference sources should be stored in separate folder. If `DATA` directory does not exist, seisflows would automatically generate synthetic data using model from `MODEL_TRUE`.

MODEL_INIT PATH contains model file for initial iteraion.

MODEL_TRUE PATH contains true model for generating synthetic data.

PRECOND PATH to user supplied diagonal preconditioner. Seisflows will rescale model parameters based on user supplied weights. See *seisflows.plugins.preconds.diagonal module*.

MASK PATH to mask file for gradient scaling. Mask needs to be stored mimicking the file format in which models are stored.

SPECFEM_DATA PATH to SPECFEM `DATA` directory which contians Par_file, SOURCE, and other necessary inputs.

SPECFEM_BIN PATH to SPECFEM `bin` directory which contains binary executable command of SPECFEM solver.

CHAPTER

FIVE

EXAMPLE

5.1 Examples: Available For Download

We have prepared a 2D waveform inversion example that is inexpensive enough to run on almost any laptop, desktop, or cluster.

Some additional examples are available for download. Please review the instructions for the 2D checkerboard test case to get a sense for how to run these other inversions.

Some 2D examples based on the Marmousi model are available here.

A 3D Cartesian checkerboard example is available here.

A 3D global 1-chunk example is available here. Please note, the compressed archive for this example is very large (> 0.5 GB). *[No longer available because of file size.]*

At a minimimum, one processer is required for the 2D Marmousi examples, 16 processors are required for the 3D Cartesian example, and 64 processors are required for the global 1-chunk example. See here for more information about running inversions in parallel.

Note: File hosting services are provided by my alma mater. The download server may become temporarily unavailable due to system maintenance or permanently unavailable due to expiration of my account.

5.2 Examples: Available Locally

Users with accounts on "tiger.princeton.edu" can run the following inversions without having to download files or recompile executables.

2D Regional and Global

- North America
- Southern California
- Global
- Deep Earth

2D Near Surface

- Marmousi offshore
- Marmousi onshore
- overthrust offshore
- overthrust onshore

- BP anticline
- BP salt diapir

3D Cartesian

- checkerboard

3D Global

- mideast

CHAPTER

SIX

MODULE

6.1 seisflows.optimize package

6.1.1 seisflows.optimize.base module

class seisflows.optimize.base.**base**
Bases: object

Nonlinear optimization abstract base class

Base class on top of which steepest descent, nonlinear conjugate, quasi- Newton and Newton methods can be implemented. Includes methods for both search direction and line search.

To reduce memory overhead, vectors are read from disk rather than passed from calling routines. For example, at the beginning of compute_direction the current gradient is read from 'g_new' and the resulting search direction is written to 'p_new'. As the inversion progresses, other information is stored as well.

Variables m_new - current model m_old - previous model m_try - line search model f_new - current objective function value f_old - previous objective function value f_try - line search function value g_new - current gradient direction g_old - previous gradient direction p_new - current search direction p_old - previous search direction

check()
Checks parameters, paths, and dependencies

compute_direction()
Computes search direction

dot(*x*, *y*)
Computes inner product between vectors

finalize_search()
Prepares algorithm machinery and scratch directory for next model upate

initialize_search()
Determines first step length in line search

load(*filename*)

loadtxt(*filename*)

model_cutoff(*m*)

restart()
Restarts nonlinear optimization algorithm

Keeps current position in model space, but discards history of nonlinear optimization algorithm in an attempt to recover from numerical stagnation

retry_status()
Determines if restart is worthwhile

After failed line search, determines if restart is worthwhile by checking, in effect, if search direction was the same as gradient direction

save(*filename*, *array*)

savetxt(*filename*, *scalar*)

setup()
Sets up nonlinear optimization machinery

update_search()
Updates line search status and step length

Status codes status > 0 : finished status == 0 : not finished status < 0 : failed

6.1.2 seisflows.optimize.LBFGS module

class seisflows.optimize.LBFGS.**LBFGS**
Bases: *seisflows.optimize.base.base*

Limited memory BFGS algorithm

check()
Checks parameters, paths, and dependencies

compute_direction()
Computes search direction

restart()
Restarts nonlinear optimization algorithm

Keeps current position in model space, but discards history of nonlinear optimization algorithm in an attempt to recover from numerical stagnation

setup()
Sets up nonlinear optimization machinery

6.1.3 seisflows.optimize.NLCG module

class seisflows.optimize.NLCG.**NLCG**
Bases: *seisflows.optimize.base.base*

Nonlinear conjugate gradient method

check()
Checks parameters, paths, and dependencies

compute_direction()
Computes search direction

restart()
Restarts nonlinear optimization algorithm

Keeps current position in model space, but discards history of nonlinear optimization algorithm in an attempt to recover from numerical stagnation

setup()
Sets up nonlinear optimization machinery

6.1.4 seisflows.optimize.steepest_descent module

class seisflows.optimize.steepest_descent.**steepest_descent**
Bases: *seisflows.optimize.base.base*

Steepest descent method

check()
Checks parameters, paths, and dependencies

compute_direction()
Computes search direction

restart()
Restarts nonlinear optimization algorithm

Keeps current position in model space, but discards history of nonlinear optimization algorithm in an attempt to recover from numerical stagnation

restarted = False

setup()
Sets up nonlinear optimization machinery

6.2 seisflows.plugins package

6.2.1 seisflows.plugins.line_search package

seisflows.plugins.line_search.backtrack module

class seisflows.plugins.line_search.backtrack.**Backtrack**(*step_count_max=10*, *step_len_max=inf*, *path='/Users/niyiyu/Documents/GitHub/seisflows/d*
Bases: *seisflows.plugins.line_search.bracket.Bracket*

Implements backtracking linesearch

Variables x - list of step lenths from current line search f - correpsonding list of function values gtg - dot product of gradient with itself gtp - dot product of gradient and search direction

Status codes status > 0 : finished status == 0 : not finished status < 0 : failed

calculate_step()
Determines step length and search status

seisflows.plugins.line_search.base module

class seisflows.plugins.line_search.base.**Base**(*step_count_max=10*, *step_len_max=inf*, *path='/Users/niyiyu/Documents/GitHub/seisflows/docs'*)
Bases: object

Abstract base class for line search

Variables x - list of step lenths from current line search f - correpsonding list of function values m - how many step lengths in current line search? n - how many model updates in optimization problem? gtg - dot product of gradient with itself gtp - dot product of gradient and search direction

Status codes status > 0 : finished status == 0 : not finished status < 0 : failed

calculate_step()

clear_history()
Clears line search history

initialize(*step_len*, *func_val*, *gtg*, *gtp*)

search_history(*sort=True*)
A convenience function, collects information needed to determine search status and calculate step length

update(*step_len*, *func_val*)

class seisflows.plugins.line_search.base.**Writer**(*path='./output.optim'*)
Bases: object

Utility for writing one or more columns to text file

newline()

write_header()

seisflows.plugins.line_search.bracket module

class seisflows.plugins.line_search.bracket.**Bracket**(*step_count_max=10*, *step_len_max=inf*, *path='/Users/niyiyu/Documents/GitHub/seisflows/docs'*)
Bases: *seisflows.plugins.line_search.base.Base*

Implements bracketing line search

Variables x - list of step lenths from current line search f - correpsonding list of function values gtg - dot product of gradient with itself gtp - dot product of gradient and search direction

Status codes status > 0 : finished status == 0 : not finished status < 0 : failed

calculate_step()
Determines step length and search status

6.2.2 seisflows.plugins.optimize package

seisflows.plugins.optimize.LBFGS module

class seisflows.plugins.optimize.LBFGS.**LBFGS**(*path='.'*, *load=<function loadnpy>*, *save=<function savenpy>*, *memory=5*, *thresh=0.0*, *maxiter=inf*, *precond=None*)
Bases: object

Limited-memory BFGS algorithm

Includes optional safeguards: periodic restarting and descent conditions.

To conserve memory, most vectors are read from disk rather than passed from a calling routine.

apply(*q*, *S=[]*, *Y=[]*)
Applies L-BFGS inverse Hessian to given vector

check_status(*g*, *r*)

restart()
Discards history and resets counters

update()
Updates L-BFGS algorithm history

seisflows.plugins.optimize.LCG module

class seisflows.plugins.optimize.LCG.**LCG**(*path*, *load=<function loadnpy>*, *save=<function savenpy>*, *thresh=inf*, *maxiter=inf*, *precond=None*)

Bases: object

CG solver

apply_precond(*r*)

check_status(**args*, ***kwargs*)

initialize()

update(*ap*)

seisflows.plugins.optimize.NLCG module

class seisflows.plugins.optimize.NLCG.**NLCG**(*path='.'*, *load=<function loadnpy>*, *save=<function savenpy>*, *thresh=1.0*, *maxiter=inf*, *precond=None*)

Nonlinear conjugate gradient method

restart()
Restarts algorithm

seisflows.plugins.optimize.NLCG.**check_conjugacy**(*g_new*, *g_old*)

seisflows.plugins.optimize.NLCG.**check_descent**(*p_new*, *g_new*)

seisflows.plugins.optimize.NLCG.**fletcher_reeves**(*g_new*, *g_old*, *precond=<function <lambda>>*)

seisflows.plugins.optimize.NLCG.**pollak_ribere**(*g_new*, *g_old*, *precond=<function <lambda>>*)

seisflows.plugins.optimize.PLCG module

class seisflows.plugins.optimize.PLCG.**LBFGS_**(*path='.'*, *load=<function loadnpy>*, *save=<function savenpy>*, *memory=5*, *thresh=0.0*, *maxiter=inf*, *precond=None*)

Bases: *seisflows.plugins.optimize.LBFGS.LBFGS*

Adapts L-BFGS from nonlinear optimization to preconditioning

class seisflows.plugins.optimize.PLCG.**PLCG**(*path*, *eta=1.0*, ***kwargs*)

Bases: *seisflows.plugins.optimize.LCG.LCG*

Preconditioned truncated-Newton CG solver

Adds preconditioning and adaptive stopping to LCG base class

apply_precond(*r*)

check_status(*ap*, *verbose=True*)
Checks Eisenstat-Walker termination status

6.2.3 seisflows.plugins.preconds package

seisflows.plugins.preconds.diagonal module

class seisflows.plugins.preconds.diagonal.**Diagonal**
Bases: object

User supplied diagonal preconditioner

Rescales model parameters based on user supplied weights

6.2.4 seisflows.plugins.solver package

seisflows.plugins.solver.specfem2d module

seisflows.plugins.solver.specfem2d.**smooth_legacy**(*input_path=''*, *output_path=''*, *parameters=[]*, *span=0.0*)

seisflows.plugins.solver.specfem2d.**write_receivers**(*coords*, *path='.'*)
Writes receiver information to text file

seisflows.plugins.solver.specfem2d.**write_sources**(*coords*, *path='.'*, *ws=1.0*, *suffix=''*)
Writes source information to text file TODO this has to be adapted for new versions of specfem because the source file format has changed

seisflows.plugins.solver.specfem3d module

seisflows.plugins.solver.specfem3d.**write_receivers**(*h*)
Writes receiver information to text file

seisflows.plugins.solver.specfem3d.**write_sources**(*PAR*, *h*, *path='.'*)
Writes source information to text file

seisflows.plugins.solver.specfem3d_globe module

seisflows.plugins.solver.specfem3d_globe.**write_parameters**(*par*, *version*)
Writes parameters to text file

seisflows.plugins.solver.specfem3d_globe.**write_receivers**(*h*)
Writes receiver information to text file

seisflows.plugins.solver.specfem3d_globe.**write_sources**(*PAR*, *h*, *path='.'*)
Writes source information to text file

6.2.5 seisflows.plugins.solver_io package

seisflows.plugins.solver_io.adios module

seisflows.plugins.solver_io.adios.**mread**(*path*, *parameters*, *iproc*, *prefix=''*, *suffix=''*)
Multiparameter read, callable by a single mpi process

seisflows.plugins.solver_io.adios.**read**(*path*, *parameter*, *iproc*)
Reads from ADIOS container

seisflows.plugins.solver_io.adios.**write**(*v*, *path*, *parameter*, *iproc*)
Writes to ADIOS container

seisflows.plugins.solver_io.fortran_binary module

seisflows.plugins.solver_io.fortran_binary.**copy_slice**(*src*, *dst*, *iproc*, *parameter*)
Copies SPECFEM model slice

seisflows.plugins.solver_io.fortran_binary.**read_slice**(*path*, *parameters*, *iproc*)
Reads SPECFEM model slice(s) Such as, for example : proc000005_vp.bin In that specific case it would be : read_slice(path, 'vp', 5)

seisflows.plugins.solver_io.fortran_binary.**write_slice**(*data*, *path*, *parameters*, *iproc*)
Writes SPECFEM model slice

6.2.6 seisflows.plugins.adjoint module

seisflows.plugins.adjoint.**Acceleration**(*syn*, *obs*, *nt*, *dt*)

seisflows.plugins.adjoint.**Amplitude**(*syn*, *obs*, *nt*, *dt*)
Cross correlation amplitude

seisflows.plugins.adjoint.**Displacement**(*syn*, *obs*, *nt*, *dt*)

seisflows.plugins.adjoint.**Envelope**(*syn*, *obs*, *nt*, *dt*, *eps=0.05*)
Envelope difference (Yuan et al 2015, eq 16)

seisflows.plugins.adjoint.**Envelope2**(*syn*, *obs*, *nt*, *dt*, *eps=0.0*)
Envelope amplitude ratio (Yuan et al 2015, eq B-2, B-3)

seisflows.plugins.adjoint.**Envelope3**(*syn*, *obs*, *nt*, *dt*, *eps=0.0*)
Envelope cross-correlation lag (Yuan et al 2015, eqs B-2, B-5)

seisflows.plugins.adjoint.**InstantaneousPhase**(*syn*, *obs*, *nt*, *dt*, *eps=0.05*)
Instantaneous phase (from Bozdag et al. 2011, eq 27)

seisflows.plugins.adjoint.**InstantaneousPhase2**(*syn*, *obs*, *nt*, *dt*, *eps=0.0*)

seisflows.plugins.adjoint.**Traveltime**(*syn*, *obs*, *nt*, *dt*)
Cross correlation traveltime (Tromp et al 2005, eq 45)

seisflows.plugins.adjoint.**TraveltimeInexact**(*syn*, *obs*, *nt*, *dt*)
Much faster (but possibly inaccurate) version of Traveltime function

seisflows.plugins.adjoint.**Velocity**(*syn*, *obs*, *nt*, *dt*)

seisflows.plugins.adjoint.**Waveform**(*syn*, *obs*, *nt*, *dt*)
Waveform difference (Tromp et al 2005, eq 9)

6.2.7 seisflows.plugins.misfit module

seisflows.plugins.misfit.**Acceleration**(*syn*, *obs*, *nt*, *dt*)

seisflows.plugins.misfit.**Amplitude**(*syn*, *obs*, *nt*, *dt*)
Cross correlation amplitude

seisflows.plugins.misfit.**Displacement**(*syn*, *obs*, *nt*, *dt*)

seisflows.plugins.misfit.**Envelope**(*syn*, *obs*, *nt*, *dt*, *eps=0.05*)
Envelope difference (Yuan et al 2015, eq 9)

seisflows.plugins.misfit.**Envelope2**(*syn*, *obs*, *nt*, *dt*, *eps=0.0*)
Envelope amplitude ratio (Yuan et al 2015, eq B-1)

seisflows.plugins.misfit.**Envelope3**(*syn*, *obs*, *nt*, *dt*, *eps=0.0*)
Envelope cross-correlation lag (Yuan et al 2015, eqs B-4)

seisflows.plugins.misfit.**InstantaneousPhase**(*syn*, *obs*, *nt*, *dt*, *eps=0.05*)
Instantaneous phase from Bozdag et al. 2011

seisflows.plugins.misfit.**InstantaneousPhase2**(*syn*, *obs*, *nt*, *dt*, *eps=0.0*)

seisflows.plugins.misfit.**Traveltime**(*syn*, *obs*, *nt*, *dt*)
Compute cross correlation traveltime between two traces suposing that they contain only one arrival

seisflows.plugins.misfit.**TraveltimeInexact**(*syn*, *obs*, *nt*, *dt*)
Much faster (but possibly inaccurate) version of Traveltime function

seisflows.plugins.misfit.**Velocity**(*syn*, *obs*, *nt*, *dt*)

seisflows.plugins.misfit.**Waveform**(*syn*, *obs*, *nt*, *dt*)
Waveform difference

6.2.8 seisflows.plugins.readers module

seisflows.plugins.readers.**ascii**(*path*, *filenames*)
Reads SPECFEM3D-style ascii data

seisflows.plugins.readers.**readBigSuFile**(*nameOfFile*, *nt*, *format='SU'*, *byteorder='<'*)
This function is a hack to read .su file containing too many samples per traces. In the su format only 2 bytes per trace are dedicated to encoding for the number of samples (as signed int, see: http://lists.swapbytes.de/archives/obspy-users/2017-March/002359.html). Even if it's an old format it's still extremely stupid. This prove the lack of vision the designer of this format had at that time. They could have chosen 8 bytes or 16 bytes it was no big deal... They've cost me a day's work. But let us forget about the past. This limits the size of the traces to 32768 samples (NSTEP beween -32768 to 32768). Let us now suppose that we have NSTEP = 80000 samples per trace. We still want to use Obspy. The problem is that the NSTEP written in the .su file does not make any sense anymore and it is read by the obspy.read function! We thus rewrote a quick version of this function replacing the number of point by PAR.NT It is mainly copy-pasted from Obspy source code.

seisflows.plugins.readers.**su**(*path*, *filename*)
Reads Seismic Unix files. Hardwired ''

Function readBigSuFile is a hack to read su file containing too many samples per trace. In the su format only 2 bytes per trace are dedicated to encoding for the number of samples (as signed int, see: http://lists.swapbytes.de/archives/obspy-users/2017-March/002359.html). Even if it's an old format it's still extremely stupid. This proove the lack of vision the designer of this format had at that time. They could have chosen 8 bytes or 16 bytes it was no big deal... They've cost me a day's work. But let us forget about the past. This limits the size of the traces to 32768 samples (NSTEP beween -32768 to 32768). Let us now suppose that we have NSTEP = 40000 samples per trace. We still want to use Obspy. The problem is that the NSTEP written in the .su file does not make any sense anymore and it is read by the obspy.read function! We thus rewrote a quick version of this function from Obspy replacing the number of point by PAR.NT

6.2.9 seisflows.plugins.wavelets module

seisflows.plugins.wavelets.**gabor**(*nt*, *df*, *fp*)

seisflows.plugins.wavelets.**ricker**(*nt*, *dt*, *fp*)

6.2.10 seisflows.plugins.writers module

seisflows.plugins.writers.**ascii**(*stream*, *path*, *filenames*)
Write ascii signal file

seisflows.plugins.writers.**su**(*stream*, *path*, *filename*)
Write Seismic Unix files. Function writeBigSuFile is a hack to write a .su file when the number of samples per trace is two big. In the su format only 2 bytes per trace are dedicated to encoding for the number of samples (as signed int, see: http://lists.swapbytes.de/archives/obspy-users/2017-March/002359.html). Even if it's an old format it's still extremely stupid. This proove the lack of vision the designer of this format had at that time. They could have chosen 8 bytes or 16 bytes it was no big deal... They've cost me a day's work. But let us forget about the past. This limits the size of the traces in the header to maximum 32768. We use Obspy to write the file with dummy values there instead of the real number of sample (that we now anyway : it is PAR.NT). We thus rewrote a quick version of this function from Obspy replacing the number of point by PAR.NT

seisflows.plugins.writers.**writeBigSuFile**(*stream*, *path*, *byteorder='<'*)
This function is a hack to write a .su file when the number of samples per trace is two big. In the su format only 2 bytes per trace are dedicated to encoding for the number of samples (as signed int, see: http://lists.swapbytes.de/archives/obspy-users/2017-March/002359.html). Even if it's an old format it's still extremely stupid. This proove the lack of vision the designer of this format had at that time. They could have chosen 8 bytes or 16 bytes it was no big deal... They've cost me a day's work. But let us forget about the past. This limits the size of the traces in the header to maximum 32768. We use Obspy to write the file with dummy values there instead of the real number of sample (that we now anyway : it is PAR.NT). We thus rewrote a quick version of this function from Obspy replacing the number of point by PAR.NT This is mostly copy-pastes from Obspy source code

6.3 seisflows.postprocess package

6.3.1 seisflows.postprocess.base module

class seisflows.postprocess.base.**base**
Bases: object

Regularization, smoothing, sharpening, masking and related operations on models or gradients

check()
Checks parameters and paths

process_kernels(*path*, *parameters*)
Sums kernels from individual sources, with optional smoothing

Input path directory containing sensitivity kernels

Input parameters list of material parameters e.g. ['vp','vs']

setup()
Placeholder for initialization or setup tasks

write_gradient(*path*)
Combines contributions from individual sources and material parameters to get the gradient, and optionally applies user-supplied scaling

Input path directory from which kernels are read and to which gradient is written

6.3.2 seisflows.postprocess.default module

class seisflows.postprocess.default.**default**
Bases: *seisflows.postprocess.base.base*

Default postprocesing option

Provides default image processing and regularization functions for models or gradients

6.4 seisflows.preprocess package

6.4.1 seisflows.preprocess.base module

class seisflows.preprocess.base.**base**
Bases: object

Data preprocessing class

Provides data processing functions for seismic traces, with options for data misfit, filtering, normalization and muting

apply_csg_mute(*traces*)

apply_filter(*traces*)

apply_filter_backwards(*traces*)

apply_mute(*traces*)

apply_normalize(*traces*)

check()
Checks parameters and paths

check_filter()
Checks filter settings

check_mute()
Checks mute settings

check_normalize()

csg_mute(*seismo_w*, *T*, *T1*)

get_network_size(*traces*)

get_receiver_coords(*traces*)

get_source_coords(*traces*)

get_time_scheme(*traces*)
FIXME: extract time scheme from trace headers rather than parameters file. Note from Alexis Bottero : it is actually better like this in my opinion because this allows for longer traces to be processed. Indeed, in su format only 2 bytes are dedicated to the number of samples which is supposed to be stored as an unsigned int. The maximum NT which can be stored in the header is then 32762 whereas there is no limit in principle.

prepare_eval_grad(*path='.'*)
Prepares solver for gradient evaluation by writing residuals and adjoint traces

Input path directory containing observed and synthetic seismic data

setup()
Sets up data preprocessing machinery

sum_residuals(*files*)
Sums squares of residuals

Input files list of single-column text files containing residuals

Output total_misfit sum of squares of residuals

write_adjoint_traces(*path*, *syn*, *obs*, *channel*)
Writes "adjoint traces" required for gradient computation

Input path location "adjoint traces" will be written

Input syn obspy Stream object containing synthetic data

Input obs obspy Stream object containing observed data

Input channel channel or component code used by writer

write_residuals(*path*, *syn*, *obs*)
Computes residuals

Input path location "adjoint traces" will be written

Input syn obspy Stream object containing synthetic data

Input obs obspy Stream object containing observed data

6.4.2 seisflows.preprocess.default module

class seisflows.preprocess.default.**default**
Bases: *seisflows.preprocess.base.base*

Default preprocesing class

Provides data processing functions for seismic traces, with options for data misfit, filtering, normalization and muting

6.4.3 seisflows.preprocess.double_difference module

class seisflows.preprocess.double_difference.**double_difference**
Bases: *seisflows.preprocess.base.base*

Double-difference data processing class

Adds double-difference data misfit functions to base class

adjoint_dd(*si*, *sj*, *t0*, *nt*, *dt*)
Returns contribution to adjoint source from a single double difference measurement

apply_weights(*traces*)

check()
Checks parameters, paths, and dependencies

distance(*x1*, *y1*, *x2*, *y2*)

load_weights()

shift(*v*, *it*)
Shifts time series a given number of steps

sum_residuals (*files*)
Sums squares of residuals

write_adjoint_traces (*path*, *syn*, *dat*, *channel*)
Computes adjoint traces from observed and synthetic traces

write_residuals (*path*, *syn*, *dat*)
Computes residuals from observations and synthetics

6.5 seisflows.solver package

6.5.1 seisflows.solver.base module

class seisflows.solver.base.**base**
Bases: object

Provides an interface through which solver simulations can be set up and run and a parent class for SPECFEM2D, SPECFEM3D and SPECFEM3D_GLOBE subclasses

This class supports only acoustic and isotropic elastic inversions. For additional options, see github.com/rmodrak/seisflows-multiparameter

eval_func, eval_grad, apply_hess These methods deal with evaluation of the misfit function or its derivatives. Together, they provide the primary interface through which SeisFlows interacts with SPECFEM2D/3D

forward, adjoint These methods allow direct access to low-level SPECFEM2D/3D components, providing an alternative interface through which to interact with the solver

setup, generate_data, generate_model One-time operations performed at beginning of an inversion or migration

initialize_solver_directories, initialize_adjoint_traces

SPECFEM2D/3D requires a particular directory structure in which to run and particular file formats for models, data, and parameter files. These methods help put in place all these prerequisites

load, save For reading and writing SPECFEM2D/3D models and kernels. On the disk, models and kernels are stored as binary files, and in memory, as dictionaries with different keys corresponding to different material parameters

split, merge Within the solver routines, it is natural to store models as dictionaries. Within the optimization routines, it is natural to store models as vectors. Two methods, 'split' and 'merge', are used to convert back and forth between these two representations

combine, smooth Utilities for combining and smoothing kernels

adjoint()
Calls adjoint solver

apply_hess (*path=''*)
Computes action of Hessian on a given model vector. (A gradient evaluation must have already been carried out.)

Input path : directory to which output files are exported

check()
Checks parameters and paths

check_mesh_properties(*path=None*)
path contains binary files such as: proc000000_z.bin, proc000000_x.bin, proc000000_vs.bin proc000000_vp.bin, proc000000_rho.bin, proc000001_z.bin, proc000001_x.bin, proc000001_vs.bin ... These will be read to get the number of processors used, the number of gll points and the coordinates of those points

check_solver_parameter_files()

check_source_names()
Determines names of sources by applying wildcard rule to user- supplied input files If source_prefix is 'SOURCE' and that in specfem DATA folder are the files SOURCE_00001, SOURCE_00002, SOURCE_00003, ... Then this will build the list names = ['00001','00002','00003',...] If, for ex, taskid is 1 the function returns ['00001', '00002']

clean()

combine(*input_path=''*, *output_path=''*, *parameters=[]*)
Sums individual source contributions. Wrapper over xcombine_sem utility.

cwd

data_filenames

eval_func(*path=''*, *export_traces=False*, *write_residuals=True*)
Performs forward simulations needed for misfit function evaluation

Input path : directory from which model is imported

Input export_traces : save or discard traces?

eval_grad(*path=''*, *export_traces=False*)
Evaluates gradient by carrying out adjoint simulations. (A function evaluation must already have been carried out.)

Input path : directory from which model is imported

Input export_traces : save or discard traces?

export_kernels(*path*)

export_model(*path, parameters=['rho', 'vp', 'vs']*)

export_residuals(*path*)

export_traces(*path*, *prefix='traces/obs'*)

forward()
Calls forward solver

generate_data(**args*, ***kwargs*)
Generates data

generate_mesh(**args*, ***kwargs*)
Performs meshing and database generation

import_model(*path*)

import_traces(*path*)

initialize_adjoint_traces()
Puts in place "adjoint traces" expected by SPECFEM Adjoint traces are initialized by writing zeros for all channels. (even the ones that are not actually in use, as required by specfem) Ex: Ux, Uy, Uz even if only Uy is used Channels actually in use during an inversion or migration will be overwritten with nonzero values later on.

initialize_solver_directories()
Creates directory structure expected by SPECFEM3D, copies executables, and prepares input files. Executables must be supplied by user as there is currently no mechanism for automatically compiling from source.

io
Solver IO module

kernel_databases

load(*path*, *parameters=[]*, *prefix=''*, *suffix=''*)
Loads SPECFEM2D/3D models or kernels

Input path : directory from which model is read

Input parameters : list of material parameters to be read (if empty, defaults to self.parameters)

Input prefix : optional filename prefix

Input suffix : optional filename suffix, eg '_kernel'

Output dict : model or kernels indexed by material parameter and processor rank, ie dict[parameter][iproc]

merge(*model*, *parameters=[]*)
Converts model from dictionary to vector representation

mesh_properties

model_databases

rename_data(*path*)
Works around conflicting data filename conventions

rename_kernels()
Works around conflicting kernel filename conventions

save(*dict, path, parameters=['vp', 'vs', 'rho'], prefix='', suffix=''*)
Saves SPECFEM2D/3D models or kernels

Input dict : model stored as a dictionary or Container

Input path : directory to which model is written

Input parameters : list of material parameters to be written

Input prefix : optional filename prefix

Input suffix : optional filename suffix, eg '_kernel'

setup()
Prepares solver for inversion or migration Sets up directory structure expected by SPECFEM and copies or generates seismic data to be inverted or migrated

smooth(*input_path=''*, *output_path=''*, *parameters=[]*, *span=0.0*)
Smooths kernels by convolving them with a Gaussian. Wrapper over xsmooth_sem utility.

source_name

source_names

source_prefix

split(*m*, *parameters=[]*)
Converts model from vector to dictionary representation

taskid

6.5.2 seisflows.solver.specfem2d module

class seisflows.solver.specfem2d.**specfem2d**
Bases: *seisflows.solver.base.base*

Python interface for SPECFEM2D

See base class for method descriptions

adjoint()
Calls SPECFEM2D adjoint solver

check()
Checks parameters and paths

check_solver_parameter_files()
Checks solver parameters

data_filenames

export_model(*path*)

forward(*path='traces/syn'*)
Calls SPECFEM2D forward solver

generate_data(***model_kwargs*)
Generates data (perform meshing and database generation first)

generate_mesh(*model_path=None*, *model_name=None*, *model_type='gll'*)
Performs meshing and database generation

import_model(*path*)

initialize_adjoint_traces()
Puts in place "adjoint traces" expected by SPECFEM Adjoint traces are initialized by writing zeros for all channels. (even the ones that are not actually in use, as required by specfem) Ex: Ux, Uy, Uz even if only Uy is used Channels actually in use during an inversion or migration will be overwritten with nonzero values later on.

kernel_databases

model_databases

source_prefix

6.5.3 seisflows.solver.specfem3d module

class seisflows.solver.specfem3d.**specfem3d**
Bases: *seisflows.solver.base.base*

Python interface for SPECFEM3D

See base class for method descriptions

adjoint()
Calls SPECFEM3D adjoint solver

check()
Checks parameters and paths

check_solver_parameter_files()
Checks solver parameters

data_filenames

data_wildcard

eval_func(**args*, ***kwargs*)
Performs forward simulations needed for misfit function evaluation

Input path : directory from which model is imported

Input export_traces : save or discard traces?

forward(*path='traces/syn'*)
Calls SPECFEM3D forward solver

generate_data(***model_kwargs*)
Generates data

generate_mesh(*model_path=None*, *model_name=None*, *model_type='gll'*)
Performs meshing and database generation

initialize_adjoint_traces()
Puts in place "adjoint traces" expected by SPECFEM Adjoint traces are initialized by writing zeros for all channels. (even the ones that are not actually in use, as required by specfem) Ex: Ux, Uy, Uz even if only Uy is used Channels actually in use during an inversion or migration will be overwritten with nonzero values later on.

kernel_databases

model_databases

rename_data()
Works around conflicting data filename conventions

source_prefix

write_parameters()

write_receivers()

write_sources()

6.5.4 seisflows.solver.specfem3d_globe module

6.6 seisflows.system package

6.6.1 seisflows.system.base module

class seisflows.system.base.**base**
Bases: object

Abstract base class

check()
Checks parameters and paths

checkpoint(*path*, *classname*, *method*, *args*, *kwargs*)
Writes information to disk so tasks can be executed remotely

run(*classname*, *method*, **args*, ***kwargs*)
Runs task multiple times

run_single(*classname*, *method*, **args*, ***kwargs*)
Runs task a single time

submit()
Submits workflow

taskid()
Provides a unique identifier for each running task

6.6.2 seisflows.system.lsf_lg module

class seisflows.system.lsf_lg.**lsf_lg**
Bases: *seisflows.system.base.base*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

Intermediate files are written to a global scratch path PATH.SCRATCH, which must be accessible to all compute nodes.

Optionally, users can provide a local scratch path PATH.LOCAL if each compute node has its own local filesystem.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

job_id_list(*stdout*)

job_status(*classname*, *method*, *jobs*)

mpiexec()
Specifies MPI executable used to invoke solver

run(*classname*, *method*, *hosts='all'*, ***kwargs*)
Runs task multiple times in embarrassingly parallel fasion

Executes classname.method(*args, **kwargs) NTASK times, each time on NPROC cpu cores

run_single(*classname*, *method*, *hosts='all'*, ***kwargs*)
Runs task multiple times in embarrassingly parallel fasion

Executes classname.method(*args, **kwargs) NTASK times, each time on NPROC cpu cores

save_kwargs(*classname*, *method*, *kwargs*)

submit(*workflow*)
Submits workflow

taskid()
Provides a unique identifier for each running task

timestamp()

6.6.3 seisflows.system.lsf_sm module

6.6.4 seisflows.system.multicore module

class seisflows.system.multicore.**multicore**
Bases: *seisflows.system.serial.serial*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

run(*classname*, *method*, **args*, ***kwargs*)
Runs task multiple times in embarrassingly parallel fasion

Executes classname.method(*args, **kwargs) NTASK times, each time on NPROC cpu cores

run_single(*classname*, *method*, **args*, ***kwargs*)
Runs task a single time

save_kwargs(*classname*, *method*, *kwargs*)

submit(*workflow*)
Submits job

6.6.5 seisflows.system.multithreaded module

class seisflows.system.multithreaded.**multithreaded**
Bases: *seisflows.system.multicore.multicore*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

6.6.6 seisflows.system.pbs_lg module

class seisflows.system.pbs_lg.**pbs_lg**
Bases: *seisflows.system.base.base*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

Intermediate files are written to a global scratch path PATH.SCRATCH, which must be accessible to all compute nodes.

Optionally, users can provide a local scratch path PATH.LOCAL if each compute node has its own local filesystem.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

job_array_args(*hosts*)

job_array_cmd(*classname*, *method*, *hosts*)

job_array_status(*classname*, *method*, *jobs*)
Determines completion status of one or more jobs

mpiexec()
Specifies MPI executable used to invoke solver

run(*classname*, *method*, *hosts='all'*, ***kwargs*)
Runs task multiple times in embarrassingly parallel fasion

Executes classname.method(*args, **kwargs) NTASK times, each time on NPROC cpu cores

save_kwargs(*classname*, *method*, *kwargs*)

submit(*workflow*)
Submits workflow

submit_job_array(*classname*, *method*, *hosts='all'*)

taskid()
Provides a unique identifier for each running task

6.6.7 seisflows.system.pbs_sm module

class seisflows.system.pbs_sm.**pbs_lg**
Bases: *seisflows.system.base.base*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

Intermediate files are written to a global scratch path PATH.SCRATCH, which must be accessible to all compute nodes.

Optionally, users can provide a local scratch path PATH.LOCAL if each compute node has its own local filesystem.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

hostlist()
Generates list of allocated cores

mpiexec()
Specifies MPI executable used to invoke solver

run(*classname*, *method*, *hosts='all'*, ***kwargs*)
Runs embarrassingly parallel tasks

Executes the following multiple times: classname.method(*args, **kwargs)

system.taskid serves to provide each running task a unique identifier

save_kwargs(*classname*, *method*, *kwargs*)

submit(*workflow*)
Submits workflow

taskid()
Provides a unique identifier for each running task

6.6.8 seisflows.system.serial module

class seisflows.system.serial.**serial**
Bases: *seisflows.system.base.base*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

mpiexec()
Specifies MPI executable used to invoke solver

progress(*taskid*)
Provides status update

run(*classname*, *method*, *hosts='all'*, ***kwargs*)
Executes method from classname multiple times in serial taskid is used to identified a given task (one source)

run_single(*classname*, *method*, **args*, ***kwargs*)
Runs task a single time

submit(*workflow*)
Submits job

taskid()
Return the value of the environment variable SEISFLOWS_TASKID which provides a unique identifier for each running task

6.6.9 seisflows.system.slurm_lg module

class seisflows.system.slurm_lg.**slurm_lg**
Bases: *seisflows.system.base.base*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

Intermediate files are written to a global scratch path PATH.SCRATCH, which must be accessible to all compute nodes.

Optionally, users can provide a local scratch path PATH.LOCAL if each compute node has its own local filesystem.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

job_array_status(*classname*, *method*, *jobs*)
Determines completion status of job or job array

job_id_list(*stdout*, *ntask*)
Parses job id list from sbatch standard output

job_status(*job*)
Queries completion status of a single job

mpiexec()
Specifies MPI executable used to invoke solver

run(*classname*, *method*, **args*, ***kwargs*)
Runs task multiple times in embarrassingly parallel fasion

Executes classname.method(*args, **kwargs) NTASK times, each time on NPROC cpu cores

run_single(*classname*, *method*, **args*, ***kwargs*)
Runs task a single time

Executes classname.method(*args, **kwargs) a single time on NPROC cpu cores

submit(*workflow*)
Submits workflow

taskid()
Provides a unique identifier for each running task

6.6.10 seisflows.system.slurm_sm module

class seisflows.system.slurm_sm.**slurm_sm**
Bases: *seisflows.system.base.base*

An interface through which to submit workflows, run tasks in serial or parallel, and perform other system functions.

By hiding environment details behind a python interface layer, these classes provide a consistent command set across different computing environments.

Intermediate files are written to a global scratch path PATH.SCRATCH, which must be accessible to all compute nodes.

Optionally, users can provide a local scratch path PATH.LOCAL if each compute node has its own local filesystem.

For important additional information, please see http://seisflows.readthedocs.org/en/latest/manual/manual.html#system-configuration

check()
Checks parameters and paths

hostlist()
Generates list of allocated cores

mpiexec()
Specifies MPI executable used to invoke solver

run(*classname*, *method*, **args*, ***kwargs*)
Runs task multiple times in embarrassingly parallel fasion

run_single(*classname*, *method*, **args*, ***kwargs*)
Runs task a single time

save_kwargs(*classname*, *method*, *kwargs*)

submit(*workflow*)
Submits workflow

taskid()
Provides a unique identifier for each running task

6.6.11 seisflows.system.tiger_lg module

class seisflows.system.tiger_lg.**tiger_lg**
Bases: *seisflows.system.slurm_lg.slurm_lg*

Specially designed system interface for tiger.princeton.edu

See parent class for more information.

check()
Checks parameters and paths

submit(**args*, ***kwargs*)
Submits job

6.6.12 seisflows.system.tiger_sm module

class seisflows.system.tiger_sm.**tiger_sm**
Bases: *seisflows.system.slurm_sm.slurm_sm*

Specially designed system interface for tiger.princeton.edu

See parent class for more information.

check()
Checks parameters and paths

submit(**args*, ***kwargs*)
Submits job

6.7 seisflows.tools package

6.7.1 seisflows.tools.array module

seisflows.tools.array.**count_zeros**(*a*)
Counts number of zeros in a list or array

seisflows.tools.array.**grid2mesh**(*V*, *grid*, *mesh*)
Interpolates from structured coordinates (grid) to unstructured coordinates (mesh)

seisflows.tools.array.**gridsmooth**(*Z*, *span*)
Smooths values on 2D rectangular grid

seisflows.tools.array.**loadnpy**(*filename*)
Loads numpy binary file.

seisflows.tools.array.**mesh2grid**(*v*, *mesh*)
Interpolates from an unstructured coordinates (mesh) to a structured coordinates (grid)

seisflows.tools.array.**meshsmooth**(*v*, *mesh*, *span*)
Smooths values on 2D unstructured mesh

seisflows.tools.array.**savenpy**(*filename*, *v*)
Saves numpy binary file.

seisflows.tools.array.**sortrows**(*a*, *return_index=False*, *return_inverse=False*)
Sorts rows of numpy array

seisflows.tools.array.**stack**(**args*)

seisflows.tools.array.**uniquerows**(*a*, *sort_array=False*, *return_index=False*)
Finds unique rows of numpy array

6.7.2 seisflows.tools.err module

exception seisflows.tools.err.**ParameterError**(**args*)
Bases: exceptions.ValueError

6.7.3 seisflows.tools.graphics module

seisflows.tools.graphics.**get_regular_ticks**(*v*, *interval*)
Returns regular tick intervals.

seisflows.tools.graphics.**plot_gll**(*x*, *y*, *z*, *vmin=None*, *vmax=None*)
Plots values on 2D unstructured GLL mesh

seisflows.tools.graphics.**plot_many_gll**(*x*, *y*, *z*, *vmin=None*, *vmax=None*)
Plots values on big 2D unstructured GLL mesh (in that case tricontourf does not work)

seisflows.tools.graphics.**plot_section**(*stream*, *ax=None*, *cmap='seismic'*, *clip=100*, *title=''*, *x_interval=1.0*, *y_interval=1.0*)
Plots a seismic section from an obspy stream.

Parameters

- **stream** (*Obspy stream object*) – Obspy stream object created from a SU data file
- **ax** (*Matplotlib Axes object*) – Optional axis object

- **cmap** (*str*) – Matplotlib colormap option.
- **clip** (*float*) – Percentage value (0-100) for amplitude clipping
- **title** (*str*) – plot title
- **x_interval** (*float*) – Offset axis tick interval in km
- **y_interval** (*float*) – Time axis tick interval in km

Raises NotImplementedError – If stream object does not have SU format

seisflows.tools.graphics.**plot_vector**(*t*, *v*, *xlabel=''*, *ylabel=''*, *title=''*)
Plots a vector or time series.

Parameters

- **v** (*ndarray, ndims = 1/2*) – Vector or time series to plot
- **xlabel** (*str*) – x axis label
- **ylabel** (*str*) – y axis label
- **title** (*str*) – plot title

Raises ValueError – If dimensions of v are greater than 2

6.7.4 seisflows.tools.math module

seisflows.tools.math.**angle**(*x*, *y*)

seisflows.tools.math.**backtrack2**(*f0*, *g0*, *x1*, *f1*, *b1=0.1*, *b2=0.5*)
Safeguarded parabolic backtrack

seisflows.tools.math.**backtrack3**(*f0*, *g0*, *x1*, *f1*, *x2*, *f2*)
Safeguarded cubic backtrack

seisflows.tools.math.**dot**(*x*, *y*)

seisflows.tools.math.**gauss2**(*X*, *Y*, *mu*, *sigma*, *normalize=True*)
Evaluates Gaussian over points of X,Y

seisflows.tools.math.**grad**(*V*, *h=[]*)
Evaluates derivatives on a 2D rectangular grid

seisflows.tools.math.**hilbert**(*w*)

seisflows.tools.math.**lsq2**(*x*, *f*)
Parabolic least squares fit

seisflows.tools.math.**nabla**(*V*, *h=[]*)
Returns sum of first-order spatial derivatives of a function defined on a 2D rectangular grid; generalizes Laplacian

seisflows.tools.math.**nabla2**(*V*, *h=[]*)
Returns sum of second-order spatial derivatives of a function defined on a 2D rectangular grid; generalizes Laplacian

seisflows.tools.math.**polyfit2**(*x*, *f*)
Parabolic fit

seisflows.tools.math.**tv**(*Z*, *h=[]*, *epsilon=1e-06*)

6.7.5 seisflows.tools.msg module

6.7.6 seisflows.tools.seismic module

class seisflows.tools.seismic.**Container**
Bases: collections.defaultdict

Dictionary-like object for holding models or kernels

class seisflows.tools.seismic.**Minmax**
Bases: collections.defaultdict

Keeps track of min,max values of model or kernel

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class seisflows.tools.seismic.**Writer**(*path='./output.stat'*)
Bases: object

Utility for appending values to text files

seisflows.tools.seismic.**call_solver**(*mpiexec*, *executable*, *output='solver.log'*)
Calls MPI solver executable

A less complicated version, without error catching, would be subprocess.call(mpiexec +' '+ executable, shell=True)

seisflows.tools.seismic.**getpar**(*key*, *file='DATA/Par_file'*, *sep='='*, *cast=<type 'str'>*, *noOutput=False*)
Reads parameter from text file

seisflows.tools.seismic.**setpar**(*key*, *val*, *filename='DATA/Par_file'*, *path='.'*, *sep='='*)
Writes parameter to text file

6.7.7 seisflows.tools.signal module

seisflows.tools.signal.**correlate**(*u*, *v*)

seisflows.tools.signal.**mask**(*slope*, *const*, *offset*, *time_scheme*, *length=400*)
Constructs tapered mask that can be applied to trace to mute early or late arrivals.

seisflows.tools.signal.**mute_early_arrivals**(*traces*, *slope*, *const*, *time_scheme*, *s_coords*, *r_coords*)
Applies tapered mask to record section, muting early arrivals

Signals arriving before

SLOPE * || s - r || + CONST

are muted, where slope is has units of velocity**-1, CONST has units of time, and || s - r || is distance between source and receiver.

seisflows.tools.signal.**mute_late_arrivals**(*traces*, *slope*, *const*, *time_scheme*, *s_coords*, *r_coords*)
Applies tapered mask to record section, muting late arrivals

Signals arriving after

SLOPE * || s - r || + CONST

are muted, where SLOPE is has units of velocity**-1, CONST has units of time, and || s - r || is distance between source and receiver.

seisflows.tools.signal.**mute_long_offsets**(*traces*, *dist*, *s_coords*, *r_coords*)
Mutes traces having

|| s - r || > DIST

where || s - r || is the offset between source and receiver and DIST is a user-supplied cutoff

seisflows.tools.signal.**mute_short_offsets**(*traces*, *dist*, *s_coords*, *r_coords*)
Mutes traces having

|| s - r || < DIST

where || s - r || is the offset between source and receiver and DIST is a user-supplied cutoff

seisflows.tools.signal.**sconvolve**(*s*, *h*, *w*, *inplace=True*)

seisflows.tools.signal.**tukeywin**(*nt*, *imin*, *imax*, *alpha=0.05*)

6.7.8 seisflows.tools.tools module

class seisflows.tools.tools.**Struct**(**args*, ***kwargs*)
Bases: dict

seisflows.tools.tools.**call**(**args*, ***kwargs*)

seisflows.tools.tools.**diff**(*list1*, *list2*)
Difference between unique elements of lists

seisflows.tools.tools.**divides**(*i*, *j*)
True if j divides i

seisflows.tools.tools.**exists**(*names*)
Wrapper for os.path.exists

seisflows.tools.tools.**findpath**(*name*)
Resolves absolute path of module

seisflows.tools.tools.**getset**(*arg*)

seisflows.tools.tools.**iterable**(*arg*)

seisflows.tools.tools.**loadjson**(*filename*)
Load object using json

seisflows.tools.tools.**loadnpy**(*filename*)
Loads numpy binary file.

seisflows.tools.tools.**loadobj**(*filename*)
Load object using pickle

seisflows.tools.tools.**loadpy**(*filename*)

seisflows.tools.tools.**loadtxt**(*filename*)
Load scalar from text file

seisflows.tools.tools.**loadyaml**(*filename*)

seisflows.tools.tools.**module_exists**(*name*)

seisflows.tools.tools.**nproc**()

seisflows.tools.tools.**package_exists**(*name*)

seisflows.tools.tools.**pkgpath**(*name*)

seisflows.tools.tools.**savejson**(*filename*, *obj*)
Save object using json

seisflows.tools.tools.**savenpy**(*filename*, *v*)
Saves numpy binary file.

seisflows.tools.tools.**saveobj**(*filename*, *obj*)
Save object using pickle

seisflows.tools.tools.**savetxt**(*filename*, *v*)
Save scalar to text file

seisflows.tools.tools.**timestamp**()

6.7.9 seisflows.tools.unix module

seisflows.tools.unix.**cat**(*src*, **dst*)
Open a file and print it. If file dst is given the file is printed into dst

seisflows.tools.unix.**cd**(*path*)
Change current directory to the one given

seisflows.tools.unix.**cp**(*src=''*, *dst=''*)
Copy all (files or directories, as list or tupples) given in src to directory (or file) dst

seisflows.tools.unix.**hostname**()

seisflows.tools.unix.**ln**(*src*, *dst*)

seisflows.tools.unix.**ls**(*path*)

seisflows.tools.unix.**mkdir**(*dirs*)

seisflows.tools.unix.**mv**(*src=''*, *dst=''*)

seisflows.tools.unix.**rename**(*old*, *new*, *names*)

seisflows.tools.unix.**rm**(*path=''*)

seisflows.tools.unix.**select**(*items*, *prompt=''*)

seisflows.tools.unix.**touch**(*filename*, *times=None*)

seisflows.tools.unix.**which**(*name*)

6.8 seisflows.workflow package

6.8.1 seisflows.workflow.base module

class seisflows.workflow.base.**base**
Bases: object

Workflow abstract base class

check()
Checks parameters and paths

checkpoint()
Writes information to disk so workflow can be resumed following a break

main()
Main routine

Execution of a workflow is equivalent to stepping through workflow.main

6.8.2 seisflows.workflow.inversion module

class seisflows.workflow.inversion.**inversion**
Bases: *seisflows.workflow.base.base*

Waveform inversion base class

Peforms iterative nonlinear inversion and provides a base class on top of which specialized strategies can be implemented.

To allow customization, the inversion workflow is divided into generic methods such as 'initialize', 'finalize', 'evaluate_function', 'evaluate_gradient', which can be easily overloaded.

Calls to forward and adjoint solvers are abstracted through the 'solver' interface so that various forward modeling packages can be used interchangeably.

Commands for running in serial or parallel on a workstation or cluster are abstracted through the 'system' interface.

check()
Checks parameters and paths

checkpoint()
Writes information to disk so workflow can be resumed following a break

clean()
Cleans directories in which function and gradient evaluations were carried out

compute_direction()
Computes search direction

evaluate_function()
Performs forward simulation to evaluate objective function

evaluate_gradient()
Performs adjoint simulation to evaluate gradient

finalize()
Saves results from current model update iteration

initialize()
Prepares for next model update iteration

line_search()
Conducts line search in given search direction

Status codes status > 0 : finished status == 0 : not finished status < 0 : failed

main()
Carries out seismic inversion

save_gradient()

save_kernels()

save_model()

save_residuals()

save_traces()

setup()
Lays groundwork for inversion. Runs setup method for preprocess, postprocess, optimize and also for the solver through the system: system.run('solver','setup') This will copy user supplied data or generate it from a 'real' model supplied This will setup the mesh for the initial model supplied If path LOCAL is supplied this is done even at iteration > 1

write_gradient(*path=''*, *suffix=''*)
Writes gradient in format expected by nonlinear optimization library

write_misfit(*path=''*, *suffix=''*)
Writes misfit in format expected by nonlinear optimization library

write_model(*path=''*, *suffix=''*)
Writes model in format expected by solver

6.8.3 seisflows.workflow.migration module

class seisflows.workflow.migration.**migration**
Bases: *seisflows.workflow.base.base*

Migration base class.

In the terminology of seismic exploration, implements a 'reverse time migration'.

check()
Checks parameters and paths

main()
Migrates seismic data

prepare_model()

save_kernels()

save_kernels_sum()

save_traces()

6.8.4 seisflows.workflow.test_adjoint module

seisflows.workflow.test_adjoint.**DotProductLHS**(*keys*, *x*, *y*)

seisflows.workflow.test_adjoint.**DotProductRHS**(*keys*, *x*, *y*)

class seisflows.workflow.test_adjoint.**test_adjoint**
Bases: *seisflows.workflow.base.base*

check()
Checks parameters and paths

event

main()
Main routine

Execution of a workflow is equivalent to stepping through workflow.main

prepare_model()

6.8.5 seisflows.workflow.test_forward module

class seisflows.workflow.test_forward.**test_forward**
Bases: *seisflows.workflow.base.base*

Tests solver by running forward simulation

check()
Checks parameters and paths

main()
Generates seismic data

6.8.6 seisflows.workflow.test_optimize module

6.8.7 seisflows.workflow.test_postprocess module

class seisflows.workflow.test_postprocess.**test_postprocess**
Bases: *seisflows.workflow.base.base*

Postprocessing class

check()
Checks parameters and paths

main()
Writes gradient of objective function

6.8.8 seisflows.workflow.test_preprocess module

class seisflows.workflow.test_preprocess.**test_preprocess**
Bases: *seisflows.workflow.base.base*

Signal processing integration test

check()
Checks parameters and paths

main()
Tests data processing methods

save(*data*, *filename*)

test_adjoint(*dat*, *syn*)

test_filter(*dat*)

test_misfit(*dat*, *syn*)

test_mute(*dat*)

test_normalize(*dat*)

test_reader()

test_writer(*data*)

6.8.9 seisflows.workflow.test_system module

class seisflows.workflow.test_system.**test_system**
Bases: *seisflows.workflow.base.base*

Tests system interface

check()
Checks parameters and paths

hello(*msg='Hello from %d'*)
Prints hello message

main()
Main routine

Execution of a workflow is equivalent to stepping through workflow.main

6.8.10 seisflows.workflow.thrifty_inversion module

class seisflows.workflow.thrifty_inversion.**thrifty_inversion**
Bases: *seisflows.workflow.inversion.inversion*

Thrifty inversion subclass

Provides savings over conventional inversion by carrying over forward simulations from line search

The results of 'inversion' and 'thrifty_inversion' should be exactly the same

clean()
Cleans directories in which function and gradient evaluations were carried out

initialize()
Prepares for next model update iteration

status = 0

update_status()

6.9 seisflows.config module

class seisflows.config.**Dict**(*newdict*)
Bases: object

Dictionary-like object for holding parameters or paths

update(*newdict*)

class seisflows.config.**Null**(**args*, ***kwargs*)
Bases: object

Always and reliably does nothing

seisflows.config.**config**()
Instantiates SeisFlows objects and makes them globally accessible by registering them in sys.modules

seisflows.config.**custom_import**(**args*)
Imports SeisFlows module and extracts class of same name. For example,

custom_import('workflow', 'inversion')

imports 'seisflows.workflow.inversion' and, from this module, extracts class 'inversion'.

seisflows.config.**load**(*path*)
Imports session from disk

seisflows.config.**save**()
Exports session to disk Write files: seisflows_parameters.json, seisflows_paths.json, seisflows_system.p, seisflows_preprocess.p, seisflows_solver.p, seisflows_postprocess.p, seisflows_optimize.p, seisflows_workflow.p

seisflows.config.**tilde_expand**(*mydict*)
Expands tilde character in path strings

PYTHON MODULE INDEX

S

INDEX

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W