AITar

Release 2.0

AITar Development Team

Mar 26, 2024

CONTENTS

1 AlTar 2.0 Release **1**
- 1.1 About AlTar . . . 1
- 1.2 Guides . . . 1
- 1.3 Tutorials . . . 1
- 1.4 Support . . . 2
- 1.5 Copyright . . . 2

2 Bayesian Inference for Inverse Problems **3**
- 2.1 Inverse problem . . . 3
- 2.2 Bayesian approach . . . 3
- 2.3 CATPMIP algorithm . . . 4
- 2.4 References . . . 4

3 Installation Guide **5**
- 3.1 Overview . . . 5
- 3.2 Supported Platforms . . . 5
  - 3.2.1 Hardware . . . 5
  - 3.2.2 Operation systems . . . 6
  - 3.2.3 Prerequisites . . . 6
- 3.3 Downloads . . . 6
- 3.4 Install with CMake . . . 7
  - 3.4.1 General steps . . . 7
  - 3.4.2 CMake Options . . . 9
    - 3.4.2.1 Installation path . . . 9
    - 3.4.2.2 Enable/disable CUDA . . . 9
    - 3.4.2.3 Target GPU architecture(s) . . . 9
    - 3.4.2.4 C++ Compiler . . . 10
    - 3.4.2.5 CUDA Compiler . . . 10
    - 3.4.2.6 BLAS Library . . . 10
    - 3.4.2.7 Library search path . . . 11
    - 3.4.2.8 Build type . . . 11
    - 3.4.2.9 Show compiling details . . . 11
    - 3.4.2.10 More options . . . 11
- 3.5 Conda method (Linux/MacOSX) . . . 11
  - 3.5.1 Install Anaconda/Miniconda . . . 11
  - 3.5.2 Install Conda packages . . . 12
  - 3.5.3 C++ Compiler . . . 12
  - 3.5.4 CUDA compiler (nvcc) . . . 13
  - 3.5.5 Download pyre and AlTar . . . 13
  - 3.5.6 Install pyre . . . 13

3.5.7 Install AlTar . . . 14
3.5.8 MPI setup . . . 14
3.6 Linux Systems . . . 15
3.6.1 Ubuntu 18.04/20.04 . . . 15
3.6.1.1 Install prerequisites . . . 15
3.6.1.2 Install pyre/AlTar . . . 15
3.6.2 RHEL/CentOS 7 . . . 16
3.6.2.1 Install prerequisites . . . 16
3.6.2.2 Install pyre/AlTar . . . 16
3.6.3 Linux with software modules . . . 16
3.7 Docker container . . . 17
3.8 Install with the mm build tool . . . 18
3.8.1 Download `mm` build tool . . . 18
3.8.2 Prepare a `config.mm` file . . . 18
3.8.3 Install pyre . . . 19
3.8.4 Install AlTar . . . 20
3.9 Tests and Examples . . . 21

4 User Guide **23**
4.1 Overview . . . 23
4.2 QuickStart . . . 24
4.2.1 Prepare the configuration file . . . 24
4.2.2 Prepare input files . . . 25
4.2.3 Run an AlTar application . . . 26
4.2.4 Collect and analyze results . . . 27
4.3 Pyre Basics . . . 28
4.3.1 Protocols and Components . . . 28
4.3.2 Pyre Config Format (`.pfg`) . . . 30
4.4 AlTar Framework . . . 31
4.4.1 Application . . . 31
4.4.2 Controller/Annealer . . . 32
4.4.2.1 Worker/AnnealingMethod . . . 33
4.4.2.2 Sampler . . . 33
4.4.2.3 Scheduler . . . 37
4.4.2.4 Archiver (Output) . . . 38
4.4.3 Job . . . 39
4.4.3.1 Configurable Attributes . . . 39
4.4.3.2 Simulation Size . . . 40
4.4.3.3 Single Thread Configuration . . . 41
4.4.3.4 Multiple Threads on One Computer . . . 41
4.4.3.5 Multiple Threads Across Several Computers . . . 42
4.4.3.6 GPU Configurations . . . 43
4.4.4 Model . . . 44
4.4.5 (Prior) Distributions . . . 44
4.4.5.1 Uniform . . . 45
4.4.5.2 Gaussian . . . 45
4.4.5.3 Truncated Gaussian . . . 46
4.4.5.4 Preset . . . 46
4.4.6 Other Distributions . . . 46
4.5 Models . . . 46
4.5.1 Static Slip Inversion . . . 46
4.5.1.1 Static Source model . . . 46
4.5.1.2 Input . . . 47
4.5.1.3 Configurations . . . 48

4.5.1.4 Output . . . 52
4.5.1.5 Moment Distribution . . . 52
4.5.1.6 Forward Model Application . . . 53
4.5.1.7 Utilities . . . 55
4.5.2 Static Slip Inversion with Cp: Epistemic Uncertainties . . . 56
4.5.3 Kinematic Slip Inversion . . . 57
4.5.3.1 Kinematic Source Model . . . 57
4.5.3.2 Joint Kinematic-Static Inversion . . . 58
4.5.3.3 Configurations (Kinematic Model only) . . . 59
4.5.3.4 Configurations (Joint inversion) . . . 62
4.5.3.5 Examples . . . 63
4.5.3.6 Forward Model Application (new version) . . . 64
4.5.3.7 Forward Model Application (old version) . . . 65

5 Programming Guide **67**
5.1 Introduction . . . 67
5.2 Code Organization . . . 67
5.3 Bayesian Model . . . 69
5.4 Model with the BayesianL2 template . . . 71
5.4.1 Parametersets(psets) . . . 71
5.4.2 Data observations(dataobs) with L2-norm . . . 72
5.4.3 Forward modeling . . . 73
5.5 C/C++/CUDA extension modules . . . 74
5.6 CUDA Models . . . 75
5.7 Data Types and Structures . . . 75
5.7.1 Configurable properties . . . 75
5.7.2 Matrix/Vector (GSL) . . . 76
5.7.2.1 Convert altar array to gsl_vector . . . 76
5.7.2.2 Basic matrix/vector operations . . . 76
5.7.2.3 Interfacing as numpy arrays . . . 76

6 Common Issues **77**
6.1 Installation Issues . . . 77
6.1.1 Cannot find `gmake` . . . 77
6.1.2 Cannot find `cublas_v2.h` . . . 77
6.2 Run-time Issues . . . 77
6.2.1 Locales . . . 77
6.2.2 Base case name . . . 78
6.2.3 Configuration Parser Error . . . 78
6.2.4 MPI launcher error . . . 78
6.2.5 Intel MKL Library . . . 78

7 API Reference **79**
7.1 `altar` . . . 79
7.1.1 Subpackages . . . 79
7.1.1.1 `altar.actions` . . . 79
7.1.1.2 `altar.bayesian` . . . 81
7.1.1.3 `altar.cuda` . . . 98
7.1.1.4 `altar.data` . . . 123
7.1.1.5 `altar.distributions` . . . 126
7.1.1.6 `altar.ext` . . . 130
7.1.1.7 `altar.models` . . . 130
7.1.1.8 `altar.norms` . . . 183
7.1.1.9 `altar.shells` . . . 184

7.1.1.10 altar.simulations 190
7.1.2 Submodules 198
7.1.2.1 altar.meta 198
7.1.3 Package Contents 199
7.1.3.1 Functions 199

8 Indices and tables **201**

Python Module Index **203**

Index **205**

CHAPTER

ONE

ALTAR 2.0 RELEASE

1.1 About AlTar

AlTar(Al-Tar) is a software package to solve inverse problems with Bayesian inference. It is named after the Spanish-French physicist Albert Tarantola, for his pioneering work on the Bayesian perspective of inverse problems.

AlTar uses primarily the Cascading Adaptive Tempered Metropolis In Parallel (CATMIP) algorithm, which is designed to efficiently simulate problems with high-dimensional spaces on parallel computers, include Graphics Processing Units(GPUs).

1.2 Guides

(work in progress, currently for AlTar 2.0 CUDA version only)

Readthedocs (html | pdf | epub):

- Installation Guide
- User Guide
- Programming Guide
- API Reference

Source on github.

1.3 Tutorials

Tutorials presented with jupyter notebooks:

- An introduction to AlTar/pyre applications: HelloApp
- An overview of the AlTar framework
- Static slip inversion: a toy model with epistemic uncertainties
  - Step 1: prepare the input files
  - Step 2: run AlTar2

1.4 Support

- Issues @ github
- Common Issues
- Slack Discussion Group (currently only for developers)

1.5 Copyright

CHAPTER

TWO

BAYESIAN INFERENCE FOR INVERSE PROBLEMS

2.1 Inverse problem

An inverse problem in science is to infer a set of unknown parameters, $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_m\}$, from data observations $\mathbf{d} = \{d_1, d_2, \ldots, d_n\}$. For example, in Seismology, we deduce the earthquake rupture information (parameterized as $\boldsymbol{\theta}$) from ground motions ($\mathbf{d}$) measured by seismometers, GPS, InSAR and other geodetic surveys. In most cases, the forward problem $\mathbf{d} = G(\boldsymbol{\theta})$ is well defined but the inverse $\boldsymbol{\theta} = G^{-1}(\mathbf{d})$ is not. This happens to linear systems $G(\boldsymbol{\theta}) = \mathbf{G}\boldsymbol{\theta}$ when the observation matrix $\mathbf{G}$ is ill-posed, and nonlinear systems where the inverse is inherently difficult.

2.2 Bayesian approach

Bayesian inference offers a statistical solution to inverse problems by modeling unknown parameters $\boldsymbol{\theta}$ as random variables subject to a conditional probability $P(\boldsymbol{\theta}|\mathbf{d})$. According to Bayes' theorem, the probability is given by

$$
\begin{aligned}
P(\boldsymbol{\theta}|\mathbf{d}) &= \frac{P(\boldsymbol{\theta})P(\mathbf{d}|\boldsymbol{\theta})}{P(\mathbf{d})}, \\
P(\boldsymbol{\theta}|\mathbf{d}) &: \text{posterior, the probability of observing } \boldsymbol{\theta} \text{ given that } \mathbf{d} \text{ is true}, \\
P(\boldsymbol{\theta}) &: \text{prior, the probability of observing } \boldsymbol{\theta} \text{ without regard to } \mathbf{d}, \\
P(\mathbf{d}|\boldsymbol{\theta}) &: \text{likelihood, the probability of observing } \mathbf{d} \text{ given that } \boldsymbol{\theta} \text{ is true}, \\
P(\mathbf{d}) &: \text{the probability of observing } \mathbf{d} \text{ independent of } \boldsymbol{\theta}.
\end{aligned}
$$

Since $P(\mathbf{d})$ is the same for all possible $\boldsymbol{\theta}$ and we treat it as a constant 1. The likelihood $P(\mathbf{d}|\boldsymbol{\theta})$ can be determined from the forward modeling, for example, assuming a form of Gaussian probability density,

$$
P(\mathbf{d}|\boldsymbol{\theta}) \propto e^{-\frac{1}{2}[\mathbf{d}-G(\boldsymbol{\theta})]^T C_\chi^{-1}[\mathbf{d}-G(\boldsymbol{\theta})]},
$$

where the covariance matrix C_χ captures noises/errors in observations $\mathbf{d}$, as well as uncertainties in the forward function $G(\boldsymbol{\theta})$.

The set of model parameters $\boldsymbol{\theta}$ with the maximum posterior probability (MAP), or the mean or median model, provides an estimate solution to the inverse problem. Compared to other point estimators, the Bayesian approach has several advantages. By fully evaluating the posterior probability densities, the Bayesian approach can also address situations when several sets of $\boldsymbol{\theta}$ have equal or comparable probabilities, or the posterior distribution is not unimodal. In addition, the Bayesian approach accommodates uncertainty quantification, by including various types of uncertainties in the calculation.

The Bayesian approach is extremely intensive computationally, since it requires repeating the forward modeling for all possible $\boldsymbol{\theta}$. With the improved sampling algorithms and computing powers, e.g., GPU, it now becomes feasible for many inverse problems with high dimensional parameter space and/or complex forward models.

2.3 CATPMIP algorithm

The posterior distribution $P(\boldsymbol{\theta}|\mathbf{d})$ can be determined, for example, by evaluating $P(\boldsymbol{\theta})P(\mathbf{d}|\boldsymbol{\theta})$ for all possible $\boldsymbol{\theta}$. Instead of uniformly sampling over the entire θ-space, we rely on Markov-Chain Monte-Carlo (MCMC) methods, which draw efficiently *weighted* samples pursuant to a prescribed probability distribution.

The CATMIP (Cascading Adaptive Transitional Metropolis in Parallel) algorithm belongs to a class of MCMC methods which use transitioning: samples are initially drawn from the prior distribution and subsequently *annealed* to the posterior through a series of transient distributions,

$$P_m(\boldsymbol{\theta}|\mathbf{d}) = P(\boldsymbol{\theta})P(\mathbf{d}|\boldsymbol{\theta})^{\beta_m},$$

where β_m (in analogy to the inverse of temperature in statistical physics) is gradually increased from $\beta_0 = 0$ to $\beta_M = 1$ in M steps.

The procedure of CATMIP is described as follows:

1. For the $m = 0$ β-step, with $\beta_0 = 0$, generate N_s random samples from the distribution $P_0(\boldsymbol{\theta}|\mathbf{d}) = P(\boldsymbol{\theta})$, as seeds of N_s parallel chains.
2. Determine β_{m+1} for the next β-step, e.g., from the statistics of the samples from the β_m-step. CATMIP uses an annealing schedule based on the coefficient of variance (COV) of the importance weights $w_i = P(\mathbf{d}|\boldsymbol{\theta}_i)^{\beta_{m+1}-\beta_m}$. The targeted COV is typically set as 1, or the effective sample size (ESS) equals to 50%.
3. Perform a resampling of samples based on their importance weights $\{w_i\}$: some samples with small weights may be discarded while some samples with large weights are duplicated. The total number of samples is kept the same, as seeds for N_s chains in the β_{m+1}-step.
4. With the new β_{m+1}, a burn-in MCMC process is preformed with the Metropolis-Hastings algorithm. New samples are proposed from the distribution $P_m(\boldsymbol{\theta}|\mathbf{d})$, assuming a multivariate Gaussian form, and accepted/rejected subject to the probability density $P_{m+1}(\boldsymbol{\theta}|\mathbf{d})$. The acceptance ratio is also recorded, which is used to scale the jump distances for proposals in the next β-step.
5. Repeat Steps 2-4 until $\beta_M = 1$ is reached, or the desired equilibrium distribution $P(\boldsymbol{\theta}|\mathbf{d})$ is achieved.

Step 4 presents the majority of the computational load. Since each chain is updated independently, the computation is embarrassingly parallel, which makes CATMIP an ideal algorithm to be implemented on parallel computers, including GPUs.

2.4 References

1. Albert Tarantola, *Inverse Problem Theory and Methods for Model Parameter Estimation*, SIAM, 2005. ISBN: 978-0-89871-572-9.
2. Sarah E. Minson, Mark Simons, and James L. Beck, *Bayesian inversion for finite fault earthquake source models I — theory and algorithm*, Geophysical Journal International, Vol. 194, 1701 (2013).

CHAPTER

THREE

INSTALLATION GUIDE

3.1 Overview

AlTar is developed upon the pyre framework. Both packages may be installed from the source code with the CMake build tool. More installation methods, e.g., binaries, will be provided in the future.

In brief, the installation steps consist of:

1. check *Supported Platforms* and install *Prerequisites*;
2. *download* the source packages from github;
3. follow the *Installation Guide* to compile/install `pyre` and `altar`.

Step-by-step instructions are also provided for some representative systems:

- *Conda method recommended method for Linux, Linux Clusters and MacOSX*
- *Ubuntu 18.04/20.04 a standard platform*
- *RHEL/CentOS 7 a standard platform*
- *Linux with environmental modules for Linux clusters*
- *Docker container out-of-the-box delivery*

3.2 Supported Platforms

3.2.1 Hardware

- CPU: `x86_64` (Intel/AMD), `ppc64` (IBM), `arm64` (Apple Silicon/ARM Neoverse/Fujitsu A64FX/…)
- GPU: NVIDIA graphics cards with CUDA-support, with compute capabilities >=3.0
  - Server/Workstation GPUs - Tesla K40, K80, P100, V100, A100, …
  - Workstation graphic Cards - Quadro K6000, M6000, P6000, GV100, RTX x000, …
  - Gaming graphic cards GTX10x0, RTX20x0, RTX30x0, …

Note: AlTar supports both single and double precision GPU computations. Most Quadro and gaming cards have limited double precision computing cores. However, single-precision simulation is sufficient for most models.

3.2.2 Operation systems

- Linux: any distribution should work though not all have been tested;
- Linux clusters: with MPI support and a job queue scheduler (PBS/Slurm);
- MacOSX: with MacPorts or conda;
- Windows: not tested; Windows Subsystem for Linux or Cygwin should work.

Note: AlTar is designed for large scale simulations. We recommend clusters or a workstation with multiple GPUs for simulating compute-intensive models.

3.2.3 Prerequisites

AlTar and pyre have several dependencies:

Required:

- `Python3 >= 3.6` with additional Python packages `numpy` and `h5py`.
- C/C++ compiler: `GCC >= gcc7`, or `clang >= 7`, with C++17 support. Note also that CUDA Toolkit may require certain versions of C/C++ compiler, see CUDA Documentation for more details.
- `GSL >= 1.15`, various numerical libraries including linear algebra and statistics.
- `HDF5 >= 1.10`, a data management system for large and complex data sets.
- `Postgresql`, a SQL database management system (only the library, the server itself is not required).
- `make >= 4`, build tool.
- `cmake >= 3.14`, build tool. Numpy component in FindPython is only supported after 3.14.

Optional:

- `MPI` for multi-thread computations on single machine or cluster system. The recommended option is `openmpi > 1.10` with CXX support (note that on many cluster systems, openmpi is compiled without the `--enable-mpi-cxx` option and therefore doesn't have the *libmpi_cxx.so* library). Other MPI implementations such as MPICH, Intel MPI are also supported.
- `CUDA toolkit >= 10.0` for GPU-accelerated computations. Additional libraries including `cublas`, `curand`, and `cusolver` are also required.
- An accelerated `BLAS` library, such as `atlas`, `openblas`, or `mkl`. Otherwise, the `gslcblas` library, as included in `GSL`, will be used by default.

3.3 Downloads

Please choose a directory where you plan to put all the source files, e.g., `${HOME}/tools/src`,

```
mkdir -p ${HOME}/tools/src
cd ${HOME}/tools/src
```

and download the source packages of pyre and AlTar from their github repositories (main branch):

```
git clone https://github.com/pyre/pyre.git
git clone https://github.com/AlTarFramework/altar.git
```

Currently, some CUDA extensions to pyre and AlTar are not fully merged to the main branch. To install and run the CUDA version of AlTar 2.0, you need to download pyre and altar packages from pyre cuda branch and altar cuda branch instead:

```
git clone https://github.com/lijun99/pyre.git
git clone https://github.com/lijun99/altar.git
```

Note: Pyre is under active development and sometimes the newest version doesn't work properly for AlTar. AlTar users are recommended to obtain pyre from the pyre cuda branch even if only CPU modules are used.

Upon successful downloads, you shall observe two directories `pyre`, `altar` under `${HOME}/tools/src` directory.

3.4 Install with CMake

3.4.1 General steps

This section provides a general instruction on installation procedures. Please refer to the following sections for more system-specific instructions.

Compile and install PYRE at first, with the following commands,

```
# enter the source directory
cd ${HOME}/tools/src/pyre
# create a build directory
mkdir build && cd build
# call cmake to generate make files
cmake .. -DCMAKE_INSTALL_PREFIX=TARGET_DIR -DCMAKE_CUDA_ARCHITECTURES="xx"
# compile
make  # or make -j to use multi-threads
# install
make install # or sudo make install
```

By default, without using `-DCMAKE_INSTALL_PREFIX`, CMake installs the package to `/usr/local`, . If you plan to install the packages to another directory `TARGET_DIR`, you may use the `-DCMAKE_INSTALL_PREFIX` option. It is always a good practice to specify the targeted GPU architecture by, e.g, `-DCMAKE_CUDA_ARCHITECTURES="60"` (targeting NVIDIA Tesla P100 GPU) or `-DCMAKE_CUDA_ARCHITECTURES="35;60"` (targeting both K40/K80 and P100 GPUs). Please refer to *CMake Options* for more details and more options.

The installed files will appear as

```
<install_prefix>
   |--- bin  # executable shell scripts
   |   |- pyre, pyre-config ...
   |- defaults # default configuration files
   |   |- pyre.pfg, merlin.pfg
   |- include # c/c++ header files
   |   |- portinfo, <pyre>
   |- lib # shared libraries
   |   |- libjournal.so libpyre.so ... (or .dylib for Mac)
```

(continues on next page)

(continued from previous page)

```
   |- packages # python packages/scripts
       |- <pyre>, <merlin>, <journal> ...
```

You may also run a few tests to check whether pyre is properly installed.

First, set up the environmental variables (you may also consider to add them to your `.bashrc` or `.cshrc`),

```
# for bash/zsh
export PATH=/usr/local/bin:${PATH}
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
export PYTHONPATH=/usr/local/packages:${PYTHONPATH}
# for csh/tcsh
setenv PATH "/usr/local/bin:$PATH"
setenv LD_LIBRARY_PATH "/usr/local/lib:$LD_LIBRARY_PATH"
setenv PYTHONPATH "/usr/local/packages:$PYTHONPATH"
```

then run commands such as

```
# check pyre module import
python3 -c 'import pyre'
# check cuda module if enabled
# an error will be reported if the module couldn't find a GPU device
python3 -c 'import cuda'
# show the pyre installation directory
pyre-config --prefix
```

There are more test scripts under the source package `${HOME}/tools/src/pyre/tests`.

After installing PYRE and setting up properly the PATHs, you may proceed to compile/install AlTar, with the same procedure,

```
# enter the source directory
cd ${HOME}/tools/src/altar
# create a build directory
mkdir build && cd build
# call cmake to generate make files
cmake .. -DCMAKE_INSTALL_PREFIX=TARGET_DIR -DCMAKE_CUDA_ARCHITECTURES="xx"
# compile
make  # or make -j  to use multi-threads
# install
make install # or sudo make install
```

By default, AlTar is also installed to `/usr/local`. If you choose to install to another directory, you may use the same `-DCMAKE_INSTALL_PREFIX` as for PYRE. By doing so, all the PATHs only need to be set once.

To test whether AlTar is properly installed, you may run the following commands

```
# check altar module import
python3 -c 'import altar'
# show the altar installation directory
altar about prefix
```

There are also tests available in `examples` directories under each model in the source package, for example, `$(HOME)/tools/src/altar/models/linear/examples`.

3.4.2 CMake Options

Here are some commonly used options to control the compilation/installation.

3.4.2.1 Installation path

```
cmake -DCMAKE_INSTALL_PREFIX=${HOME}/tools ..
```

By default, `cmake` installs the compiled package to `/usr/local`. If you plan to install it to another system directory, or your home directory (for users who don't have admin access), such as ${HOME}/tools as shown above. Remember to set properly the environmental variables `PATH`, `LD_LIBRARY_PATH` and `PYTHONPATH`. If you use `Conda`, you may use `-DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX`.

3.4.2.2 Enable/disable CUDA

```
cmake -DWITH_CUDA=ON (or OFF) ..
```

By default, *WITH_CUDA=ON* for the cuda branch version and *WITH_CUDA=OFF* for the main branch version. To enable CUDA extensions, you will also need the CUDA Toolkit. If not found, `cmake` will automatically turn *WITH_CUDA=OFF*.

3.4.2.3 Target GPU architecture(s)

Note: We recommend specifying a proper GPU architecture with `-DCMAKE_CUDA_ARCHITECTURES="xx"` or `-DCMAKE_CUDA_FLAGS="-arch=sm_xx"`. It not only ensures the efficient GPU executable code for your device, but also avoids the issue of code incompatibilities. CUDA Toolkit 9 and 10 use `sm_30` as the default target architecture, while CUDA 11 uses `sm_52`. The compiled code will continue to run on GPU devices with higher compute capabilities, but not on GPU devices with lower compute capabilities, reporting an error `no kernel image is available for execution on the device`. This happens, e.g., when you have a K40 (`sm_35`) and use a `sm_52` flag (default by CUDA 11). AlTar does not support CUDA on Mac, and `-DCMAKE_CUDA_FLAGS=...` will be neglected if provided.

To specify the targeted GPU architecture(s),

```
# target one architecture
cmake -DCMAKE_CUDA_ARCHITECTURES="60" ..
# target multiple architectures
cmake -DCMAKE_CUDA_ARCHITECTURES="35;60" ..
```

Note that `CUDA_ARCHITECTURES` is only available on CMake 3.18 and later versions. For earlier versions, you may use `CUDA_FLAGS` instead,

```
# target one architecture
cmake -DCMAKE_CUDA_FLAGS="-arch=sm_60" ..
# target multiple architectures
cmake -DCMAKE_CUDA_FLAGS="-gencode arch=compute_35,code=sm_35 -gencode arch=compute_
↪60,code=sm_60" ..
```

`CUDA_FLAGS` may also be used to pass other compiling options to CUDA compiler `nvcc`.

You may find out which type(s) of GPU are installed by running

```
nvidia-smi
```

Compute capabilities for some common NVIDIA GPUs are K40/80 (`sm_35`), V100 (`sm_70`), A100 (`sm_80`), GTX1050/1070/1080 ((`sm_61`), RTX 2060/2070/2080 (`sm_75`), RTX 3060/3070/3080(`sm_86`. More details can be found at NVIDIA website. If pyre is already installed, you may also use its cuda utilities to find out:

```
python3 -c "import cuda; [print(f'Device {device.id} {device.name} has compute␣
↪capability {device.capability}') for device in cuda.devices]"
```

3.4.2.4 C++ Compiler

To specify the C++ compiler, e.g., */usr/bin/g++*, you may use

```
cmake -DCMAKE_CXX_COMPILER=/usr/bin/g++ ..
```

Note that pyre requires a GCC>=7 for c++17 support.

C++ compiler may also be specified from the environmental variable `CXX`, for example,

```
# bash/zsh
export CXX = "/usr/bin/g++"
# csh/tcsh
setenv CXX  "/usr/bin/g++"
```

3.4.2.5 CUDA Compiler

To specify the CUDA compiler `nvcc`, e.g., */usr/local/cuda-11.3/bin/nvcc*, you may use

```
cmake -DCMAKE_CUDA_COMPILER=/usr/local/cuda-11.3/bin/nvcc ..
```

C++ compiler may also be specified from the environmental variable `CUDACXX`, for example,

```
# bash/zsh
export CUDACXX = "/usr/local/cuda-11.3/bin/nvcc -arch=sm_60"
# csh/tcsh
setenv CUDACXX  "/usr/local/cuda-11.3/bin/nvcc -arch=sm_60"
```

3.4.2.6 BLAS Library

Pyre requires a BLAS library for its `gsl` module. CMake searches automatically an available BLAS library by default. If none is found, the `gslcblas` library included with GSL package will be used. You may also specify which BLAS library to use by

```
cmake .. -DBLA_VENDOR=vendor
```

where `vendor` can be `Generic`` (``libblas.so`), `ATLAS`, `Intel10_64lp`, `OpenBLAS` .... You may also add `-DCMAKE_PREFIX_PATH=/path/to/blas` to enforce a search path.

3.4.2.7 Library search path

To specify the locations of a prerequisite library instead of the default one, for example, on some Linux systems, `cmake` may find and use libraries from `/usr/` instead of the libraries provided by conda, you may use

```
cmake -DCMAKE_PREFIX_PATH=${CONDA_PREFIX} ..
```

to enforce libraries installed under Conda to be used.

For more than one paths, use a semicolon separated list, *-DCMAKE_PREFIX_PATH="PATH1;PATH2;PATH3"*.

3.4.2.8 Build type

```
cmake -DCMAKE_BUILD_TYPE=Release (or Debug) ..
```

For the Debug build type, the `-g` compiler flag will be added to generate debugging information. For the Release type, the `-O3` optimization flag will be added. If none is specified, the default flags of `g++` are used.

3.4.2.9 Show compiling details

By default, the compiling step `make` only shows one line summary of each file being compiled. To the detailed compiling command and options, you may use

```
make VERBOSE=1
```

3.4.2.10 More options

For more options of `cmake`, please check CMake Documentation.

3.5 Conda method (Linux/MacOSX)

3.5.1 Install Anaconda/Miniconda

Conda(Anaconda/Miniconda) offers an easy way to install Python, packages and libraries on different platforms, especially for users without the admin privilege to their computers. We recommend a full version of Anaconda3. If disk space is an issue or you plan to use a virtual environment, you may use Miniconda instead.

For MacOSX with Apple Silicon, you may install the native `arm64` version from Miniforge.

If Anaconda3 is not installed, please download and follow the instructions to install it. You may choose to install it under you home directory `${HOME}/anaconda3` (default) or a system directory, e.g., `/opt/anaconda3`. The path to the Anaconda3 is set as an environmental variable `CONDA_PREFIX`. To check whether Anaconda3 is properly installed and loaded, you may try the following commands

```
$ which conda
/opt/anaconda3/bin/conda
$ which python3
/opt/anaconda3/bin/python3
$ echo ${CONDA_PREFIX}
/opt/anaconda3
```

You may also create a virtual environment

```
$ conda create -n altar
$ conda activate altar
$ which python3
/opt/anaconda3/envs/altar/bin/python3
$ echo ${CONDA_PREFIX}
/opt/anaconda3/envs/altar
```

3.5.2 Install Conda packages

Install the required libraries and packages by Conda:

```
$ conda install git make cmake hdf5 h5py openmpi gsl openblas postgresql numpy scipy
```

3.5.3 C++ Compiler

You will also need a c++ compiler.

- **Ubuntu 18.04/20.04** GCC 7.4.0/9.3.0 is installed by default and is sufficient. If GCC/G++ are not installed, run

```
sudo apt install gcc g++
```

- **Redhat/CentOS 7** The system default compiler GCC 4.x doesn't support C++17. Higher versions of GCC are offered through `devtoolset`. Please follow instructions for Redhat or CentOS to install, e.g., `devtoolset-7`. An alternative is to use GNU compilers provided by Conda, see below.
- **MacOSX** You will need to install either the full version of Xcode or the (compact) Command Line Tools. Xcode can be installed from the App Store. To install the Command Line Tools, run

```
sudo xcode-select --install
# To select or switch compilers,
sudo xcode-select --switch /Library/Developer/CommandLineTools/
```

- **Conda GNU Compilers** Conda also offers compiler packages, which work well for most Linux/MacOSX(Intel) systems,

```
# for Linux x86_64
conda install gcc_linux-64 gxx_linux-64
# for Mac (Intel Only)
conda install clang_osx-64 clangxx_osx-64
# for Mac Big Sur with Xcode 12 (Intel only), you need to use clang-11,
conda install clang_osx-64=11.0.0 clangxx_osx-64=11.0.0 -c conda-forge
# for Mac with Apple Silicon, please use only Command Line Tools or Xcode
```

If you would like to use a c++ compiler other than the default version, or the version (auto) discovered by `cmake`, you may use `-DCMAKE_CXX_COMPILER=...` to specify the compiler.

3.5.4 CUDA compiler (nvcc)

CUDA Toolkit integrates tools to develop GPU applications, including the compiler (`nvcc`), libraries (`libcudart.so`, `libcublas.so` ...). If CUDA is installed, you may obtain and install CUDA Toolkit following the NVIDIA Toolkit Documentation.

CUDA Toolkit is usually installed to `/usr/local/cuda`. On Linux clusters, many version of CUDA toolkit may be provided as modules. You may select a version by

```
module load cuda/11.3
```

Conda also provides a CUDA Toolkit package,

```
conda install cudatoolkit
```

which is installed to `$CONDA_PREFIX` directory.

You may check the CUDA Toolkit installation by

```
# check the nvcc availability and path
$ which nvcc
/usr/local/cuda/bin/nvcc
# check the CUDA Toolkit version
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Cuda compilation tools, release 11.3, V11.3.109
```

`CMake` discovers the default `nvcc` command to compile CUDA programs. You may also specify anther CUDA compiler by `-DCMAKE_CUDA_COMPILER=/path/to/nvcc`.

Note that NVIDIA driver, including the CUDA driver (`libcuda.so`), is required on a GPU workstation or GPU nodes in a cluster. NVIDIA drivers can only be installed/updated by *root* users. You may check their availability and versions by the command `nvidia-smi`. CUDA shared libraries should also be available on GPU workstations.

3.5.5 Download pyre and AlTar

Please download the source packages of pyre/AlTar from github following the *Download instructions*. Taking CUDA branch versions as an example,

```
mkdir -p ${HOME}/tools/src
cd ${HOME}/tools/src
git clone https://github.com/lijun99/pyre.git
git clone https://github.com/lijun99/altar.git
```

3.5.6 Install pyre

With Conda, we recommend installing pyre and AlTar to `$CONDA_PREFIX`, so that both packages are loaded automatically when conda or conda venv is activated. We need an extra step to make a symbolic link to `lib/python3.x/site-packages`,

```
# the python command returns the path of site-packages, and we link it as $CONDA_
↪PREFIX/packages
ln -sf `python3 -c 'import site; print(site.getsitepackages()[0])'` $CONDA_PREFIX/
↪packages
```

Compile and install pyre

```
cd ${HOME}/tools/src/pyre
mkdir build && cd build
cmake .. -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX -DCMAKE_PREFIX_PATH=$CONDA_PREFIX -
↪DCMAKE_CUDA_ARCHITECTURES=native -DBLA_VENDOR=OpenBLAS -DPython3_EXECUTABLE=$CONDA_
↪PREFIX/bin/python3

make -j && make install
```

where `INSTALL_PREFIX` is the installation path and `PREFIX_PATH` is the path to search the prerequisite packages. Replace `native` with appropriate compute capability number(s) for your GPU(s). See *GPU architecture(s)* for more details.

Note: `FindPython3` in new versions of `cmake` sometimes finds the system python3 interpreter instead of the conda installed one, please add `-DPython3_EXECUTABLE=$CONDA_PREFIX/bin/python3` as above to assist the search. The new standard is to use `FindPython` instead; we will update the pyre/altar cmake files.

3.5.7 Install AlTar

Since pyre is installed to `$CONDA_PREFIX`, there is no need to set the PATHs. We proceed to compile and install AlTar, with the same procedure,

```
cd ${HOME}/tools/src/altar
mkdir build && cd build
cmake .. -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX -DCMAKE_PREFIX_PATH=$CONDA_PREFIX -
↪DCMAKE_CUDA_ARCHITECTURES=native -DPython3_EXECUTABLE=$CONDA_PREFIX/bin/python3

make -j && make install
```

Please read *CMake Options* if you have some problems or need more customizations. Please also read *Installation instructions* on how to make tests.

For future runs, you may simply activate conda or the conda venv to load AlTar,

```
# activate altar if it is installed in a venv
conda activate altar
# test
altar about
```

3.5.8 MPI setup

AlTar runs MPI jobs by automatically forking multiple threads and invoking `mpirun` command with any AlTar Application, a capability offered by pyre `mpi` shell. However, pyre sometimes does not recognize the Conda-installed openMPI. You will need to create manually a configuration file, `mpi.pfg`, either under `$(HOME)/.pyre` directory or under the current job directory, as

```
; mpi.pfg file

mpi.shells.mpirun:
  ; mpi implementation
  mpi = openmpi#mpi_conda

; mpi configuration
pyre.externals.mpi.openmpi # mpi_conda:
```

(continues on next page)

(continued from previous page)

```
  version = 4.0.5
  launcher = mpirun
  prefix = /opt/anaconda3/envs/altar
  bindir = {mpi_conda.prefix}/bin
  incdir = {mpi_conda.prefix}/include
  libdir = {mpi_conda.prefix}/lib
```

You need to replace `/opt/anaconda3/envs/altar` with the actual path of your `$CONDA_PREFIX`, which can be revealed by the command `echo $CONDA_PREFIX`.

Another option is to insert these lines to your job configuration file, without creating a separate `mpi.pfg` file.

This setup procedure also applies to other MPIs not automatically recognized by pyre, e.g., loaded by environmental modules.

3.6 Linux Systems

We recommend Conda methods for all Linux systems. If you prefer to use the standard Linux packages, please follow the instructions in this section.

3.6.1 Ubuntu 18.04/20.04

3.6.1.1 Install prerequisites

```
$ sudo apt update && sudo apt install -y gcc g++ python3 python3-dev python3-numpy␣
↪python3-scipy python3-h5py libgsl-dev libopenblas-dev libpq-dev postgresql-server-
↪dev-all libopenmpi-dev libhdf5-serial-dev make git
```

For Ubuntu 18.04 only: the system CMake version is 3.10; you need to manually upgrade cmake from Kitware Repo, e.g.,

```
$ sudo wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null␣
↪| sudo apt-key add -
$ sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ bionic main'
$ sudo apt-get update
$ sudo apt-get install cmake
```

To install/run CUDA modules, you will also need to install CUDA Toolkit if it is not pre-installed. See *CUDA compiler (nvcc)* for more details.

3.6.1.2 Install pyre/AlTar

Please follow the instructions in *General steps*.

3.6.2 RHEL/CentOS 7

3.6.2.1 Install prerequisites

Enable EPEL repo

```
yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

and install prerequisites

```
yum install -y python3 python3-devel hdf5-devel gsl-devel postgresql-devel openmpi␣
↪openmpi-devel git environment-modules
# install numpy/scipy/h5py via pip
pip3 install numpy scipy h5py
# load openmpi
module load mpi
# install cmake from Kitware
wget https://github.com/Kitware/CMake/releases/download/v3.19.3/cmake-3.19.3-Linux-
↪x86_64.sh
sh cmake-3.19.3-Linux-x86_64.sh  --skip-license --prefix=/usr/local
```

Install C/C++ compiler

```
# 1. Install a package with repository for your system:
# On CentOS, install package centos-release-scl available in CentOS repository:
sudo yum install centos-release-scl

# On RHEL, enable RHSCL repository for you system:
sudo yum-config-manager --enable rhel-server-rhscl-7-rpms

# 2. Install the collection:
sudo yum install devtoolset-7

# 3. Start using software collections:
scl enable devtoolset-7 bash
```

3.6.2.2 Install pyre/AlTar

Please follow the instructions in *General steps*.

3.6.3 Linux with software modules

Many clusters use software modules to load libraries and software packages, e.g.,

```
# list available modules
module av
# load a certain module
module load cuda/10.2
# list loaded modules
module list
# show the loaded module information
module show cuda
```

Please load all necessary modules as listed in *Prerequisites*. You may then follow the *General steps* above to install pyre and AlTar.

Since modules are set up differently in different computers, we only provide a general prescription if CMake fails to locate the prerequisite package from auto search.

You may provide the package path to CMake by,

- `-DCMAKE_PREFIX_PATH`, which specifies the package installation prefix to be searched. Files are expected to be arranged in a standard fashion under prefix, `bin`, `includes`, `lib`. If the files are not arranged in the standard way, you may use options below,
- `-DCMAKE_INCLUDE_PATH`, which specifies the search path(s) for header files;
- `-DCMAKE_LIBRARY_PATH`. which specifies the search paths(s) for library files.

Each of these three parameters can be a semicolon-separated list to include more than one paths, e.g., `-DCMAKE_PREFIX_PATH="/PATH/To/GSL;/PATH/To/HDF5;/PATH/To/MPI`.

CMake uses various builtin *Find modules* to search various packages, while each *Find module* may use some *hints* to locate the package. For example, `FindGSL` uses `GSL_ROOT_DIR`, and `FindMPI` uses `MPIEXEC_EXECUTABLE` or `MPI_HOME`. These mint may be passed as environmental variables `export GSL_ROOT_DIR=...` or as cmake options, e.g., `-DGSL_ROOT_DIR=...`.

Note: Many clusters have their own recommended MPI packages which are optimized for the specific type of interconnects. Before using these pre-installed MPI packages, please check whether they have `cxx` devel support, or `libmpi_cxx.so` is available, which is required by AlTar.

3.7 Docker container

You may follow the steps below to build a docker image, based on a NVIDIA cuda 10.2 image with Ubuntu.

```
wget https://gitlab.com/nvidia/container-images/cuda/raw/master/dist/ubuntu18.04/10.2/
↪runtime/Dockerfile
docker build --build-arg IMAGE_NAME=nvidia/cuda . -t cuda/nvidia:10.2
docker exec -it cuda/nvidia:10.2
apt update && apt install -y gcc g++ python3 python3-dev python3-numpy python3-numpy-
↪dev python3-h5py libgsl-dev libopenblas-dev libpq-dev postgresql-server-dev-all␣
↪libopenmpi-dev libhdf5-serial-dev make git wget software-properties-common locales
locale-gen --no-purge --lang en_US.UTF-8 && update-locale LANG=en_US.UTF-8 LANGUAGE
wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null | apt-
↪key add - && apt-add-repository 'deb https://apt.kitware.com/ubuntu/ bionic main' &&
↪ apt-get update && apt install -y cmake
apt install -y cuda-compiler-10-2 cuda-cudart-dev-10-2 cuda-curand-dev-10-2 libcublas-
↪dev cuda-cusolver-dev-10-2
ln -sf /usr/lib/python3/dist-packages /usr/local/packages
cd /usr/local/src
git clone https://github.com/lijun99/pyre.git
git clone https://github.com/lijun99/altar.git
cd /usr/local/src/pyre && mkdir build && cd build && cmake .. && make all && make␣
↪install
cd /usr/local/src/altar && mkdir build && cd build && cmake .. && make all && make␣
↪install
echo ': "${LANG:=en_US.UTF-8}"; export LANG' >> /etc/profile
```

Open another terminal, find out the *CONTAINER ID* for this image, named *cuda/nvidia:10.2*, and commit the changes to a new image

```
$ docker commit CONTAINER_ID altar:2.0.2-cuda
```

To run AlTar from the container

```
$ docker run --gpus all -ti -v ${PWD}:/mnt altar:2.0.2-cuda
```

which also mounts the current directory as /mnt in the virtual system. You may go to your job directory and run AlTar from there.

If you meet a `UnicodeDecodeError`, you will need to `export LANG=en_US.UTF-8` at first. See *Locales* for more details.

OpenMPI may issue a warning to run MPI jobs as a *root* user, you may add the `--allow-run-as-root` option to your job configuration file as follows,

```
; for parallel runs
mpi.shells.mpirun:
    extra = -mca btl self,tcp --allow-run-as-root
```

3.8 Install with the mm build tool

The mm build tool (please note that it is different from the old mm, or config build tool) is another powerful tool to build hybrid python/c/c++/cuda applications.

3.8.1 Download `mm` build tool

```
cd ${HOME}/tools/src
git clone https://github.com/aivazis/mm.git
```

3.8.2 Prepare a `config.mm` file

The `mm` build tool requires a `config.mm` file to locate dependent libraries or packages. Taking Ubuntu 18.04 as an example, the `config.mm` file appear as

```
# file config.mm

# gsl
gsl.dir = /usr
gsl.incpath = /usr/include
gsl.libpath = /usr/lib/x86_64-linux-gnu

# mpi
mpi.dir = /usr/lib/x86_64-linux-gnu/openmpi/
mpi.binpath = /usr/bin
mpi.incpath = /usr/lib/x86_64-linux-gnu/openmpi/include
mpi.libpath = /usr/lib/x86_64-linux-gnu/openmpi/lib
mpi.flavor = openmpi
mpi.executive = mpirun

# hdf5
hdf5.dir = /usr
```

(continues on next page)

(continued from previous page)

```
hdf5.incpath = /usr/include
hdf5.libpath = /usr/lib/x86_64-linux-gnu

# postgresql
libpq.dir = /usr
libpq.incpath = /usr/include/postgresql
libpq.libpath = /usr/lib/x86_64-linux-gnu

# openblas
openblas.dir = /usr
openblas.libpath = /usr/lib/x86_64-linux-gnu

# python3
python.version = 3.6
python.dir = /usr
python.binpath = /usr/bin
python.incpath = /usr/include/python3.6m
python.libpath = /usr/lib/python3.6

# numpy
numpy.dir = /usr/lib/python3/dist-packages/numpy/core

# cuda
cuda.dir = /usr/local/cuda
cuda.binpath = /usr/local/cuda/bin
cuda.incpath = /usr/local/cuda/include
cuda.libpath = /usr/local/cuda/lib64 /usr/lib/x86_64-linux-gnu/
cuda.libraries := cudart cublas curand cusolver

# end of file
```

You may leave the `config.mm` file in the `pyre/.mm`, `altar/.mm` directories, or in the `${HOME}/.mm` directory to be shared by all projects.

Examples of *config.mm* files are available at config.mm.

3.8.3 Install pyre

After preparing all required libraries/packages and the `config.mm` file (in `pyre/.mm` or `${HOME}/.mm`), you need to compile and install pyre at first.

Make an alias of the mm command, in `bash`

```
$ alias mm='python3 ${HOME}/tools/src/mm/mm.py'
```

or in `csh/tcsh`,

```
$ alias mm 'python3 ${HOME}/tools/src/mm/mm.py'
```

Now, you can compile `pyre` by

```
$ cd ${HOME}/tools/src/pyre
$ mm
```

By default, the compiled files are located at `${HOME}/tools/src/pyre/products/debug-shared-linux-x86_64`. If you need to customize the installation, you can check the options offered by

`mm` by

```
$ mm --help
```

For example, if you prefer to install pyre to a system folder, you may use `--prefix` option, such as

```
$ mm --prefix=/usr/local
```

After compiling/installation, you need to set up some environmental variables for other applications to access `pyre`, for example, create a `${HOME}/.pyre.rc` for `bash`,

```
# file .pyre.rc
export PYRE_DIR=${HOME}/tools/src/pyre/products/debug-shared-linux-x86_64
export PATH=${PYRE_DIR}/bin:$PATH
export LD_LIBRARY_PATH=${PYRE_DIR}/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${PYRE_DIR}/packages:$PYTHONPATH
export MM_INCLUDES=${PYRE_DIR}/include
export MM_LIBPATH=${PYRE_DIR}/lib
# end of file
```

or `${HOME}/.pyre.cshrc` for `csh/tcsh`,

```
# file .pyre.cshrc
setenv PYRE_DIR "${HOME}/tools/src/pyre/products/debug-shared-linux-x86_64"
setenv PATH "${PYRE_DIR}/bin:$PATH"
setenv LD_LIBRARY_PATH "${PYRE_DIR}/lib:$LD_LIBRARY_PATH"
setenv PYTHONPATH "${PYRE_DIR}/packages:$PYTHONPATH"
setenv MM_INCLUDES "${PYRE_DIR}/include"
setenv MM_LIBPATH "${PYRE_DIR}/lib"
# end of file
```

You will also need to append `pyre` configurations to `${HOME}/.mm/config.mm` or `altar/.mm/config.mm` or any other application who requires `pyre`,

```
# append to the following lines to an existing config.mm
# pyre
pyre.dir =  ${HOME}/tools/src/pyre/products/debug-shared-linux-x86_64
pyre.libraries := pyre journal ${if ${value cuda.dir}, pyrecuda}
```

3.8.4 Install AlTar

First, make sure that you have a prepared `config.mm` file, which also includes the `pyre` configuration, in either `altar/.mm/` or `${HOME}/.mm` directory.

Follow the same step to compile AlTar,

```
$ cd ${HOME}/tools/src/altar
$ mm
```

Similar to `pyre` installation, the AlTar products are located at `${HOME}/tools/src/altar/products/debug-shared-linux-x86_64`, or the directory specified by `mm --prefix=`.

Also, you need to set up some environmental variables for `altar` as well, for example, create a `${HOME}/.altar2.rc` for `bash`,

```
# file .altar2.rc
export ALTAR2_DIR=${HOME}/tools/src/altar/products/debug-shared-linux-x86_64
export PATH=${ALTAR2_DIR}/bin:$PATH
export LD_LIBRARY_PATH=${ALTAR2_DIR}/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${ALTAR2_DIR}/packages:$PYTHONPATH
# end of file
```

or `${HOME}/.altar2.cshrc` for `csh/tcsh`,

```
# file .altar2.cshrc
setenv ALTAR2_DIR "${HOME}/tools/src/altar/products/debug-shared-linux-x86_64"
setenv PATH "${ALTAR2_DIR}/bin:$PATH"
setenv LD_LIBRARY_PATH "${ALTAR2_DIR}/lib:$LD_LIBRARY_PATH"
setenv PYTHONPATH "${ALTAR2_DIR}/packages:$PYTHONPATH"
# end of file
```

Before running an altar/pyre application, you need to load the altar/pyre environmental settings

```
$ source ${HOME}/.pyre.rc
$ source ${HOME}/.altar2.rc
```

3.9 Tests and Examples

Pyre tests are available at `${HOME}/tools/src/pyre/tests`.

AlTar examples are are available for each model.

For details how to run AlTar applications, please refer to *User Guide*.

CHAPTER

FOUR

USER GUIDE

4.1 Overview

AlTar is a software package implementing the *CATPMIP algorithm* and other Markov-Chain Monte-Carlo algorithms for Bayesian inference. It adopts the component-based software architecture and the job management system from the pyre framework, aiming to make high performance scientific computations more accessible to researchers with minimum knowledge of software engineering.

An AlTar application includes the following root components,

- the Bayesian framework (controller/annealer), which performs the MCMC sampling of the Bayesian posterior distribution;
- a model, which performs the forward modelling and feeds the resulting data likelihood to the Bayesian framework;
- job, which manages the size of a simulation and its deployment to different platforms;

while each component can be configured from the configuration file, e.g., switching between different algorithms/implementations, turning on/off features, and of course, changing the value of a parameter.

To run AlTar simulations, it is usually done by a simple command

```
anAlTarApp --config=anAlTarApp.pfg
```

where `anAlTarApp` is an AlTar application tailed for a given inverse problem, while `anAlTarApp.pfg` is a configuration file you where specify settings for your simulation.

We have provided examples with each implemented inverse problem. You may use them as templates to prepare your own simulation. This Guides aims to provide detailed instructions on how to configure each component, which is organized as follows,

- *Quickstart*: to demonstrate how to run AlTar simulations with a linear inverse problem;
- *An introduction to pyre*: to offer a brief introduction to components and the `.pfg` configuration file format;
- *The AlTar framework*: how to configure the components of the MCMC simulation;
- *Job Management*: how to configure the job deployment to different platforms;
- Models: to provide instructions on how to prepare data and run simulations for inverse problems in geophysics,
  - *Static inversion of earthquake source models*;
  - Static inversion with Cp (forward model uncertainty);
  - *Kinematic inversion of earthquake source models*;
  - Mogi source model for volcanoes (TBD);
  - Compound dislocation model (CDM) for volcanoes (TBD).

Users may also develop new models for other inverse problems. A *Programming Guide* is also provided.

4.2 QuickStart

As a quick start, we use the linear model as an example to demonstrate how to run Bayesian MCMC simulations with AlTar. :

1. Prepare a configuration file, e.g., `linear.pfg`, to specify various parameters and settings;
2. Prepare input data files required for the model, e.g., for the linear model, the observed data, the data covariance and the Green's function;
3. Run a dedicated AlTar application, e.g., `linear` for the linear model;
4. Collect and analyze the simulation results.

The linear model example demonstrated here comes with the AlTar source package, under the directory models/linear/examples. It is also available as a jupyter notebook in Tutorials:linear.

4.2.1 Prepare the configuration file

A configuration file is used to pass various settings to an AlTar application. Here is an example for the linear model,

Listing 1: linear.pfg

```
;
; michael a.g. aïvázis
; orthologue
; (c) 1998-2020 all rights reserved
;

; the application
linear:
    ; the model
    model = linear
    ; the linear model configurations
    model:
        ; the directory for input files
        case = patch-9
        ; the number of parameters
        parameters = 18

        ; the number of observations
        observations = 108
        ; the data observations file
        data = data.txt
        ; the data covariance file
        cd = cd.txt

        ; prior distribution for parameters
        ; prior is used to calculate the prior probability
        ;    and check ranges during the simulation
        prior = gaussian
        ; prior configurations
        prior:
            parameters = {linear.model.parameters}
```

(continues on next page)

(continued from previous page)

```
            center = 0.0
            sigma = 0.5
        ; prep is used to initialize the samples in the beginning of the simulation
        ; it can be different from prior
        prep = uniform
        prep:
            parameters = {linear.model.parameters}
            support = (-0.5, 0.5)

    ; controller/annealer, use the default CATMIP annealer
    controller:
        ; archiver, use the default HDF5 achiver
        archiver:
            output_dir = results ; results output directory
            output_freq = 1 ; output frequency in annealing beta steps

    ; run configuration
    job.tasks = 1 ; number of tasks per host
    job.gpus = 0  ; number of gpus per task
    job.chains = 2**10 ; number of chains per task

; end of file
```

The `.pfg` (pyre config) files follow a human-readable data-serialization format similar to YAML, where the data-structure hierarchy is maintained by whitespace indentation (or by full/partial paths, such as *job.tasks*, see *Pyre Config Format (.pfg)* for more detailed instructions).

The name of the AlTar application (instance), `linear`, is set as the root. Configurable components of an AlTar application include

- `model`, for model specific configurations, such as the prior distributions of the model parameters, parameters in the forward model, and the data observations;
- `job`, which configures the size of the simulation, and how the job will be deployed, e.g., single or multiple threads, single machine or multi-node cluster, CPU or GPU;
- `controller`, for configurations to control the Bayesian MCMC procedure.

Note: If a component is not specified in the configuration, its *default* value/implementation will be used instead.

Model configurations vary depending on its own forward problem: model-specific instructions are provided in the respective sections of this Guide. Instructions for the main framework, such as `job` and `controller`, can be found in the *AlTar Framework* section.

4.2.2 Prepare input files

While simple configurations can be specified in the configuration file, large sets of data are passed to the AlTar application by data files. Different model may require different categories of data, in different input format.

For the linear model, the data likelihood is computed as

$$P(\mathbf{d}|\boldsymbol{\theta}) = \frac{1}{\sqrt{(2\pi)^m \det(C_d)}} \times \exp\left[-\frac{1}{2}\left(\mathbf{d}-\mathbf{d}^{pred}\right)^T C_d^{-1}\left(\mathbf{d}-\mathbf{d}^{pred}\right)\right]$$

where $\boldsymbol{\theta}$ is a vector with n unknown model parameters, $\mathbf{d}$ a vector with m observations, the covariance C_d a $m \times m$ matrix representing the data uncertainties. The data prediction $\mathbf{d}^{pred}$ is given by the forward model

$$\mathbf{d}^{pred} = \mathcal{G}\boldsymbol{\theta}.$$

where the Green's function $\mathcal{G}$, is a $n \times m$ matrix.

The computation requires three users' input, $\mathbf{d}$, C_d, and $\mathcal{G}$, as plain text files `data.txt`, `cd.txt` and `green.txt`. The location of the files can be specified by the `linear.model.case` parameter in the configuration file, while the file names can be specified by `linear.model.data`, `linear.model.cd` and `linear.model.green`.

4.2.3 Run an AlTar application

For each model, we have provided a dedicated command for running AlTar simulations (in fact, you can run `altar` for all models, but it may require some changes to the configuration file). The dedicated command for the linear model is `linear`, which is a Python script as shown below

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# michael a.g. aïvázis (michael.aivazis@para-sim.com)
#
# (c) 2010-2020 california institute of technology
# (c) 2013-2020 parasim inc
# all rights reserved
#

# get the package
import altar

# make a specialized app that uses this model by default
class Linear(altar.shells.application, family='altar.applications.linear'):
    """
    A specialized AlTar application that exercises the Linear model
    """

    # user configurable state
    model = altar.models.model(default='linear')
    model.doc = "the AlTar model to sample"


# bootstrap
if __name__ == "__main__":
    # build an instance of the default app
    app = Linear(name="linear")
    # invoke the main entry point
    status = app.run()
    # share
    raise SystemExit(status)


# end of file
```

It defines a `Linear` application class and provides a `main` entry point for execution. The `linear` can be run as any other shell commands, but you do need to run it at the directory where the `linear.pfg` and the `case (patch-9)` directory are located,

```
linear
```

and the simulation begins.

If you would like to use a different script file other than `linear.pfg`

```
linear --config=linear2.pfg
```

or if you would like to pass/change a parameter from command lines, e.g., to increase the number of Markov chains

```
linear --job.chains=2**10
```

More run options will be explained in the *AlTar Framework* section.

4.2.4 Collect and analyze results

AlTar offers several options how to output the simulation results. The default is an HDF5 archiver, which outputs the simulation results from each β-step to HDF5 files located at `results` directory. Data in these HDF5 files, named as `step_nnn.h5`, can be viewed by a HDF Viewer, such as HDFView, HDFCompass.

For each `step_nnn.h5`, the following structures are used

```
+---------- step_nnn.h5 ------
├── Annealer ; annealing data
|   ├── beta ; the beta value
|   └── covariance ; the covariance matrix for Gaussian proposal
├── Bayesian ; the Bayesian probabilities
|   ├── prior ; prior probability for all samples, vector (samples)
|   ├── likelihood ; data likelihood for all samples
|   └── posterior ; posterior probability for all samples
└── ParameterSets ;  samples
    └── theta ; samples of model parameters, 2d array (samples, parameters)
```

```
import h5py
import numpy
```

You may draw a histogram of the posterior to check its distribution. Since the log values of the probabilities are used and saved, the distribution will normally show a lognormal form. You may also do some statistics on the samples, for example, mean and standard deviations. If the posterior assumes a Gaussian distribution, the mean model provides an estimated solution to the linear inverse problem. Some of the data analysis programs are also included with AlTar.

AlTar is a software package developed to perform Bayesian inference to inverse problems with the Markov Chain Monte-Carlo methods. It consists of

- a main framework which performs the Bayesian MCMC, and controls the job deployment;
- a model which performs the forward modeling and feeds the data likelihood results to the Bayesian framework. Model implementations for various inverse problems are included. Users may add new models by

An AlTar application integrates a model with the main framework and serves as the main program to run simulations.

4.3 Pyre Basics

AlTar's architecture is based on the pyre framework. Before the official documentation for pyre is released, we offer here a brief introduction to pyre, its programming design and its configuration file format which are necessary for AlTar users.

4.3.1 Protocols and Components

Python offers a modular programming design with modules and classes while pyre extends the functionalities of python classes to facilitate their integrations and configurations, through *components*.

A prescribed functionality is defined as a *protocol* (similar to an abstract class). For example, various distributions are used in AlTar, mainly serving as prior distributions. We first define a protocol,

```
# the protocol
class Distribution(altar.protocol, family="altar.distributions"):
    """
    The protocol that all AlTar probability distributions must satisfy
    """

    # required behaviors
    @altar.provides
    def initialize(self, **kwds):
        """
        Initialize with the given random number generator
        """

    # model support
    @altar.provides
    def initializeSample(self, theta):
        """
        Fill my portion of {theta} with initial random values from my distribution.
        """

    @altar.provides
    def priorLikelihood(self, theta, prior):
        """
        Fill my portion of {prior} with the likelihoods of the samples in {theta}
        """

    @altar.provides
    def verify(self, theta, mask):
        """
        Check whether my portion of the samples in {theta} are consistent with my␣
↪constraints, and
        update {mask}, a vector with zeroes for valid samples and non-zero for␣
↪invalid ones
        """

    ... ...

    # framework hooks
    @classmethod
    def pyre_default(cls):
        """
        Supply a default implementation
        """
```

(continues on next page)

(continued from previous page)

```
        # use the uniform distribution
        from .Uniform import Uniform as default
        # and return it
        return default
```

where `@altar.provides` decorator specifies the behaviors (methods) that its implementations must define. An implementation, for example, the Uniform distribution, is defined as a *component*,

```
class Uniform(altar.component, family="altar.distributions.uniform"):
    """
    The uniform probability distribution
    """

    # user configurable state
    parameters = altar.properties.int()
    parameters.doc = "the number of model parameters that i take care of"

    offset = altar.properties.int(default=0)
    offset.doc = "the starting point of my parameters in the overall model state"

    # user configurable state
    support = altar.properties.array(default=(0,1))
    support.doc = "the support interval of the prior distribution"


    # protocol obligations
    @altar.export
    def initialize(self, rng):
        """
        Initialize with the given random number generator
        """
        # set up my pdf
        self.pdf = altar.pdf.uniform(rng=rng.rng, support=self.support)
        # all done
        return self

    @altar.export
    def initializeSample(self, theta):
        """
        Fill my portion of {theta} with initial random values from my distribution.
        """
        # grab the portion of the sample that's mine
        θ = self.restrict(theta=theta)
        # fill it with random numbers from my initializer
        self.pdf.matrix(matrix=θ)
        # and return
        return self


    @altar.export
    def verify(self, theta, mask):
        """
        Check whether my portion of the samples in {theta} are consistent with my␣
↪constraints, and
        update {mask}, a vector with zeroes for valid samples and non-zero for␣
↪invalid ones
```

(continues on next page)

(continued from previous page)

```
        """

        ... ...

        # all done; return the rejection map
        return mask

    ... ...

    # private data
    pdf = None # the pdf implementation
```

where the required behaviors specific to the Uniform distribution are defined. Besides behaviors, a component may also include attributes such as

- Properties, configurable parameters in terms of basic Python data type, such as an integer [defined as `altar.properties.int()`;
- Sub-Components, configurable attributes in terms of components;
- Non-configurable attributes, regular Python objects such as static properties or objects determined at runtime, e.g., the *pdf* function in `Distribution`.

Components are building blocks of AlTar. For example, Distribution can be used as the prior distribution in a Bayesian model,

```
class Bayesian(altar.component, family="altar.models.bayesian", implements=model):
"""
The base class of AlTar models that are compatible with Bayesian explorations
"""

    prior = altar.distributions.distribution()
    prior.doc = "the prior distribution"

    ... ...
```

Here, prior is a configurable component, for which users can specify at runtime by `model.prior=uniform` or any other distributions implementing the Distribution-protocol. Since the protocol defines the uniform distribution as its default implementation, if none is specified at runtime, the uniform distribution is used by default.

Note also that *components* are abstract methods and can be only be instantiated by an AlTar application instance. If you create a component instance in a Python shell, it will not behave as a regular Python class.

4.3.2 Pyre Config Format (`.pfg`)

Configurations of properties and components can be passed to the program as command line arguments, or more conveniently, by a configuration file. Three types of configuration files are supported by pyre/AlTar: `.pml` (XML-style), `.cfg` (an INI-style format used in AlTar 1.1), and `.pfg` (YAML/JSON-style). We recommend `.pfg` for its human-readable data serialization format.

An example of `.pfg` file is provided in *QuickStart*, for the linear model.

Some basic rules of `.pfg` format are

- Whitespace indentation is used for denoting structure, or hierarchy; however, tab characters are not allowed.
- Hierarchy of components can be specified by indentation, or by explicit full path, or by a combination of partial path with indentation. For example, these three configurations are equivalent:

```
; method 1: all by indentation
linear:
    job:
        tasks = 1
        gpus = 0
        chains = 2**10

; method 2: all by full path
linear.job.tasks = 1
linear.job.gpus = 0
linear.job.chains = 2**10

; method 3: combination with partial path and indentation
linear:
    job.tasks = 1
    job.gpus = 0
    job.chains = 2**10
```

- If a component is not specified or listed in the configuration file, a default value/implementation specified in the Python program will be used instead.
- Strings such as paths, names, don't need quotation marks.

4.4 AlTar Framework

4.4.1 Application

See API Reference: `altar.shells.application`

An AlTar application is the *root* component which integrates all the components in AlTar. The main components of an AlTar application are as follows.

class Application (*altar.application*, *family="altar.shells.application"*)

controller = altar.bayesian.controller()

Value
altar.bayesian.Annealer (default)

Description
the MCMC simulation processor, *Annealer* for CATMIP algorithm;

model = altar.models.model()

Value
altar.models.linear(), altar.models.cuda.static(), …

Description
performs the forward modelling and computes the Bayesian probability densities: the prior, the data likelihood and the posterior;

job = altar.simulations.run()

Description
manages the simulation size and job deployment;

rng = altar.simulations.rng()

Description
the random number generator shared by all processes;

monitors = altar.properties.dict(schema=altar.simulations.monitor())

Description
a collection of event handlers, such as reporter, profiler.

An AlTar application executes the simulation by defining a required `main` entry point:

```
@altar.export
def main(self, *args, **kwds):
    """
    The main entry point
    """

    # initialize various components
    self.job.initialize(application=self)
    self.rng.initialize()
    self.controller.initialize(application=self)
    self.model = self.model.initialize(application=self)
    # sample the posterior distribution
    return self.model.posterior(application=self)
```

which initializes different components and invokes the `model.posterior` to perform the MCMC sampling.

An AlTar application is also the engager of the pyre framework, as inherited from `pyre.application` or `pyre.plexus`, which performs

- registering all protocols and components in a database;
- reading/loading configuration files;
- instantiating all components into *regular* Python objects;
- invoking the proper shell (MPI, SLURM) to deploy the job.

4.4.2 Controller/Annealer

See API Reference: *altar.bayesian.Annealer*

A Bayesian controller uses an annealing schedule and MCMC to approximate the posterior distribution of a model. The current Annealer uses exclusively the CATMIP algorithm (more controllers implementing other algorithms will be added to a future release). It includes the following configurable components

- `sampler = altar.bayesian.sampler()`, the MCMC sampler. The default is a `Metropolis` sampler with fixed chain length based on Metropolis-Hastings algorithm. Another sampler implemented is `AdaptiveMetropolis` which targets a fixed acceptance ratio and varies the chain length targeting a fixed effective sample size.
- `scheduler = altar.bayesian.scheduler()`, the generator of the annealing schedule. The default and currently only implemented scheduler is based on the Coefficient of Variance (COV) of the data likelihood densities.
- `dispatcher = altar.simulations.dispatcher(default=Notifier)`, currently only serves the purpose of profiling.
- `archiver = altar.simulations.archiver(default=Recorder)`, the archiver of simulation state. The default recorder saves the simulation state to HDF5 files.

and another component determined at runtime from the job configurations,

- `worker` (AnnealingMethod): as AlTar simulations can be performed with either single thread or multiple threads, on CPUs or GPUs, the Controller uses different workers where various deployment-dependence procedures are differentiated. For example, the multiple thread processor, `MPIAnnealing` needs to include additional procedures to collect/distribute samples for all threads.

The Annealer's behaviors include

- `deduceAnnealingMethod` which uses the job configuration to determine the worker.
- `posterior` which defines the MCMC procedures with an annealing schedule.

4.4.2.1 Worker/AnnealingMethod

See API Reference: *altar.bayesian.AnnealingMethod*

A worker(AnnealingMethod) defines each procedure of annealing which may be platform dependent. There are three types of workers:

- `SequentialAnnealing`, a single thread CPU processor
- `CUDAAnnealing`, a single thread GPU processor
- `MPIAnnealing`, a multiple thread processor which uses SequentialAnnealing(CPU) or CUDAAnnealing(GPU) as slave workers.

Each worker also keeps a set of simulation state data, such as `beta` (the inverse temperature), `theta` (the random samples), `prior/data/posterior` (the Bayesian densities), in an object `CoolingStep`.

Worker is not directly user configurable; it is determined automatically by the job configuration.

4.4.2.2 Sampler

See API Reference: *altar.bayesian.Sampler*

Starting with N_s number of chains/samples (processed in parallel), a sampler performs MC updates of the samples pursuant to a given distribution for several steps (length of the chain). For finite β, the target distribution is the transient distribution $P_m(\boldsymbol{\theta}|\mathbf{d}) = P(\boldsymbol{\theta})P(\mathbf{d}|\boldsymbol{\theta})^{\beta_m}$, while the sampling serves as a burn-in process. When $\beta = 1$ is reached, the sampler samples the posterior distribution $P(\boldsymbol{\theta}|\mathbf{d})$.

The default sampler is a CPU `Metroplis` sampler. To use other samplers, e.g., for CUDA simulations, users need to specify it in the controller block of the configuration file

```
ApplicationInstance:
    controller:
        sampler = altar.cuda.bayesian.metropolis
        sampler:
            ; sampler attributes
            ... ...
```

Metropolis

See API Reference: `altar.bayesian.Metropolis`

Algorithm

- New samples are proposed with a Gaussian kernel,

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \alpha\boldsymbol{\delta}$$
$$\boldsymbol{\delta} \sim N(0, \boldsymbol{\Sigma})$$

  where $\boldsymbol{\Sigma}$ is the (weighted) covariance matrix of starting samples (from the previous β step), and α is a scaling factor adjusting the jump distance. In CATMIP, α is adjusted by the acceptance rate (from the previous β-step):

$$\alpha = acceptanceWeight * acceptanceRate + rejectionWeight$$

  Since the acceptance rate varies between 0 and 1, `rejectionWeight` and `rejectionWeight+acceptanceWeight` offer as the lower and upper limits of the scaling factor α.

- Decide whether to accept the proposed samples with the Metropolis–Hastings algorithm.
- Repeat the MC updates for a fixed N_c-number of times.

Configurable attributes

scaling
: float, the initial value of α, default=0.1

acceptanceWeight, rejectionWeight
: float, ratios to adjust the value of α during the run, defaults=8/9, 1/9

steps
: integer, the MC update steps in each β-step (the length of each chain), configured by `job.steps`.

Configuration examples

```
ApplicationInstanceName:
    controller:
        sampler = altar.bayesian.metropolis
        sampler:
            scaling = 0.2
            acceptanceWeight = 0.1
            rejectionWeight = 0.9
    ; the length of chains
    job.steps = 2**12
```

Metropolis (CUDA Version)

See API Reference: `altar.cuda.bayesian.cudaMetropolis`

The CUDA version follows the same procedure as above, but includes more control on the scaling factor.

Configurable attributes

scaling
: float; the initial value of α; default=0.1

acceptanceWeight, rejectionWeight
float; ratios to adjust the value of α during the run, defaults=8/9, 1/9

useFixedScaling
bool; if `True`, the initial `scaling` will be used for all β-steps; default= `False`

scalingMin, scalingMax
float; the min/max values of the scaling factor; default=0.01, 1

steps
integer, the MC update steps in each β-step (the length of each chain), configured by `job.steps`.

Configuration examples

```
ApplicationInstanceName:
    controller:
        sampler = altar.cuda.bayesian.metropolis
        sampler:
            scaling = 0.2
            acceptanceWeight = 0.99
            rejectionWeight = 0.01
            useFixedScaling = False
            scalingMin = 0.1
            scalingMax = 0.5
    ; the length of chains
    job.steps = 2**12
```

In this example, `scaling` is set to 0.2 in the beginning. During the run, `scaling = acceptanceWeight*R + rejectionWeight`, where R is the acceptance rate from the previous β-step, or `scaling` $\in [0.01, 1]$. `scalingMin` and `scalingMax` further adjust the range as [0.1, 0.5].

AdaptiveMetropolis

See API Reference: *`altar.cuda.bayesian.cudaAdaptiveMetropolis`* (for CUDA only)

Algorithm

In an AdaptiveMetropolis sampler, there are a few variations from the Metropolis sampler,

1. After a certain number of MC updates, `corr_check_steps`, the correlation between the current samples and the initial samples are computed. If the correlation is smaller than a threshold value, `target_correlation`, or the samples become sufficiently de-correlated, we can stop MC updates for the current β-step. A `max_mc_steps` sets the maximum number of MC updates if the correlation threshold value cannot be achieved.
2. The scaling factor α targets an optimal acceptance rate, `target_acceptance_rate`, with a feedback function

$$\alpha_{j+1} = \alpha_j \exp[-gain * (acceptanceRate_j - target\_acceptance\_rate)]$$

where j labels the β-step. The initial value is set as

$$\alpha_0 = scaling/\sqrt{parameters}$$

It is shown that an optimal value for α is $2.38/\sqrt{d}$, where d is the dimension of parameter space, or `parameters`.

3. (*New in 2.0.2*) Sometimes, it is useful to use more MC steps when β is small, or *vice versa*. We introduce a new `max_mc_steps_stage2` to be used for the maximum MC steps when $\beta >$ `beta_step2`.

Configurable Attributes

scaling
float, default=2.38, initial scaling factor for Gaussian proposal, to be normalized by the square root of the number of parameters

parameters
integer, default=1, the total number of parameters in simulation: since the controller is initialized before the model, users need to manually provide this information to the sampler (we will try to eliminate this requirement in the next update).

scaling_min, scaling_max
float, default=(0.01, 1), the minimum and maximum values allowed for the scaling factor

target_acceptance_rate
float, default=0.234, the targeted acceptance rate

gain
float, default=None (determined by `target_acceptance_rate`), the feedback gain constant

max_mc_steps
integer, default=100000, the maximum Monte-Carlo steps for one beta step

min_mc_steps
integer, default=1000, the minimum Monte-Carlo steps for one beta step

beta_stage2
float, default=0.1, the start beta value to use another maximum MC steps

max_mc_steps_stage2
integer, default=None (to be set the same as `max_mc_steps`), the maximum Monte-Carlo steps for one beta step when `beta` > `beta_stage2`

corr_check_steps
integer, default=1000, the Monte-Carlo steps to compute and check the correlation

target_correlation
float, default=0.6, the threshold of correlation to stop the chain updates

Configuration examples

```
ApplicationInstanceName:
    controller:
        sampler = altar.cuda.bayesian.adaptivemetropolis  ; only for CUDA
        sampler:
            scaling = 2.38
            parameters = 399 ; number of parameters in model
            min_mc_steps = 3000
            max_mc_steps = 10000
            corr_check_steps = 1000
            target_correlation = 0.6
            beta_stage2 = 0.1
            max_mc_steps_stage2 = 5000
```

In this example, the initial scaling factor is set to `scaling` / sqrt(`parameters`) or $2.38/\sqrt{399} = 0.12$ (you can also set `scaling` directly to 0.12, and use the default value 1 for `parameters`). After `min_mc_steps` (3,000), the correlation between current samples and initial samples are computed every `corr_check_steps` (1,000). If the correlation is less than `target_correlation` (0.6), the MC update stops and the simulation proceeds to next β step. If the `target_correlation` cannot be achieved by a certain number of steps, `max_mc_steps` (10,000) for $\beta <=$ `beta_stage2` (0.1) while `max_mc_steps_stage2` (5,000) for $\beta >$ `beta_stage2` (0.1), the MC update is forced to stop and the simulation proceeds to next β step.

4.4.2.3 Scheduler

A scheduler regulates how β increases between different β-steps. The default (and currently the only option) is *COV Scheduler*.

COV Scheduler

See API Reference: `altar.bayesian.COV`

Algorithm

The COV (Coefficient of Variation) scheduler targets the effective sample size from resampling between different transient distributions,

$$P_m(\boldsymbol{\theta}|\mathbf{d}) = P(\boldsymbol{\theta})P(\mathbf{d}|\boldsymbol{\theta})^{\beta_m},$$

At the m-stage, samples $\theta_{m,k}$ are generated with the target equilibrium distribution $P_m(\boldsymbol{\theta}|\mathbf{d})$, where $k = 1, 2, \ldots, N_s$ and N_s is the total number of samples (chains). At the $m+1$-stage, the sampling targets $P_{m+1}(\boldsymbol{\theta}|\mathbf{d})$ as the new equilibrium distribution. To sample a distribution with samples generated from another distribution is called as importance sampling, with the importance weight

$$\begin{aligned} w(\theta_{m,k}) &= \frac{P_{m+1}(\theta_{m,k}|\mathbf{d})}{P_m(\boldsymbol{\theta}|\mathbf{d})} \\ &= P(\mathbf{d}|\theta_{m,k})^{\beta_{m+1}-\beta_m} \end{aligned}$$

while the effective sample size (ESS) from the resampling is associated with the COV of $w(\theta_{m,k})$,

$$\begin{aligned} ESS &= \frac{N_s}{1+COV(w)}, \\ COV(w) &= \frac{\bar{w}}{\sigma} \\ \bar{w} &= \frac{1}{N_s}\sum_k w(\theta_{m,k}) \\ \sigma &= \frac{1}{N_s-1}\sum_k [w(\theta_{m,k}) - \bar{w}]^2. \end{aligned}$$

In COV Scheduler, we choose a β_{m+1} so that COV is of order unity, e.g., $COV = 1$, or $ESS = N_s/2$.

Configurable Attributes

target
: float, default=1.0, the target value for COV

solver
: `altar.bayesian.solver()`, values= `grid` (default), `brent`; the δβ solver based on the grid search algorithm (grid) or the Brent minimizer (brent)

check_positive_definiteness
: bool, values = `True` (default), `False`; whether to check the positive definiteness of Σ matrix and condition it accordingly

min_eigenvalue_ratio
: float, default=0.001; if checking the positive definiteness, the minimal eigenvalue to be set as a ratio of the maximum eigenvalue

Configuration examples

```
ApplicationInstanceName:
    controller:
        scheduler: ; default is COV
            target = 2.0
            solver = brent
            check_positive_definiteness = False
```

4.4.2.4 Archiver (Output)

See API Reference: `altar.simulations.Archiver`

The Archiver saves progress information. The default is an H5Recorder which saves the Bayesian statistical data to HDF5 files.

H5Recorder

See API Reference: `altar.bayesian.H5Recorder`

H5Recorder saves the random samples and their Bayesian probability densities from each β-step to an HDF5 file, `output_dir/step_nnn.h5`.

```
+---------- step_nnn.h5 ------
├── Annealer ; annealing data
|   ├── beta ; the beta value
|   └── covariance ; the covariance matrix for Gaussian proposal
├── Bayesian ; the Bayesian probabilities
|   ├── prior ; (log) prior probability for all samples, vector (samples)
|   ├── likelihood ; (log) data likelihood for all samples
|   └── posterior ; (log) posterior probability for all samples
└── ParameterSets ;  samples
    └── theta ; samples of model parameters, 2d array (samples, parameters)
```

If you use a list of parameter sets, e.g., in Static Inversion, {`strike_slip`, `dip_slip`, `insar_ramp`}, their names will be used for their data sets,

```
└── ParameterSets ;
    └── strike_slip ; samples of strike slips, 2d array (samples, number of patches)
    └── dip_slip ; samples of dip slips, 2d array (samples, number of patches)
    └── insar_ramp ; samples of insar ramp parameters to be fitted, 2d array (samples,
↪ number of ramp parameters)
```

H5Recorder also records the MCMC statistics from each β-step to a file `output_dir/BetaStatistics.txt`. An example for the linear model is as follows

```
iteration, beta, scaling, (accepted, invalid, rejected)
0, 0, 0.3, (0, 0, 0)
1, 0.00015000000000000001, 0.5925347222222223, (2773, 0, 2347)
2, 0.00037996549999999997, 0.3166666666666665, (1184, 0, 3936)
3, 0.00069984391104, 0.5828125, (2717, 0, 2403)
4, 0.0011195499765973632, 0.3454861111111111, (1350, 0, 3770)
5, 0.0016789230286104687, 0.5607638888888888, (2590, 0, 2530)
6, 0.0025175127332664362, 0.3590277777777778, (1428, 0, 3692)
7, 0.0037843154920951883, 0.5447916666666667, (2498, 0, 2622)
8, 0.005577503724209416, 0.3751736111111111, (1521, 0, 3599)
```

(continues on next page)

(continued from previous page)

```
9, 0.008163002214526472, 0.5303819444444444, (2415, 0, 2705)
10, 0.012229533905446913, 0.37986111111111115, (1548, 0, 3572)
11, 0.017958602608795324, 0.5154513888888889, (2329, 0, 2791)
12, 0.026207750346881442, 0.38125, (1556, 0, 3564)
13, 0.03808801579264949, 0.5, (2240, 0, 2880)
14, 0.05444051952417445, 0.40243055555555557, (1678, 0, 3442)
15, 0.07760672679583216, 0.4793402777777778, (2121, 0, 2999)
16, 0.11173527790438637, 0.42065972222222225, (1783, 0, 3337)
17, 0.16503116123012318, 0.4588541666666667, (2003, 0, 3117)
18, 0.24518816975203137, 0.41944444444444445, (1776, 0, 3344)
19, 0.3470877668355071, 0.46753472222222225, (2053, 0, 3067)
20, 0.5037867027949854, 0.40625, (1700, 0, 3420)
21, 0.7176546338903467, 0.47465277777777776, (2094, 0, 3026)
22, 1.0, 0.41770833333333335, (1766, 0, 3354)
```

which shows how β evolves from β-step iterations, as well the scaling parameter :math:`alpha`, and the MC acceptance (accepted/rejected = proposals accepted/rejected by Metropolis-Hastings algorithm, invalid = proposals rejected due to being out of range for ranged priors).

Configurable Attributes

output_dir
: path(string), default="results"; the directory to save results

output_freq
: integer, default=1; the frequency to write step data to files, e.g., if you only want to save data for every 3 β-steps, you may choose `output_freq=3`. The final β-step, i.e., $\beta = 1$ will be always saved as `step_final.h5`.

Configuration examples

```
ApplicationInstanceName:
    controller:
        archiver: ; default is H5Recorder
            output_dir = results/static_chain_1024
            output_freq = 3
```

4.4.3 Job

See API Reference: `altar.simulations.Job`

The `job` component in an AlTar application controls the size of the simulation as well as its deployment to different platforms.

4.4.3.1 Configurable Attributes

```
name = altar.properties.str(default="sample")
name.doc = "the name of the job; used as a stem for making filenames, etc."

mode = altar.properties.str()
mode = "the programming model"

hosts = altar.properties.int(default=1)
hosts.doc = "the number of hosts to run on"
```

(continues on next page)

(continued from previous page)

```
tasks = altar.properties.int(default=1)
tasks.doc = "the number of tasks per host"

gpus = altar.properties.int(default=0)
gpus.doc = "the number of gpus per task"

gpuprecision = altar.properties.str(default="float64")
gpuprecision.doc = "the precision of gpu computations"
gpuprecision.validators = altar.constraints.isMember("float64", "float32")

gpuids = altar.properties.list(schema=altar.properties.int())
gpuids.default = None
gpuids.doc = "the list of gpu ids for parallel jobs"

chains = altar.properties.int(default=1)
chains.doc = "the number of chains per worker"

steps = altar.properties.int(default=20)
steps.doc = 'the length of each Markov chain'

tolerance = altar.properties.float(default=1.0e-3)
tolerance.doc = "convergence tolerance for β->1.0"
```

4.4.3.2 Simulation Size

For a single thread simulation, the job size is determined by the number of `chains` (per thread), which are processed as a batch. More chains offer better phase space exploration. But the number of chains may be limited by the computer memory size (CPU or GPU). Since the memory usage also depends on the number of parameters and the type of model, users are encouraged to try some chain sizes at first (stop the simulation after one or two beta steps) to determine an optimal setting of `chains` for their computer system.

Large-scale simulations can be distributed to multiple threads. Distributing the total number of chains from one thread (sequentially) to multiple threads (in parallel) may also reduce the computation time (wall time). The number of threads are controlled by two parameters `hosts` - the number of hosts (nodes), and `tasks` - the number of threads per host. The total number of chains is therefore `hosts*tasks*chains`.

The multi-threading in AlTar is achieved by MPI. An AlTar application is capable of deploying itself automatically to multiple MPI threads in one or more computers/nodes so that users don't need to run `mpirun`, `qsub`, or `sbatch` explicitly.

The Metropolis sampler uses `job.steps` to control the number of MC updates in each β-step. (*It might be better to move this setting directly to sampler*). This procedure serves as a burn-in to equilibrate samples from one distribution with β_m to another with β_{m+1}. Larger `steps` allow more equilibration but are not required in CATMIP: the β-increment (or the total number of β-steps) will be adjusted.

4.4.3.3 Single Thread Configuration

The default job setting is to run AlTar with one CPU thread; you don't need to provide any settings in the configuration file. However, if you prefer to keep `hosts` and `tasks` entries to be modified later, set them to be 1 explicitly.

```
ApplicationInstanceName:
    job:
        hosts = 1 ; number of hosts/nodes
        tasks = 1 ; number of threads per host
        chains = 2**12 ; number of Markov chains per thread.
        steps = 2**10 ; length of the Markov chain
```

4.4.3.4 Multiple Threads on One Computer

To run a multi-threaded simulation on a single computer, you need to adjust the `tasks` setting, as well as to specify a `mpi.shells.mpirun` shell instead,

From the configuration file

```
ApplicationInstanceName:
    job:
        tasks = 8

    shell = mpi.shells.mpirun

; additional configurations for mpi shell, if needed
mpi.shells.mpirun # altar.plexus.shell:
    extra = -mca btl self,tcp
```

or from the command line

```
$ AlTarApp --job.tasks=8 --shell=mpi.shells.mpirun
```

If you use an MPI package not installed under the system directory, you may need to provide its configuration to AlTar/pyre framework. For example, for OpenMPI installed with Anaconda, the following additional configurations are required

```
mpi.shells.mpirun:
  ; mpi implementation
  mpi = openmpi#mpi_conda

; mpi configuration
pyre.externals.mpi.openmpi # mpi_conda:
  version = 3.0
  launcher = mpirun
  prefix = /opt/anaconda3
  bindir = {mpi_conda.prefix}/bin
  incdir = {mpi_conda.prefix}/include
  libdir = {mpi_conda.prefix}/lib
```

Since the MPI package information is common for running all jobs on a computer, you may choose to save the above configuration to an `mpi.pfg` file, either under `${HOME}/.pyre` directory (searchable by all AlTar/pyre applications), or under the work directory with the AlTar application configurable file, e.g., `linear.pfg`.

Decide the max number of tasks/threads on a computer from the number of physical cores, not from the total (virtual) threads. Hyperthreading may increase the number of available threads, but it might not increase performance for compute-intensive models, e.g., with matrix multiplications.

4.4.3.5 Multiple Threads Across Several Computers

If a batch scheduler is not required, you may still use the `mpi.shells.mpirun` shell with the additional `hostfile` configuration. A hostfile is a simple text file with hosts/nodes specified, e.g., `my_hostfile`

```
# This is an example hostfile.  Comments begin with #
192.168.1.101 slots=16
192.168.1.102 slots=16
192.168.1.103 slots=16
```

To use the hostfile with `mpi.shells.mpirun` shell, e.g., with 4 hosts and 8 threads per host,

```
ApplicationInstanceName:
    job:
        hosts = 4
        tasks = 8
    shell = mpi.shells.mpirun
    shell:
        hostfile = my_hostfile
```

Or from the commmand line,

```
$ AlTarApp --job.hosts=4 --job.tasks=8 --shell=mpi.shells.mpirun --shell.hostfile=my_
↪hostfile
```

If a batch scheduler is available, e.g., Slurm Workload Manager, use an `mpi.shells.slurm` shell instead. An example configuration is as follows

```
ApplicationInstanceName:
    job:
        hosts = 4
        tasks = 8
    shell = mpi.shells.slurm

; for parallel runs
mpi.shells.slurm :
    submit = True ; if True, submit the job for execution, if not, a slurm script is␣
↪generated
    queue = gpu ; the name of the queue
```

Or from the command line

```
$ AlTarApp --job.hosts=4 --job.tasks=8 --shell=mpi.shells.slurm --shell.queue=gpu
```

If your Slurm Manager requires additional configurations, you can use `submit=False`, modify the generated Slurm script, and use `sbatch` to submit the job.

4.4.3.6 GPU Configurations

The GPU support in AlTar is implemented with CUDA, and therefore is limited to NVIDIA graphical cards.

To choose GPU or CPU If you plan to run AlTar simulations on GPU, you may enable it by

```
ApplicationInstanceName:
    job.gpus = 1 ; number of GPU per task,  0 = use gpu
```

AlTar also checks the availability of `cuda` modules (software) and compatible CUDA devices (hardware). If either is unavailable, AlTar enforces `job.gpus = 0`, or using CPU instead.

Currently, the `cuda` modules are not fully integrated with cpu modules. You may need to check whether the model has a cuda implementation, and also select explicitly some cuda components, for example, for the Static Inversion,

```
slipmodel: ; the Application Instance Name

    model = altar.models.seismic.cuda.static
    ; define parametersets
    psets:
        strikeslip = altar.cuda.models.parameterset
        dipslip = altar.cuda.models.parameterset

        strikeslip:
            count = {slipmodel.model.patches}
            prior = altar.cuda.distributions.gaussian
            prior.mean = 0
            prior.sigma = 0.5
        ... ...

    controller:
        sampler = altar.cuda.bayesian.metropolis

    ... ...
```

In the next release, we will try to merge cuda modules with cpu modules so that a single ``jobs.gpus`` flag can switch between CPU and GPU computations.

To use multiple GPUs GPUs (device) rely on CPU (host) for job dispatching and data copies from/to GPU memories. AlTar runs on one GPU per (CPU) thread, therefore, the number of GPUs used for simulation is tuned by the number of threads per host, `job.tasks`, and/or the number of hosts, `job.hosts`. `job.gpus` is always 1 for GPU simulations.

To deploy the simulation to 8 GPUs in one computer/node, the configuration is

```
ApplicationInstanceName:
    job.hosts = 1 ; number of hosts
    job.tasks = 8 ; number of threads per host
    job.gpus = 1 ; number of gpus per thread
```

To deploy the simulation to 4 nodes with 8 GPUs per node, the configuration is

```
ApplicationInstanceName:
    job.hosts = 4 ; number of hosts
    job.tasks = 8 ; number of threads per host
    job.gpus = 1 ; number of gpus per thread
```

For a computer/node with multiple GPUs, the job is distributed sequentially to the first available GPUs (`gpuids` = 0, 1,2, 3 …). If you plan to assign the job to specific GPUs, you can use `job.gpuids` to specify them,

```
ApplicationInstanceName:
    job.hosts = 4 ; number of hosts
    job.tasks = 2 ; number of threads per host
    job.gpus = 1 ; number of gpus per thread
    job.gpuids = [2, 3]
```

Another method is to use the environmental variable `CUDA_VISIBLE_DEVICES`. For example,

```
# bash
$ export CUDA_VISIBLE_DEVICES=2,3
# csh/tcsh
$ setenv CUDA_VISIBLE_DEVICES 2,3
```

which makes only GPU2 and GPU3 visible for applications, appearing as `gpuids=[0, 1]`. With this method, you don't need to set `gpuids`.

To select the precision AlTar supports both single and double precision GPU computations (the CPU computation is always double precision). However, most NVIDIA gaming cards only have single precision processors. In our tests on Tesla cards, single precision computations run twice faster than double precisions. If you want to choose between single and double precisons, you may use the `job.gpuprecison` flag, such as

```
ApplicationInstanceName:
    job:
        hosts = 1 ; number of hosts
        tasks = 2 ; number of threads per host
        gpus = 1 ; number of gpus per thread
        gpuprecision = float32 ; double(float64) or single(float32) precision for gpu␣
↪computations
```

4.4.4 Model

The Model component in AlTar Framework defines the forward problem, and computes the data likelihood accordingly. Each model needs an implementation for a given inverse problem. See *Models* for details guides on implemented models.

Users may also develop their own models, following the guide in *Bayesian Model*.

4.4.5 (Prior) Distributions

There are several probability distributions defined in AlTar, serving as prior distributions.

For a prior distribution *altar.distributions.distribution*, the following methods are required

```
# model support
@altar.provides
def initializeSample(self, theta):
    """
    Fill my portion of {theta} with initial random values from my distribution.
    """

@altar.provides
def priorLikelihood(self, theta, prior):
    """
    Fill my portion of {prior} with the likelihoods of the samples in {theta}
    """
```

(continues on next page)

(continued from previous page)

```
@altar.provides
def verify(self, theta, mask):
    """
    Check whether my portion of the samples in {theta} are consistent with my␣
↪constraints, and
    update {mask}, a vector with zeroes for valid samples and non-zero for invalid␣
↪ones
    """
```

In AlTar, the logarithmic values of the Bayesian probability densities are processed. Therefore, in addition to `priorLikelihood`, a `verify` method is needed for certain ranged distributions to check whether the proposed samples fall outside the range.

For a parameter to be sampled, the distributions for generating the initial samples and computing the prior probability densities during the simulation may be different. For example, in static inversion of earthquakes, you may want to use a Moment Scale distribution to generate (strike or dip) slips whose sum is consistent with a given moment magnitude scale, while during the simulation, while a simple uniform distribution for `priorLikelihood` and `verify`.

Please note that AlTar processes samples in batch, where θ is a 2d array `shape=(samples, parameters)`. Also, in a specific Model, there may be different parameter sets which observe different prior distributions. Therefore, the methods in a prior distribution are responsible for its own portion of parameters (selected columns of θ) and for a batched samples (rows of θ).

4.4.5.1 Uniform

The probability density function (PDF) for a uniform distribution is

$$\begin{aligned} f(x; a, b) &= \frac{1}{b-a}, \text{for } x \in [a, b] \\ &= 0, \text{otherwise} \end{aligned}$$

where $[a, b]$ is the support or range.

Example

```
prior = uniform
prior:
    support = (0, 1)
```

4.4.5.2 Gaussian

The PDF for the Gaussian (normal) distribution is defined as

$$f(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ and σ^2 are mean (center) and variance, respectively.

Example

```
prior = gaussian
prior:
    mean = 0
    sigma = 2
```

4.4.5.3 Truncated Gaussian

The truncated Gaussian distribution is derived from the Gaussian distribution but is only finite within the support range.

Example

(Currently only implemented in CUDA).

```
prior = altar.cuda.distributions.tgaussian
prior:
    support = (-1, 1)
    mean = 0
    sigma = 2
```

4.4.5.4 Preset

The `Preset` distribution is used to load initial samples from pre-calculated ones. Therefore, it only serves as a preparation (`prep`) distribution. The currently support input format is HDF5, as the default output for AlTar simulation results.

Example

(Currently only implemented in CUDA).

For example, in the earthquake (seismic) inversion, we have samples of `strikeslip` generated from the static inversion and would like to load them for the kinematic inversion,

```
prep = altar.cuda.distributions.preset ; load preset samples
prep.input_file = theta_cascaded.h5 ; file name
prep.dataset = ParameterSets/strikeslip ; dataset name in h5
```

4.4.6 Other Distributions

More prior distributions can be easily added. You may follow the existing distributions as examples. Or please write to us so that we add them to the package.

4.5 Models

4.5.1 Static Slip Inversion

4.5.1.1 Static Source model

The finite fault earthquake source models infer the spatial distribution and temporal evolution of coseismic slips at fault plane(s) from the observed surface or subsurface displacements.

In the static source model, the spatial distributions of coseismic slips are determined. We model the fault plane(s) as a set of patches; each patch treated as a point source with two orthogonal displacement components, slips along the strike and dip directions. Each slip on the fault plane can be translated into surface deformation, e.g., by the Okada model, which is derived from a Green's function solution to the elastic half space problem. The observed surface deformation at a given location is the linear combination due to (strike/dip) slips of all patches.

Note: For the static inversion, the patches can be of any shape or area as long as each patch can be treated a single point source. Note that the kinematic inversion currently only supports a rectangle fault divided into $n_{dd} \times n_{as}$ square patches. If you plan for a joint static-kinematic inversion, you need to run the static inversion on the square patches as well.

Therefore, the forward model can be expressed as a linear model

$$\mathbf{d}^{pred} = \mathcal{G}\boldsymbol{\theta}.$$

where $\boldsymbol{\theta}$ (also denoted as $\mathbf{m}$ in geophysics literatures) is a vector with $N_{param} = 2N_{patch}$ components, representing the slips along the strike and dip directions for N_{patch} patches; $\mathbf{d}$ is a vector of observed surface deformations (may include vertical and east, north horizontal components) at different locations, with N_{obs} observations; and $\mathcal{G}$ is a $2N_{patch} \times N_{obs}$ matrix, pre-calculated Green's functions connecting a slip source to a deformation component at an observation location.

A generalized forward model could also include other linear parameters, for example, the InSAR ramp parameters (a, b, c), used to fit the spurious ramp-like displacement fields $a + bx + cy$ from InSAR interferograms, where x and y are the locations of the data in local Cartesian coordinates.

4.5.1.2 Input

To run the static inversion, in addition to an AlTar configuration file (see next section), you are required to prepare three input files,

data.txt
: the observed data with N_{obs} observations, a vector in one row or one column.

cd.txt
: covariance of data representing measurement errors/uncertainties, prepared in a $N_{obs} \times N_{obs}$ matrix (could have off-diagonal terms for correlated errors). A constant could be used (with *cd_std* option) if all data are uncorrelated and have the same variance. Epistemic uncertainties such as elastic heterogeneity and fault geometry uncertainties may also be included, see Static Inversion Cp for more details.

green.txt
: the pre-calculated Green's functions, prepared in a $N_{obs} \times N_{param}$ matrix, with N_{param} as the leading dimension, i.e., data arranged in row-major as

```
G[Obs1][Param1] G[Obs1][Param2] ... G[Obs1][ParamNparam]
G[Obs2][Param2] G[Obs2][Param2] ... G[Obs2][ParamNparam]
... ...
G[ObsNobs][Param1] G[ObsNobs][Param2] ... G[ObsNobs][ParamNparam]
```

You may use any other names for these files but need to specify them in the configuration file. These files are arranged under the same directory which can be specified by `case` setting in the configuration file.

In addition to plain text inputs with suffix `.txt`, AlTar 2 also accepts binary data, including

- Raw binary format with suffix `.bin` or `.dat`. The precision should be consistent with the numerical precision specified by `job.gpuprecision`. Its size should match the size of the corresponding quantity. For example, the Green's function has in total $N_{obs} \times N_{param}$ elements. They will be reshaped by the program.
- HDF5 files with suffix `.h5`. HDF5 files are preferred because they include the data shape and precision in metadata. AlTar2 performs the reshaping and precision conversions if necessary.

AlTar2 recognizes the data format automatically by the file suffixes.

There are some software packages which can pre-calculate the Green's functions and/or prepare the input files for AlTar, e.g., CSI. Please refer to its manual and tutorials for more details.

4.5.1.3 Configurations

A configuration file for the static inversion appears as

```
; application instance name
slipmodel:

    ; model to be sampled
    model = altar.models.seismic.cuda.static
    model:

        ; the name of the test case, also as the directory for input files
        case = 9patch

        ; number of patches
        patches = 9

        ; green's function (observations, parameters)
        green = static.gf.h5

        ; data observations
        dataobs = altar.cuda.data.datal2
        dataobs:
            observations = 108
            data_file = static.data.h5
            cd_file = static.Cd.h5
            ; or use a constant cd
            ; cd_std = 1e-4

        ; list of parameter sets
        ; the order should be consistent with the green's function
        psets_list = [strikeslip, dipslip, ramp]

        ; define parameter sets
        psets:
            strikeslip = altar.cuda.models.parameterset
            dipslip = altar.cuda.models.parameterset
            ramp = altar.cuda.models.parameterset

            strikeslip:
                count = {slipmodel.model.patches}
                prior = altar.cuda.distributions.gaussian
                prior.mean = 0
                prior.sigma = 0.5

            dipslip:
                count = {slipmodel.model.patches}
                prep = altar.models.seismic.cuda.moment
                prep:
                    Mw_mean = 7.3
                    Mw_sigma = 0.2
                    Mu = [30] ; in GPa
                    area = [400] ; patch area in km^2
                prior = altar.cuda.distributions.uniform
                prior.support = (-0.5, 20)

            ramp:
                count = 3
```

(continues on next page)

(continued from previous page)

```
                prior = altar.cuda.distribution.uniform
                prior.support = (-1, 1)

    controller:
        sampler = altar.cuda.bayesian.metropolis
        archiver:
            output_dir = results/static ; output directory
            output_freq = 3 ; output frequency in beta steps


    ; run configuration
    job:
        tasks = 1 ; number of tasks per host
        gpus = 1  ; number of gpus per task
        gpuprecision = float32 ; double(float64) or single(float32) precision for gpu␣
↪computations
        ;gpuids = [0] ; a list gpu device ids for tasks on each host, default␣
↪range(job.gpus)
        chains = 2**10 ; number of chains per task
        steps = 1000 ; MC burn-in steps for each beta step
```

We explain each section below.

Application Instance Name

We use a shell command `slipmodel` for all seismic slip models, including static and kinematic inversions, which uses `slipmodel` as the application instance name. Therefore, please use `slipmodel` as the root in the configuration file. By the pyre convention, the shell command searches and loads configurations from the file `slipmodel.pfg` in current path. If you name your configuration file as `slipmodel.pfg`, you may simply run

```
$ slipmodel
```

to invoke simulations for any slip models. If you want to name the configuration file as something else, e.g., `static.pfg`, `static_mpi.pfg`, or `Nepal_static.pfg`, you may specify the configuration file from the command line by the `--config` option,

```
$ slipmodel  --config=static.pfg
```

Model

For static inversion, you need to specify `model = altar.models.seismic.cuda.static` (or the CPU version, `model=altar.models.seismic.static`).

Model Attributes

case
: the directory where all input files are located;

patches
: the number of patches, or point sources;

green
: the file name for the Green's functions, as prepared from the instructions above;

dataobs
: a component to process the data observations and calculate the data likelihood with L2 norm, with details provided in *Data Observations*;

psets_lists
: a list of parameter sets, the order will be used for many purposes, e.g., enforcing the order of parameters in θ;

psets
: components to describe the parameter sets, with details provided in *Parameter Sets*.

Data Observations

The observed data are handled by a component named `dataobs`. We use exclusively the L2 norm for the likelihood computation because it accommodates the uncertainty quantification from the data covariance matrix (Cd). Therefore,

```
dataobs = altar.cuda.data.data12
dataobs:
    observations = 108
    data_file = static.data.h5
    cd_file = static.Cd.h5
    ; cd_std = 1e-2
```

For the data observations with the data covariance matrix `data12`, the following attributes are required

observations
: the number of data observations

data_file
: the name of the file containing the data observations, a vector with `observations` elements

cd_file
: the name of the file containing the data covariance, a matrix with `observations x observations` elements

cd_std
: if the data covariance has only constant diagonal elements, you may use this option instead of `cd_file`.

Parameter Sets

A parameter set is a group of parameters which share the same prior distributions and are arranged continuously in $\boldsymbol{\theta}$. In static model, we use the following parameter sets `strikeslip`, `dipslip`, and optionally, `ramp` (you may use any other names for the parameter sets as long as they are intuitive).

The order of the parameter sets in $\boldsymbol{\theta}$ is enforced by the attribute `psets_list`,

```
psets_list = [strikeslip, dipslip, ramp]
```

If the number of patches is 9 and there are 3 InSAR ramp parameters for one set of interferograms. The 21 parameters in $\boldsymbol{\theta}$ are (0-8), strike slips of 9 patches; (9-17), dip slipd of 9 patches; and (18-20), ramp parameters. The order of the parameter sets can be varied, but has to be consistent with that in the Green's function matrix.

For each parameter set, you need to define it as a parameterset, e.g., `strikeslip = altar.cuda.models.parameterset` or (`strikeslip = contiguous` for CPU models). Its attributes include

count
the number of parameters in this set. `{slipmodel.model.patches}` is another way to assign values with pre-defined parameters;

prior
the prior distribution to initialize random samples in the beginning, and compute prior probabilities during the sampling process. See Prior Distributions for choices of priors.

- `prep` (optional), a distribution to initialize samples only. If it is defined, `prep` distribution is used to initialize samples while `prior` distribution is used for computing prior probabilities. If `prep` is not defined, `prior` distribution is used for both.

Example

For dip-slip faults, you may use a `uniform` prior to limit the range of dip slips while using a *Moment Distribution* to initialize samples so that the moment magnitude is consistent with an estimate scale M_w.

```
dipslip = altar.cuda.models.parameterset
dipslip:
    count = {slipmodel.model.patches}
    prep = altar.models.seismic.cuda.moment
    prep:
        Mw_mean = 7.3 ; mean moment magnitude scale
        Mw_sigma = 0.2 ; sd for moment magnitude scale
        Mu = [30] ; in GPa
        area = [400] ; patch area in km^2
    prior = altar.cuda.distributions.uniform
    prior:
        support = (-0.5, 20)
```

Meanwhile, a Gaussian distribution centered at 0 may be used for strike slips

```
strikeslip = altar.cuda.models.parameterset
strikeslip:
    count = {cudastatic.model.patches}
    prior = altar.cuda.distributions.gaussian
    prior:
        mean = 0
        sigma = 0.5
```

Since the same distribution is also used to initialize samples, no `prep` setting is needed.

For InSAR ramps, either a uniform or a Gaussian prior can be used

```
ramp = altar.cuda.models.parameterset
ramp:
    count = 3
    prior = altar.cuda.distributions.uniform
    prior.support = (-0.5, 0.5)
```

If you prefer to use different priors for different patches, for example, to limit the range of slips far away from the hypocenter, you can further divide the strikeslip/dipslip into several parameter sets, such as

```
psets_list = [strikeslip_p1-3, strikeslip_p4-6, strikeslip_p7-9, ...]
```

and define each parameter set by specifying its count and range.

Controller

Please refer to *Controller/Annealer* for Bayesian framework configurations. You may use this section to choose and customize the `sampler` - to process MCMC (e.g, `altar.cuda.bayesian.metropolis` or `altar.cuda.bayesian.adaptivemetropolis`), `archiver` - to record the results (default is `H5Recorder`), and `scheduler` - to control the annealing schedule (default is `COV` scheduler).

Job

Please refer to *Job* on details how to deploy AlTar simulation to different platforms, e.g., single GPU, multiple GPUs on one computer (with mpi), or multiple GPUs distributed in different nodes of a cluster (with mpi and PBS/Slurm scheduler).

4.5.1.4 Output

By default, the static inversion simulation results are stored in HDF5 files, see *H5Recorder* for more details.

4.5.1.5 Moment Distribution

For strike (dip) faults, we may want the generated seismic moment from all strike (dip) slips to be consistent with the estimated moment magnitude scale M_w,

$$M_w = (\log M_0 - 9.1)/1.5$$

M_0 is the scalar seismic moment, defined by

$$M_0 = \mu \sum_{p=1}^{N_{patch}} A_p D_p$$

where μ is the shear modulus of the rocks involved in the earthquake (in pascals), A_p and D_p are the area (in square meters) and the slip (in meters) of a patch.

A `Moment` distribution is designed to generate random slips for this purpose : it generates a random M_w from a Gaussian distribution $M_w \sim N(Mw_{mean}, Mw_{\sigma})$, then distributes the corresponding M_0/μ to different patches with a Dirichlet distribution (i.e., the sum is a constant), and divides the values by the patch area to obtain slips.

Example

The Moment distribution is used as a `prep` distribution to initialize samples in a parameter set,

```
prep = altar.models.seismic.cuda.moment
prep:
    Mw_mean = 7.3 ; mean moment magnitude scale
    Mw_sigma = 0.2 ; sd for moment magnitude scale
    Mu = [30] ; in GPa
    area = [400] ; patch area in km^2
```

Attributes

Mw_mean
the mean value of the moment magnitude scale.

Mw_sigma
the standard deviation of the moment magnitude scale.

Mu
: the shear modulus of the rocks (in GPa), a list with N_{patch} elements. If only one element, the same value will be used for all patches.

area
: the patch area (in square kilometers), also a list with N_{patch} elements. If the areas for all patches are the same, you may input only one value `area = [400]`. If the areas are different, you may input the list as `area = [400, 300, 200, 300, ...]`, or use a file option below.

area_patch_file
: a text file as input for patch areas, a vector with N_{patch} elements, e.g., `area_patch_file = area.txt`.

slip_sign
: `positive` (default) or `negative`. By default, the moment distribution generates all positive slips, i.e., the displacement is along the positive axis along the dip or strike direction. If the slips are along the opposite direction, use `negative` to generate negative slips.

Note also since `Mu` and `area` appear as products for each patch, you may also use, e.g., `Mu=[1]` and input their products to `area` or `area_patch_file`.

4.5.1.6 Forward Model Application

When analyzing the results, you may need to run the forward problem once for an obtained mean, median, or MAP model, or a synthetic model, to produce data predictions and compare with data observations. For the static model, it is straightforward: obtain the mean model (vector), read the Green's function (matrix), and perform a matrix-vector multiplication.

AlTar2 also provides an option to run the forward problem only instead of the full-scale Bayesian simulation, with a slightly modified configuration file. Please follow the steps below.

The first step is to prepare a model file, e.g., `static_mean_model.txt`, including a set of parameters (a vector of N_{param} elements), and copy the file to the input directory - the `case` directory. To obtain the mean model from the simulations, see *Utilities* below.

The second step is to modify the configuration file, e.g, `static.pfg`, by adding the following settings under `model` configuration,

```
slipmodel:

    ; model to be sampled
    model = altar.models.seismic.cuda.static
    model:

        ; settings for running forward problem only
        ; forward theta input
        theta_input = static_mean_model.txt
        ; forward output file
        forward_output = static_forward_prediction.h5

        ... ...
        ; the rest is the same
```

theta_input
: the input model file, a text, binary or HDF5 file.

theta_dataset
: to specify the dataset name in an HDF5 file.

forward_output
the output file including the predicted data in HDF5 format.

Note that the forward problem option runs with one GPU (and one thread), please make adjustment to the `job` configuration if necessary,

```
job:
    tasks = 1 ; number of tasks per host
    gpus = 1  ; number of gpus per task
    gpuprecision = float32 ; double(float64) or single(float32) precision for gpu␣
↪computations
    ;gpuids = [0] ; a list gpu device ids for tasks on each host, default range(job.
↪gpus)
    ... ...
```

The third step is to run a command

```
$ slipmodel.plexus forward --config=static.pfg
```

Check the generated `static_forward_prediction.h5` file for the predicted data from the input model.

Please check the examples directory from the source code for a complete sample.

`slipmodel.plexus` is a new AlTar application which supports multiple options/workflows how to run the program, a functionality provided by the pyre plexus application class. It currently offers three options,

```
$ slipmodel.plexus about # show application info
$ slipmodel.plexus sample --config=...  # full simulation, equivalent to slipmodel␣
↪command
$ slipmodel.plexus forward --config=... # forward modeling only
$ slipmodel.plexus  #  call about (as default) to show application info
```

The same configuration file can be used for either options. To run the Bayesian simulations, you may use the same file and run

```
$ slipmodel --config=static.pfg
# or
$ slipmodel.plexus sample --config=static.pfg
```

the settings for the forward problem option have no effect on the simulation workflow and vice versa.

4.5.1.7 Utilities

We also provide some utilities (in Python) which may be useful to analyze the data or data conversions. Some of scripts may require user modification. Before we can finalize them into standard features, these utilities are currently located at seismic/examples/utils directory of the source code.

HDF5 Converter tool

We recommend HDF5 as the input format. A conversion tool `H5Converter` is provided if you need to convert any `.txt` or `.bin` files (e.g., from AlTar 1.1) to `.h5`.

Examples

Convert a text file to hdf5

```
H5Converter --inputs=static.gf.txt
```

Convert a binary file to hdf5, additional information such as the precision (default=float32) and the shape (default = 1d vector and will be reshaped to 2d in program if needed) of the output can be added by

```
H5Converter --inputs=kinematicG.gf.bin --precision='float32' --shape=[100,11000]
```

Merge several files into one hdf5, e.g., to prepare the sensitivity kernels for Cp,

```
H5Converter --inputs=[static.kernel.pertL1.txt,static.kernel.pertL2.txt] --
↪output=static.kernel.h5
```

For help on all available options

```
H5Converter --help
```

Plot histograms of Bayesian probabilities

You may want to check the distributions of the (log) prior/likelihood/posterior probabilities, which usually a good indication for the simulation performance. The utility is named `plotBayesian`.

Taking the source code example as an example,

```
# go to the result directory
cd results/static
# run plotBayesian for the final step output,
# which shows the histograms of (log) prior/likelihood/posterior
../../utils/plotBayesian
# to show output from a different beta step
../../utils/plotBayesian --step=step_000.h5
# to change the number of bins for the histogram
../../utils/plotBayesian --bin=20
```

The `plotBayesian` utility generates a `Bayesian_histograms.pdf` file for the plot. You may change the script to show the plot on GUI directly or save it to a different format.

Compute the mean model

`meanModelStatic.py` under the `utils` directory can be used to compute the mean model of the samples in a given beta step. It also serves as a tool to convert AlTar2 H5 output to AlTar-1.1 H5 output.

```
# go the result directory
cd results/static
# run the utility
python3 ../../utils/meanModelStatic.py
```

which prints out the mean values and variances of all parameters, as well as saving them to text files. It also converts `step_final.h5` to AlTar-1.1 H5 format, `step_final_v1.h5`.

For a different beta step output, you need to change `input` and `output` in the script.

The script can also be used for other models, not limited to `static`: you will just need to change `psets_list` to the same as your simulation script.

Compare the data predications and observations

`checkDataPrediction.py` reads the data predictions from the forward modeling application and compares them with the input data observations. You may need to change the input files in the script.

Convert AlTar2 output to AlTar-1.1 output

AlTar2's H5 output differs from AlTar-1.1 H5 output in rearranging the samples in their parameter sets. You may use the `meanModelStatic.py` utility above to convert them.

4.5.2 Static Slip Inversion with Cp: Epistemic Uncertainties

In addition to observational errors, one can calculate epistemic uncertainties (Minson et al., 2013), which affect the forward model and are related to our imperfect knowledge of the Earth interior. Epistemic uncertainties will stem from a various set of parameters, but the most prominent uncertainties will derive from elastic heterogeneity (Duputel et al., 2014) and fault geometry (Ragon et al., 2018, 2019).

To estimate epistemic uncertainties, we assume that the real surface displacement follows a Gaussian distribution centred on the predictions with a prediction covariance matrix Cp that depends on the resulting source model. The misfit covariance matrix characterizing the misfit function becomes `Cx = Cd + Cp`.

First approach: static Cp

The prediction covariance matrix `Cp` can be included in the inversion following two different approaches. In the first, `Cp` is calculated a priori and included in the inversion process within `Cx`. In this case, the full `Cx` matrix replaces the `Cd` matrix:

```
    dataobs = altar.cuda.data.data12
    dataobs:
        observations = 1000
        data_file = d.txt
        cd_file = cx.txt ; Cx contains both Cd and Cp
```

Second approach: Updated Cp

The alternative approach, implemented in AlTar, is to update Cp with interim models at each step of the tempered inversion. The covariance matrix `Cp` is calculated from 3 variables: a sensitivity kernel for every investigated parameter,

the standard deviation of the a priori distribution of investigated parameters, and an initial model chosen a priori. In this approach, `Cp` is re-calculated at each tempering step by choosing the mean of tested samples as the assumed initial model.

```
    ; sensitivity kernels
    nCmu = 8 ; number of investigated parameters
    cmu_file = Cmu.h5 ; standard deviation matrix
    kmu_file = Kelastic.h5 ; tensor of sensitivity kernels (matrix if only one␣
↪investigated parameter)
    initial_model_file = inimodel.txt ; initial model vector
```

Additionnally, `Cp` can be incorporated in the inversion at any tempering step. The tempering step will be selected with its corresponding beta value, parameterized by `beta_cp_start`. If `beta_cp_start = 0.01`, then `Cp` will be incorporated from the beginning of the inversion process. If `beta_cp_start = 0.1` or `0.5`, then `Cp` will be incorporated after a few tempering steps, or once beta reaches 0.5, respectively. `beta_use_initial_model` will indicate if the initial model should be used (`1`) or not (`0`). If not, then a unit uniform initial model is used.

```
    beta_cp_start = 0.01 ; beta value to start incorporating Cp
    beta_use_initial_model = 1 ; use provided initial model
```

4.5.3 Kinematic Slip Inversion

4.5.3.1 Kinematic Source Model

The kinematic source model studies also the temporal evolution of coseismic slips. The kinematic source model currently implemented in AlTar is formulated as follows.

The fault plane, as a rectangular area, is divided into $N_{dd} \times N_{as}$ square patches (dd=down dip, as=along strike), and time is divided into N_t intervals. The kinematic slip function $\mathbf{M}_b(\vec{\xi}, t)$ (big M) can be in general written as

$$\mathbf{M}_b(\vec{\xi}, t) = \mathbf{D}(\vec{\xi}) S(t - T_R(\vec{\xi}); T_r(\vec{\xi})),$$

where $\vec{\xi}, t$ label the patch and time, respectively, and

- $\mathbf{D}(\vec{\xi})$ is the coseismic slip vector, also subject to the static source model inversion;
- $T_R(\vec{\xi})$ is the initial rupture time of each patch, determined by solving the Eikonal equation with a Fast Sweeping algorithm, given the location of the hypocenter H_0 and the rupture velocity $V_r(\vec{\xi})$ (assumed to be isotropic and the same within each patch);
- $T_r(\vec{\xi})$ is the slip duration (rise time) for each patch;
- $S(t, T_r)$ is the source time function which is finite only for $t \in [0, T_r]$ and integrated to 1: we choose a triangular source time function.

To produce smooth synthetics, we refine each patch into $N_{mesh} \times N_{mesh}$ grids when solving the eikonal equation, while dividing each time interval into N_{pt} points. $\mathbf{M}_b(\vec{\xi}, t)$ are then interpolated and integrated from these finer spatial and temporal meshes.

The predicted observations can be calculated by,

$$d_{prediction}(\vec{x}, t) = \sum_{\vec{\xi}, t'} \mathcal{G}_b(\vec{x}, \vec{\xi}; t - t') \mathbf{M}_b(\vec{\xi}, t')$$

which is a linear model (note that the Eikonal equation is non-linear). Here, $\mathcal{G}_b(\vec{x}, \vec{\xi}; t - t')$ (big G) are the kinematic Green's functions relating the source $\mathbf{M}_b(\vec{\xi}, t')$ to an observation location at a given time $(\vec{x}, t)$. The kinematic Green's functions are pre-calculated as inputs to AlTar. There are some existing software packages for the calculation, e.g., the frequency-wavenumber integration code of Zhu and Rivera, or AXITRA.

In summary, the kinematic inversion uses the following source parameters

- the two components of coseismic slip $\mathbf{D}(\vec{\xi})$ ($2 \times N_{dd} \times N_{as}$ elements),
- the rupture velocity $V_r(\vec{\xi})$ ($N_{dd} \times N_{as}$ elements),
- the rise time $T_r(\vec{\xi})$ ($N_{dd} \times N_{as}$ elements),
- the location of the hypocenter $\mathbf{H}_0$ (2 elements),

and the forward model

$$d_{pred} = G(\mathbf{D}(\vec{\xi}), V_r(\vec{\xi}), T_r(\vec{\xi}), \mathbf{H}_0)$$

is preformed in two steps,

- to obtain $\mathbf{M}_b(\vec{\xi}, t')$ from the Eikonal equation sovler,
- to calculate $d_{pred} = \mathcal{G}_b \mathbf{M}_b$.

4.5.3.2 Joint Kinematic-Static Inversion

Because the slips $\mathbf{D}(\vec{\xi})$ are subject to both static and kinematic observations, we in general run static and kinematic models together, by a model ensemble with or without cascading.

The joint Bayesian probability for static and kinematic models can be written as

$$P(\boldsymbol{\theta}_c, \boldsymbol{\theta}_s, \boldsymbol{\theta}_k | \mathbf{d}_s, \mathbf{d}_k) = P(\boldsymbol{\theta}_c) P(\boldsymbol{\theta}_s) P(\boldsymbol{\theta}_k) P(\mathbf{d}_s | \boldsymbol{\theta}_c, \boldsymbol{\theta}_s) P(\mathbf{d}_k | \boldsymbol{\theta}_c, \boldsymbol{\theta}_k).$$

where $\boldsymbol{\theta}_c = [strikeslip, dipslip]$ as shared (common) parameters, $\boldsymbol{\theta}_s = [ramp]$, and $\boldsymbol{\theta}_k = [risetime, rupturevelocity, hypocenter]$. The data likelihoods are computed from

- the static model with observations $\mathbf{d}_s$ and the forward model $\mathbf{d}_s^{pred} = G_s(\boldsymbol{\theta}_c, \boldsymbol{\theta}_s)$,
- the kinematic model with $\mathbf{d}_k$ and $\mathbf{d}_k^{pred} = G_k(\boldsymbol{\theta}_c, \boldsymbol{\theta}_k)$.

In the annealing schemes such as CATMIP, we can introduce transitioning distributions

$$P_{\beta_s,\beta_k}(\boldsymbol{\theta}_c, \boldsymbol{\theta}_s, \boldsymbol{\theta}_k | \mathbf{d}_s, \mathbf{d}_k) = P(\boldsymbol{\theta}_c) P(\boldsymbol{\theta}_s) P(\boldsymbol{\theta}_k) [P(\mathbf{d}_s | \boldsymbol{\theta}_c, \boldsymbol{\theta}_s)]^{\beta_s} [P(\mathbf{d}_k | \boldsymbol{\theta}_c, \boldsymbol{\theta}_k)]^{\beta_k}.$$

where both β_s and β_k vary from 0 to 1, either jointly or independently. Depending on how β_s, β_k vary, AlTar supports two schemes:

- In the non-cascading scheme, we set $\beta = \beta_s = \beta_k$, i.e., both increasing at the same pace, while their increment in the COV scheduler is determined by the coefficient of variation of the weights $w = [P(\mathbf{d}_s | \boldsymbol{\theta}_c, \boldsymbol{\theta}_s) P(\mathbf{d}_k | \boldsymbol{\theta}_c, \boldsymbol{\theta}_k)]^{\beta_{m+1} - \beta_m}$.
- In the cascading scheme, we run AlTar twice:
  1. In the first run, we perform inversion on the static model only, or producing samples of $\boldsymbol{\theta}_c$ and $\boldsymbol{\theta}_s$ pursuant to the posterior distribution of the static model $P(\boldsymbol{\theta}_c, \boldsymbol{\theta}_s | \mathbf{d}_s) = P(\boldsymbol{\theta}_c) P(\boldsymbol{\theta}_s) [P(\mathbf{d}_s | \boldsymbol{\theta}_c, \boldsymbol{\theta}_s)]$.
  2. In the second run, we run static and kinematic models together, by setting $\beta_s = 1$ (`cascaded = True`) and varying β_k from 0 to 1 (`cascaded = False`). Here, the samples of $\boldsymbol{\theta}_c$ and $\boldsymbol{\theta}_s$ from the static inversion are used as the initial samples for the joint inversion. The increment of β_k in the COV scheduler is determined by the coefficient of variation of the weights $w = [P(\mathbf{d}_k | \boldsymbol{\theta}_c, \boldsymbol{\theta}_k)]^{\beta_{k,m+1} - \beta_{k,m}}$.

With the optimized and (usually greatly) reduced search range of $\boldsymbol{\theta}_c$ and $\boldsymbol{\theta}_s$ in parameter space from the static inversion, the cascaded joint-static-kinematic inversion runs more efficiently than the non-cascading scheme. In general, the cascading scheme is recommended for all model ensembles if there is a computation-intensive model (such as the kinematic model) present.

4.5.3.3 Configurations (Kinematic Model only)

We illustrate the settings of the kinematic model by assuming to run it alone (in practice this is rarely adopted).

An example configuration file

The configuration file (`kinematicg_only.pfg`) for the kinematic model appears as

```
; application instance name
slipmodel:

    ; model to be sampled
    model = altar.models.seismic.cuda.kinematicg
    model:

        dataobs:
            observations = 14148 ; number of observed data points
            data_file = kinematicG.data.h5
            cd_std = 5.0e-3
            ; or cd_file = kinematicG.cd.h5 if using a file input

        ; fixed model parameters
        ; green's function (2*Ndd*Nas*Nt, observations)
        ; [Nt][2(strike/dip)][Nas][Ndd] with leading dimensions on the right
        green = kinematicG.gf.h5

        Ndd = 3 ; patches along dip
        Nas = 3 ; patches along strike
        Nmesh = 30 ; mesh points for each patch
        dsp = 20.0 ; length for each patch, km
        Nt = 90 ; number of time intervals
        Npt = 2 ; mesh points for each time interval
        dt = 1.0 ; time unit for each interval, second
        ; initial starting time for each patch, in addition to the fast sweeping␣
↪calculated arrival time
        t0s = [0.0] * {slipmodel.model.patches}

        ; parameters to be simulated
        ; provide a list at first, serving as their orders in theta
        psets_list = [strikeslip, dipslip, risetime, rupturevelocity, hypocenter]

        ; define each parameterset
        psets:
            strikeslip = altar.cuda.models.parameterset
            dipslip = altar.cuda.models.parameterset
            risetime = altar.cuda.models.parameterset
            rupturevelocity = altar.cuda.models.parameterset
            hypocenter = altar.cuda.models.parameterset

            ; variables for patches are arranged along dip direction at first␣
↪[Nas][Ndd]
            strikeslip:
                count = {slipmodel.model.patches}
                prep = altar.cuda.distributions.preset ; load preset samples
                prep.input_file = theta_cascaded.h5 ; file name
                prep.dataset = ParameterSets/strikeslip ; dataset name in h5
```

(continues on next page)

(continued from previous page)

```
            prior = altar.cuda.distributions.gaussian
            prior.mean = 0
            prior.sigma = 0.5

        dipslip:
            count = {slipmodel.model.patches}
            prep = altar.cuda.distributions.preset
            prep.input_file = theta_cascaded.h5 ; file name
            prep.dataset = ParameterSets/dipslip ; dataset name in h5
            prior = altar.cuda.distributions.uniform
            prior.support = (-0.5, 20.0)

        risetime:
            count = {slipmodel.model.patches}
            prior = altar.cuda.distributions.uniform
            prior.support = (10.0, 30.0)

        rupturevelocity:
            count = {slipmodel.model.patches}
            prior = altar.cuda.distributions.uniform
            prior.support= (1.0, 6.0)

        ; along strike(first), dip directions
        ; could be separated into 2 for dip and strike direction
        hypocenter:
            count = 2
            prior = altar.cuda.distributions.gaussian
            prior.mean = 20.0
            prior.sigma = 5.0
```

Parameter Sets

The parameter sets or `psets` for the kinematic models are `psets_list = [strikeslip, dipslip, risetime, rupturevelocity, hypocenter]`.

- The names the parameter sets can be changed per your preference, e.g., `strike_slip`, `StrikeSlip`. But the order of the parameter sets must be preserved because the forward model uses the order to map appropriate parameters. `strikeslip` and `dipslip` may be switched as long as their order is consistent with the Green's functions.
- `strikeslip` and `dipslip` are two components of the cumulative slip displacement. If you prefer to load their initial samples from the static inversion results, use the `altar.cuda.distributions.preset` distribution for `prep`, see Preset distribution for more details. Only `HDF5` format is accepted for Preset prior and therefore, its dataset name `prep.dataset=ParameterSets/strikeslip` is also required. If you choose to generate samples from a given distribution, e.g., gaussian/moment scale distributions, please follow the *Parameter Sets* example in static inversion to set their `prep` and `prior` distributions. The slips are usually in unit of meters.
- `risetime` (in unit of seconds) and `rupturevelocity` (in unit of km/s) are rupture duration and velocities for each patch. As they are positive, usually uniform or truncated gaussian distributions are used as their priors.
- `strikeslip`, `dipslip`, `risetime` and `rupturevelocity` are defined for each patch and their counts are the same as the number of patches. The sequence of patches is arranged as, for $N_{dd} \times N_{as}$ patches, $(as_0, dd_0), (as_0, dd_1), ...(as_0, dd_{Ndd-1}), (as_1, dd_0), ..., (as_{Nas-1}, dd_{Ndd-1})$. Or `dd` is the leading dimension.
- `hypocenter` (in unit of km) is the location of the hypocenter measured from the **CENTER** of the (as_0, dd_0) patch (note that it's not the origin or the corner), in unit of kilometers. If the distances down dip (dd) and along

strike (as) directions are different, you may separate them as two parameter sets `hypo_as` and `hypo_dd`, with `as` component being first.

Input files

The kinematic model requires the following input files

green
: the kinematic Green's functions, with the `shape=(2*Ndd*Nas*Nt, observations)`. The `observations` is the number of observed data points, and is the leading dimension. `[Nt][2(strike/dip)][Nas][Ndd]` labels the spatial-temporal source displacements with leading dimensions on the right (or which comes first):

```
(t=0, strike, as_0, dd_0, obs_0), (t=0, strike, as_0, dd_0, obs_1), ..., (t=0, strike,
↪ as_0, dd_0, obs_{Nobs-1})
(t=0, strike, as_0, dd_1, obs_0), (t=0, strike, as_0, dd_1, obs_1), ..., (t=0, strike,
↪ as_0, dd_1, obs_{Nobs-1})
... ...
(t=0, strike, as_0, dd_{Ndd-1}, obs_0), (t=0, strike, as_0, dd_{Ndd-1}, obs_1), ..., ␣
↪(t=0, strike, as_0, dd_{Ndd-1}, obs_{Nobs-1})
(t=0, strike, as_1, dd_0, obs_0), (t=0, strike, as_1, dd_0, obs_1), ..., (t=0, strike,
↪ as_1, dd_0, obs_{Nobs-1})
... ...
(t=0, strike, as_{Nas-1}, dd_{Ndd-1}, obs_0), (t=0, strike, as_{Nas-1}, dd_{Ndd-1},␣
↪obs_1), ..., (t=0, strike, as_{Nas-1}, dd_{Ndd-1}, obs_{Nobs-1})
(t=0, dip, as_0, dd_0, obs_0), (t=0, dip, as_0, dd_0, obs_1), ..., (t=0, dip, as_0,␣
↪dd_0, obs_{Nobs-1})
... ...
(t=0, dip, as_{Nas-1}, dd_{Ndd-1}, obs_0), (t=0, dip, as_{Nas-1}, dd_{Ndd-1}, obs_1),␣
↪..., (t=0, dip, as_{Nas-1}, dd_{Ndd-1}, obs_{Nobs-1})
(t=1, strike, as_0, dd_0, obs_0), (t=1, strike, as_0, dd_0, obs_1), ..., (t=1, strike,
↪ as_0, dd_0, obs_{Nobs-1})
... ...
(t={Nt-1}, dip, as_{Nas-1}, dd_{Ndd-1}, obs_0), (t={Nt-1}, dip, as_{Nas-1}, dd_{Ndd-1}
↪, obs_1), ..., (t={Nt-1}, dip, as_{Nas-1}, dd_{Ndd-1}, obs_{Nobs-1})

You need to follow the above order when preparing the Green's functions as it's the␣
↪order how big-M is arranged in the forward model.
```

dataobs.data_file
: 1d vector of observed data.

dataobs.cd_file
: the data covariance matrix with `shape=(observations, observations)`. If not available, a constant `dataobs.cd_std` may be used instead.

The input files can be a text file (.txt), a raw binary (.bin or .dat) or an HDF5 (.h5) file, with its format recognized by the file suffix.

Other attributes

Ndd
integer, number of patches down the dip direction

Nas
integer, number of patches along the strike direction

Nmesh
integer, number of mesh points for each patch, i.e., each patch is divided into $N_{mesh} \times N_{mesh}$ grids for solving the Eikonal equation

dsp
float, the length for each patch, in km

Nt
integer, number of time intervals, should be long enough to cover the rupture process

Npt
integer, number of mesh points for each time interval

dt
float, time unit for each time interval, in second

t0s
a list of floats with N_{patch} elements, initial starting time for each patch, in addition to the fast sweeping calculated arrival time. If configured properly, they can reduce the total number of time intervals needed for the computation.

4.5.3.4 Configurations (Joint inversion)

The configuration for the joint kinematic-static inversion (`kinematicg.pfg`) appears as

```
model = altar.models.seismic.cuda.cascaded
model:
    ; parameters to be simulated (priors)
    ; provide a list at first, serving as their orders in theta
    psets_list = [strikeslip, dipslip, ramp, risetime, rupturevelocity, hypocenter]
    ; define parametersets
    psets:
        ; define the prior for each parameter set
        ; use preset prior to load samples from static inversion for cascading scheme
        ; or use regular priors for non-cascading scheme
        strikeslip = ... ...
        dipslip = ... ...
        ... ...

    ; the model ensemble
    models:
        static = altar.models.seismic.cuda.static
        kinematic = altar.models.seismic.cuda.kinematicg

        static:
            cascaded = True ; or False for non-cascading scheme
            psets_list = [strikeslip, dipslip, ramp]
            ; other static model configurations
            ... ...
```

(continues on next page)

(continued from previous page)

```
        kinematic:
            cascaded = False ; default setting for model
            psets_list = [strikeslip, dipslip, risetime, rupturevelocity, hypocenter]
            ; other kinematic model configurations
            ... ...
```

Here, the main model is a model ensemble `altar.models.seismic.cuda.cascaded`, while its embedded-models `[static, kinematic]` listed as elements of the attribute `models` (a dict).

The parametersets are properties of the main model and are processed by the main model for sample initializations and prior probability computations. Each embedded-model only requires a `psets_list` attribute to extract a sub set of parameters from `model.psets` for its own forward modelling, with the data likelihood computed with respect to its own data observations. The main model collects the data likelihood from all embedded models and assembles them into the Bayesian posterior.

The configuration for each embedded model will be the same as when running it independently, except for an extra flag `cascaded` (default=``False``) to control the cascading scheme.

For the non-cascading scheme with $\beta_s = \beta_k = \beta$ varying from 0 to 1 simultaneously, set

```
static:
    cascaded=False
kinematic:
    cascaded=False
```

while for the cascading scheme with $\beta_s = 1$, and $\beta_k = \beta$ varying from 0 to 1, after running the static inversion,

```
static:
    cascaded=True
kinematic:
    cascaded=False
```

4.5.3.5 Examples

The examples for the joint static and kinematic inversion are available at models/seismic/examples. Input files for both static and kinematic models are stored under the `9patch` directory.

Cascading Scheme

The first step is to run the static inversion only:

```
$ slipmodel --config=static.pfg
```

Please refer to *Static Slip Inversion* for more details.

The results are saved in the directory `results/static` specified by the config `controller.archiver.output_dir`, which include HDF5 files for all or selected annealing steps. The final step ($\beta = 1$) results are saved in `step_final.h5`. Copy that file to `9patch` directory so that the final samples of strike/dip slips serve as initial samples for the joint inversion:

```
$ cp results/static/step_final.h5 9patch/theta_cascaded.h5
```

Please also note that the number of chains `job.chains` in the static inversion should be the same or larger than that of the joint inversion so that there are enough samples available.

We now can run the joint static-kinematic inversion,

```
$ slipmodel --config=kinematicg.pfg
```

The results for the jointly inversion will be saved to `results/cascaded`, or any other directory by changing `controller.archiver.output_dir` in `kinematicg.pfg`.

Non-cascading Scheme

For the non-cascading scheme, you don't need the step to run static inversion.

You may edit the `kinematicg.pfg` file (or make a copy at first),

- change the `static.cascaded` to `False`;
- change the `prep` distributions for `strikeslip`, `dipslip`, and `ramp` from `preset` to appropriate distributions, e.g., copying them from `static.pfg` file.
- change the output directory `controller.archiver.output_dir` to, e.g., `results/non-cascaded`.

Then run the joint inversion:

```
$ slipmodel --config=kinematicg.pfg
```

In general, the non-cascading takes long iterations to converge and therefore is slower than the cascading scheme.

Please refer to the *AlTar Framework* for the Bayesian MCMC framework options and job/output controls. For example, the Adaptive Metropolis Sampler in general has better performance than the fixed-length Metropolis Sampler, which can be selected by setting `sampler=altar.cuda.bayesian.adapativemetropolis` in the configuration file.

4.5.3.6 Forward Model Application (new version)

It is essentially the same as the static *Forward Model Application*. The steps are,

1) prepare a file with a set of parameters in `case` input directory;
2) add the forward problem settings to the configuration file `kinematicg.pfg` and change the `job` configuration to run with one GPU,

```
; the model
model = altar.models.seismic.cuda.cascaded
model:

    ; settings for running forward problem only
    ; forward theta input
    theta_input = kinematicG_mean_model.txt
    ; forward output file
    forward_output = forward_prediction.h5

... ...

job:
tasks = 1 ; number of tasks per host
gpus = 1  ; number of gpus per task
gpuprecision = float32 ; double(float64) or single(float32) precision for gpu␣
↪computations
;gpuids = [0] ; a list gpu device ids for tasks on each host, default range(job.gpus)
```

3) run the plexus command,

```
$ slipmodel.plexus forward --config=kinematicg.pfg
```

4) the predicted data from both static and kinematic models, as well as the bigM, will be saved to one `forward_prediction.h5` file.

An example is available at examples/kinematicg.pfg.

Note also that the same script can be used for Bayesian simulation, with the commands,

```
$ slipmodel --config=kinematicg.pfg
# or
$ slipmodel.plexus sample --config=kinematicg.pfg
```

See *Forward Model Application* for more details.

4.5.3.7 Forward Model Application (old version)

Note: This section describes an old implementation, which will be depreciated in the next release.

When analyzing the results, you may need to run the forward model once for the obtained mean-model or any set of parameters, to produce data predictions in comparison with data observations. Since the kinematic forward model is not straightforward, we provide an additional application for running the forward model only, named `kinematicForwardModel`.

An example configuration file is available as examples/kinematicg_forward.pfg. You may use the `model` configuration copied from `kinematicg.pfg`, with extra settings

```
; theta input
theta_input = kinematicG_synthetic_theta.txt

; output h5 file name
; data prediction is 1d vector with dimension observations
data_output = kinematicG_synthetic_data.h5
; Mb is 1d vector arranged as [Nt][2(strike/dip)][Nas][Ndd] with leading dimensions␣
↪on the right
mb_output = kinematicG_synthetic_mb.h5
```

where `theta_input` is the input of a mean model or any synthetic model, and `data_out` and `mb_output` are output file names for the data predictions and the big M (you can create an animation from it to observe the rupture process).

The forward model application may be run as

```
$ kinematicForwardModel --config=kinematicg_forward.pfg
```

CHAPTER

FIVE

PROGRAMMING GUIDE

5.1 Introduction

This guide aims to help developers to write their own models for AlTar.

- AlTar's framework is developed in Python while the compute-intensive routines are developed as C/C++ or CUDA(for GPUs) extension modules, to offer a user-friendly interface without scarifying the efficiency.
- AlTar follows the `components`-based programming model of pyre. Components, as Python classes with extended functionalities, not only facilitate the modularized software development, but also offer user with great convenience to config settings and parameters *at runtime*:
  - to choose between different implementations of a prescribed functionality (protocol), e.g., to choose different Metropolis sampling algorithms;
  - to turn features on or off, e.g., the debugger, profiler;
  - parameters and other attributes are also implemented as configurable properties (traits).
- AlTar utilizes the job management system from pyre to self-deploy the simulations to different platforms, single thread or multiple threads, single machine or clusters, CPU or GPUs.

5.2 Code Organization

New models can be added to the `altar/models` directory. We recommend the AlTar/pyre convention to organize the source code inside the directory.

The simplest model can be constructed all by Python. The minimum set of files includes, e.g., for a new model named `regression`,

```
altar/models/regression # root
├── CMakeLists.txt # cmake script
├── bin  # binary commands
│   └── regression # command to invoke AlTar simulation for this model
├── regression # python scripts
│   ├── __init__.py # export modules at the package level
│   ├── meta.py.in # metadata includes version/copyright/authors ...
│   └── Regression.py # the main Python program
└── examples # (optional) provide examples to users
    ├── regression.pfg # an example of the configuration file
    └── synthetic # a directory includes data files for the example
```

If C/C++/Fortran routines are needed, which are wrapped as extension modules in Python, these additional files can be arranged as

```
altar/models/regression # root
├── lib # C/C++/Fortran routines compiled as shared libraries
│   └── libregression # compiled to INSTALL_DIR/lib/libregression.so(dylib)
│       ├── version.h # version header file
│       ├── version.cc  # version code
│       ├── Source.h  # C++ source code header file
│       ├── Source.icc  # C++ source code header include file for static methods/
↪variables
│       └── Source.cc # # C++ source code
├── ext  # CPython extension modules
│   └── regression # compiled to regression.cpython-{}.so
│       ├── capsules.h # Python Capsule (pass pointers of C++ objects) definitions
│       ├── regression.cc  # extension module definition
│       ├── source.h  # header file for wrappers
│       ├── source.cc  # CPython wrappers for C/C++/Fortran routines/classes
│       ├── metadata.h  # metadata header file includes version ...
│       └── metadata.cc # # metadata source file includes version ...
└── regression # python scripts
    └── ext  # to host the regression.cpython-{}.so product
        └── __init__.py # export extension modules at the package level
```

GPU/CUDA models can be constructed in the same fashion. The following files will be added,

```
altar/models/regression # root
├── lib # C/C++/Fortran routines compiled as shared libraries
│   └── libcudaregression # compiled to INSTALL_DIR/lib/libcudaregression.so(dylib)
│       ├── version.h # version header file
│       ├── version.cc  # version code
│       ├── Source.h  # CUDA source code header file
│       ├── Source.icc  # CUDA source code header include file for static methods/
↪variables
│       └── Source.cu # # CUDA source code
├── ext  # CPython extension modules
│   └── cudaregression # compiled to cudaregression.cpython-{}.so
│       ├── capsules.h # Python Capsule (pass pointers of C++ objects) definitions
│       ├── regression.cc  # extension module definition
│       ├── source.h  # header file for wrappers
│       ├── source.cc  # CPython wrappers for C/C++/Fortran routines/classes
│       ├── metadata.h  # metadata header file includes version ...
│       └── metadata.cc # # metadata source file includes version ...
└── regression # python scripts
    ├──cuda #
    │   ├── __init__.py # export modules at the package level
    │   ├── meta.py.in # metadata includes version/copyright/authors ...
    │   └── cudaRegression.py # the main Python program
    └── ext  # to host the regression.cpython-{}.so product
        └── __init__.py # export cuda extension modules as well
```

Additional compile/build scripts are also required, for either CMake or (new) MM build tools.

For `CMAKE`, in addition to the `CMakeLists.txt` file under the model directory, these files need to be added to modified,

```
altar # root directory
├── .cmake # for cmake
│   └── altar_regression.cmake # functions to build packages/lib/modules/driver(bin)
└── CMakeLists.txt # to include the new model
```

For (new) MM build tool,

```
altar # root directory
└── .mm # for cmake
   ├── regression.def # new model configurations
   └── altar.mm # to include the new model
```

Details for the above files will be illustrated by specific examples in the following sections.

5.3 Bayesian Model

An AlTar application can be broadly separated into two parts, the MCMC framework for Bayesian statistics and a Bayesian Model who is responsible for calculating the prior/data likelihood/posterior probabilities for a given set of parameters θ.

Specifically, a Model is required to provide the methods as listed in the Model protocol,

```
# services for the simulation controller
@altar.provides
def initialize(self, application):
    """
    Initialize the state of the model given a {problem} specification
    """

@altar.provides
def initializeSample(self, step):
    """
    Fill {step.theta} with an initial random sample from my prior distribution.
    """

@altar.provides
def priorLikelihood(self, step):
    """
    Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior
    distribution
    """

@altar.provides
def dataLikelihood(self, step):
    """
    Fill {step.data} with the likelihoods of the samples in {step.theta} given the␣
↪available
    data. This is what is usually referred to as the "forward model"
    """

@altar.provides
def posteriorLikelihood(self, step):
    """
    Given the {step.prior} and {step.data} likelihoods, compute a generalized␣
↪posterior using
    {step.beta} and deposit the result in {step.post}
    """

@altar.provides
def likelihoods(self, step):
    """
    Convenience function that computes all three likelihoods at once given the␣
```

(continues on next page)

(continued from previous page)

```
↪current {step}
    of the problem
    """

@altar.provides
def verify(self, step, mask):
    """
    Check whether the samples in {step.theta} are consistent with the model␣
↪requirements and
    update the {mask}, a vector with zeroes for valid samples and non-zero for␣
↪invalid ones
    """

# notifications
@altar.provides
def top(self, annealer):
    """
    Notification that a β step is about to start
    """

@altar.provides
def bottom(self, annealer):
    """
    Notification that a β step just ended
    """
```

and these methods are called by the AlTar framework at respective places. (`@altar.provides` decorated methods are similar to C++ virtual functions for which the derived classes must declare).

- `initialize` is to initialize various parameters and settings of the Model, as also being required for other components in AlTar. It takes `application`, the root component, as inputs in order to pull information from other components, e.g. obtaining the number of chains and the processor information (cpu/gpu) from the `job` component.
- Most other methods take CoolingStep (step) as inputs, which stores the process data including
  - the parameters being sampled, θ, in 2d array `shape=(samples, parameters)`,
  - the prior, data likelihood, and posterior probability densities for samples, each in 1d vector `shape=samples`.

  Note that AlTar processes a batch of samples (number of Markov Chains) in parallel.
- `initializeSample` is to generate random samples from a prior distribution in the beginning of the simulations. Samples can also be loaded from pre-calculated values using the `preset` prior.
- `priorLikelihood` computes the prior probability densities from the given prior distribution(s). During the simulation, when new proposed samples fall outside of the support of certain ranged distributions, the density is 0 and therefore the proposals are invalid. Because AlTar uses the logarithmic value of the densities, we need an extra `verify` method to check the ranged priors.
- `dataLikelihood` computes the data likelihood. It performs
  - the forward modeling, calculating the data predictions from θ,
  - computes the residual between data predictions and observations,
  - and return the data likelihood with a given Norm function (e.g., L2-Norm).
- `posteriorLikelihood` computes the posterior probability densities from prior and data likelihood. For transitional *posterior* distributions in annealing schemes, it is simply $prior + \beta * data$.

- `top` and `bottom` methods are hooks for developers to insert model-specific procedures before or after each annealing step. For example, for models considering model uncertainties, these methods are the places to invoke computing Cp and updating the covariance matrix (Cd+Cp).

Many models may share the same procedures to compute the prior, data likelihood, and posterior; they may differ in forward modeling. We offer some templates to simplify the model development.

5.4 Model with the BayesianL2 template

A BayesianL2 template assumes a fixed structure of contiguous parameter sets and the data observations with L2-norm. With these assumptions, all required model methods are pre-defined while developers are only required to define a `forwardModel` method which performs the forward modeling. This template offers the easiest approach to write a new model.

We use the linear regression model to demonstrate how to construct a Bayesian model with the BayesianL2 template in the following. The linear regression model fits a group of data (x_n, y_n) with a linear function

$$y = slope \times x + intercept + \epsilon_i$$

where `slope` and `intercept` are parameters to be sampled while ϵ_i are random noises. The Python program for this example is available at models/regression/regression/Linear.py.

5.4.1 Parametersets(psets)

In the linear regression model, $\theta = [slope, intercept]$. In principle, slope and intercept could have different prior distributions. Each set of parameters with the same prior distribution is defined as a (contiguous) parameter set. θ can therefore be described as a parametersets (psets) with each parameter set arranged sequentially and contiguously in 1d vector (or columns in batched samples). In Python, `psets` is a `dict` with a collection of `contiguous` parameter set objects. We further define a list `psets_list` to assure the orders of parameter sets in θ (it is also useful in model ensembles to select model-relevant parameter sets from the global `psets`).

A description of `psets` in the configuration file `linear.pfg` appears as

```
; parameter sets
psets_list = [slope, intercept]
psets:
    slope = contiguous
    intercept = contiguous

    slope:
        count = 1
        prep = uniform
        prep.support = (0, 5)
        prior = uniform
        prior.support = (0, 5)

    intercept:
        count = 1
        prep = uniform
        prep.support = (0, 5)
        prior = uniform
        prior.support = (0, 5)
```

where slope and intercept are each described as a `contiguous` parameter set and each set has its own `prep` (to initialize random samples and `prior` (for verify and prior probability) distributions.

We use the static earthquake inversion as another example, where the sampling parameters are dip-slip (D_d), strike-slip (D_s) displacements for N-patches, and if necessary the inSAR ramping parameters (R). Each is described by a `contiguous` parameter set and assembled as a `psets`. (Each row of) θ is therefore $(D_{d1}, D_{d2}, \ldots, D_{dN}, D_{s1}, D_{s2}, \ldots, D_{sN}, R_1, R_2, \ldots)$. The order of sets D_d, D_s and R can be switched as long as it is consistent with the forward modeling, e.g., the Green's functions. If you want to use different priors for strike slips in different patches, you may separate D_s into several parameter sets.

With `psets`, various methods related to parameters, including `initializeSamples`, `verify` and `priorLikelihood`, are pre-defined in BayesianL2 template. Users only need to specify its included parameter sets in the configuration file.

5.4.2 Data observations(dataobs) with L2-norm

L2-norm is recommended for models which need to incorporate various uncertainties. The data likelihood is computed as

$$\log(DataLikelihood) = -\frac{1}{2}\left[d_{pred} - d_{obs}\right] C_\chi^{-1} \left[d_{pred} - d_{obs}\right]^T + C$$

where $d_{pred} = G(\theta)$ are the data predictions from the forward model, d_{pred} the data observations, C_χ the covariance matrix capturing data (Cd) and/or model(Cp) uncertainties, and C a normalization constant depending on the determinant of C_χ.

In BayesianL2 template, the observed data are described by a `dataobs` object with L2-norm. It includes

- observed data points (`dataobs.dataobs`) in 1d vector with `shape=observations`, `observations` is the number of observed data points;
- data covariance matrix (`dataobs.cd`) in 2d array with `shape=(observations, observations)`. (If only constant diagonal elements are available, use `cd_std` instead).

`dataobs` is responsible for

- loading the data observations (d_{obs}) and the data covariance (C_d),
- calculating the Cholesky decomposition of the inverse C_d and saving it in `dataobs`,
- when called by the `model.dataLikelihood`, computing the L2-norm (likelihood) between data predictions and observations with L2-norm,
- for Cp models, updating the covariance matrix $C_\chi = C_d + C_p$ (though still denoted as C_d),
- when needed, merging the covariance matrix with data in `initialize`, as controlled by a flag `dataobs.merge_cd_with_data=True/False`. This procedure improves performance greatly for models when the covariance matrix can also be merged with model parameters, such as the Green's functions in the linear model, by avoiding repeating the matrix-vector (or matrix-matrix for batched) multiplication.

For the linear regression model, the data points (x_n, y_n) don't fit perfectly into the `dataobs` description. Instead, we treat y_n as data observations and x_n as model parameters. We need to initialize them with the `initialize` method,

```
@altar.export
def initialize(self, application):
    """
    Initialize the state of the model
    """

    # call the super class initialization
    super().initialize(application=application)
    # super class method loads and initializes dataobs with y_n
```

(continues on next page)

(continued from previous page)

```
    # model specific initialization after superclass
    # grab data
    self.x = self.loadFile(self.x_file)
    self.y = self.dataobs.dataobs

    ... ...
```

so that `x` and `y` are now accessible by the forward modeling.

Their descriptions in the configuration file `linear.pfg` appear as, e.g.,

```
case = synthetic ; input directory
dataobs:
    observations = 200
    data_file = y.txt
    cd_std = 1.e-2
x_file = x.txt
```

where (x_n, y_n) are separated into two text files (raw binary and H5 input files are also supported by the `loadFile` function).

With L2-norm `dataobs`, the `dataLikelihood` method can be defined straightforwardly: it calls a `forwardModel` defined for a specific model and with the data predictions or residuals it calls dataobs' norm method to compute the likelihood.

5.4.3 Forward modeling

As shown above, with `psets` and `dataobs`, the BayesianL2 template only requires developers to write a `forwardModel` or `forwardModelBatched` method to compute the data predictions from a given set of θ.

Developers have the options to

- Choose whether to implement `forwardModelBatched` or `forwardModel`. The `forwardModelBatched` computes the data predictions for all samples. A pre-defined version iterates over all samples (rows of θ) and calls `forwardModel` which computes the data predictions for one sample. The batched mode sometimes improves the speed, e.g., in the linear model, one may use the matrix-matrix product (a routine commonly optimized for both CPU and GPU) to compute $d = G\theta$. In this case, a customized `forwardModelBatched` method may be defined to override the one pre-defined in BayesianL2.
- Choose to simply compute the data predictions or return the residuals between predictions and observations, depending on the performance or convenience; This is controlled by a flag `return_residual = True/False` which can be specified either in `model.initialize` code or in the configuration file `model.return_residual=True/False`.
- How to incorporate the data covariance (Cd). If the model uncertainties (Cp) are also considered, please refer to the examples such as `models/seismic/staticCp`.

For the linear regression model, the simplest implementation is to use the `forwardModel` method for a single set of parameters, and the code appears as

```
def forwardModel(self, theta, prediction):
    """
    Forward Model
    :param theta: sampling parameters for one sample
    :param prediction: data prediction or residual (prediction - observation)
    :return:
```

(continues on next page)

(continued from previous page)

```
    """

    # grab the parameters from theta

    slope = theta[0]
    intercept = theta[1]

    # calculate the residual between data prediction and observation
    size = self.observations
    for i in range(size):
        prediction[i] = slope * self.x[i] + intercept - self.y[i]

    # all done
    return self
```

we also need to specify the flag `return_residual = True` accordingly.

We can also use the batch method `forwardModelBatched`, where we can construct a 2d array `G=[[x1, x2, ..., xN], [1, 1, ... ,1]]`, and simply use the matrix-matrix product `gemm` to calculate the predictions for all samples $d_{pred} = G\theta$.

We now complete a new model with the BayesianL2 template. Note that if either parameter sets or L2-`dataobs` descriptions doesn't fit your model, you may write your own methods following the `Model` protocol.

Please also note that vectors/matrices in AlTar are based on GSL, while they can operate as numpy arrays. But if you would like to use some numpy/scipy functions on numpy arrays, on you may create a numpy ndarray view or copy from GSL vectors/matrices. See *Matrix/Vector (GSL)* for more details.

5.5 C/C++/CUDA extension modules

For certain procedures, the Python code might not be efficient and you might want to write them in other high performance programming languages. For example, the Linear Algebra libraries (BLAS, LAPACK) are written in FORTRAN/C while are accessible in Python by extension modules.

While there are many convenient tools to write Python wrappers for C/C++/Fortran/CUDA functions, such as `cython`, `SWIG`, `Boost.Python`, `pybind11`, we recommend the native CPython method.

Let's use an example of vector copy. Define in `copy.h`:

```
// vector_copy
extern const char * const vector_copy__name__;
extern const char * const vector_copy__doc__;
PyObject * vector_copy(PyObject *, PyObject *);
```

The source code `copy.cc`

```
PyObject *
vector_copy(PyObject *, PyObject * args) {
// the arguments
PyObject * sourceCapsule;
PyObject * destinationCapsule;
// unpack the argument tuple
int status = PyArg_ParseTuple(
                              args, "O!O!:vector_copy",
                              &PyCapsule_Type, &destinationCapsule,
```

(continues on next page)

(continued from previous page)

```
                              &PyCapsule_Type, &sourceCapsule
                              );
// if something went wrong
if (!status) return 0;
// bail out if the source capsule is not valid
if (!PyCapsule_IsValid(sourceCapsule, capsule_t)) {
    PyErr_SetString(PyExc_TypeError, "invalid vector capsule for source");
    return 0;
}
// bail out if the destination capsule is not valid
if (!PyCapsule_IsValid(destinationCapsule, capsule_t)) {
    PyErr_SetString(PyExc_TypeError, "invalid vector capsule for destination");
    return 0;
}

// get the vectors
gsl_vector * source =
    static_cast<gsl_vector *>(PyCapsule_GetPointer(sourceCapsule, capsule_t));
gsl_vector * destination =
    static_cast<gsl_vector *>(PyCapsule_GetPointer(destinationCapsule, capsule_t));
// copy the data
gsl_vector_memcpy(destination, source);

// return None
Py_INCREF(Py_None);
return Py_None;
}
```

5.6 CUDA Models

TBD

5.7 Data Types and Structures

5.7.1 Configurable properties

```
altar.properties.array
altar.properties.bool
altar.properties.catalog
altar.properties.complex
altar.properties.converter
altar.properties.date
altar.properties.decimal
altar.properties.dict
altar.properties.dimensional
altar.properties.envpath
altar.properties.envvar
altar.properties.facility
altar.properties.float
altar.properties.identity
altar.properties.inet
```

(continues on next page)

(continued from previous page)

```
altar.properties.int
altar.properties.istream
altar.properties.list
altar.properties.normalizer
altar.properties.object
altar.properties.ostream
altar.properties.path
altar.properties.paths
altar.properties.property
altar.properties.set
altar.properties.str
altar.properties.strings
altar.properties.time
altar.properties.tuple
altar.properties.uri
altar.properties.uris
altar.properties.validator
```

5.7.2 Matrix/Vector (GSL)

The Matrix/Vector is based on GSL Matrix/Vector.

GSL matrix shape (size1, size2) -> (rows, cols) and is row-major.

5.7.2.1 Convert altar array to gsl_vector

```
array = altar.properties.array(default=(0, 0, 0,0))
gvector = altar.vector(shape=4)
for index, value in enumerate(array): gvector[index] = value
```

5.7.2.2 Basic matrix/vector operations

```
mat = altar.matrix(shape=(rows, cols)) # create a new matrix with dimension rows x␣
↪cols (row-major)
mat.zero() # initialize 0 to each element
mat.fill(number) #
mat_clone = mat.clone() #
mat1.copy(mat2)
```

5.7.2.3 Interfacing as numpy arrays

As there are more utilities available for numpy `ndarray`, you may view or copy GSL vectors/matrices are numpy arrays.

```
# create a gsl vector
gslv = altar.vector(shape=10).fill(1)
# create a numpy array view (data changes to npa_view will change gslv)
npa_view = gslv.ndarray()
# create a numpy array view (data changes to npa_view don't affect gslv)
npa_copy = gslv.ndarray(copy=True)
```

CHAPTER

SIX

COMMON ISSUES

6.1 Installation Issues

6.1.1 Cannot find `gmake`

When the command of GNU make is `make` instead of `gmake`, please set the environmental variable

```
$ export GNU_MAKE=make # for bash
$ setenv GNU_MAKE make # for csh/tcsh
```

or set the variable when calling mm,

```
$ GNU_MAKE=make mm
```

6.1.2 Cannot find `cublas_v2.h`

For certain Linux systems, NVIDIA installer installs `cublas` to the system directory `/usr/include` and `/usr/lib/x86_64-linux-gnu` instead of `/usr/local/cuda`. In this case, please add the include and library paths to `cuda.incpath` and `cuda.libpath` in `config.mm` file.

6.2 Run-time Issues

6.2.1 Locales

Error: UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 18: ordinal not in range(128)

You might need to set the `LANG` variable,

```
$ export LANG=en_US.UTF-8
```

if `en_US.UTF-8` locale is not installed, update your locale by

```
$ sudo apt install locales
$ sudo locale-gen --no-purge --lang en_US.UTF-8
$ sudo update-locale LANG=en_US.UTF-8 LANGUAGE
```

6.2.2 Base case name

Error: altar: bad case name: 'patch-9'

The AlTar App cannot find the configuration file (usually) or the input file directory. Please go to the job directory with the configuration and input files, and run the App again. If the configuration is not named as `theAlTarApp.pfg`, you need `--config=YourConfigFile.pfg` option, e.g.,

```
linear --config=my_linear_model.pfg
```

6.2.3 Configuration Parser Error

Error: File "/opt/anaconda3/envs/altar/lib/python3.9/site-packages/pyre/parsing/Scanner.py", line 71, in pyre_tokenize

match = stream.match(scanner=self, tokenizer=tokenizer)

This is usually due to a bad format in configuration file. For example, `.pfg` files do not recognize TABs; please check your file for possible TABs and replace them with SPACEs. See *Pyre Config Format (.pfg)* for more details.

6.2.4 MPI launcher error

Error: launcher = self.mpi.launcher

AttributeError: 'NoneType' object has no attribute 'launcher'

This happens when AlTar cannot locate the `mpirun` command. It can be solved by manually setting up an `mpi.pfg` file. See *MPI setup* for more details.

6.2.5 Intel MKL Library

Error: Intel MKL FATAL ERROR: Cannot load libmkl_avx2.so.1 or libmkl_def.so.1.

This is due to a Conda issue with MKL libraries. The solution is to preload certain MKL libraries before running AlTar applications,

```
LD_PRELOAD=$CONDA_PREFIX/lib/libmkl_core.so:$CONDA_PREFIX/lib/libmkl_sequential.so␣
↪altarApp --config=configFile
```

CHAPTER

SEVEN

API REFERENCE

This page contains auto-generated API reference documentation[1].

7.1 altar

7.1.1 Subpackages

7.1.1.1 altar.actions

Submodules

altar.actions.About

Module Contents

Classes

class altar.actions.About.**About**(*name*, *spec*, *plexus*, ***kwds*)

Bases: altar.panel()

Display information about this application

root

tip = specify the portion of the namespace to display

name(*self*, *plexus*, ***kwds*)

Print the name of the app for configuration purposes

home(*self*, *plexus*, ***kwds*)

Print the application home directory

prefix(*self*, *plexus*, ***kwds*)

Print the application installation directory

models(*self*, *plexus*, ***kwds*)

Print the altar model directory

[1] Created with sphinx-autoapi

when (*self*, *plexus*, ***kwds*)
Print the build timestamp

etc (*self*, *plexus*, ***kwds*)
Print the application configuration directory

version (*self*, *plexus*, ***kwds*)
Print the version of the altar package

copyright (*self*, *plexus*, ***kwds*)
Print the copyright note of the altar package

credits (*self*, *plexus*, ***kwds*)
Print out the license and terms of use of the altar package

license (*self*, *plexus*, ***kwds*)
Print out the license and terms of use of the altar package

nfs (*self*, *plexus*, ***kwds*)
Dump the application configuration namespace

pfs (*self*, *plexus*, ***kwds*)
Dump the application private filesystem

vfs (*self*, *plexus*, ***kwds*)
Dump the application virtual filesystem

altar.actions.Forward

Module Contents

Classes

class altar.actions.Forward.**Forward**(*name*, *spec*, *plexus*, ***kwds*)
Bases: altar.panel()

Sample the posterior distribution of a model

default (*self*, *plexus*, ***kwds*)
Sample the model posterior distribution

altar.actions.Sample

Module Contents

Classes

class altar.actions.Sample.**Sample**(*name*, *spec*, *plexus*, ***kwds*)
Bases: altar.panel()

Sample the posterior distribution of a model

default (*self*, *plexus*, ***kwds*)
Sample the model posterior distribution

Package Contents

Functions

altar.actions.**about**()

altar.actions.**sample**()

altar.actions.**forward**()

7.1.1.2 altar.bayesian

Submodules

altar.bayesian.Annealer

Module Contents

Classes

class altar.bayesian.Annealer.**Annealer**(*name*, *locator*, ***kwds*)

Bases: altar.component

A Bayesian controller that uses an annealing schedule and MCMC to approximate the posterior distribution of a model

sampler

doc = the sampler of the posterior distribution

scheduler

doc = the generator of the annealing schedule

dispatcher

doc = the event dispatcher that activates the registered handlers

archiver

doc = the archiver of simulation state

model

worker

info

warning

error

debug

firewall

initialize (*self*, *application*)
Initialize me and my parts given an {application} context

posterior (*self*, *model*)
Sample the posterior distribution

deduceAnnealingMethod (*self*, *job*)
Instantiate an annealing method compatible the user choices

sequential (*self*)
Instantiate the plain sequential annealing method

cuda (*self*)
Instantiate a CUDA aware annealing method

threaded (*self*, *threads*, *worker*)
Instantiate the multi-threaded annealing method

mpi (*self*, *worker*)
Instantiate the MPI aware annealing method

altar.bayesian.AnnealingMethod

Module Contents

Classes

class altar.bayesian.AnnealingMethod.**AnnealingMethod** (*annealer*, ***kwds*)
Base class for the various annealing implementation strategies

step

iteration = 0

wid = 0

workers

initialize (*self*, *application*)
Initialize me and my parts given an {application} context

start (*self*, *annealer*)
Start the annealing process from scratch

abstract restart (*self*, *annealer*)
Start the annealing process from a checkpoint

top (*self*, *annealer*)
Notification that we are at the beginning of an update

cool (*self*, *annealer*)
Push my state forward along the cooling schedule

walk (*self*, *annealer*)
Explore configuration space by walking the Markov chains

resample (*self*, *annealer*, *statistics*)

Analyze the acceptance statistics and take the problem state to the end of the annealing step

archive (*self*, *annealer*, *scaling*, *stats*)

Notify archiver to record

bottom (*self*, *annealer*)

Notification that we are at the bottom of an update

finish (*self*, *annealer*)

Notification that the simulation is over

altar.bayesian.Brent

Module Contents

Classes

class altar.bayesian.Brent.**Brent** (*name*, *locator*, ***kwds*)

Bases: altar.component

A δβ solver based on the GSL implementation of the Brent algorithm

tolerance

doc = the fractional tolerance for achieving convergence

maxiter

doc = the maximum number of iterations while looking for a δβ

cov

initialize (*self*, *application*, *scheduler*)

Initialize me and my parts given an {application} context and a {scheduler}

solve (*self*, *llk*, *weight*)

Compute the next temperature in the cooling schedule :param llk: data log-likelihood :param weight: the normalized weight :return: β, cov

altar.bayesian.COV

Module Contents

Classes

class altar.bayesian.COV.**COV** (*name*, *locator*, ***kwds*)

Bases: altar.component

Annealing schedule based on attaining a particular value for the coefficient of variation (COV) of the data likelihood; after Ching[2007].

The goal is to compute a proposed update δβ_m to the temperature β_m such that the vector of weights w_m given by

w_m := π(D|θ_m)^{δβ_m}

has a particular target value for

COV(w_m) := <w_m> / sqrt{<(w_m-<w_m>)^2>}

target

doc = the target value for COV

solver

doc = the δβ solver

check_positive_definiteness

doc = whether to check the positive definiteness of Σ matrix and condition it accordingly

min_eigenvalue_ratio

doc = the desired minimal eigenvalue of Σ matrix, as a ratio to the max eigenvalue

beta_resampling_start

doc = the beta threshold to start the resampling procedure

w

cov = 0.0

uniform

rng

initialize (*self*, *application*)

Initialize me and my parts given an {application} context

update (*self*, *step*)

Push {step} forward along the annealing schedule

updateTemperature (*self*, *step*)

Generate the next temperature increment

computeCovariance (*self*, *step*)

Compute the parameter covariance Σ of the sample in {step}

Σ = c_m^2 sum_{i in samples} ilde{w}_{i} θ_i θ_i^T} - bar{θ} bar{θ}^T

where

bar{θ} = sum_{i in samples} ilde{w}_{i} θ_{i}

The covariance Σ gets used to build a proposal pdf for the posterior

rank (*self*, *step*)

Rebuild the sample and its statistics sorted by the likelihood of the parameter values

resampling (*self*, *step*)

Rebuild the sample and its statistics sorted by the likelihood of the parameter values

conditionCovariance (*self*, *Σ*)

Make sure the covariance matrix Σ is symmetric and positive definite

computeSampleMultiplicities (*self*, *step*)

Prepare a frequency vector for the new samples given the scaled data log-likelihood in {w} for this cooling step

buildHistogramRanges (*self*, *w*)

Build histogram bins based on the scaled data log-likelihood

altar.bayesian.CUDAAnnealing

Module Contents

Classes

class altar.bayesian.CUDAAnnealing.**CUDAAnnealing** (*annealer*, ***kwds*)

Bases: *altar.bayesian.AnnealingMethod.AnnealingMethod*

Implementation that takes advantage of CUDA on gpus to accelerate the computation

wid = 0

workers = 1

device

gstep

initialize (*self*, *application*)

initialize worker

cuInitialize (*self*, *application*)

Initialize the cuda worker

start (*self*, *annealer*)

Start the annealing process

altar.bayesian.Controller

Module Contents

Classes

class altar.bayesian.Controller.**Controller**

Bases: altar.protocol

The protocol that all AlTar controllers must implement

dispatcher

doc = the event dispatcher that activates the registered handlers

archiver

doc = the archiver of simulation state

posterior(*self*, *model*)

Sample the posterior distribution of the given {model}

initialize(*self*, *application*)

Initialize me and my parts given an {application} context

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.bayesian.CoolingStep

Module Contents

Classes

class altar.bayesian.CoolingStep.**CoolingStep**(*beta*, *theta*, *likelihoods*, *sigma=None*, ***kwds*)

Encapsulation of the state of the calculation at some particular β value

beta

theta

prior

data

posterior

sigma

mean

std

classmethod start(*cls*, *annealer*)

Build the first cooling step by asking {model} to produce a sample set from its initializing prior, compute the likelihood of this sample given the data, and compute a (perhaps trivial) posterior

classmethod allocate(*cls*, *annealer*)

classmethod alloc(*cls*, *samples*, *parameters*)

Allocate storage for the parts of a cooling step

clone(*self*)

Make a new step with a duplicate of my state

computePosterior(*self*)

Compute the posterior from prior, data, and beta

statistics(*self*)

Compute the statistics of samples :return:

print (*self*, *channel*, *indent=' ' * 2*)

Print info about this step

save_hdf5 (*self*, *path=None*, *iteration=None*, *psets=None*)

Save Coolinging Step to HDF5 file :param step altar.bayesian.CoolingStep: :param path altar.primitives.path:

Returns
None

load_hdf5 (*self*, *path=None*, *iteration=0*)

load CoolingStep from HDF5 file

altar.bayesian.Grid

Module Contents

Classes

class altar.bayesian.Grid.**Grid**(*name*, *locator*, ***kwds*)

Bases: altar.component

A δβ solver based on a naive grid search

tolerance

doc = the fractional tolerance for achieving convergence

maxiter

doc = the maximum number of iterations while looking for a δβ

cov

initialize (*self*, *application*, *scheduler*)

Initialize me and my parts given an {application} context and a {scheduler}

solve (*self*, *llk*, *weight*)

Compute the next temperature in the cooling schedule :param llk: data log-likelihood :param weight: the normalized weight :return: β, cov

altar.bayesian.H5Recorder

Module Contents

Classes

class altar.bayesian.H5Recorder.**H5Recorder**(*name*, *locator*, ***kwds*)

Bases: altar.component

H5Recorder stores the intermediate simulation state to HDF5 files

output_dir

doc = the directory to save results

output_freq

doc = the frequency to write step data to files

statistics

initialize(*self*, *application*)

Initialize me given an {application} context

record(*self*, *step*, *iteration*, *psets*, ***kwds*)

Record the final state of the calculation

recordstep(*self*, *step*, *stats*, *psets*)

Record step to file for ce

saveStats(*self*)

Save the statistics information to file

altar.bayesian.MPIAnnealing

Module Contents

Classes

class altar.bayesian.MPIAnnealing.**MPIAnnealing**(*annealer*, *worker*, *communicator=None*, ***kwds*)

Bases: *altar.bayesian.AnnealingMethod.AnnealingMethod*

A distributed implementation of the annealing method that uses MPI

manager = 0

worker

initialize(*self*, *application*)

Initialize me and my parts given an {application} context

start(*self*, *annealer*)

Start the annealing process

top(*self*, *annealer*)

Notification that we are at the beginning of a β update

cool(*self*, *annealer*)

Push my state forward along the cooling schedule

walk(*self*, *annealer*)

Explore configuration space by walking the Markov chains

resample(*self*, *annealer*, *statistics*)

Analyze the acceptance statistics and take the problem state to the end of the annealing step

archive(*self*, *annealer*, *scaling*, *stats*)

Notify archiver to record annealer information

bottom (*self*, *annealer*)
Notification that we are at the end of a β update

finish (*self*, *annealer*)
Shut down the annealing process

collect (*self*)
Assemble my global state

partition (*self*)
Distribute my global state

altar.bayesian.Metropolis

Module Contents

Classes

class altar.bayesian.Metropolis.**Metropolis**(*name*, *locator*, ***kwds*)
Bases: altar.component

The Metropolis algorithm as a sampler of the posterior distribution

scaling

doc = the parameter covariance Σ is scaled by the square of this

acceptanceWeight

doc = the weight of accepted samples during covariance rescaling

rejectionWeight

doc = the weight of rejected samples during covariance rescaling

steps = 1

uniform

uninormal

sigma_chol

dispatcher

initialize (*self*, *application*)
Initialize me and my parts given an {application} context

samplePosterior (*self*, *annealer*, *step*)
Sample the posterior distribution

resample (*self*, *annealer*, *statistics*)
Update my statistics based on the results of walking my Markov chains

prepareSamplingPDF (*self*, *annealer*, *step*)
Re-scale and decompose the parameter covariance matrix, in preparation for the Metropolis update

walkChains (*self*, *annealer*, *step*)

Run the Metropolis algorithm on the Markov chains

displace (*self*, *sample*)

Construct a set of displacement vectors for the random walk from a distribution with zero mean and my covariance

adjustCovarianceScaling (*self*, *accepted*, *rejected*, *unlikely*)

Compute a new value for the covariance sacling factor based on the acceptance/rejection ratio

altar.bayesian.Notifier

Module Contents

Classes

class altar.bayesian.Notifier.**Notifier**(***kwds*)

Bases: altar.component

A dispatcher of events generated during the annealing process

start = simulationStart

samplePosteriorStart = samplePosteriorStart

prepareSamplingPDFStart = prepareSamplingPDFStart

prepareSamplingPDFFinish = prepareSamplingPDFFinish

betaStart = betaStart

walkChainsStart = walkChainsStart

chainAdvanceStart = chainAdvanceStart

verifyStart = verifyStart

verifyFinish = verifyFinish

priorStart = priorStart

priorFinish = priorFinish

dataStart = dataStart

dataFinish = dataFinish

posteriorStart = posteriorStart

posteriorFinish = posteriorFinish

acceptStart = acceptStart

acceptFinish = acceptFinish

chainAdvanceFinish = chainAdvanceFinish

walkChainsFinish = walkChainsFinish

resampleStart = resampleStart

resampleFinish = resampleFinish

betaFinish = betaFinish

samplePosteriorFinish = samplePosteriorFinish

finish = simulationFinish

initialize(*self*, *application*)

Initialize me given an {application} context

register(*self*, *monitor*)

Enable {monitor} as an observer of simulation events

notify(*self*, *event*, *controller*)

Notify all handlers that are waiting for {event}

altar.bayesian.Profiler

Module Contents

Classes

class altar.bayesian.Profiler.**Profiler**(***kwds*)

Bases: altar.component

Profiler times the various simulation phases

seed

doc = a template for the filename with the timing results

default = prof-{{wid:05}}-{{beta:03}}x{{parameters:03}}x{{chains:06}}x{{steps:03}}.csv

pfs

initialize(*self*, *application*)

Initialize me given an {application} context

simulationStart(*self*, *controller*, ***kwds*)

Handler invoked when the simulation is about to start

samplePosteriorStart(*self*, *controller*, ***kwds*)

Handler invoked at the beginning of sampling the posterior

prepareSamplingPDFStart(*self*, *controller*, ***kwds*)

Handler invoked at the beginning of the preparation of the sampling PDF

prepareSamplingPDFFinish(*self*, *controller*, ***kwds*)

Handler invoked at the end of the preparation of the sampling PDF

betaStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of the beta step

walkChainsStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of the chain walk

chainAdvanceStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of a single step of chain walking

chainAdvanceFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of a single step of chain walking

verifyStart (*self*, *controller*, ***kwds*)
Handler invoked before we start verifying the generated sample

verifyFinish (*self*, *controller*, ***kwds*)
Handler invoked after we are done verifying the generated sample

priorStart (*self*, *controller*, ***kwds*)
Handler invoked before we compute the prior

priorFinish (*self*, *controller*, ***kwds*)
Handler invoked after we compute the prior

dataStart (*self*, *controller*, ***kwds*)
Handler invoked before we compute the data likelihood

dataFinish (*self*, *controller*, ***kwds*)
Handler invoked after we compute the data likelihood

posteriorStart (*self*, *controller*, ***kwds*)
Handler invoked before we assemble the posterior

posteriorFinish (*self*, *controller*, ***kwds*)
Handler invoked after we assemble the posterior

acceptStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of sample accept/reject

acceptFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of sample accept/reject

resampleStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of resampling

resampleFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of resampling

walkChainsFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of the chain walk

betaFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of the beta step

samplePosteriorFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of sampling the posterior

simulationFinish (*self*, *controller*, ***kwds*)
Handler invoked when the simulation is about to finish

save (*self*, *controller*)
Save the times collected by my timers

altar.bayesian.Recorder

Module Contents

Classes

class altar.bayesian.Recorder.**Recorder** (*name*, *locator*, ***kwds*)
Bases: altar.component

Recorder stores the intermediate simulation state in memory

output_dir

doc = the directory to save results

output_freq

doc = the frequency to write step data to files

statistics

initialize (*self*, *application*)
Initialize me given an {application} context

record (*self*, *step*, *iteration*, *psets*, ***kwds*)
Record the final state of the calculation

recordstep (*self*, *step*, *stats*, *psets*)
Record step to file for ce

saveStats (*self*)
Save the statistics information to file

altar.bayesian.Sampler

Module Contents

Classes

class altar.bayesian.Sampler.**Sampler**
Bases: altar.protocol

The protocol that all AlTar samplers must implement

initialize (*self*, *application*)
Initialize me and my parts given an {application} context

samplePosterior (*self*, *controller*, *step*)
Sample the posterior distribution

resample (*self*, *controller*, *statistics*)

Update my statistics based on the results of walking my Markov chains

classmethod pyre_default (*cls*, ***kwds*)

Supply a default implementation

altar.bayesian.Scheduler

Module Contents

Classes

class altar.bayesian.Scheduler.**Scheduler**

Bases: altar.protocol

The protocol that all AlTar schedulers must implement

initialize (*self*, *application*)

Initialize me and my parts given an {application} context

update (*self*, *step*)

Push {step} forward along the annealing schedule

updateTemperature (*self*, *step*)

Generate the next temperature increment

computeCovariance (*self*, *step*)

Compute the parameter covariance of the sample in the {step}

rank (*self*, *step*)

Rebuild the sample and its statistics sorted by the likelihood of the parameter values

classmethod pyre_default (*cls*, ***kwds*)

Supply a default implementation

altar.bayesian.SequentialAnnealing

Module Contents

Classes

class altar.bayesian.SequentialAnnealing.**SequentialAnnealing** (*annealer*, ***kwds*)

Bases: *altar.bayesian.AnnealingMethod.AnnealingMethod*

Implementation that assumes its state is the global state of the solver, and therefore it is able to compute the statistical properties of the sample distribution

wid = 0

workers = 1

start (*self*, *annealer*)

Start the annealing process

altar.bayesian.Solver

Module Contents

Classes

class altar.bayesian.Solver.**Solver**

Bases: altar.protocol

The protocol that all δβ solvers must implement

tolerance

doc = the fractional tolerance for achieving convergence

initialize(*self*, *application*, *scheduler*)

Initialize me and my parts given an {application} context and a {scheduler}

solve(*self*, *llk*, *weight*)

Compute the next temperature in the cooling schedule

classmethod pyre_default(*cls*, ***kwds*)

Provide a default implementation

altar.bayesian.ThreadedAnnealing

Module Contents

Classes

class altar.bayesian.ThreadedAnnealing.**ThreadedAnnealing**(*annealer*, ***kwds*)

Bases: *altar.bayesian.AnnealingMethod.AnnealingMethod*

Annealing method that uses threads on the local machine

Package Contents

Classes

Functions

class altar.bayesian.**controller**

Bases: altar.protocol

The protocol that all AlTar controllers must implement

dispatcher

doc = the event dispatcher that activates the registered handlers

archiver

doc = the archiver of simulation state

posterior(*self*, *model*)
Sample the posterior distribution of the given {model}

initialize(*self*, *application*)
Initialize me and my parts given an {application} context

classmethod pyre_default(*cls*, ***kwds*)
Supply a default implementation

class altar.bayesian.**sampler**
Bases: altar.protocol

The protocol that all AlTar samplers must implement

initialize(*self*, *application*)
Initialize me and my parts given an {application} context

samplePosterior(*self*, *controller*, *step*)
Sample the posterior distribution

resample(*self*, *controller*, *statistics*)
Update my statistics based on the results of walking my Markov chains

classmethod pyre_default(*cls*, ***kwds*)
Supply a default implementation

class altar.bayesian.**scheduler**
Bases: altar.protocol

The protocol that all AlTar schedulers must implement

initialize(*self*, *application*)
Initialize me and my parts given an {application} context

update(*self*, *step*)
Push {step} forward along the annealing schedule

updateTemperature(*self*, *step*)
Generate the next temperature increment

computeCovariance(*self*, *step*)
Compute the parameter covariance of the sample in the {step}

rank(*self*, *step*)
Rebuild the sample and its statistics sorted by the likelihood of the parameter values

classmethod pyre_default(*cls*, ***kwds*)
Supply a default implementation

class altar.bayesian.**monitor**
Bases: altar.protocol

The protocol that all AlTar simulation monitors must implement

Monitors respond to simulation events by generating user diagnostics to report progress

initialize(*self*, *application*)
Initialize me given an {application} context

classmethod pyre_default(*cls*, ***kwds*)
Supply a default implementation

class altar.bayesian.**archiver**
Bases: altar.protocol

The protocol that all AlTar simulation archivers must implement

Archivers persist intermediate simulation state and can be used to restart a simulation

initialize(*self*, *application*)
Initialize me given an {application} context

record(*self*, *step*)
Record the final state of the simulation

classmethod pyre_default(*cls*, ***kwds*)
Supply a default implementation

class altar.bayesian.**solver**
Bases: altar.protocol

The protocol that all δβ solvers must implement

tolerance

doc = the fractional tolerance for achieving convergence

initialize(*self*, *application*, *scheduler*)
Initialize me and my parts given an {application} context and a {scheduler}

solve(*self*, *llk*, *weight*)
Compute the next temperature in the cooling schedule

classmethod pyre_default(*cls*, ***kwds*)
Provide a default implementation

altar.bayesian.**annealer**()

altar.bayesian.**cov**()

altar.bayesian.**brent**()

altar.bayesian.**grid**()

altar.bayesian.**metropolis**()

altar.bayesian.**profiler**()

altar.bayesian.**recorder**()

7.1.1.3 altar.cuda

Subpackages

altar.cuda.bayesian

Submodules

altar.cuda.bayesian.cudaAdaptiveMetropolis

Module Contents

Classes

class altar.cuda.bayesian.cudaAdaptiveMetropolis.**cudaAdaptiveMetropolis**(*name*, *locator*, ***kwds*)

Bases: altar.component

The Adaptive Metropolis algorithm from Thomas Catanach Compared with traditional MCMC, there are two modifications: 1. After certain steps (corr_check_steps), the correlation between the current and the starting samples is computed. If the correlation is less than a threshold value (target_correlation), i.e., the chains are effectively de-correlated, the MCMC stops. max_mc_steps provides a maximum MCMC step 2. The scaling factor (scaling) targets an optimal acceptance rate

scaling

doc = scaling factor σ for Gaussian proposal ~ N(0, σ^2 Σ), initial value 2.38/sqrt(N_d)

scaling_min

doc = the minimum value of the scaling factor

scaling_max

doc = the maximum value of the scaling factor

parameters

doc = total number of parameters N_d

target_acceptance_rate

doc = the targeted acceptance rate

gain

doc = Feedback gain constant

max_mc_steps

doc = the maximum Monte-Carlo steps for one beta step

min_mc_steps

doc = the minimum Monte-Carlo steps for one beta step

max_mc_steps_stage2

doc = the maximum Monte-Carlo steps at stage 2, or beta> beta_stage2

beta_stage2

doc = beta value to start stage 2, i.e., to use a different max_mc_steps

corr_check_steps

doc = the Monte-Carlo steps to compute the de

target_correlation

doc = the threshold of correlation to stop the chain

mcsteps = 1

dispatcher

ginit = False

gstep

gcandidate

gproposal

gsigma_chol

gvalid_indices

gvalid_samples

ginvalid_flags

gacceptance_flags

precision

gdice

curng

initialize(*self*, *application*)

Initialize me and my parts given an {application} context

cuInitialize(*self*, *application*)

samplePosterior(*self*, *annealer*, *step*)

Sample the posterior distribution :param annealer - the controller: :param step - cpu CoolingStep:

Returns

statistics (accepted/rejected/invalid) or (accepted/unlikely/rejected)

resample(*self*, *annealer*, *statistics*)

Update my statistics based on the results of walking my Markov chains

prepareSamplingPDF (*self*, *annealer*, *step*)

Re-scale and decompose the parameter covariance matrix, in preparation for the Metropolis update

finishSamplingPDF (*self*, *step*)

procedures after sampling, e.g, copy data back to cpu

walkChains (*self*, *annealer*, *step*)

Run the Metropolis algorithm on the Markov chains :param annealer: cudaAnnealer :param step: cudaCoolingStep

Returns

statistics = (accepted, rejected, unlikely)

displace (*self*, *displacement*)

Construct a set of displacement vectors for the random walk from a distribution with zero mean and my covariance

adjustCovarianceScaling (*self*, *accepted*, *invalid*, *rejected*)

Compute a new value for the covariance sacling factor based on the acceptance/rejection ratio

allocateGPUData (*self*, *samples*, *parameters*)

initialize gpu work data

static gainFunction (*x*)

Compute the optimal gain constant from a target acceptranceRate {x}

altar.cuda.bayesian.cudaCoolingStep

Module Contents

Classes

class altar.cuda.bayesian.cudaCoolingStep.**cudaCoolingStep** (*beta*, *theta*, *likelihoods*, ***kwds*)

Encapsulation of the state of the calculation at some particular β value

beta

theta

prior

data

posterior

precision

classmethod start (*cls*, *annealer*)

Build the first cooling step by asking {model} to produce a sample set from its initializing prior, compute the likelihood of this sample given the data, and compute a (perhaps trivial) posterior

classmethod alloc (*cls*, *samples*, *parameters*, *dtype*)

Allocate storage for the parts of a cooling step

clone (*self*)

Make a new step with a duplicate of my state

computePosterior (*self*, *batch=None*)

(Re-)Compute the posterior from prior, data, and (updated) beta

copyFromCPU (*self*, *step*)

Copy cpu step to gpu step

copyToCPU (*self*, *step*)

copy gpu step to cpu step

altar.cuda.bayesian.cudaMetropolis

Module Contents

Classes

class altar.cuda.bayesian.cudaMetropolis.**cudaMetropolis** (*name*, *locator*, ***kwds*)

Bases: altar.component

The Metropolis algorithm as a sampler of the posterior distribution

scaling

doc = the parameter covariance Σ is scaled by the square of this

acceptanceWeight

doc = the weight of accepted samples during covariance rescaling

rejectionWeight

doc = the weight of rejected samples during covariance rescaling

useFixedScaling

doc = whether to use a fixed scaling

scalingMin

doc = the minimum value of the scaling factor

scalingMax

doc = the maximum value of the scaling factor

mcsteps = 1

dispatcher

ginit = False

gstep

gcandidate

gproposal

gsigma_chol

gvalid_indices

gvalid_samples

ginvalid_flags

gacceptance_flags

precision

gdice

curng

initialize (*self*, *application*)

Initialize me and my parts given an {application} context

cuInitialize (*self*, *application*)

samplePosterior (*self*, *annealer*, *step*)

Sample the posterior distribution :param annealer - the controller: :param step - cpu CoolingStep:

Returns

statistics (accepted/rejected/invalid) or (accepted/unlikely/rejected)

resample (*self*, *annealer*, *statistics*)

Update my statistics based on the results of walking my Markov chains

prepareSamplingPDF (*self*, *annealer*, *step*)

Re-scale and decompose the parameter covariance matrix, in preparation for the Metropolis update

finishSamplingPDF (*self*, *step*)

procedures after sampling, e.g, copy data back to cpu

walkChains (*self*, *annealer*, *step*)

Run the Metropolis algorithm on the Markov chains :param annealer: cudaAnnealer :param step: cudaCoolingStep

Returns

statistics = (accepted, rejected, unlikely)

displace (*self*, *displacement*)

Construct a set of displacement vectors for the random walk from a distribution with zero mean and my covariance

adjustCovarianceScaling (*self*, *accepted*, *invalid*, *rejected*)

Compute a new value for the covariance sacling factor based on the acceptance/rejection ratio

allocateGPUData (*self*, *samples*, *parameters*)

initialize gpu work data

altar.cuda.bayesian.cudaMetropolisVaryingSteps

Module Contents

Classes

class altar.cuda.bayesian.cudaMetropolisVaryingSteps.**cudaMetropolisVaryingSteps**(*name*, *locator*, ***kwds*)

Bases: altar.component

The Metropolis algorithm as a sampler of the posterior distribution

scaling

doc = the parameter covariance Σ is scaled by the square of this

acceptanceWeight

doc = the weight of accepted samples during covariance rescaling

rejectionWeight

doc = the weight of rejected samples during covariance rescaling

max_mc_steps

doc = the maximum Monte-Carlo steps for one beta step

corr_check_steps

doc = the Monte-Carlo steps to compute the de

target_correlation

doc = the threshold of correlation to stop the chain

mcsteps = 1

dispatcher

ginit = False

gstep

gcandidate

gproposal

gsigma_chol

gvalid_indices

gvalid_samples

ginvalid_flags

gacceptance_flags

precision

gdice

curng

initialize (*self*, *application*)

Initialize me and my parts given an {application} context

cuInitialize (*self*, *application*)

samplePosterior (*self*, *annealer*, *step*)

Sample the posterior distribution :param annealer - the controller: :param step - cpu CoolingStep:

Returns

statistics (accepted/rejected/invalid) or (accepted/unlikely/rejected)

resample (*self*, *annealer*, *statistics*)

Update my statistics based on the results of walking my Markov chains

prepareSamplingPDF (*self*, *annealer*, *step*)

Re-scale and decompose the parameter covariance matrix, in preparation for the Metropolis update

finishSamplingPDF (*self*, *step*)

procedures after sampling, e.g, copy data back to cpu

walkChains (*self*, *annealer*, *step*)

Run the Metropolis algorithm on the Markov chains :param annealer: cudaAnnealer :param step: cudaCoolingStep

Returns

statistics = (accepted, rejected, unlikely)

displace (*self*, *displacement*)

Construct a set of displacement vectors for the random walk from a distribution with zero mean and my covariance

adjustCovarianceScaling (*self*, *accepted*, *invalid*, *rejected*)

Compute a new value for the covariance sacling factor based on the acceptance/rejection ratio

allocateGPUData (*self*, *samples*, *parameters*)

initialize gpu work data

Package Contents

Classes

Functions

class altar.cuda.bayesian.**controller**

Bases: altar.protocol

The protocol that all AlTar controllers must implement

dispatcher

doc = the event dispatcher that activates the registered handlers

archiver

doc = the archiver of simulation state

posterior(*self*, *model*)

Sample the posterior distribution of the given {model}

initialize(*self*, *application*)

Initialize me and my parts given an {application} context

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.cuda.bayesian.**sampler**

Bases: altar.protocol

The protocol that all AlTar samplers must implement

initialize(*self*, *application*)

Initialize me and my parts given an {application} context

samplePosterior(*self*, *controller*, *step*)

Sample the posterior distribution

resample(*self*, *controller*, *statistics*)

Update my statistics based on the results of walking my Markov chains

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.cuda.bayesian.**scheduler**

Bases: altar.protocol

The protocol that all AlTar schedulers must implement

initialize(*self*, *application*)

Initialize me and my parts given an {application} context

update(*self*, *step*)

Push {step} forward along the annealing schedule

updateTemperature(*self*, *step*)

Generate the next temperature increment

computeCovariance(*self*, *step*)

Compute the parameter covariance of the sample in the {step}

rank(*self*, *step*)

Rebuild the sample and its statistics sorted by the likelihood of the parameter values

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.cuda.bayesian.**metropolis**()

altar.cuda.bayesian.**metropolisvaryingsteps**()

altar.cuda.bayesian.**adaptivemetropolis**()

altar.cuda.data

Submodules

altar.cuda.data.cudaDataL2

Module Contents

Classes

class altar.cuda.data.cudaDataL2.**cudaDataL2**(*name*, *locator*, ***kwds*)

Bases: *altar.data.DataL2.DataL2*

The observed data with L2 norm

data_file

doc = the name of the file with the observations

observations

doc = the number of observed data

cd_file

doc = the name of the file with the data covariance matrix

cd_std

doc = the constant covariance for data, sigma^2

dtype_cd

doc = the data type (float32/64) for Cd computations if different from others

norm

default

doc = l2 norm for calculating likelihood

merge_cd_to_data = True

normalization = 0

precision

gdataObsBatch

gcd_inv

initialize (*self*, *application*)

Initialize data obs from model

cuEvalLikelihood (*self*, *prediction*, *likelihood*, *residual=True*, *batch=None*)

compute the datalikelihood for prediction :param prediction: (samples x observations) input of predicted data :param likelihood: (samples) pre-allocated likelihood/norm :param residual: whether prediction is already subtracted by observed data :param batch: number of (first few) samples to be computed :return: likelihood

release_cd (*self*)

release gcd_inv

loadFile (*self*, *filename*, *shape*, *dataset=None*, *dtype=None*)

Load an input file to a numpy array (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
output numpy.array

initializeCovariance (*self*)

initialize gpu data and data covariance

updateCovariance (*self*, *cp=None*)

Update the data covariance C_chi = Cd + Cp :param cp: cuda matrix with shape(obs, obs), data covariance due to model uncertainty :return:

checkPositiveDefiniteness (*self*, *matrix*, *name=None*)

Check positive definiteness of a GPU matrix :param matrix: a real symmetric (GPU) matrix :return: true or false

mergeCdtoData (*self*, *cd_inv*, *data*)

Merge the data covariance matrix to observed data :param cd_inv: the inverse of covariance matrix in Cholesky-decomposed form, with Lower matrix filled :param data: raw observed data :return: cd_inv*data, a cuda vector

Package Contents

Classes

Functions

class altar.cuda.data.**data**

Bases: altar.protocol

The protocol that all AlTar norms must satify

initialize(*self*, *application*)

initialize data

classmethod pyre_default(*cls*, ***kwds*)

Provide a default norm in case the user hasn't selected one

altar.cuda.data.**data2**()

altar.cuda.distributions

Submodules

altar.cuda.distributions.cudaDistribution

Module Contents

Classes

class altar.cuda.distributions.cudaDistribution.**cudaDistribution**(*name*, *locator*, ***kwds*)

Bases: *altar.distributions.Base.Base*

The base class for probability distributions

parameters

doc = the number of model parameters that belong to me

offset

doc = the starting point of my parameters in the overall model state

device

idx_range

precision

initialize(*self*, *rng*)

Initialize with the given random number generator

verify(*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

cuInitialize(*self*, *application*)

cuda specific initialization

cuInitSample(*self*, *theta*)

cuda process to initialize random samples

cuVerify (*self*, *theta*, *mask*)

cuda process to verify the validity of samples

cuEvalPrior (*self*, *theta*, *prior*)

cuda process to compute the prior

altar.cuda.distributions.cudaGaussian

Module Contents

Classes

class altar.cuda.distributions.cudaGaussian.**cudaGaussian** (*name*, *locator*, ***kwds*)

Bases: *altar.cuda.distributions.cudaDistribution.cudaDistribution*

The cuda gaussian probability distribution

mean

doc = the mean value

sigma

doc = the standard deviation

cuInitSample (*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

cuVerify (*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones :param theta cuArray: :type theta cuArray: samples x total_parameters

cuEvalPrior (*self*, *theta*, *prior*, *batch*)

Fill my portion of {likelihood} with the likelihoods of the samples in {theta}

altar.cuda.distributions.cudaPreset

Module Contents

Classes

class altar.cuda.distributions.cudaPreset.**cudaPreset**

Bases: *altar.cuda.distributions.cudaDistribution.cudaDistribution*

The cuda preset distribution - initialize samples from a file Note that a preset distribution cannot be used for prior_run.

input_file

doc = input file in hdf5 format

dataset

doc = the name of dataset in hdf5

rank = 0

precision

ifs

error

initialize (*self*, *application*)

Initialize with the given random number generator

cuInitialize (*self*, *application*)

cuda initialize distribution :param application: :return:

cuInitSample (*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

_loadhdf5 (*self*, *theta*)

load from hdf5 file

altar.cuda.distributions.cudaTGaussian

Module Contents

Classes

class altar.cuda.distributions.cudaTGaussian.**cudaTGaussian**

Bases: *altar.cuda.distributions.cudaDistribution.cudaDistribution*

The cuda gaussian probability distribution

mean

doc = the mean value

sigma

doc = the standard deviation

support

doc = the support interval of the truncated gaussian distribution

cuInitSample (*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

cuVerify (*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones :param theta cuArray: :type theta cuArray: samples x total_parameters

cuEvalPrior (*self*, *theta*, *prior*, *batch*)

Fill my portion of {likelihood} with the likelihoods of the samples in {theta}

altar.cuda.distributions.cudaUniform

Module Contents

Classes

class altar.cuda.distributions.cudaUniform.**cudaUniform**(*name*, *locator*, ***kwds*)

Bases: *altar.cuda.distributions.cudaDistribution.cudaDistribution*

The cuda uniform probability distribution

support

doc = the support interval of the prior distribution

cuInitSample(*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

cuVerify(*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones :param theta cuArray: :type theta cuArray: samples x total_parameters

cuEvalPrior(*self*, *theta*, *prior*, *batch*)

Fill my portion of {likelihood} with the likelihoods of the samples in {theta}

Package Contents

Classes

Functions

class altar.cuda.distributions.**distribution**

Bases: altar.protocol

The protocol that all AlTar probability distributions must satisfy

parameters

doc = the number of model parameters that i take care of

offset

doc = the starting point of my parameters in the overall model state

initialize(*self*, ***kwds*)

Initialize with the given random number generator

initializeSample(*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

priorLikelihood(*self*, *theta*, *prior*)

Fill my portion of {prior} with the likelihoods of the samples in {theta}

verify(*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

sample(*self*)

Sample the distribution using a random number generator

density(*self*, *x*)

Compute the probability density of the distribution at {x}

vector(*self*, *vector*)

Fill {vector} with random values

matrix(*self*, *matrix*)

Fill {matrix} with random values

classmethod pyre_default(*cls*)

Supply a default implementation

class altar.cuda.distributions.**cudaDistribution**(*name*, *locator*, ***kwds*)

Bases: *altar.distributions.Base.Base*

The base class for probability distributions

parameters

doc = the number of model parameters that belong to me

offset

doc = the starting point of my parameters in the overall model state

device

idx_range

precision

initialize(*self*, *rng*)

Initialize with the given random number generator

verify(*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

cuInitialize(*self*, *application*)

cuda specific initialization

cuInitSample(*self*, *theta*)

cuda process to initialize random samples

cuVerify(*self*, *theta*, *mask*)

cuda process to verify the validity of samples

cuEvalPrior(*self*, *theta*, *prior*)

cuda process to compute the prior

altar.cuda.distributions.**uniform**()

altar.cuda.distributions.**gaussian**()

altar.cuda.distributions.**tgaussian**()

altar.cuda.distributions.**preset**()

altar.cuda.ext

altar.cuda.models

Submodules

altar.cuda.models.cudaBayesian

Module Contents

Classes

class altar.cuda.models.cudaBayesian.**cudaBayesian**(*name*, *locator*, ***kwds*)

Bases: *altar.models.Bayesian.Bayesian*

The base class of AlTar models that are compatible with Bayesian explorations

parameters

doc = the number of model degrees of freedom

cascaded

doc = whether the model is cascaded (annealing temperature is fixed at 1)

embedded

doc = whether the model is embedded in an ensemble of models

psets_list

doc = list of parameter sets, used to set orders

psets

default

doc = an ensemble of parameter sets in the model

dataobs

default

doc = observed data

case

doc = the directory with the input files

idx_map

default

doc = the indices for model parameters in whole theta set

return_residual

doc = the forward model returns residual(True) or prediction(False)

forwardonly

doc = whether to run the simulation or the forward problem only

theta_input

doc = the theta input file with a vector of parameters

theta_dataset

doc = the name/path of the theta dataset in h5 file

forward_output

dpc = the name/path of the file to save forward problem results

observations

device

precision

ifs

gidx_map

gtheta

initialize(*self*, *application*)

Initialize the state of the model given an {application} context

cuInitialize(*self*, *application*)

cuda interface

posterior(*self*, *application*)

Sample my posterior distribution

cuInitSample(*self*, *theta*)

Fill {theta} with an initial random sample from my prior distribution.

cuVerify(*self*, *theta*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

cuEvalPrior(*self*, *theta*, *prior*, *batch*)

Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

cuEvalLikelihood(*self*, *theta*, *likelihood*, *batch*)

calculate data likelihood and add it to step.prior or step.data

cuEvalPosterior(*self*, *step*, *batch*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods(*self*, *annealer*, *step*, *batch=None*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

updateModel(*self*, *annealer*)

Update Model parameters if needed :param annealer: :return: default is False

mountInputDataspace(*self*, *pfs*)

Mount the directory with my input files

loadFile(*self*, *filename*, *shape=None*, *dataset=None*, *dtype=None*)

Load an input file to a numpy array (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
output numpy.array

loadFileToGPU(*self*, *filename*, *shape=None*, *dataset=None*, *out=None*, *dtype=None*)

Load an input file to a gpu (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
out altar.cuda.matrix/vector

restricted(*self*, *theta*, *batch*)

extract theta which contains model's own parameters

forwardProblem(*self*, *application*, *theta=None*)
Perform the forward modeling with given {theta}

altar.cuda.models.cudaBayesianEnsemble

Module Contents

Classes

class altar.cuda.models.cudaBayesianEnsemble.**cudaBayesianEnsemble**(*name*, *locator*, ***kwds*)

Bases: *altar.models.Bayesian.Bayesian*

A collection of AlTar models that comprise a single model

models

doc = the collection of models in this ensemble

parameters

doc = the number of model degrees of freedom

psets_list

doc = list of parameter sets, used to set orders

psets

doc = an ensemble of parameter sets in the model

case

doc = the directory with the input files

forwardonly

doc = whether to run the simulation or the forward problem only

theta_input

doc = the theta input file with a vector of parameters

theta_dataset

doc = the name/path of the theta dataset in h5 file

forward_output

doc = the name/path of the file to save forward problem results

datallk

initialize(*self*, *application*)
Initialize the state of the model given an {application} context

cuInitialize(*self*, *application*)

cuda initialization

posterior(*self*, *application*)

Sample my posterior distribution

cuInitSample(*self*, *theta*)

Fill {theta} with an initial random sample from my prior distribution.

cuVerify(*self*, *theta*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

cuEvalPrior(*self*, *theta*, *prior*, *batch*)

Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

cuEvalLikelihood(*self*, *step*, *batch*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

cuEvalPosterior(*self*, *step*, *batch*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

updateModel(*self*, *annealer*)

Update model parameters if needed :param annealer: :return:

likelihoods(*self*, *annealer*, *step*, *batch*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

mountInputDataspace(*self*, *pfs*)

Mount the directory with my input files

loadFile(*self*, *filename*, *shape=None*, *dataset=None*, *dtype=None*)

Load an input file to a numpy array (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
output numpy.array

loadFileToGPU(*self*, *filename*, *shape=None*, *dataset=None*, *out=None*, *dtype=None*)

Load an input file to a gpu (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
out altar.cuda.matrix/vector

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

altar.cuda.models.cudaParameterEnsemble

Module Contents

Classes

class altar.cuda.models.cudaParameterEnsemble.**cudaParameterEnsemble**

Bases: altar.cuda.models.cudaParameter.cudaParameter

An Ensemble of parameter sets

psets

doc = an ensemble of parameter sets in the model

initialize(*self*, *application*)

Initialize my distributions

cuInitialize(*self*, *application*)

cuda initialize

cuInitSample(*self*, *theta*, *batch=None*)

Fill {theta} with an initial random sample from my prior distribution.

cuEvalPrior(*self*, *theta*, *prior*, *batch=None*)

Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

cuVerify(*self*, *theta*, *mask*, *batch=None*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

altar.cuda.models.cudaParameterSet

Module Contents

Classes

class altar.cuda.models.cudaParameterSet.**cudaParameterSet**(*name*, *locator*, ***kwds*)

Bases: *altar.models.Contiguous.Contiguous*

A contiguous parameter set

count

doc = the number of parameters in this set

prior

doc = the prior distribution

prep

doc = the distribution to use to initialize this parameter set

offset = 0

cuInitialize(*self*, *application*)

cuda initialization

cuInitSample(*self*, *theta*, *batch=None*)

Fill {theta} with an initial random sample from my prior distribution.

cuEvalPrior(*self*, *theta*, *prior*, *batch=None*)

Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

cuVerify(*self*, *theta*, *mask*, *batch=None*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

cuRestrict(*self*, *theta*)

Return my portion of the sample matrix {theta}

Package Contents

Classes

Functions

class altar.cuda.models.**model**

Bases: altar.protocol

The protocol that all AlTar models must implement

posterior(*self*, *application*)

Sample my posterior distribution

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

posteriorLikelihood(*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods(*self*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

top(*self*, *annealer*)

Notification that a β step is about to start

bottom(*self*, *annealer*)

Notification that a β step just ended

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.cuda.models.**parameters**

Bases: altar.protocol

The protocol that all AlTar parameter sets must implement

count

doc = the number of parameters in this set

prior

doc = the prior distribution

prep

doc = the distribution to use to initialize this parameter set

initialize(*self*, *model*, *offset*)

Initialize the parameter set given the {model} that owns it

initializeSample(*self*, *theta*)

Fill {theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *theta*, *priorLLK*)

Fill {priorLLK} with the likelihoods of the samples in {theta} in my prior distribution

verify(*self*, *theta*, *mask*)

Check whether the samples in {theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.cuda.models.**bayesian**()

altar.cuda.models.**bayesianensemble**()

altar.cuda.models.**parameterset**()

altar.cuda.norms

Submodules

altar.cuda.norms.cudaL2

Module Contents

Classes

class altar.cuda.norms.cudaL2.**cudaL2**(*name*, *locator*, ***kwds*)

Bases: *altar.norms.L2.L2*

The L2 norm

cuEval(*self*, *data*, *out=None*, *batch=None*, *cdinv=None*)

Compute the L2 norm of the given data ||x|| :param data - matrix: :type data - matrix: samples x observations :param batch - number of samples to be computed: :type batch - number of samples to be computed: first rows :param cdinv - inverse covariance matrix: :type cdinv - inverse covariance matrix: observations x observations) in its Cholesky decomposed form (Upper Triangle

Returns
out - norm vector (samples)

cuEvalLikelihood(*self*, *data*, *constant=0.0*, *out=None*, *batch=None*, *cdinv=None*)

Compute the L2 norm data likelihood of the given data const - ||x||^2/2 :param data - matrix: :type data - matrix: samples x observations :param batch - number of samples to be computed: :type batch - number of samples to be computed: first rows :param constant - normalization constant: :param cdinv - inverse covariance matrix: :type cdinv - inverse covariance matrix: observations x observations) in its Cholesky decomposed form (Upper Triangle

Returns
out - data likelihood vector (samples)

Package Contents

Classes

Functions

class altar.cuda.norms.**norm**

Bases: altar.protocol

The protocol that all AlTar norms must satify

eval(*self*, *v*, ***kwds*)

Compute the L2 norm of the given vector

classmethod pyre_default(*cls*, ***kwds*)

Provide a default norm in case the user hasn't selected one

altar.cuda.norms.**l2**()

Package Contents

Functions

altar.cuda.**get_current_device**()

Return current cuda device

altar.cuda.**use_device**(*id*)

Set current device to device with id

altar.cuda.**curand_generator**()

altar.cuda.**cublas_handle**()

altar.cuda.**copyright**()

Return the altar copyright note

altar.cuda.**license**()

Print the altar license

altar.cuda.**version**()

Return the altar version

altar.cuda.**credits**()

Print the acknowledgments

7.1.1.4 altar.data

Submodules

altar.data.DataL2

Module Contents

Classes

class altar.data.DataL2.**DataL2**(*name*, *locator*, ***kwds*)

Bases: altar.component

The observed data with L2 norm

data_file

doc = the name of the file with the observations

observations

doc = the number of observed data

cd_file

doc = the name of the file with the data covariance matrix

cd_std

doc = the constant covariance for data

merge_cd_with_data

doc = whether to merge cd with data

norm

default

doc = the norm to use when computing the data log likelihood

normalization = 0

ifs

samples

dataobs

dataobs_batch

cd

cd_inv

error

initialize (*self*, *application*)

Initialize data obs from model

evalLikelihood (*self*, *prediction*, *likelihood*, *residual=True*, *batch=None*)

compute the datalikelihood for prediction (samples x observations)

dataobsBatch (*self*)

Get a batch of duplicated dataobs

loadData (*self*)

load data and covariance

initializeCovariance (*self*, *cd*)

For a given data covariance cd, compute L2 likelihood normalization, inverse of cd in Cholesky decomposed form, and merge cd with data observation, d-> L*d with cd^{-1} = L L* :param cd: :return:

updateCovariance (*self*, *cp*)

Update data covariance with cp, cd -> cd + cp :param cp: a matrix with shape (obs, obs) :return:

computeNormalization (*self*, *observations*, *cd*)

Compute the normalization of the L2 norm

computeCovarianceInverse (*self*, *cd*)

Compute the inverse of the data covariance matrix

altar.data.DataObs

Module Contents

Classes

class altar.data.DataObs.**DataObs**

Bases: altar.protocol

The protocol that all AlTar norms must satify

initialize (*self*, *application*)

initialize data

classmethod pyre_default (*cls*, ***kwds*)

Provide a default norm in case the user hasn't selected one

Package Contents

Classes

Functions

class altar.data.**data**

Bases: altar.protocol

The protocol that all AlTar norms must satify

initialize(*self*, *application*)
initialize data

classmethod pyre_default(*cls*, ***kwds*)
Provide a default norm in case the user hasn't selected one

class altar.data.**DataL2**(*name*, *locator*, ***kwds*)
Bases: altar.component

The observed data with L2 norm

data_file

doc = the name of the file with the observations

observations

doc = the number of observed data

cd_file

doc = the name of the file with the data covariance matrix

cd_std

doc = the constant covariance for data

merge_cd_with_data

doc = whether to merge cd with data

norm

default

doc = the norm to use when computing the data log likelihood

normalization = 0

ifs

samples

dataobs

dataobs_batch

cd

cd_inv

error

initialize(*self*, *application*)
Initialize data obs from model

evalLikelihood(*self*, *prediction*, *likelihood*, *residual=True*, *batch=None*)
compute the datalikelihood for prediction (samples x observations)

dataobsBatch(*self*)
Get a batch of duplicated dataobs

loadData (*self*)

load data and covariance

initializeCovariance (*self*, *cd*)

For a given data covariance cd, compute L2 likelihood normalization, inverse of cd in Cholesky decomposed form, and merge cd with data observation, d-> L*d with cd^{-1} = L L* :param cd: :return:

updateCovariance (*self*, *cp*)

Update data covariance with cp, cd -> cd + cp :param cp: a matrix with shape (obs, obs) :return:

computeNormalization (*self*, *observations*, *cd*)

Compute the normalization of the L2 norm

computeCovarianceInverse (*self*, *cd*)

Compute the inverse of the data covariance matrix

altar.data.**datal2**()

7.1.1.5 altar.distributions

Submodules

altar.distributions.Base

Module Contents

Classes

class altar.distributions.Base.**Base** (*name*, *locator*, ***kwds*)

Bases: altar.component

The base class for probability distributions

parameters

doc = the number of model parameters that belong to me

offset

doc = the starting point of my parameters in the overall model state

pdf

abstract initialize (*self*, *rng*)

Initialize with the given random number generator

initializeSample (*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

priorLikelihood (*self*, *theta*, *likelihood*)

Fill my portion of {likelihood} with the likelihoods of the samples in {theta}

abstract verify (*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

sample (*self*)
Sample the distribution using a random number generator

density (*self*, *x*)
Compute the probability density of the distribution at {x}

vector (*self*, *vector*)
Fill {vector} with random values

matrix (*self*, *matrix*)
Fill {matrix} with random values

restrict (*self*, *theta*)
Return my portion of the {theta}

altar.distributions.Distribution

Module Contents

Classes

class altar.distributions.Distribution.**Distribution**
Bases: altar.protocol

The protocol that all AlTar probability distributions must satisfy

parameters

doc = the number of model parameters that i take care of

offset

doc = the starting point of my parameters in the overall model state

initialize (*self*, ***kwds*)
Initialize with the given random number generator

initializeSample (*self*, *theta*)
Fill my portion of {theta} with initial random values from my distribution.

priorLikelihood (*self*, *theta*, *prior*)
Fill my portion of {prior} with the likelihoods of the samples in {theta}

verify (*self*, *theta*, *mask*)
Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

sample (*self*)
Sample the distribution using a random number generator

density (*self*, *x*)
Compute the probability density of the distribution at {x}

vector (*self*, *vector*)
Fill {vector} with random values

matrix (*self*, *matrix*)

Fill {matrix} with random values

classmethod pyre_default (*cls*)

Supply a default implementation

altar.distributions.Gaussian

Module Contents

Classes

class altar.distributions.Gaussian.**Gaussian** (*name*, *locator*, ***kwds*)

Bases: *altar.distributions.Base.Base*

The Gaussian probability distribution

mean

doc = the mean value of the distribution

sigma

doc = the standard deviation of the distribution

initialize (*self*, *rng*)

Initialize with the given random number generator

verify (*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

altar.distributions.Uniform

Module Contents

Classes

class altar.distributions.Uniform.**Uniform** (*name*, *locator*, ***kwds*)

Bases: *altar.distributions.Base.Base*

The uniform probability distribution

support

doc = the support interval of the prior distribution

initialize (*self*, *rng*)

Initialize with the given random number generator

verify (*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

altar.distributions.UnitGaussian

Module Contents

Classes

class altar.distributions.UnitGaussian.**UnitGaussian**(*name*, *locator*, ***kwds*)

Bases: *altar.distributions.Base.Base*

Special case of the Gaussian probability distribution with σ = 1

initialize(*self*, *rng*)

Initialize with the given random number generator

verify(*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

Package Contents

Classes

Functions

class altar.distributions.**distribution**

Bases: altar.protocol

The protocol that all AlTar probability distributions must satisfy

parameters

doc = the number of model parameters that i take care of

offset

doc = the starting point of my parameters in the overall model state

initialize(*self*, ***kwds*)

Initialize with the given random number generator

initializeSample(*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

priorLikelihood(*self*, *theta*, *prior*)

Fill my portion of {prior} with the likelihoods of the samples in {theta}

verify(*self*, *theta*, *mask*)

Check whether my portion of the samples in {theta} are consistent with my constraints, and update {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

sample(*self*)

Sample the distribution using a random number generator

density(*self*, *x*)

Compute the probability density of the distribution at {x}

vector(*self*, *vector*)
Fill {vector} with random values

matrix(*self*, *matrix*)
Fill {matrix} with random values

classmethod pyre_default(*cls*)
Supply a default implementation

altar.distributions.**uniform**()

altar.distributions.**gaussian**()

altar.distributions.**ugaussian**()

7.1.1.6 altar.ext

7.1.1.7 altar.models

Subpackages

altar.models.cdm

Subpackages

altar.models.cdm.ext

Submodules

altar.models.cdm.CDM

Module Contents

Classes

class altar.models.cdm.CDM.**CDM**(*name*, *locator*, ***kwds*)
Bases: *altar.models.bayesian*

An implementation of the Compound Dislocation Model, Nikhoo et al. [2017]

psets

doc = the model parameter meta-data

observations

doc = the number of model degrees of freedom

norm

default

doc = the norm to use when computing the data log likelihood

```
case
doc = the directory with the input files
displacements
doc = the name of the file with the displacements
covariance
doc = the name of the file with the data covariance
nu
doc = the Poisson ratio
mode
doc = the implementation strategy
validators
parameters = 0
strategy
ifs
d
los
oid
points
cd
xIdx = 0
yIdx = 0
dIdx = 0
openingIdx = 0
aXIdx = 0
aYIdx = 0
aZIdx = 0
omegaXIdx = 0
omegaYIdx = 0
omegaZIdx = 0
offsetIdx = 0
cd_inv
```

normalization = 1

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

initializeParameterSets(*self*)

Initialize my parameter sets

mountInputDataspace(*self*, *pfs*)

Mount the directory with my input files

loadInputs(*self*)

Load the data in the input files into memory

computeNormalization(*self*)

Compute the normalization of the L2 norm

computeCovarianceInverse(*self*)

Compute the inverse of my data covariance

meta(*self*)

Persist the sample layout by recording the parameter set metadata

show(*self*, *job*, *channel*)

Place model information in the supplied {channel}

altar.models.cdm.CUDA

Module Contents

Classes

class altar.models.cdm.CUDA.**CUDA**

A strategy for computing the data log likelihood that is written in pure python

source

initialize(*self*, *application*, *model*)

Initialize the strategy with {model} information

dataLikelihood(*self*, *model*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data.

altar.models.cdm.Data

Module Contents

Classes

class altar.models.cdm.Data.**Data**(*name*, ***kwds*)

Bases: altar.tabular.sheet

The layout of the input data file

oid

doc = an integer identifying the data source

x

doc = the EW coordinate of the location of the source

y

doc = the NS coordinate of the location of the source

d

doc = the displacement projected along the line of sight (LOS)

theta

doc = the azimuthal angle of the LOS vector to the observing craft

phi

doc = the polar angle of the LOS vector to the observing craft

read(*self*, *uri*)

Load a data set from a CSV file

write(*self*, *uri*)

Save my data into a CSV file

altar.models.cdm.Fast

Module Contents

Classes

class altar.models.cdm.Fast.**Fast**

A strategy for computing the data log likelihood that is written in pure python

source

initialize(*self*, *model*, ***kwds*)

Initialize the strategy with {model} information

dataLikelihood(*self*, *model*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data.

altar.models.cdm.Native

Module Contents

Classes

class altar.models.cdm.Native.**Native**

A strategy for computing the data log likelihood that is written in pure python

initialize(*self*, ***kwds*)

Initialize the strategy

dataLikelihood(*self*, *model*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data.

altar.models.cdm.Source

Module Contents

Classes

class altar.models.cdm.Source.**Source**(*x=x*, *y=y*, *d=d*, *ax=ax*, *ay=ay*, *az=az*, *omegaX=omegaX*, *omegaY=omegaY*, *omegaZ=omegaZ*, *opening=opening*, *v=v*, ***kwds*)

The source response for a Compound Dislocation Model in an elastic half space.

x = 0

y = 0

d = 0

ax = 0

ay = 0

az = 0

omegaX = 0

omegaY = 0

omegaZ = 0

opening = 0

v = 0.25

displacements(*self*, *locations*, *los*)

Compute the expected displacements at a set of observation locations from a compound (triaxial) dislocation source at depth.

altar.models.cdm.libcdm

Module Contents

Functions

altar.models.cdm.libcdm.**cosd**(*angd*)

altar.models.cdm.libcdm.**sind**(*angd*)

altar.models.cdm.libcdm.**norm**(*v*)

altar.models.cdm.libcdm.**CDM**(*X*, *Y*, *X0*, *Y0*, *depth*, *ax*, *ay*, *az*, *omegaX*, *omegaY*, *omegaZ*, *opening*, *nu*)

CDM calculates the surface displacements and potency associated with a CDM that is composed of three mutually orthogonal rectangular dislocations in a half-space.

CDM: Compound Dislocation Model RD: Rectangular Dislocation EFCS: Earth-Fixed Coordinate System RDCS: Rectangular Dislocation Coordinate System ADCS: Angular Dislocation Coordinate System (The origin of the RDCS is the RD centroid. The axes of the RDCS are aligned with the strike, dip and normal vectors of the RD, respectively.)

INPUTS X and Y: Horizontal coordinates of calculation points in EFCS (East, North, Up). X and Y must have the same size.

X0, Y0 and depth: Horizontal coordinates (in EFCS) and depth of the CDM centroid. The depth must be a positive value. X0, Y0 and depth have the same unit as X and Y.

omegaX, omegaY and omegaZ: Clockwise rotation angles about X, Y and Z axes, respectively, that specify the orientation of the CDM in space. The input values must be in degrees.

ax, ay and az: Semi-axes of the CDM along the X, Y and Z axes, respectively, before applying the rotations. ax, ay and az have the same unit as X and Y.

opening: The opening (tensile component of the Burgers vector) of the RDs that form the CDM. The unit of opening must be the same as the unit of ax, ay and az.

nu: Poisson's ratio.

OUTPUTS ue, un and uv: Calculated displacement vector components in EFCS. ue, un and uv have the same unit as opening and the CDM semi-axes in inputs.

DV: Potency of the CDM. DV has the unit of volume (the unit of displacements, opening and CDM semi-axes to the power of 3).

Example: Calculate and plot the vertical displacements on a regular grid.

[X,Y] = numpy.meshgrid(-7:.02:7,-5:.02:5); X0 = 0.5; Y0 = -0.25; depth = 2.75; omegaX = 5; omegaY = -8; omegaZ = 30; ax = 0.4; ay = 0.45; az = 0.8; opening = 1e-3; nu = 0.25; import te[[ue,un,uv,DV] = CDM(X,Y,X0,Y0,depth,omegaX,omegaY,omegaZ,ax,ay,az,… opening,nu); figure surf(X,Y,reshape(uv,size(X)),'edgecolor','none') view(2) axis equal axis tight set(gcf,'renderer','painters')

Reference journal article: Nikkhoo, M., Walter, T. R., Lundgren, P. R., Prats-Iraola, P. (2016): Compound dislocation models (CDMs) for volcano deformation analyses. Submitted to Geophysical Journal International

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

I appreciate any comments or bug reports.

Mehdi Nikkhoo Created: 2015.5.22 Last modified: 2016.10.18

Section 2.1, Physics of Earthquakes and Volcanoes Department 2, Geophysics Helmholtz Centre Potsdam German Research Centre for Geosciences (GFZ)

email: mehdi.nikkhoo@gfz-potsdam.de mehdi.nikkhoo@gmail.com

website: http://www.volcanodeformation.com

Converted from Matlab to Python April 2018 by Eric Gurrola Jet Propulsion Lab/Caltech

altar.models.cdm.libcdm.**RDdispSurf**(*X*, *Y*, *P1*, *P2*, *P3*, *P4*, *opening*, *nu*)

RDdispSurf calculates surface displacements associated with a rectangular dislocation in an elastic half-space.

altar.models.cdm.libcdm.**CoordTrans**(*x1*, *x2*, *x3*, *A*)

CoordTrans transforms the coordinates of the vectors, from x1x2x3 coordinate system to X1X2X3 coordinate system. "A" is the transformation matrix, whose columns e1,e2 and e3 are the unit base vectors of the x1x2x3. The coordinates of e1,e2 and e3 in A must be given in X1X2X3. The transpose of A (i.e., A') will transform the coordinates from X1X2X3 into x1x2x3.

altar.models.cdm.libcdm.**AngSetupFSC**(*X*, *Y*, *bX*, *bY*, *bZ*, *PA*, *PB*, *nu*)

AngSetupSurf calculates the displacements associated with an angular dislocation pair on each side of an RD in a half-space.

altar.models.cdm.libcdm.**AngDisDispSurf**(*y1*, *y2*, *beta*, *b1*, *b2*, *b3*, *nu*, *a*)

AngDisDispSurf calculates the displacements associated with an angular dislocation in a half-space.

altar.models.cdm.meta

Module Contents

altar.models.cdm.meta.**date = 2021-07-16T04:55:48Z**

altar.models.cdm.meta.**major = 2**

altar.models.cdm.meta.**minor = 0**

altar.models.cdm.meta.**micro = 1**

altar.models.cdm.meta.**revision = c4eeb4c**

altar.models.cdm.meta.**version**

altar.models.cdm.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.cdm.meta.**banner**

altar.models.cdm.meta.**header**

altar.models.cdm.meta.**license**

altar.models.cdm.meta.**acknowledgments**

Package Contents

Classes

Functions

class altar.models.cdm.**source**(*x=x*, *y=y*, *d=d*, *ax=ax*, *ay=ay*, *az=az*, *omegaX=omegaX*, *omegaY=omegaY*, *omegaZ=omegaZ*, *opening=opening*, *v=v*, ***kwds*)

The source response for a Compound Dislocation Model in an elastic half space.

x = 0

y = 0

d = 0

ax = 0

ay = 0

az = 0

omegaX = 0

omegaY = 0

omegaZ = 0

opening = 0

v = 0.25

displacements(*self*, *locations*, *los*)

Compute the expected displacements at a set of observation locations from a compound (triaxial) dislocation source at depth.

class altar.models.cdm.**data**(*name*, ***kwds*)

Bases: altar.tabular.sheet

The layout of the input data file

oid

doc = an integer identifying the data source

x

doc = the EW coordinate of the location of the source

y

doc = the NS coordinate of the location of the source

d

doc = the displacement projected along the line of sight (LOS)

theta

doc = the azimuthal angle of the LOS vector to the observing craft

phi

doc = the polar angle of the LOS vector to the observing craft

read(*self*, *uri*)
Load a data set from a CSV file

write(*self*, *uri*)
Save my data into a CSV file

altar.models.cdm.**cdm**()

altar.models.cudalinear

Submodules

altar.models.cudalinear.cudaLinear

Module Contents

Classes

class altar.models.cudalinear.cudaLinear.**cudaLinear**(*name*, *locator*, ***kwds*)
Bases: *altar.cuda.models.cudaBayesian.cudaBayesian*

Cuda implementation of a linear model A linear model is defined as data = G theta

dataobs

default

doc = the observed data

green

doc = the name of the file with the Green functions

GF

gGF

gDataPred

cublas_handle

initialize (*self*, *application*)

Initialize the state of the model given a {problem} specification

_forwardModel (*self*, *theta*, *prediction*, *batch*, *observation=None*)

Linear Forward Model prediction= G theta

cuEvalLikelihood (*self*, *theta*, *likelihood*, *batch*)

to be loaded by super class cuEvalLikelihood which already decides where the local likelihood is added to

loadGF (*self*)

Load the data in the input files into memory

prepareGF (*self*)

copy green function to gpu and merge cd with green function

altar.models.cudalinear.meta

Module Contents

altar.models.cudalinear.meta.**date = 2021-07-16T04:55:48Z**

altar.models.cudalinear.meta.**major = 2**

altar.models.cudalinear.meta.**minor = 0**

altar.models.cudalinear.meta.**micro = 1**

altar.models.cudalinear.meta.**revision = c4eeb4c**

altar.models.cudalinear.meta.**version**

altar.models.cudalinear.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.cudalinear.meta.**banner**

altar.models.cudalinear.meta.**header**

altar.models.cudalinear.meta.**license**

altar.models.cudalinear.meta.**acknowledgments**

Package Contents

Functions

altar.models.cudalinear.**cudalinear**()

altar.models.emhp

Submodules

altar.models.emhp.EMHP

Module Contents

Classes

class altar.models.emhp.EMHP.**EMHP**(*name*, *locator*, ***kwds*)

Bases: *altar.models.bayesian*

A diagnostic tool

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

File {step.theta} with an initial random sample form my prior distribution

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

altar.models.emhp.meta

Module Contents

altar.models.emhp.meta.**date = 2021-07-16T04:55:48Z**

altar.models.emhp.meta.**major = 2**

altar.models.emhp.meta.**minor = 0**

altar.models.emhp.meta.**micro = 1**

altar.models.emhp.meta.**revision = c4eeb4c**

altar.models.emhp.meta.**version**

altar.models.emhp.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.emhp.meta.**banner**

altar.models.emhp.meta.**header**

altar.models.emhp.meta.**license**

altar.models.emhp.meta.**acknowledgments**

Package Contents

Functions

altar.models.emhp.**emhp**()

altar.models.gaussian

Subpackages

altar.models.gaussian.ext

Submodules

altar.models.gaussian.Gaussian

Module Contents

Classes

class altar.models.gaussian.Gaussian.**Gaussian**(***kwds*)

Bases: *altar.models.bayesian*

A model that emulates the probability density for a single observation of the model parameters. The observation is treated as normally distributed around a given mean, with a covariance constructed out of its eigenvalues and a rotation in configuration space. Currently, only two dimensional parameter spaces are supported.

parameters

doc = the number of model degrees of freedom

support

doc = the support interval of the prior distribution

prep

doc = the distribution used to generate the initial sample

prior

doc = the prior distribution

μ

doc = the location of the central value of the observation

λ

doc = the eigenvalues of the covariance matrix

ψ

doc = the orientation of the covariance semi-major axis

peak

σ_inv

normalization = 1

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

altar.models.gaussian.meta

Module Contents

altar.models.gaussian.meta.**date = 2021-07-16T04:55:48Z**

altar.models.gaussian.meta.**major = 2**

altar.models.gaussian.meta.**minor = 0**

altar.models.gaussian.meta.**micro = 1**

altar.models.gaussian.meta.**revision = c4eeb4c**

altar.models.gaussian.meta.**version**

altar.models.gaussian.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010-
California Institute of Technology ALL RIGHTS RESERVED**

altar.models.gaussian.meta.**banner**

altar.models.gaussian.meta.**header**

altar.models.gaussian.meta.**license**

altar.models.gaussian.meta.**acknowledgments**

Package Contents

Functions

altar.models.gaussian.**gaussian**()

altar.models.linear

Submodules

altar.models.linear.Linear

Module Contents

Classes

class altar.models.linear.Linear.**Linear**(*name*, *locator*, ***kwds*)

Bases: *altar.models.bayesian*

parameters

doc = the number of parameters in the model

observations

doc = the number of data samples

prep

default

doc = the distribution used to generate the initial sample

prior

default

doc = the prior distribution

norm

default

doc = the norm to use when computing the data log likelihood

case

doc = the directory with the input files

green

doc = the name of the file with the Green functions

data

doc = the name of the file with the observations

cd

doc = the name of the file with the data covariance matrix

ifs

G

d

Cd

Cd_inv

residuals

normalization = 1

initialize (*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample (*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution.

priorLikelihood (*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood (*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify (*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

mountInputDataspace (*self*, *pfs*)

Mount the directory with my input files

loadInputs (*self*)

Load the data in the input files into memory

computeCovarianceInverse (*self*, *cd*)

Compute the inverse of the data covariance matrix

computeNormalization (*self*, *observations*, *cd*)

Compute the normalization of the L2 norm

initializeResiduals (*self*, *samples*, *data*)

Prime the matrix that will hold the residuals (G θ - d) for each sample by duplicating the observation vector as many times as there are samples

altar.models.linear.meta

Module Contents

altar.models.linear.meta.**date = 2021-07-16T04:55:48Z**

altar.models.linear.meta.**major = 2**

altar.models.linear.meta.**minor = 0**

altar.models.linear.meta.**micro = 1**

altar.models.linear.meta.**revision = c4eeb4c**

altar.models.linear.meta.**version**

altar.models.linear.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.linear.meta.**banner**

altar.models.linear.meta.**header**

altar.models.linear.meta.**license**

altar.models.linear.meta.**acknowledgments**

Package Contents

Functions

altar.models.linear.**linear**()

altar.models.mogi

Subpackages

altar.models.mogi.ext

Submodules

altar.models.mogi.CUDA

Module Contents

Classes

class altar.models.mogi.CUDA.**CUDA**

A strategy for computing the data log likelihood that is written in pure python

source

initialize (*self*, *application*, *model*)

Initialize the strategy with {model} information

dataLikelihood (*self*, *model*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data.

altar.models.mogi.Data

Module Contents

Classes

class altar.models.mogi.Data.**Data** (*name*, ***kwds*)

Bases: altar.tabular.sheet

The layout of the input data file

oid

doc = an integer identifying the data source

x

doc = the EW coordinate of the location of the source

y

doc = the NS coordinate of the location of the source

d

doc = the displacement projected along the line of sight (LOS)

theta

doc = the azimuthal angle of the LOS vector to the observing craft

phi

doc = the polar angle of the LOS vector to the observing craft

read (*self*, *uri*)

Load a data set from a CSV file

write (*self*, *uri*)

Save my data into a CSV file

altar.models.mogi.Fast

Module Contents

Classes

class altar.models.mogi.Fast.**Fast**

A strategy for computing the data log likelihood that is written in pure python

source

initialize(*self*, *model*, ***kwds*)

Initialize the strategy with {model} information

dataLikelihood(*self*, *model*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data.

altar.models.mogi.Mogi

Module Contents

Classes

class altar.models.mogi.Mogi.**Mogi**(*name*, *locator*, ***kwds*)

Bases: *altar.models.bayesian*

An implementation of Mogi[1958]

The surface displacement calculation for a pressure point source in an elastic half space.

Currently, {mogi} is implemented as a four parameter model: x,y,depth locate the point source, and {dV} provides the point source strength as the volume change. It can easily become a five parameter model by including the Poisson ratio of the elastic material to the list of free parameters.

psets

doc = the model parameter meta-data

observations

doc = the number of model degrees of freedom

norm

default

doc = the norm to use when computing the data log likelihood

case

doc = the directory with the input files

displacements

doc = the name of the file with the displacements

covariance

doc = the name of the file with the data covariance

nu

doc = the Poisson ratio

mode

doc = the implementation strategy

validators

parameters = 0

strategy

ifs

d

los

oid

points

cd

xIdx = 0

yIdx = 0

dIdx = 0

sIdx = 0

offsetIdx = 0

cd_inv

normalization = 1

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

initializeParameterSets (*self*)
Initialize my parameter sets

mountInputDataspace (*self*, *pfs*)
Mount the directory with my input files

loadInputs (*self*)
Load the data in the input files into memory

computeNormalization (*self*)
Compute the normalization of the L2 norm

computeCovarianceInverse (*self*)
Compute the inverse of my data covariance

meta (*self*)
Persist the sample layout by recording the parameter set metadata

show (*self*, *job*, *channel*)
Place model information in the supplied {channel}

altar.models.mogi.Native

Module Contents

Classes

class altar.models.mogi.Native.**Native**
A strategy for computing the data log likelihood that is written in pure python

initialize (*self*, ***kwds*)
Initialize the strategy

dataLikelihood (*self*, *model*, *step*)
Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data.

altar.models.mogi.Source

Module Contents

Classes

class altar.models.mogi.Source.**Source** (*x=x*, *y=y*, *d=d*, *dV=dV*, *nu=nu*, ***kwds*)
An implementation of Mogi[1958]

The surface displacement calculation for a point pressure source in an elastic half space.

x = 0

y = 0

d = 0

dV = 0

nu = 0.25

displacements(*self*, *locations*, *los*)

Compute the expected displacements from a point pressure source at a set of observation locations

altar.models.mogi.meta

Module Contents

altar.models.mogi.meta.**date = 2021-07-16T04:55:48Z**

altar.models.mogi.meta.**major = 2**

altar.models.mogi.meta.**minor = 0**

altar.models.mogi.meta.**micro = 1**

altar.models.mogi.meta.**revision = c4eeb4c**

altar.models.mogi.meta.**version**

altar.models.mogi.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.mogi.meta.**banner**

altar.models.mogi.meta.**header**

altar.models.mogi.meta.**license**

altar.models.mogi.meta.**acknowledgments**

Package Contents

Classes

Functions

class altar.models.mogi.**source**(*x=x*, *y=y*, *d=d*, *dV=dV*, *nu=nu*, ***kwds*)

An implementation of Mogi[1958]

The surface displacement calculation for a point pressure source in an elastic half space.

x = 0

y = 0

d = 0

dV = 0

nu = 0.25

displacements(*self*, *locations*, *los*)

Compute the expected displacements from a point pressure source at a set of observation locations

class altar.models.mogi.**data**(*name*, ***kwds*)

Bases: altar.tabular.sheet

The layout of the input data file

oid

doc = an integer identifying the data source

x

doc = the EW coordinate of the location of the source

y

doc = the NS coordinate of the location of the source

d

doc = the displacement projected along the line of sight (LOS)

theta

doc = the azimuthal angle of the LOS vector to the observing craft

phi

doc = the polar angle of the LOS vector to the observing craft

read(*self*, *uri*)

Load a data set from a CSV file

write(*self*, *uri*)

Save my data into a CSV file

altar.models.mogi.**mogi**()

altar.models.regression

Submodules

altar.models.regression.Linear

Module Contents

Classes

class altar.models.regression.Linear.**Linear**(*name*, *locator*, ***kwds*)

Bases: *altar.models.BayesianL2.BayesianL2*

Linear Regression model y= ax +b

x_file

doc = the input file for x variable

x

y

initialize (*self*, *application*)

Initialize the state of the model

forwardModel (*self*, *theta*, *prediction*)

Forward Model :param theta: sampling parameters for one sample :param prediction: data prediction or residual (prediction - observation) :return: none

altar.models.regression.meta

Module Contents

altar.models.regression.meta.**date = 2021-07-16T04:55:48Z**

altar.models.regression.meta.**major = 2**

altar.models.regression.meta.**minor = 0**

altar.models.regression.meta.**micro = 1**

altar.models.regression.meta.**revision = c4eeb4c**

altar.models.regression.meta.**version**

altar.models.regression.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.regression.meta.**banner**

altar.models.regression.meta.**header**

altar.models.regression.meta.**license**

altar.models.regression.meta.**acknowledgments**

Package Contents

Classes

Functions

class altar.models.regression.**model**

Bases: altar.protocol

The protocol that all AlTar models must implement

posterior (*self*, *application*)

Sample my posterior distribution

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

posteriorLikelihood(*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods(*self*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

top(*self*, *annealer*)

Notification that a β step is about to start

bottom(*self*, *annealer*)

Notification that a β step just ended

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.models.regression.**linear**()

altar.models.seismic

Subpackages

altar.models.seismic.actions

Submodules

altar.models.seismic.actions.About

Module Contents

Classes

class altar.models.seismic.actions.About.**About**(*name*, *spec*, *plexus*, ***kwds*)

Bases: altar.panel()

Display information about this application

root

tip = specify the portion of the namespace to display

name(*self*, *plexus*, ***kwds*)

Print the name of the app for configuration purposes

home(*self*, *plexus*, ***kwds*)

Print the application home directory

prefix(*self*, *plexus*, ***kwds*)

Print the application installation directory

models(*self*, *plexus*, ***kwds*)

Print the altar model directory

when(*self*, *plexus*, ***kwds*)

Print the build timestamp

etc(*self*, *plexus*, ***kwds*)

Print the application configuration directory

version(*self*, *plexus*, ***kwds*)

Print the version of the altar package

copyright(*self*, *plexus*, ***kwds*)

Print the copyright note of the altar package

credits(*self*, *plexus*, ***kwds*)

Print out the license and terms of use of the altar package

license(*self*, *plexus*, ***kwds*)

Print out the license and terms of use of the altar package

nfs(*self*, *plexus*, ***kwds*)

Dump the application configuration namespace

pfs(*self*, *plexus*, ***kwds*)

Dump the application private filesystem

vfs(*self*, *plexus*, ***kwds*)

Dump the application virtual filesystem

altar.models.seismic.actions.Forward

Module Contents

Classes

class altar.models.seismic.actions.Forward.**Forward**(*name*, *spec*, *plexus*, ***kwds*)

Bases: altar.panel()

Sample the posterior distribution of a model

default(*self*, *plexus*, ***kwds*)

Sample the model posterior distribution

altar.models.seismic.actions.Sample

Module Contents

Classes

class altar.models.seismic.actions.Sample.**Sample**(*name*, *spec*, *plexus*, ***kwds*)

Bases: altar.panel()

Sample the posterior distribution of a model

default(*self*, *plexus*, ***kwds*)

Sample the model posterior distribution

Package Contents

Functions

altar.models.seismic.actions.**about**()

altar.models.seismic.actions.**sample**()

altar.models.seismic.actions.**forward**()

altar.models.seismic.cuda

Submodules

altar.models.seismic.cuda.cudaCascaded

Module Contents

Classes

class altar.models.seismic.cuda.cudaCascaded.**cudaCascaded**(*name*, *locator*, ***kwds*)

Bases: *altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble*

Cascaded inversion of a model ensemble

altar.models.seismic.cuda.cudaKinematicG

Module Contents

Classes

class altar.models.seismic.cuda.cudaKinematicG.**cudaKinematicG**(*name*, *locator*, ***kwds*)

Bases: *altar.cuda.models.cudaBayesian.cudaBayesian*

KinematicG model with cuda

dataobs

default

doc = the observed data

green

doc = the name of the file with the Green functions

Nas

doc = number of patches along strike direction

Ndd

doc = number of patches along dip direction

Nmesh

doc = number of mesh points for each patch for fastsweeping

dsp

doc = the distance unit for each patch, in km

Nt

doc = number of time intervals for kinematic process

Npt

doc = number of mesh points for each time interval for fastsweeping

dt

doc = the time unit for each time interval (in s)

t0s

doc = the start time for each patch

cmodel

GF

gGF

gDprediction

cublas_handle

NGbparameters

gt0s

initialize (*self*, *application*)

Initialize the state of the model given a {problem} specification

forwardModelBatched (*self*, *theta*, *gf*, *prediction*, *batch*, *observation=None*)

KinematicG forward model in batch: cast Mb(x,y,t) :param theta: matrix (samples, parameters), sampling parameters :param gf: matrix (2*Ndd*Nas*Nt, observations), kinematicG green's function :param prediction: matrix (samples, observations), the predicted data or residual between predicted and observed data :param batch: integer, the number of samples to be computed batch<=samples :param observation: matrix (samples, observations), duplicates of observed data :return: prediction as predicted data(observation=None) or residual (observation is provided)

forwardModel (*self*, *theta*, *gf*, *prediction*, *observation=None*)

KinematicG forward model for single sample: cast Mb(x,y,t) :param theta: vector (parameters), sampling parameters :param gf: matrix (2*Ndd*Nas*Nt, observations), kinematicG green's function :param prediction: vector (observations), the predicted data or residual between predicted and observed data :param observation: vector (observations), duplicates of observed data :return: prediction as predicted data(observation=None) or residual (observation is provided)

castSlipsOfTime (*self*, *theta*, *Mb=None*)

Compute Mb (slips of patches over time) from a given set of parameters :param theta: a vector arranged in [slip (strike and dip), risetime, …] :param Mb: :return: Mb

linearGM (*self*, *gf*, *Mb*, *prediction=None*, *observation=None*)

Perform prediction = Gb * Mb :param Gb: :param Mb: :param prediction: :return: prediction

cuEvalLikelihood (*self*, *theta*, *likelihood*, *batch*)

Compute the likelihood from my forward problem

mergeCovarianceToGF (*self*)

merge data covariance (cd) with green function

forwardProblem (*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

altar.models.seismic.cuda.cudaKinematicGCp

Module Contents

Classes

class altar.models.seismic.cuda.cudaKinematicGCp.**cudaKinematicGCp**

Bases: *altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG*

KinematicG inversion with Cp (prediction error due to model parameter uncertainty)

nCmu

doc = the number of model parameters with uncertainties (or to be considered)

cmu_file

doc = the covariance describing the uncertainty of model parameter, a nCmu x nCmu matrix

kmu_file

doc = the sensitivity kernel of model parameters, a hdf5 file including nCmu kernel data sets

initial_model_file

doc = the initial mean model

beta_cp_start

doc = for beta >= beta_cp_start, incorporate Cp into Cd

beta_use_initial_model

doc = for beta <= beta_use_initial_model, use initial_model instead of mean model

mean_model

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeCp(*self*)

Returns

updateModel(*self*, *annealer*)

Model method called by Sampler before Metropolis sampling for each beta step starts, employed to compute Cp and merge Cp with data covariance :param annealer: the annealer for application :return: True or False if model parameters are updated or remain the same

computeCp(*self*, *model*, *cp=None*)

Compute Cp with a mean model :param model: :return:

altar.models.seismic.cuda.cudaMoment

Module Contents

Classes

class altar.models.seismic.cuda.cudaMoment.**cudaMoment**(*name*, *locator*, ***kwds*)

Bases: *altar.cuda.distributions.cudaUniform.cudaUniform*

The probability distribution for displacements (D) conforming to a given Moment magnitude scale Mw = (log M0 - 9.1)/1.5 (Hiroo Kanamori) M0 = Mu A D It serves to initialize samples only, with combined gaussian and dirichlet distributions. It inherits uniform distribution for verification and density calculations.

area_patch_file

doc = input file for area of each patch, in unit of km^2

area

doc = area of each patch in unit of km^2, provide one value if the same for all patches

Mw_mean

doc = the mean moment magnitude scale

Mw_sigma

doc = the variance of moment magnitude scale

Mu

doc = the shear modulus for each patch in GPa, provide one value if the same for all patches

Mu_patch_file

Mu_patch_file = input file for the shear modulus of each patch, in Km^2

slip_sign

validators

doc = the sign of slips, all positive or all negative

area_patches

mu_patches

patches

rng

initialize(*self*, *application*)

Initialize with the given random number generator

cuInitialize(*self*, *application*)

cuda interface of initialization

cuInitSample(*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

altar.models.seismic.cuda.cudaStatic

Module Contents

Classes

class altar.models.seismic.cuda.cudaStatic.**cudaStatic**(*name*, *locator*, ***kwds*)

Bases: *altar.cuda.models.cudaBayesian.cudaBayesian*

cudaLinear with the new cuda framework

dataobs

default

doc = the observed data

green

doc = the name of the file with the Green functions

GF

gGF

gDataPred

cublas_handle

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

forwardModelBatched(*self*, *theta*, *green*, *prediction*, *batch*, *observation=None*)

Linear Forward Model prediction= G theta

forwardModel(*self*, *theta*, *green*, *prediction*, *observation=None*)

Static/Linear forward model prediction = green * theta :param theta: a parameter set, vector with size parameters :param green: green's function, matrix with size (observations, parameters) :param prediction: data prediction, vector with size observations :return: data prediction if observation is none; otherwise return residual

cuEvalLikelihood(*self*, *theta*, *likelihood*, *batch*)

Compute data likelihood from the forward model, :param theta: parameters, matrix [samples, parameters] :param likelihood: data likelihood P(d|theta), vector [samples] :param batch: the number of samples to be computed, batch <=samples :return: likelihood, in case of model ensembles, data likelihood of this model is added to the input likelihood

mergeCovarianceToGF(*self*)

merge data covariance (cd) with green function

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

altar.models.seismic.cuda.cudaStaticCp

Module Contents

Classes

class altar.models.seismic.cuda.cudaStaticCp.**cudaStaticCp**

Bases: *altar.models.seismic.cuda.cudaStatic.cudaStatic*

Static inversion with Cp (prediction error due to model parameter uncertainty)

nCmu

doc = the number of model parameters with uncertainties (or to be considered)

cmu_file

doc = the covariance describing the uncertainty of model parameter, a nCmu x nCmu matrix

kmu_file

doc = the sensitivity kernel of model parameters, a hdf5 file including nCmu kernel data sets

initial_model_file

doc = the initial mean model

beta_cp_start

doc = for beta >= beta_cp_start, incorporate Cp into Cd

beta_use_initial_model

doc = for beta <= beta_use_initial_model, use initial_model instead of mean model

dtype_cp

doc = single/double precision to compute Cp

gCmu

gInitModel

gMeanModel

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeCp(*self*)

Initialize Cp related :return:

updateModel(*self*, *annealer*)

Model method called by Sampler before Metropolis sampling for each beta step starts, employed to compute Cp and merge Cp with data covariance :param annealer: the annealer for application :return: True or False if model parameters are updated or remain the same

computeCp(*self*, *model*, *cp=None*)

Compute Cp with a mean model :param model: :return:

Package Contents

Classes

Functions

class altar.models.seismic.cuda.**model**(*name*, *locator*, ***kwds*)

Bases: *altar.models.Bayesian.Bayesian*

The base class of AlTar models that are compatible with Bayesian explorations

parameters

doc = the number of model degrees of freedom

cascaded

doc = whether the model is cascaded (annealing temperature is fixed at 1)

embedded

doc = whether the model is embedded in an ensemble of models

psets_list

doc = list of parameter sets, used to set orders

psets

default

doc = an ensemble of parameter sets in the model

dataobs

default

doc = observed data

case

doc = the directory with the input files

idx_map

default

doc = the indices for model parameters in whole theta set

return_residual

doc = the forward model returns residual(True) or prediction(False)

forwardonly

doc = whether to run the simulation or the forward problem only

theta_input

doc = the theta input file with a vector of parameters

theta_dataset

doc = the name/path of the theta dataset in h5 file

forward_output

dpc = the name/path of the file to save forward problem results

observations

device

precision

ifs

gidx_map

gtheta

initialize(*self*, *application*)

Initialize the state of the model given an {application} context

cuInitialize(*self*, *application*)

cuda interface

posterior(*self*, *application*)

Sample my posterior distribution

cuInitSample(*self*, *theta*)

Fill {theta} with an initial random sample from my prior distribution.

cuVerify(*self*, *theta*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

cuEvalPrior(*self*, *theta*, *prior*, *batch*)

Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

cuEvalLikelihood(*self*, *theta*, *likelihood*, *batch*)

calculate data likelihood and add it to step.prior or step.data

cuEvalPosterior(*self*, *step*, *batch*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods(*self*, *annealer*, *step*, *batch=None*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

updateModel(*self*, *annealer*)

Update Model parameters if needed :param annealer: :return: default is False

mountInputDataspace(*self*, *pfs*)

Mount the directory with my input files

loadFile(*self*, *filename*, *shape=None*, *dataset=None*, *dtype=None*)

Load an input file to a numpy array (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
output numpy.array

loadFileToGPU(*self*, *filename*, *shape=None*, *dataset=None*, *out=None*, *dtype=None*)

Load an input file to a gpu (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
out altar.cuda.matrix/vector

restricted(*self*, *theta*, *batch*)

extract theta which contains model's own parameters

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

altar.models.seismic.cuda.**moment**()

altar.models.seismic.cuda.**static**()

altar.models.seismic.cuda.**staticcp**()

altar.models.seismic.cuda.**kinematicg**()

altar.models.seismic.cuda.**cascaded**()

altar.models.seismic.ext

altar.models.seismic.shells

Submodules

altar.models.seismic.shells.Action

Module Contents

Classes

class altar.models.seismic.shells.Action.**Action**

Bases: altar.models.seismic.action

Protocol for {altar} commands

altar.models.seismic.shells.Seismic

Module Contents

Classes

class altar.models.seismic.shells.Seismic.**Seismic**(***kwds*)

Bases: altar.plexus

The main action dispatcher for the simple AlTar application

job

doc = the job input parameters

model

default

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main(*self*, **args*, ***kwds*)

The main entry point

initialize (*self*)
Initialize without running, for debug purpose only

forward (*self*)
Perform the forward problem once for a given set of parameters

pyre_banner (*self*)
Place the application banner in the {info} channel

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

altar.models.seismic.shells.cudaApplication

Module Contents

Classes

class altar.models.seismic.shells.cudaApplication.**cudaApplication**(*name=None*, ***kwds*)

Bases: altar.application

The base class for simple AlTar applications

job

doc = the job input parameters

model

default

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)
The main entry point

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

pyre_mpi (*self*)
Transfer my {job} settings to the MPI shell

Package Contents

Classes

class altar.models.seismic.shells.**seismic**(***kwds*)

Bases: altar.plexus

The main action dispatcher for the simple AlTar application

job

doc = the job input parameters

model

default

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main(*self*, **args*, ***kwds*)

The main entry point

initialize(*self*)

Initialize without running, for debug purpose only

forward(*self*)

Perform the forward problem once for a given set of parameters

pyre_banner(*self*)

Place the application banner in the {info} channel

pyre_interactiveSessionContext(*self*, *context*)

Go interactive

Submodules

altar.models.seismic.Moment

Module Contents

Classes

class altar.models.seismic.Moment.**Moment**(*name*, *locator*, ***kwds*)

Bases: *altar.distributions.Uniform.Uniform*

The probability distribution for displacements (D) conforming to a given Moment magnitude scale Mw = (log M0 - 9.1)/1.5 (Hiroo Kanamori) M0 = Mu A D It inherits uniform distribution for verification and density calculations, while generates samples for a combined gaussian and dirichlet distributions

area_patches_file

doc = input file for area of each patch, in unit of km^2

area

doc = total area in unit of km^2

Mw_mean

doc = the mean moment magnitude scale

Mw_sigma

doc = the variance of moment magnitude scale

Mu

doc = the shear modulus in unit of GPa

area_patches

patches

initialize(*self*, *rng*)

Initialize with the given random number generator

initializeSample(*self*, *theta*)

Fill my portion of {theta} with initial random values from my distribution.

altar.models.seismic.Static

Module Contents

Classes

class altar.models.seismic.Static.**Static**(*name*, *locator*, ***kwds*)

Bases: *altar.models.bayesian*

Static inversion with cuda (d = G theta) Modeled as N patches with dip and slip displacements

parametersets

doc = the set of parameters in the model

parameters

doc = total number of parameters in the model

observations

doc = the number of data samples

patches

norm

default

doc = the norm to use when computing the data log likelihood

case

doc = the directory with the input files

green_file

doc = the name of the file with the Green functions

data_file

doc = the name of the file with the observations

cd_file

doc = the name of the file with the data covariance matrix

output_path

ifs

G

d

Cd

Cd_inv

residuals

normalization = 1

processor = cpu

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution.

computePrior(*self*, *step*)

Fill {step.prior} with the densities of the samples in {step.theta} in the prior distribution

computeDataLikelihood(*self*, *step*)

Fill {step.data} with the densities of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

initializeParameterSets (*self*)

Initialize the parameter set

mountInputDataspace (*self*, *pfs*)

Mount the directory with my input files

loadInputs (*self*)

Load the data in the input files into memory

initializeCovariance (*self*, *samples*)

Compute the Cholesky decomposition of the inverse of the data covariance and merge it to data

computeCovarianceInverse (*self*, *cd*)

Compute the inverse of the data covariance matrix

computeNormalization (*self*, *observations*, *cd*)

Compute the normalization of the L2 norm

initializeResiduals (*self*, *samples*, *data*)

Prime the matrix that will hold the residuals (G θ - d) for each sample by duplicating the observation vector as many times as there are samples

update (*self*, *annealer*)

Model updating at the bottom of each annealing step Output step data

forwardModel (*theta*, *green*, *data_residuals=None*, *data_observations=None*, *batches=None*)

Forward model: compute data prediction or data residuals from a set of theta :param theta [in: :type theta [in: samples, parameters :param cuarray] parameters with shape=: :type cuarray] parameters with shape=: samples, parameters :param green [in: :type green [in: observations, parameters :param cuarray] Green's function with shape =: :type cuarray] Green's function with shape =: observations, parameters :param batches [in: :param integer: :param optional] number of samples needto be computed <=samples: :param data_observations [in: :param cuarray: :param optional] data observations: :param data_residuals[inout: :type data_residuals[inout: observations, samples :param cuarray: :type cuarray: observations, samples :param optional] data predictions or residuals shape=: :type optional] data predictions or residuals shape=: observations, samples

Returns

data predictions or residuals if data_observations is provides

altar.models.seismic.StaticCp

Module Contents

Classes

class altar.models.seismic.StaticCp.**StaticCp** (*name*, *locator*, ***kwds*)

Bases: *altar.models.seismic.Static.Static*

Linear Model with prediction uncertainty Cp, in addition to data uncertainty Cd Cp = Kp Cmu Kp^T Kp = Kmu * <θ> Kp.shape = (observations, nCmu) Cmu.shape = (nCmu, nCmu) Kmu.shape =(observations, parameters) (same as the green's function)

nCmu

doc = the number of model parameter sets

cmu_file

doc = the covariance describing the uncertainty of model parameter

kmu_file

doc = the sensitivity kernel of model parameter: input as kmu1.txt, ...

initialModel_file

doc = the initial mean model

G0

d0

Cd0

Cmu

Kmu

Cp

meanModel

initialize (*self*, *application*)
Initialize the state of the model given a {problem} specification

loadInputsCp (*self*)
Load the additional data (for Cp problem) in the input files into memory

initializeCovariance (*self*, *samples*)
initialize data covariance related variables

computeCp (*self*, *theta_mean*)
Calculate Cp

update (*self*, *annealer*)
Model update interface

altar.models.seismic.meta

Module Contents

altar.models.seismic.meta.**date = 2021-07-16T04:55:48Z**

altar.models.seismic.meta.**major = 2**

altar.models.seismic.meta.**minor = 0**

altar.models.seismic.meta.**micro = 1**

altar.models.seismic.meta.**revision = c4eeb4c**

altar.models.seismic.meta.**version**

altar.models.seismic.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.seismic.meta.**banner**

altar.models.seismic.meta.**header**

altar.models.seismic.meta.**license**

altar.models.seismic.meta.**acknowledgments**

Package Contents

Functions

altar.models.seismic.**moment**()

altar.models.seismic.**static**()

altar.models.seismic.**staticCp**()

altar.models.sir

Submodules

altar.models.sir.SIR

Module Contents

Classes

class altar.models.sir.SIR.**SIR**(*name*, *locator*, ***kwds*)

Bases: *altar.models.BayesianL2.BayesianL2*

SIR model in epidemiology This model assumes five parameters

> S0 - The initial value of susceptible population (times the population base factor) I0 - The initial value of infectious R0 - The initial value of Recovered/Deaths β - the average number of contacts per person per time γ - the rate of recovery or mortality

to generate the time sequence of S(t), I(t) and R(t) It uses the new cases per day as data observations, or S(t-1)-S(t)

population_base

doc = the base factor for population

_SIR_Rate(*self*, *S*, *I*, *N*, *β*, *γ*)

SIR Rate equation

forwardModel(*self*, *theta*, *prediction*)

Forward SIR model

altar.models.sir.meta

Module Contents

altar.models.sir.meta.**date = 2021-07-16T04:55:48Z**

altar.models.sir.meta.**major = 2**

altar.models.sir.meta.**minor = 0**

altar.models.sir.meta.**micro = 1**

altar.models.sir.meta.**revision = c4eeb4c**

altar.models.sir.meta.**version**

altar.models.sir.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.models.sir.meta.**banner**

altar.models.sir.meta.**header**

altar.models.sir.meta.**license**

altar.models.sir.meta.**acknowledgments**

Package Contents

Functions

altar.models.sir.**sir**()

Submodules

altar.models.Bayesian

Module Contents

Classes

class altar.models.Bayesian.**Bayesian**(*name*, *locator*, ***kwds*)

Bases: altar.component

The base class of AlTar models that are compatible with Bayesian explorations

offset

doc = the starting point of my state in the overall controller state

parameters

doc = the number of model degrees of freedom

psets

default

doc = an ensemble of parameter sets in the model

rng

controller

job

info

warning

error

default

firewall

initialize (*self*, *application*)

Initialize the state of the model given an {application} context

posterior (*self*, *application*)

Sample my posterior distribution

initializeSample (*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood (*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood (*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

posteriorLikelihood (*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods (*self*, *annealer*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

abstract verify (*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

top (*self*, *annealer*)

Notification that a β step is about to start

bottom (*self*, *annealer*)

Notification that a β step just ended

forwardProblem (*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

mountInputDataspace (*self*, *pfs*)
Mount the directory with my input files

restrict (*self*, *theta*)
Return my portion of the sample matrix {theta}

altar.models.BayesianL2

Module Contents

Classes

class altar.models.BayesianL2.**BayesianL2** (*name*, *locator*, ***kwds*)
Bases: *altar.models.Bayesian.Bayesian*

A (Simplified) Bayesian Model with ParameterSets and L2 data norm

parameters

doc = the number of model degrees of freedom

cascaded

doc = whether the model is cascaded (annealing temperature is fixed at 1)

embedded

doc = whether the model is embedded in an ensemble of models

psets_list

doc = list of parameter sets, used to set orders

psets

default

doc = an ensemble of parameter sets in the model

dataobs

default

doc = observed data

case

doc = the directory with the input files

idx_map

default

doc = the indices for model parameters in whole theta set

return_residual

doc = the forward model returns residual(True) or prediction(False)

observations

device

precision

ifs

initialize (*self*, *application*)

Initialize the state of the model given an {application} context

posterior (*self*, *application*)

Sample my posterior distribution

initializeSample (*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution.

verify (*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

evalPrior (*self*, *theta*, *prior*)

Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

abstract forwardModel (*self*, *theta*, *prediction*)

The forward model for a single set of parameters

forwardModelBatched (*self*, *theta*, *prediction*)

The forward model for a batch of theta: compute prediction from theta also return {residual}=True, False if the difference between data and prediction is computed

evalDataLikelihood (*self*, *theta*, *likelihood*)

calculate data likelihood and add it to step.prior or step.data

evalPosterior (*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods (*self*, *annealer*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

updateModel (*self*, *annealer*)

Update Model parameters if needed :param annealer: :return: default is False

mountInputDataspace (*self*, *pfs*)

Mount the directory with my input files

loadFile (*self*, *filename*, *shape=None*, *dataset=None*, *dtype=None*)

Load an input file to a gsl vector or matrix (for both float32/64 support) Supported format: 1. text file in '.txt' suffix, stored in prescribed shape 2. binary file with '.bin' or '.dat' suffix,

the precision must be same as the desired gpuprecision, and users must specify the shape of the data

3. **(preferred) hdf5 file in '.h5' suffix (preferred)**
   the metadata of shape, precision is included in .h5 file

Parameters

- **filename** – str, the input file name
- **shape** – list of int
- **dataset** – str, name/key of dataset for h5 input only

Returns
output gsl vector/matrix

restrict (*self*, *theta*)
Return my portion of the sample matrix {theta}

altar.models.Contiguous

Module Contents

Classes

class altar.models.Contiguous.**Contiguous** (*name*, *locator*, ***kwds*)
Bases: altar.component

A contiguous parameter set

count

doc = the number of parameters in this set

prior

doc = the prior distribution

prep

doc = the distribution to use to initialize this parameter set

offset = 0

initialize (*self*, *model*, *offset*)
Initialize my state given the {model} that owns me

initializeSample (*self*, *theta*)
Fill {theta} with an initial random sample from my prior distribution.

priorLikelihood (*self*, *theta*, *priorLLK*)
Fill {priorLLK} with the log likelihoods of the samples in {theta} in my prior distribution

verify (*self*, *theta*, *mask*)
Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

restrict (*self*, *theta*)
Return my portion of the sample matrix {theta}

altar.models.Ensemble

Module Contents

Classes

class altar.models.Ensemble.**Ensemble**(*name*, *locator*, ***kwds*)

Bases: *altar.models.Bayesian.Bayesian*

A collection of AlTar models that comprise a single model

models

doc = the collection of models in this ensemble

initialize(*self*, *application*)

Initialize the state of the model given an {application} context

initializeSample(*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

altar.models.Model

Module Contents

Classes

class altar.models.Model.**Model**

Bases: altar.protocol

The protocol that all AlTar models must implement

posterior(*self*, *application*)

Sample my posterior distribution

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood (*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

posteriorLikelihood (*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods (*self*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify (*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

top (*self*, *annealer*)

Notification that a β step is about to start

bottom (*self*, *annealer*)

Notification that a β step just ended

forwardProblem (*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

classmethod pyre_default (*cls*, ***kwds*)

Supply a default implementation

altar.models.Null

Module Contents

Classes

class altar.models.Null.**Null** (*name*, *locator*, ***kwds*)

Bases: *altar.models.Bayesian.Bayesian*

A trivial model that can be used as a base class for deriving interesting ones

parameters

doc = the number of model degrees of freedom

initializeSample (*self*, *step*)

Fill {step.θ} with an initial random sample from my prior distribution

priorLikelihood (*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood (*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

verify (*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

forwardProblem(*self*, *application*, *theta=None*)
Perform the forward modeling with given {theta}

altar.models.ParameterSet

Module Contents

Classes

class altar.models.ParameterSet.**ParameterSet**
Bases: altar.protocol

The protocol that all AlTar parameter sets must implement

count

doc = the number of parameters in this set

prior

doc = the prior distribution

prep

doc = the distribution to use to initialize this parameter set

initialize(*self*, *model*, *offset*)
Initialize the parameter set given the {model} that owns it

initializeSample(*self*, *theta*)
Fill {theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *theta*, *priorLLK*)
Fill {priorLLK} with the likelihoods of the samples in {theta} in my prior distribution

verify(*self*, *theta*, *mask*)
Check whether the samples in {theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

classmethod pyre_default(*cls*, ***kwds*)
Supply a default implementation

Package Contents

Classes

Functions

class altar.models.**model**
Bases: altar.protocol

The protocol that all AlTar models must implement

posterior(*self*, *application*)

Sample my posterior distribution

initialize(*self*, *application*)

Initialize the state of the model given a {problem} specification

initializeSample(*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

posteriorLikelihood(*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods(*self*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

top(*self*, *annealer*)

Notification that a β step is about to start

bottom(*self*, *annealer*)

Notification that a β step just ended

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.models.**parameters**

Bases: altar.protocol

The protocol that all AlTar parameter sets must implement

count

doc = the number of parameters in this set

prior

doc = the prior distribution

prep

doc = the distribution to use to initialize this parameter set

initialize(*self*, *model*, *offset*)

Initialize the parameter set given the {model} that owns it

initializeSample(*self*, *theta*)

Fill {theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *theta*, *priorLLK*)

Fill {priorLLK} with the likelihoods of the samples in {theta} in my prior distribution

verify(*self*, *theta*, *mask*)

Check whether the samples in {theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.models.**bayesian**(*name*, *locator*, ***kwds*)

Bases: altar.component

The base class of AlTar models that are compatible with Bayesian explorations

offset

doc = the starting point of my state in the overall controller state

parameters

doc = the number of model degrees of freedom

psets

default

doc = an ensemble of parameter sets in the model

rng

controller

job

info

warning

error

default

firewall

initialize(*self*, *application*)

Initialize the state of the model given an {application} context

posterior(*self*, *application*)

Sample my posterior distribution

initializeSample(*self*, *step*)

Fill {step.theta} with an initial random sample from my prior distribution.

priorLikelihood(*self*, *step*)

Fill {step.prior} with the likelihoods of the samples in {step.theta} in the prior distribution

dataLikelihood(*self*, *step*)

Fill {step.data} with the likelihoods of the samples in {step.theta} given the available data. This is what is usually referred to as the "forward model"

posteriorLikelihood(*self*, *step*)

Given the {step.prior} and {step.data} likelihoods, compute a generalized posterior using {step.beta} and deposit the result in {step.post}

likelihoods(*self*, *annealer*, *step*)

Convenience function that computes all three likelihoods at once given the current {step} of the problem

abstract verify(*self*, *step*, *mask*)

Check whether the samples in {step.theta} are consistent with the model requirements and update the {mask}, a vector with zeroes for valid samples and non-zero for invalid ones

top(*self*, *annealer*)

Notification that a β step is about to start

bottom(*self*, *annealer*)

Notification that a β step just ended

forwardProblem(*self*, *application*, *theta=None*)

Perform the forward modeling with given {theta}

mountInputDataspace(*self*, *pfs*)

Mount the directory with my input files

restrict(*self*, *theta*)

Return my portion of the sample matrix {theta}

altar.models.**null**()

altar.models.**ensemble**()

altar.models.**bayesianl2**()

altar.models.**contiguous**()

7.1.1.8 altar.norms

Submodules

altar.norms.L2

Module Contents

Classes

class altar.norms.L2.**L2**(*name*, *locator*, ***kwds*)

Bases: altar.component

The L2 norm

eval(*self*, *v*, *sigma_inv=None*)

Compute the L2 norm of the given vector, with or without a covariance matrix

withCovariance (*self*, *v*, *sigma_inv*)
Compute the L2 norm of the given vector using the given Cholesky decomposed inverse covariance matrix

altar.norms.Norm

Module Contents

Classes

class altar.norms.Norm.**Norm**
Bases: altar.protocol
The protocol that all AlTar norms must satify
eval (*self*, *v*, ***kwds*)
Compute the L2 norm of the given vector
classmethod pyre_default (*cls*, ***kwds*)
Provide a default norm in case the user hasn't selected one

Package Contents

Classes

Functions

class altar.norms.**norm**
Bases: altar.protocol
The protocol that all AlTar norms must satify
eval (*self*, *v*, ***kwds*)
Compute the L2 norm of the given vector
classmethod pyre_default (*cls*, ***kwds*)
Provide a default norm in case the user hasn't selected one

altar.norms.**l2**()

7.1.1.9 altar.shells

Submodules

altar.shells.Action

Module Contents

Classes

class altar.shells.Action.**Action**

Bases: altar.action

Protocol for {altar} commands

altar.shells.AlTar

Module Contents

Classes

class altar.shells.AlTar.**AlTar**(***kwds*)

Bases: altar.plexus

The main action dispatcher for the simple AlTar application

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main(*self*, **args*, ***kwds*)

The main entry point

initialize(*self*)

Initialize without running, for debug purpose only

pyre_banner(*self*)

Place the application banner in the {info} channel

pyre_interactiveSessionContext(*self*, *context*)

Go interactive

altar.shells.Application

Module Contents

Classes

class altar.shells.Application.**Application**(*name=None*, ***kwds*)

Bases: altar.application

The base class for simple AlTar applications

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main(*self*, **args*, ***kwds*)

The main entry point

pyre_interactiveSessionContext(*self*, *context*)

Go interactive

pyre_mpi(*self*)

Transfer my {job} settings to the MPI shell

altar.shells.cudaAlTar

Module Contents

Classes

class altar.shells.cudaAlTar.**cudaAlTar**(***kwds*)

Bases: altar.plexus

The main action dispatcher for the simple AlTar application

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)
Initialize without running, for debug purpose only

pyre_banner (*self*)
Place the application banner in the {info} channel

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

pyre_mpi (*self*)
Transfer my {job} settings to the MPI shell

altar.shells.cudaApplication

Module Contents

Classes

class altar.shells.cudaApplication.**cudaApplication**(*name=None*, ***kwds*)
Bases: altar.application

The base class for simple AlTar applications

job

doc = the job input parameters

model

default

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)
The main entry point

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

pyre_mpi (*self*)
Transfer my {job} settings to the MPI shell

Package Contents

Classes

class altar.shells.**application** (*name=None*, ***kwds*)
Bases: altar.application

The base class for simple AlTar applications

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)
The main entry point

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

pyre_mpi (*self*)
Transfer my {job} settings to the MPI shell

class altar.shells.**altar** (***kwds*)
Bases: altar.plexus

The main action dispatcher for the simple AlTar application

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)

The main entry point

initialize (*self*)

Initialize without running, for debug purpose only

pyre_banner (*self*)

Place the application banner in the {info} channel

pyre_interactiveSessionContext (*self*, *context*)

Go interactive

class altar.shells.**cudaapplication** (*name=None*, ***kwds*)

Bases: altar.application

The base class for simple AlTar applications

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)

The main entry point

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

pyre_mpi (*self*)
Transfer my {job} settings to the MPI shell

class altar.shells.**cudaaltar**(***kwds*)
Bases: altar.plexus

The main action dispatcher for the simple AlTar application

job

doc = the job input parameters

model

doc = the AlTar model to sample

rng

doc = the random number generator

controller

doc = my simulation controller

monitors

doc = a collection of event handlers

main (*self*, **args*, ***kwds*)
The main entry point

initialize (*self*)
Initialize without running, for debug purpose only

pyre_banner (*self*)
Place the application banner in the {info} channel

pyre_interactiveSessionContext (*self*, *context*)
Go interactive

pyre_mpi (*self*)
Transfer my {job} settings to the MPI shell

7.1.1.10 altar.simulations

Submodules

altar.simulations.Archiver

Module Contents

Classes

class altar.simulations.Archiver.**Archiver**

Bases: altar.protocol

The protocol that all AlTar simulation archivers must implement

Archivers persist intermediate simulation state and can be used to restart a simulation

initialize(*self*, *application*)

Initialize me given an {application} context

record(*self*, *step*)

Record the final state of the simulation

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.simulations.Dispatcher

Module Contents

Classes

class altar.simulations.Dispatcher.**Dispatcher**

Bases: altar.protocol

The protocol that all AlTar simulation dispatchers must implement

Dispatchers associate event handlers with specific aspects of the calculation and invoke them when appropriate

initialize(*self*, *application*)

Initialize me given an {application} context

altar.simulations.GSLRNG

Module Contents

Classes

class altar.simulations.GSLRNG.**GSLRNG**(***kwds*)

Bases: altar.component

The protocol for random number generators

seed

doc = the number with which to seed the generator

algorithm

doc = the random number generator algorithm

rng

initialize (*self*, ***kwds*)

Initialize the random number generator

show (*self*)

Display some information about me

altar.simulations.Job

Module Contents

Classes

class altar.simulations.Job.**Job**(*name*, *locator*, ***kwds*)

Bases: altar.component

The set of parameters in an AlTar run

name

doc = the name of the job; used as a stem for making filenames, etc.

mode

mode = the programming model

hosts

doc = the number of hosts to run on

tasks

doc = the number of tasks per host

gpus

doc = the number of gpus per task

gpuprecision

doc = the precision of gpu computations

validators

gpuids

default

doc = the list of gpu ids for parallel jobs

chains

doc = the number of chains per worker

steps

doc = the length of each Markov chain

tolerance

doc = convergence tolerance for β->1.0

initialize(*self*, *application*)

Initialize the job parameters with information from the application context

validateMachineLayout(*self*, *application*)

Adjust the machine parameters based on the {application} context and the runtime environment

altar.simulations.Monitor

Module Contents

Classes

class altar.simulations.Monitor.**Monitor**

Bases: altar.protocol

The protocol that all AlTar simulation monitors must implement

Monitors respond to simulation events by generating user diagnostics to report progress

initialize(*self*, *application*)

Initialize me given an {application} context

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.simulations.RNG

Module Contents

Classes

class altar.simulations.RNG.**RNG**

Bases: altar.protocol

The protocol for random number generators

initialize(*self*, ***kwds*)

Initialize the random number generator

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.simulations.Recorder

Module Contents

Classes

class altar.simulations.Recorder.**Recorder**(*name*, *locator*, ***kwds*)

Bases: altar.component

Recorder stores the intermediate simulation state in memory

theta

doc = the path to the file with the final posterior sample

sigma

doc = the path to the file with the final parameter correlation matrix

llk

doc = the path to the file with the final posterior log likelihood

initialize(*self*, *application*)

Initialize me given an {application} context

record(*self*, *step*, ***kwds*)

Record the final state of the calculation

altar.simulations.Reporter

Module Contents

Classes

class altar.simulations.Reporter.**Reporter**(*name*, *locator*, ***kwds*)

Bases: altar.component

Reporter reports simulation progress by using application journal channels

initialize(*self*, *application*)

Initialize me given an {application} context

simulationStart(*self*, *controller*, ***kwds*)

Handler invoked when the simulation is about to start

samplePosteriorStart(*self*, *controller*, ***kwds*)

Handler invoked at the beginning of sampling the posterior

prepareSamplingPDFStart(*self*, *controller*, ***kwds*)

Handler invoked at the beginning of the preparation of the sampling PDF

prepareSamplingPDFFinish(*self*, *controller*, ***kwds*)

Handler invoked at the end of the preparation of the sampling PDF

betaStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of the beta step

walkChainsStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of the chain walk

chainAdvanceStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of a single step of chain walking

chainAdvanceFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of a single step of chain walking

verifyStart (*self*, *controller*, ***kwds*)
Handler invoked before we start verifying the generated sample

verifyFinish (*self*, *controller*, ***kwds*)
Handler invoked after we are done verifying the generated sample

priorStart (*self*, *controller*, ***kwds*)
Handler invoked before we compute the prior

priorFinish (*self*, *controller*, ***kwds*)
Handler invoked after we compute the prior

dataStart (*self*, *controller*, ***kwds*)
Handler invoked before we compute the data likelihood

dataFinish (*self*, *controller*, ***kwds*)
Handler invoked after we compute the data likelihood

posteriorStart (*self*, *controller*, ***kwds*)
Handler invoked before we assemble the posterior

posteriorFinish (*self*, *controller*, ***kwds*)
Handler invoked after we assemble the posterior

acceptStart (*self*, *controller*, ***kwds*)
Handler invoked at the beginning of sample accept/reject

acceptFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of sample accept/reject

walkChainsFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of the chain walk

resampleStart (*self*, *controller*, ***kwds*)
Handler invoked before we start resampling

resampleFinish (*self*, *controller*, ***kwds*)
Handler invoked after we are done resampling

betaFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of the beta step

samplePosteriorFinish (*self*, *controller*, ***kwds*)
Handler invoked at the end of sampling the posterior

simulationFinish (*self*, *controller*, ***kwds*)
Handler invoked when the simulation is about to finish

altar.simulations.Run

Module Contents

Classes

class altar.simulations.Run.**Run**

Bases: altar.protocol

Protocol that specifies the job parameter set

name

doc = the name of the job; used as a stem for making filenames, etc.

mode

mode = the programming model

hosts

doc = the number of hosts to run on

tasks

doc = the number of tasks per host

gpus

doc = the number of gpus per task

chains

doc = the number of chains per worker

steps

doc = the length of each Markov chain

tolerance

doc = convergence tolerance for β->1.0

initialize(*self*, *application*)

Initialize the run components with context from {application}

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

Package Contents

Classes

Functions

class altar.simulations.**archiver**

Bases: altar.protocol

The protocol that all AlTar simulation archivers must implement

Archivers persist intermediate simulation state and can be used to restart a simulation

initialize(*self*, *application*)

Initialize me given an {application} context

record(*self*, *step*)

Record the final state of the simulation

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.simulations.**dispatcher**

Bases: altar.protocol

The protocol that all AlTar simulation dispatchers must implement

Dispatchers associate event handlers with specific aspects of the calculation and invoke them when appropriate

initialize(*self*, *application*)

Initialize me given an {application} context

class altar.simulations.**monitor**

Bases: altar.protocol

The protocol that all AlTar simulation monitors must implement

Monitors respond to simulation events by generating user diagnostics to report progress

initialize(*self*, *application*)

Initialize me given an {application} context

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.simulations.**run**

Bases: altar.protocol

Protocol that specifies the job parameter set

name

doc = the name of the job; used as a stem for making filenames, etc.

mode

mode = the programming model

hosts

doc = the number of hosts to run on

tasks

doc = the number of tasks per host

gpus

doc = the number of gpus per task

chains

doc = the number of chains per worker

steps

doc = the length of each Markov chain

tolerance

doc = convergence tolerance for β->1.0

initialize(*self*, *application*)

Initialize the run components with context from {application}

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

class altar.simulations.**rng**

Bases: altar.protocol

The protocol for random number generators

initialize(*self*, ***kwds*)

Initialize the random number generator

classmethod pyre_default(*cls*, ***kwds*)

Supply a default implementation

altar.simulations.**gsl**()

altar.simulations.**job**()

altar.simulations.**recorder**()

altar.simulations.**reporter**()

7.1.2 Submodules

7.1.2.1 altar.meta

Module Contents

altar.meta.**date = 2021-07-16T04:55:48Z**

altar.meta.**major = 2**

altar.meta.**minor = 0**

altar.meta.**micro = 1**

altar.meta.**revision = c4eeb4c**

altar.meta.**version**

altar.meta.**copyright = (c) 2013- ParaSim Inc. (c) 2010- California Institute of Technology ALL RIGHTS RESERVED**

altar.meta.**banner**

altar.meta.**header**

altar.meta.**license**

altar.meta.**acknowledgments**

7.1.3 Package Contents

7.1.3.1 Functions

altar.**package**

altar.**modelPrefix**

altar.**copyright**()
Return the altar copyright note

altar.**license**()
Print the altar license

altar.**version**()
Return the altar version

altar.**credits**()
Print the acknowledgments

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

altar, 79
altar.actions, 79
altar.actions.About, 79
altar.actions.Forward, 80
altar.actions.Sample, 80
altar.bayesian, 81
altar.bayesian.Annealer, 81
altar.bayesian.AnnealingMethod, 82
altar.bayesian.Brent, 83
altar.bayesian.Controller, 85
altar.bayesian.CoolingStep, 86
altar.bayesian.COV, 83
altar.bayesian.CUDAAnnealing, 85
altar.bayesian.Grid, 87
altar.bayesian.H5Recorder, 87
altar.bayesian.Metropolis, 89
altar.bayesian.MPIAnnealing, 88
altar.bayesian.Notifier, 90
altar.bayesian.Profiler, 91
altar.bayesian.Recorder, 93
altar.bayesian.Sampler, 93
altar.bayesian.Scheduler, 94
altar.bayesian.SequentialAnnealing, 94
altar.bayesian.Solver, 95
altar.bayesian.ThreadedAnnealing, 95
altar.cuda, 98
altar.cuda.bayesian, 98
altar.cuda.bayesian.cudaAdaptiveMetropolis, 98
altar.cuda.bayesian.cudaCoolingStep, 100
altar.cuda.bayesian.cudaMetropolis, 101
altar.cuda.bayesian.cudaMetropolisVaryingSteps, 103
altar.cuda.data, 106
altar.cuda.data.cudaDataL2, 106
altar.cuda.distributions, 108
altar.cuda.distributions.cudaDistribution, 108
altar.cuda.distributions.cudaGaussian, 109
altar.cuda.distributions.cudaPreset, 109
altar.cuda.distributions.cudaTGaussian, 110
altar.cuda.distributions.cudaUniform, 111
altar.cuda.ext, 113
altar.cuda.models, 113
altar.cuda.models.cudaBayesian, 113
altar.cuda.models.cudaBayesianEnsemble, 116
altar.cuda.models.cudaParameterEnsemble, 118
altar.cuda.models.cudaParameterSet, 119
altar.cuda.norms, 121
altar.cuda.norms.cudaL2, 121
altar.data, 123
altar.data.DataL2, 123
altar.data.DataObs, 124
altar.distributions, 126
altar.distributions.Base, 126
altar.distributions.Distribution, 127
altar.distributions.Gaussian, 128
altar.distributions.Uniform, 128
altar.distributions.UnitGaussian, 129
altar.ext, 130
altar.meta, 198
altar.models, 130
altar.models.Bayesian, 173
altar.models.BayesianL2, 175
altar.models.cdm, 130
altar.models.cdm.CDM, 130
altar.models.cdm.CUDA, 132
altar.models.cdm.Data, 133
altar.models.cdm.ext, 130
altar.models.cdm.Fast, 133
altar.models.cdm.libcdm, 135
altar.models.cdm.meta, 136
altar.models.cdm.Native, 134
altar.models.cdm.Source, 134
altar.models.Contiguous, 177
altar.models.cudalinear, 138
altar.models.cudalinear.cudaLinear, 138
altar.models.cudalinear.meta, 139

altar.models.emhp, 140
altar.models.emhp.EMHP, 140
altar.models.emhp.meta, 140
altar.models.Ensemble, 178
altar.models.gaussian, 141
altar.models.gaussian.ext, 141
altar.models.gaussian.Gaussian, 141
altar.models.gaussian.meta, 142
altar.models.linear, 143
altar.models.linear.Linear, 143
altar.models.linear.meta, 145
altar.models.Model, 178
altar.models.mogi, 145
altar.models.mogi.CUDA, 145
altar.models.mogi.Data, 146
altar.models.mogi.ext, 145
altar.models.mogi.Fast, 147
altar.models.mogi.meta, 150
altar.models.mogi.Mogi, 147
altar.models.mogi.Native, 149
altar.models.mogi.Source, 149
altar.models.Null, 179
altar.models.ParameterSet, 180
altar.models.regression, 151
altar.models.regression.Linear, 151
altar.models.regression.meta, 152
altar.models.seismic, 153
altar.models.seismic.actions, 153
altar.models.seismic.actions.About, 153
altar.models.seismic.actions.Forward, 154
altar.models.seismic.actions.Sample, 155
altar.models.seismic.cuda, 155
altar.models.seismic.cuda.cudaCascaded, 155
altar.models.seismic.cuda.cudaKinematicG, 156
altar.models.seismic.cuda.cudaKinematicGCp, 157
altar.models.seismic.cuda.cudaMoment, 158
altar.models.seismic.cuda.cudaStatic, 159
altar.models.seismic.cuda.cudaStaticCp, 160
altar.models.seismic.ext, 165
altar.models.seismic.meta, 171
altar.models.seismic.Moment, 167
altar.models.seismic.shells, 165
altar.models.seismic.shells.Action, 165
altar.models.seismic.shells.cudaApplication, 166
altar.models.seismic.shells.Seismic, 165
altar.models.seismic.Static, 168
altar.models.seismic.StaticCp, 170
altar.models.sir, 172
altar.models.sir.meta, 173
altar.models.sir.SIR, 172
altar.norms, 183
altar.norms.L2, 183
altar.norms.Norm, 184
altar.shells, 184
altar.shells.Action, 184
altar.shells.AlTar, 185
altar.shells.Application, 186
altar.shells.cudaAlTar, 186
altar.shells.cudaApplication, 187
altar.simulations, 190
altar.simulations.Archiver, 190
altar.simulations.Dispatcher, 191
altar.simulations.GSLRNG, 191
altar.simulations.Job, 192
altar.simulations.Monitor, 193
altar.simulations.Recorder, 194
altar.simulations.Reporter, 194
altar.simulations.RNG, 193
altar.simulations.Run, 196

INDEX

Symbols

_SIR_Rate() (*altar.models.sir.SIR.SIR method*), 172
_forwardModel() (*altar.models.cudalinear.cudaLinear.cudaLinear method*), 139
_loadhdf5() (*altar.cuda.distributions.cudaPreset.cudaPreset method*), 110

A

About (*class in altar.actions.About*), 79
About (*class in altar.models.seismic.actions.About*), 153
about() (*in module altar.actions*), 81
about() (*in module altar.models.seismic.actions*), 155
acceptanceWeight (*altar.bayesian.Metropolis.Metropolis attribute*), 89
acceptanceWeight (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
acceptanceWeight (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
acceptFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
acceptFinish() (*altar.bayesian.Profiler.Profiler method*), 92
acceptFinish() (*altar.simulations.Reporter.Reporter method*), 195
acceptStart (*altar.bayesian.Notifier.Notifier attribute*), 90
acceptStart() (*altar.bayesian.Profiler.Profiler method*), 92
acceptStart() (*altar.simulations.Reporter.Reporter method*), 195
acknowledgments (*in module altar.meta*), 199
acknowledgments (*in module altar.models.cdm.meta*), 137
acknowledgments (*in module altar.models.cudalinear.meta*), 139
acknowledgments (*in module altar.models.emhp.meta*), 141
acknowledgments (*in module altar.models.gaussian.meta*), 142
acknowledgments (*in module altar.models.linear.meta*), 145
acknowledgments (*in module altar.models.mogi.meta*), 150
acknowledgments (*in module altar.models.regression.meta*), 152
acknowledgments (*in module altar.models.seismic.meta*), 172
acknowledgments (*in module altar.models.sir.meta*), 173
Action (*class in altar.models.seismic.shells.Action*), 165
Action (*class in altar.shells.Action*), 184
adaptivemetropolis() (*in module altar.cuda.bayesian*), 106
adjustCovarianceScaling() (*altar.bayesian.Metropolis.Metropolis method*), 90
adjustCovarianceScaling() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis method*), 100
adjustCovarianceScaling() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
adjustCovarianceScaling() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary method*), 104
algorithm (*altar.simulations.GSLRNG.GSLRNG attribute*), 191
alloc() (*altar.bayesian.CoolingStep.CoolingStep class method*), 86
alloc() (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep class method*), 100
allocate() (*altar.bayesian.CoolingStep.CoolingStep class method*), 86
allocateGPUData() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis method*), 100
allocateGPUData() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102

allocateGPUData() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps method*), 104
altar
    module, 79
altar (*class in altar.shells*), 188
AlTar (*class in altar.shells.AlTar*), 185
altar.actions
    module, 79
altar.actions.About
    module, 79
altar.actions.Forward
    module, 80
altar.actions.Sample
    module, 80
altar.bayesian
    module, 81
altar.bayesian.Annealer
    module, 81
altar.bayesian.AnnealingMethod
    module, 82
altar.bayesian.Brent
    module, 83
altar.bayesian.Controller
    module, 85
altar.bayesian.CoolingStep
    module, 86
altar.bayesian.COV
    module, 83
altar.bayesian.CUDAAnnealing
    module, 85
altar.bayesian.Grid
    module, 87
altar.bayesian.H5Recorder
    module, 87
altar.bayesian.Metropolis
    module, 89
altar.bayesian.MPIAnnealing
    module, 88
altar.bayesian.Notifier
    module, 90
altar.bayesian.Profiler
    module, 91
altar.bayesian.Recorder
    module, 93
altar.bayesian.Sampler
    module, 93
altar.bayesian.Scheduler
    module, 94
altar.bayesian.SequentialAnnealing
    module, 94
altar.bayesian.Solver
    module, 95
altar.bayesian.ThreadedAnnealing
    module, 95
altar.bayesian.ThreadedMetropolis
    module, 98
altar.cuda.bayesian
    module, 98
altar.cuda.bayesian.cudaAdaptiveMetropolis
    module, 98
altar.cuda.bayesian.cudaCoolingStep
    module, 100
altar.cuda.bayesian.cudaMetropolis
    module, 101
altar.cuda.bayesian.cudaMetropolisVaryingSteps
    module, 103
altar.cuda.data
    module, 106
altar.cuda.data.cudaDataL2
    module, 106
altar.cuda.distributions
    module, 108
altar.cuda.distributions.cudaDistribution
    module, 108
altar.cuda.distributions.cudaGaussian
    module, 109
altar.cuda.distributions.cudaPreset
    module, 109
altar.cuda.distributions.cudaTGaussian
    module, 110
altar.cuda.distributions.cudaUniform
    module, 111
altar.cuda.ext
    module, 113
altar.cuda.models
    module, 113
altar.cuda.models.cudaBayesian
    module, 113
altar.cuda.models.cudaBayesianEnsemble
    module, 116
altar.cuda.models.cudaParameterEnsemble
    module, 118
altar.cuda.models.cudaParameterSet
    module, 119
altar.cuda.norms
    module, 121
altar.cuda.norms.cudaL2
    module, 121
altar.data
    module, 123
altar.data.DataL2
    module, 123
altar.data.DataObs
    module, 124
altar.distributions
    module, 126
altar.distributions.Base

module, 126
altar.distributions.Distribution
module, 127
altar.distributions.Gaussian
module, 128
altar.distributions.Uniform
module, 128
altar.distributions.UnitGaussian
module, 129
altar.ext
module, 130
altar.meta
module, 198
altar.models
module, 130
altar.models.Bayesian
module, 173
altar.models.BayesianL2
module, 175
altar.models.cdm
module, 130
altar.models.cdm.CDM
module, 130
altar.models.cdm.CUDA
module, 132
altar.models.cdm.Data
module, 133
altar.models.cdm.ext
module, 130
altar.models.cdm.Fast
module, 133
altar.models.cdm.libcdm
module, 135
altar.models.cdm.meta
module, 136
altar.models.cdm.Native
module, 134
altar.models.cdm.Source
module, 134
altar.models.Contiguous
module, 177
altar.models.cudalinear
module, 138
altar.models.cudalinear.cudaLinear
module, 138
altar.models.cudalinear.meta
module, 139
altar.models.emhp
module, 140
altar.models.emhp.EMHP
module, 140
altar.models.emhp.meta
module, 140
altar.models.Ensemble
module, 178
altar.models.gaussian
module, 141
altar.models.gaussian.ext
module, 141
altar.models.gaussian.Gaussian
module, 141
altar.models.gaussian.meta
module, 142
altar.models.linear
module, 143
altar.models.linear.Linear
module, 143
altar.models.linear.meta
module, 145
altar.models.Model
module, 178
altar.models.mogi
module, 145
altar.models.mogi.CUDA
module, 145
altar.models.mogi.Data
module, 146
altar.models.mogi.ext
module, 145
altar.models.mogi.Fast
module, 147
altar.models.mogi.meta
module, 150
altar.models.mogi.Mogi
module, 147
altar.models.mogi.Native
module, 149
altar.models.mogi.Source
module, 149
altar.models.Null
module, 179
altar.models.ParameterSet
module, 180
altar.models.regression
module, 151
altar.models.regression.Linear
module, 151
altar.models.regression.meta
module, 152
altar.models.seismic
module, 153
altar.models.seismic.actions
module, 153
altar.models.seismic.actions.About
module, 153
altar.models.seismic.actions.Forward
module, 154
altar.models.seismic.actions.Sample

module, 155
altar.models.seismic.cuda
module, 155
altar.models.seismic.cuda.cudaCascaded
module, 155
altar.models.seismic.cuda.cudaKinematicG
module, 156
altar.models.seismic.cuda.cudaKinematicGp
module, 157
altar.models.seismic.cuda.cudaMoment
module, 158
altar.models.seismic.cuda.cudaStatic
module, 159
altar.models.seismic.cuda.cudaStaticCp
module, 160
altar.models.seismic.ext
module, 165
altar.models.seismic.meta
module, 171
altar.models.seismic.Moment
module, 167
altar.models.seismic.shells
module, 165
altar.models.seismic.shells.Action
module, 165
altar.models.seismic.shells.cudaApplication
module, 166
altar.models.seismic.shells.Seismic
module, 165
altar.models.seismic.Static
module, 168
altar.models.seismic.StaticCp
module, 170
altar.models.sir
module, 172
altar.models.sir.meta
module, 173
altar.models.sir.SIR
module, 172
altar.norms
module, 183
altar.norms.L2
module, 183
altar.norms.Norm
module, 184
altar.shells
module, 184
altar.shells.Action
module, 184
altar.shells.AlTar
module, 185
altar.shells.Application
module, 186
altar.shells.cudaAlTar
module, 186
altar.shells.cudaApplication
module, 187
altar.simulations
module, 190
altar.simulations.Archiver
module, 190
altar.simulations.Dispatcher
module, 191
altar.simulations.GSLRNG
module, 191
altar.simulations.Job
module, 192
altar.simulations.Monitor
module, 193
altar.simulations.Recorder
module, 194
altar.simulations.Reporter
module, 194
altar.simulations.RNG
module, 193
altar.simulations.Run
module, 196
AngDisDispSurf() (*in module altar.models.cdm.libcdm*), 136
AngSetupFSC() (*in module altar.models.cdm.libcdm*), 136
Annealer (*class in altar.bayesian.Annealer*), 81
annealer() (*in module altar.bayesian*), 97
AnnealingMethod (*class in altar.bayesian.AnnealingMethod*), 82
Application (*built-in class*), 31
application (*class in altar.shells*), 188
Application (*class in altar.shells.Application*), 186
archive() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 83
archive() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
archiver (*altar.bayesian.Annealer.Annealer attribute*), 81
archiver (*altar.bayesian.controller attribute*), 95
archiver (*altar.bayesian.Controller.Controller attribute*), 85
archiver (*altar.cuda.bayesian.controller attribute*), 105
archiver (*class in altar.bayesian*), 97
archiver (*class in altar.simulations*), 197
Archiver (*class in altar.simulations.Archiver*), 190
area (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
area (*altar.models.seismic.Moment.Moment attribute*), 168
area_patch_file (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 158

area_patches (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
area_patches (*altar.models.seismic.Moment.Moment attribute*), 168
area_patches_file (*altar.models.seismic.Moment.Moment attribute*), 168
ax (*altar.models.cdm.source attribute*), 137
ax (*altar.models.cdm.Source.Source attribute*), 134
aXIdx (*altar.models.cdm.CDM.CDM attribute*), 131
ay (*altar.models.cdm.source attribute*), 137
ay (*altar.models.cdm.Source.Source attribute*), 134
aYIdx (*altar.models.cdm.CDM.CDM attribute*), 131
az (*altar.models.cdm.source attribute*), 137
az (*altar.models.cdm.Source.Source attribute*), 134
aZIdx (*altar.models.cdm.CDM.CDM attribute*), 131

B

banner (*in module altar.meta*), 199
banner (*in module altar.models.cdm.meta*), 136
banner (*in module altar.models.cudalinear.meta*), 139
banner (*in module altar.models.emhp.meta*), 140
banner (*in module altar.models.gaussian.meta*), 142
banner (*in module altar.models.linear.meta*), 145
banner (*in module altar.models.mogi.meta*), 150
banner (*in module altar.models.regression.meta*), 152
banner (*in module altar.models.seismic.meta*), 172
banner (*in module altar.models.sir.meta*), 173
Base (*class in altar.distributions.Base*), 126
bayesian (*class in altar.models*), 182
Bayesian (*class in altar.models.Bayesian*), 173
bayesian() (*in module altar.cuda.models*), 121
bayesianensemble() (*in module altar.cuda.models*), 121
BayesianL2 (*class in altar.models.BayesianL2*), 175
bayesianl2() (*in module altar.models*), 183
beta (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
beta (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep attribute*), 100
beta_cp_start (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 158
beta_cp_start (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
beta_resampling_start (*altar.bayesian.COV.COV attribute*), 84
beta_stage2 (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
beta_use_initial_model (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 158
beta_use_initial_model (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
betaFinish (*altar.bayesian.Notifier.Notifier attribute*), 91
betaFinish() (*altar.bayesian.Profiler.Profiler method*), 92
betaFinish() (*altar.simulations.Reporter.Reporter method*), 195
betaStart (*altar.bayesian.Notifier.Notifier attribute*), 90
betaStart() (*altar.bayesian.Profiler.Profiler method*), 91
betaStart() (*altar.simulations.Reporter.Reporter method*), 194
bottom() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 83
bottom() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
bottom() (*altar.cuda.models.model method*), 120
bottom() (*altar.models.bayesian method*), 183
bottom() (*altar.models.Bayesian.Bayesian method*), 174
bottom() (*altar.models.model method*), 181
bottom() (*altar.models.Model.Model method*), 179
bottom() (*altar.models.regression.model method*), 153
Brent (*class in altar.bayesian.Brent*), 83
brent() (*in module altar.bayesian*), 97
buildHistogramRanges() (*altar.bayesian.COV.COV method*), 85

C

cascaded (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
cascaded (*altar.models.BayesianL2.BayesianL2 attribute*), 175
cascaded (*altar.models.seismic.cuda.model attribute*), 162
cascaded() (*in module altar.models.seismic.cuda*), 164
case (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
case (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
case (*altar.models.BayesianL2.BayesianL2 attribute*), 175
case (*altar.models.cdm.CDM.CDM attribute*), 130
case (*altar.models.linear.Linear.Linear attribute*), 143
case (*altar.models.mogi.Mogi.Mogi attribute*), 147
case (*altar.models.seismic.cuda.model attribute*), 162
case (*altar.models.seismic.Static.Static attribute*), 169
castSlipsOfTime() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
cd (*altar.data.DataL2 attribute*), 125
cd (*altar.data.DataL2.DataL2 attribute*), 123
cd (*altar.models.cdm.CDM.CDM attribute*), 131
Cd (*altar.models.linear.Linear.Linear attribute*), 144

cd (*altar.models.linear.Linear.Linear attribute*), 144
cd (*altar.models.mogi.Mogi.Mogi attribute*), 148
Cd (*altar.models.seismic.Static.Static attribute*), 169
Cd0 (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
cd_file (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
cd_file (*altar.data.DataL2 attribute*), 125
cd_file (*altar.data.DataL2.DataL2 attribute*), 123
cd_file (*altar.models.seismic.Static.Static attribute*), 169
cd_inv (*altar.data.DataL2 attribute*), 125
cd_inv (*altar.data.DataL2.DataL2 attribute*), 123
cd_inv (*altar.models.cdm.CDM.CDM attribute*), 131
Cd_inv (*altar.models.linear.Linear.Linear attribute*), 144
cd_inv (*altar.models.mogi.Mogi.Mogi attribute*), 148
Cd_inv (*altar.models.seismic.Static.Static attribute*), 169
cd_std (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
cd_std (*altar.data.DataL2 attribute*), 125
cd_std (*altar.data.DataL2.DataL2 attribute*), 123
CDM (*class in altar.models.cdm.CDM*), 130
cdm() (*in module altar.models.cdm*), 138
CDM() (*in module altar.models.cdm.libcdm*), 135
chainAdvanceFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
chainAdvanceFinish() (*altar.bayesian.Profiler.Profiler method*), 92
chainAdvanceFinish() (*altar.simulations.Reporter.Reporter method*), 195
chainAdvanceStart (*altar.bayesian.Notifier.Notifier attribute*), 90
chainAdvanceStart() (*altar.bayesian.Profiler.Profiler method*), 92
chainAdvanceStart() (*altar.simulations.Reporter.Reporter method*), 195
chains (*altar.simulations.Job.Job attribute*), 192
chains (*altar.simulations.run attribute*), 198
chains (*altar.simulations.Run.Run attribute*), 196
check_positive_definiteness (*altar.bayesian.COV.COV attribute*), 84
checkPositiveDefiniteness() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 107
clone() (*altar.bayesian.CoolingStep.CoolingStep method*), 86
clone() (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep method*), 100
cmodel (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
Cmu (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
cmu_file (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 158
cmu_file (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
cmu_file (*altar.models.seismic.StaticCp.StaticCp attribute*), 170
collect() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 89
computeCovariance() (*altar.bayesian.COV.COV method*), 84
computeCovariance() (*altar.bayesian.scheduler method*), 96
computeCovariance() (*altar.bayesian.Scheduler.Scheduler method*), 94
computeCovariance() (*altar.cuda.bayesian.scheduler method*), 105
computeCovarianceInverse() (*altar.data.DataL2 method*), 126
computeCovarianceInverse() (*altar.data.DataL2.DataL2 method*), 124
computeCovarianceInverse() (*altar.models.cdm.CDM.CDM method*), 132
computeCovarianceInverse() (*altar.models.linear.Linear.Linear method*), 144
computeCovarianceInverse() (*altar.models.mogi.Mogi.Mogi method*), 149
computeCovarianceInverse() (*altar.models.seismic.Static.Static method*), 170
computeCp() (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinemat method*), 158
computeCp() (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp method*), 161
computeCp() (*altar.models.seismic.StaticCp.StaticCp method*), 171
computeDataLikelihood() (*altar.models.seismic.Static.Static method*), 169
computeNormalization() (*altar.data.DataL2 method*), 126
computeNormalization() (*altar.data.DataL2.DataL2 method*), 124
computeNormalization() (*altar.models.cdm.CDM.CDM method*), 132
computeNormalization() (*altar.models.linear.Linear.Linear method*), 144
computeNormalization() (*altar.models.mogi.Mogi.Mogi method*), 149
computeNormalization() (*altar.models.seismic.Static.Static method*), 170
computePosterior() (*altar.bayesian.CoolingStep.CoolingStep method*), 86
computePosterior() (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep method*), 101

computePrior() (*altar.models.seismic.Static.Static method*), 169
computeSampleMultiplicities() (*altar.bayesian.COV.COV method*), 85
conditionCovariance() (*altar.bayesian.COV.COV method*), 84
Contiguous (*class in altar.models.Contiguous*), 177
contiguous() (*in module altar.models*), 183
controller (*altar.models.bayesian attribute*), 182
controller (*altar.models.Bayesian.Bayesian attribute*), 174
controller (*altar.models.seismic.shells.cudaApplication.cudaApplication attribute*), 166
controller (*altar.models.seismic.shells.seismic attribute*), 167
controller (*altar.models.seismic.shells.Seismic.Seismic attribute*), 165
controller (*altar.shells.altar attribute*), 189
controller (*altar.shells.AlTar.AlTar attribute*), 185
controller (*altar.shells.application attribute*), 188
controller (*altar.shells.Application.Application attribute*), 186
controller (*altar.shells.cudaaltar attribute*), 190
controller (*altar.shells.cudaAlTar.cudaAlTar attribute*), 187
controller (*altar.shells.cudaapplication attribute*), 189
controller (*altar.shells.cudaApplication.cudaApplication attribute*), 187
controller (*class in altar.bayesian*), 95
Controller (*class in altar.bayesian.Controller*), 85
controller (*class in altar.cuda.bayesian*), 104
cool() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
cool() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
CoolingStep (*class in altar.bayesian.CoolingStep*), 86
CoordTrans() (*in module altar.models.cdm.libcdm*), 136
copyFromCPU() (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep method*), 101
copyright (*in module altar.meta*), 199
copyright (*in module altar.models.cdm.meta*), 136
copyright (*in module altar.models.cudalinear.meta*), 139
copyright (*in module altar.models.emhp.meta*), 140
copyright (*in module altar.models.gaussian.meta*), 142
copyright (*in module altar.models.linear.meta*), 145
copyright (*in module altar.models.mogi.meta*), 150
copyright (*in module altar.models.regression.meta*), 152
copyright (*in module altar.models.seismic.meta*), 171
copyright (*in module altar.models.sir.meta*), 173
copyright() (*altar.actions.About.About method*), 80
copyright() (*altar.models.seismic.actions.About.About method*), 154
copyright() (*in module altar*), 199
copyright() (*in module altar.cuda*), 122
copyToCPU() (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep method*), 101
corr_check_steps (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
corr_check_steps (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary attribute*), 103
cosd() (*in module altar.models.cdm.libcdm*), 135
count (*altar.cuda.models.cudaParameterSet.cudaParameterSet attribute*), 119
count (*altar.cuda.models.parameters attribute*), 120
count (*altar.models.Contiguous.Contiguous attribute*), 177
count (*altar.models.parameters attribute*), 181
count (*altar.models.ParameterSet.ParameterSet attribute*), 180
cov (*altar.bayesian.Brent.Brent attribute*), 83
cov (*altar.bayesian.COV.COV attribute*), 84
cov (*altar.bayesian.Grid.Grid attribute*), 87
COV (*class in altar.bayesian.COV*), 83
cov() (*in module altar.bayesian*), 97
covariance (*altar.models.cdm.CDM.CDM attribute*), 131
covariance (*altar.models.mogi.Mogi.Mogi attribute*), 147
Cp (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
credits() (*altar.actions.About.About method*), 80
credits() (*altar.models.seismic.actions.About.About method*), 154
credits() (*in module altar*), 199
credits() (*in module altar.cuda*), 122
cublas_handle (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
cublas_handle (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 157
cublas_handle (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
cublas_handle() (*in module altar.cuda*), 122
CUDA (*class in altar.models.cdm.CUDA*), 132
CUDA (*class in altar.models.mogi.CUDA*), 145
cuda() (*altar.bayesian.Annealer.Annealer method*), 82
cudaAdaptiveMetropolis (*class in altar.cuda.bayesian.cudaAdaptiveMetropolis*), 98
cudaaltar (*class in altar.shells*), 190
cudaAlTar (*class in altar.shells.cudaAlTar*), 186
CUDAAnnealing (*class in al-*

tar.bayesian.CUDAAnnealing), 85
cudaApplication (*class in altar.models.seismic.shells.cudaApplication*), 166
cudaapplication (*class in altar.shells*), 189
cudaApplication (*class in altar.shells.cudaApplication*), 187
cudaBayesian (*class in altar.cuda.models.cudaBayesian*), 113
cudaBayesianEnsemble (*class in altar.cuda.models.cudaBayesianEnsemble*), 116
cudaCascaded (*class in altar.models.seismic.cuda.cudaCascaded*), 155
cudaCoolingStep (*class in altar.cuda.bayesian.cudaCoolingStep*), 100
cudaDataL2 (*class in altar.cuda.data.cudaDataL2*), 106
cudaDistribution (*class in altar.cuda.distributions*), 112
cudaDistribution (*class in altar.cuda.distributions.cudaDistribution*), 108
cudaGaussian (*class in altar.cuda.distributions.cudaGaussian*), 109
cudaKinematicG (*class in altar.models.seismic.cuda.cudaKinematicG*), 156
cudaKinematicGCp (*class in altar.models.seismic.cuda.cudaKinematicGCp*), 157
cudaL2 (*class in altar.cuda.norms.cudaL2*), 121
cudaLinear (*class in altar.models.cudalinear.cudaLinear*), 138
cudalinear() (*in module altar.models.cudalinear*), 139
cudaMetropolis (*class in altar.cuda.bayesian.cudaMetropolis*), 101
cudaMetropolisVaryingSteps (*class in altar.cuda.bayesian.cudaMetropolisVaryingSteps*), 103
cudaMoment (*class in altar.models.seismic.cuda.cudaMoment*), 158
cudaParameterEnsemble (*class in altar.cuda.models.cudaParameterEnsemble*), 118
cudaParameterSet (*class in altar.cuda.models.cudaParameterSet*), 119
cudaPreset (*class in altar.cuda.distributions.cudaPreset*), 109
cudaStatic (*class in altar.models.seismic.cuda.cudaStatic*), 159
cudaStaticCp (*class in altar.models.seismic.cuda.cudaStaticCp*), 160
cudaTGaussian (*class in altar.cuda.distributions.cudaTGaussian*), 110
cudaUniform (*class in altar.cuda.distributions.cudaUniform*), 111
cuEval() (*altar.cuda.norms.cudaL2.cudaL2 method*), 121
cuEvalLikelihood() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 107
cuEvalLikelihood() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
cuEvalLikelihood() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
cuEvalLikelihood() (*altar.cuda.norms.cudaL2.cudaL2 method*), 121
cuEvalLikelihood() (*altar.models.cudalinear.cudaLinear.cudaLinear method*), 139
cuEvalLikelihood() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
cuEvalLikelihood() (*altar.models.seismic.cuda.cudaStatic.cudaStatic method*), 160
cuEvalLikelihood() (*altar.models.seismic.cuda.model method*), 163
cuEvalPosterior() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
cuEvalPosterior() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
cuEvalPosterior() (*altar.models.seismic.cuda.model method*), 163
cuEvalPrior() (*altar.cuda.distributions.cudaDistribution method*), 112
cuEvalPrior() (*altar.cuda.distributions.cudaDistribution.cudaDistribution method*), 109
cuEvalPrior() (*altar.cuda.distributions.cudaGaussian.cudaGaussian method*), 109
cuEvalPrior() (*altar.cuda.distributions.cudaTGaussian.cudaTGaussian method*), 110
cuEvalPrior() (*altar.cuda.distributions.cudaUniform.cudaUniform method*), 111
cuEvalPrior() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
cuEvalPrior() (*al-*

tar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method), 117
cuEvalPrior() (*altar.cuda.models.cudaParameterEnsemble.cudaParameterEnsemble method*), 118
cuEvalPrior() (*altar.cuda.models.cudaParameterSet.cudaParameterSet method*), 119
cuEvalPrior() (*altar.models.seismic.cuda.model method*), 163
cuInitialize() (*altar.bayesian.CUDAAnnealing.CUDAAnnealing method*), 85
cuInitialize() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis method*), 99
cuInitialize() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
cuInitialize() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps method*), 104
cuInitialize() (*altar.cuda.distributions.cudaDistribution method*), 112
cuInitialize() (*altar.cuda.distributions.cudaDistribution.cudaDistribution method*), 108
cuInitialize() (*altar.cuda.distributions.cudaPreset.cudaPreset method*), 110
cuInitialize() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
cuInitialize() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 116
cuInitialize() (*altar.cuda.models.cudaParameterEnsemble.cudaParameterEnsemble method*), 118
cuInitialize() (*altar.cuda.models.cudaParameterSet.cudaParameterSet method*), 119
cuInitialize() (*altar.models.seismic.cuda.cudaMoment.cudaMoment method*), 159
cuInitialize() (*altar.models.seismic.cuda.model method*), 163
cuInitSample() (*altar.cuda.distributions.cudaDistribution method*), 112
cuInitSample() (*altar.cuda.distributions.cudaDistribution.cudaDistribution method*), 108
cuInitSample() (*altar.cuda.distributions.cudaGaussian.cudaGaussian method*), 109
cuInitSample() (*altar.cuda.distributions.cudaPreset.cudaPreset method*), 110
cuInitSample() (*altar.cuda.distributions.cudaTGaussian.cudaTGaussian method*), 110
cuInitSample() (*altar.cuda.distributions.cudaUniform.cudaUniform method*), 111
cuInitSample() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
cuInitSample() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
cuInitSample() (*altar.cuda.models.cudaParameterEnsemble.cudaParameterEnsemble method*), 118
cuInitSample() (*altar.cuda.models.cudaParameterSet.cudaParameterSet method*), 119
cuInitSample() (*altar.models.seismic.cuda.cudaMoment.cudaMoment method*), 159
cuInitSample() (*altar.models.seismic.cuda.model method*), 163
curand_generator() (*in module altar.cuda*), 122
cuRestrict() (*altar.cuda.models.cudaParameterSet.cudaParameterSet method*), 119
curng (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
curng (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
curng (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary attribute*), 104
cuVerify() (*altar.cuda.distributions.cudaDistribution method*), 112
cuVerify() (*altar.cuda.distributions.cudaDistribution.cudaDistribution method*), 108
cuVerify() (*altar.cuda.distributions.cudaGaussian.cudaGaussian method*), 109
cuVerify() (*altar.cuda.distributions.cudaTGaussian.cudaTGaussian method*), 110
cuVerify() (*altar.cuda.distributions.cudaUniform.cudaUniform method*), 111
cuVerify() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
cuVerify() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnse method*), 117
cuVerify() (*altar.cuda.models.cudaParameterEnsemble.cudaParameterEn method*), 118

cuVerify() (*altar.cuda.models.cudaParameterSet.cudaParameterSet method*), 119
cuVerify() (*altar.models.seismic.cuda.model method*), 163

D

d (*altar.models.cdm.CDM.CDM attribute*), 131
d (*altar.models.cdm.data attribute*), 138
d (*altar.models.cdm.Data.Data attribute*), 133
d (*altar.models.cdm.source attribute*), 137
d (*altar.models.cdm.Source.Source attribute*), 134
d (*altar.models.linear.Linear.Linear attribute*), 144
d (*altar.models.mogi.data attribute*), 151
d (*altar.models.mogi.Data.Data attribute*), 146
d (*altar.models.mogi.Mogi.Mogi attribute*), 148
d (*altar.models.mogi.source attribute*), 150
d (*altar.models.mogi.Source.Source attribute*), 149
d (*altar.models.seismic.Static.Static attribute*), 169
d0 (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
data (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
data (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep attribute*), 100
data (*altar.models.linear.Linear.Linear attribute*), 143
data (*class in altar.cuda.data*), 107
data (*class in altar.data*), 124
data (*class in altar.models.cdm*), 137
Data (*class in altar.models.cdm.Data*), 133
data (*class in altar.models.mogi*), 151
Data (*class in altar.models.mogi.Data*), 146
data_file (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
data_file (*altar.data.DataL2 attribute*), 125
data_file (*altar.data.DataL2.DataL2 attribute*), 123
data_file (*altar.models.seismic.Static.Static attribute*), 169
dataFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
dataFinish() (*altar.bayesian.Profiler.Profiler method*), 92
dataFinish() (*altar.simulations.Reporter.Reporter method*), 195
DataL2 (*class in altar.data*), 125
DataL2 (*class in altar.data.DataL2*), 123
datal2() (*in module altar.cuda.data*), 108
datal2() (*in module altar.data*), 126
dataLikelihood() (*altar.cuda.models.model method*), 120
dataLikelihood() (*altar.models.bayesian method*), 182
dataLikelihood() (*altar.models.Bayesian.Bayesian method*), 174
dataLikelihood() (*altar.models.cdm.CDM.CDM method*), 132
dataLikelihood() (*altar.models.cdm.CUDA.CUDA method*), 132
dataLikelihood() (*altar.models.cdm.Fast.Fast method*), 133
dataLikelihood() (*altar.models.cdm.Native.Native method*), 134
dataLikelihood() (*altar.models.emhp.EMHP.EMHP method*), 140
dataLikelihood() (*altar.models.Ensemble.Ensemble method*), 178
dataLikelihood() (*altar.models.gaussian.Gaussian.Gaussian method*), 142
dataLikelihood() (*altar.models.linear.Linear.Linear method*), 144
dataLikelihood() (*altar.models.model method*), 181
dataLikelihood() (*altar.models.Model.Model method*), 178
dataLikelihood() (*altar.models.mogi.CUDA.CUDA method*), 146
dataLikelihood() (*altar.models.mogi.Fast.Fast method*), 147
dataLikelihood() (*altar.models.mogi.Mogi.Mogi method*), 148
dataLikelihood() (*altar.models.mogi.Native.Native method*), 149
dataLikelihood() (*altar.models.Null.Null method*), 179
dataLikelihood() (*altar.models.regression.model method*), 153
datallk (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
dataobs (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
dataobs (*altar.data.DataL2 attribute*), 125
dataobs (*altar.data.DataL2.DataL2 attribute*), 123
dataobs (*altar.models.BayesianL2.BayesianL2 attribute*), 175
dataobs (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
dataobs (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
dataobs (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
dataobs (*altar.models.seismic.cuda.model attribute*), 162
DataObs (*class in altar.data.DataObs*), 124
dataobs_batch (*altar.data.DataL2 attribute*), 125
dataobs_batch (*altar.data.DataL2.DataL2 attribute*), 123
dataobsBatch() (*altar.data.DataL2 method*), 125
dataobsBatch() (*altar.data.DataL2.DataL2 method*), 124
dataset (*altar.cuda.distributions.cudaPreset.cudaPreset*

attribute), 109
dataStart (*altar.bayesian.Notifier.Notifier attribute*), 90
dataStart() (*altar.bayesian.Profiler.Profiler method*), 92
dataStart() (*altar.simulations.Reporter.Reporter method*), 195
date (*in module altar.meta*), 198
date (*in module altar.models.cdm.meta*), 136
date (*in module altar.models.cudalinear.meta*), 139
date (*in module altar.models.emhp.meta*), 140
date (*in module altar.models.gaussian.meta*), 142
date (*in module altar.models.linear.meta*), 145
date (*in module altar.models.mogi.meta*), 150
date (*in module altar.models.regression.meta*), 152
date (*in module altar.models.seismic.meta*), 171
date (*in module altar.models.sir.meta*), 173
debug (*altar.bayesian.Annealer.Annealer attribute*), 81
deduceAnnealingMethod() (*altar.bayesian.Annealer.Annealer method*), 82
default (*altar.bayesian.Profiler.Profiler attribute*), 91
default (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
default (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
default (*altar.data.DataL2 attribute*), 125
default (*altar.data.DataL2.DataL2 attribute*), 123
default (*altar.models.bayesian attribute*), 182
default (*altar.models.Bayesian.Bayesian attribute*), 174
default (*altar.models.BayesianL2.BayesianL2 attribute*), 175
default (*altar.models.cdm.CDM.CDM attribute*), 130
default (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
default (*altar.models.linear.Linear.Linear attribute*), 143
default (*altar.models.mogi.Mogi.Mogi attribute*), 147
default (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicGattribute*), 156
default (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
default (*altar.models.seismic.cuda.model attribute*), 162
default (*altar.models.seismic.shells.cudaApplication.cudaApplication attribute*), 166
default (*altar.models.seismic.shells.seismic attribute*), 167
default (*altar.models.seismic.shells.Seismic.Seismic attribute*), 165
default (*altar.models.seismic.Static.Static attribute*), 169
default (*altar.shells.cudaApplication.cudaApplication attribute*), 187
default (*altar.simulations.Job.Job attribute*), 192
default() (*altar.actions.Forward.Forward method*), 80
default() (*altar.actions.Sample.Sample method*), 80
default() (*altar.models.seismic.actions.Forward.Forward method*), 155
default() (*altar.models.seismic.actions.Sample.Sample method*), 155
density() (*altar.cuda.distributions.distribution method*), 112
density() (*altar.distributions.Base.Base method*), 127
density() (*altar.distributions.distribution method*), 129
density() (*altar.distributions.Distribution.Distribution method*), 127
device (*altar.bayesian.CUDAAnnealing.CUDAAnnealing attribute*), 85
device (*altar.cuda.distributions.cudaDistribution attribute*), 112
device (*altar.cuda.distributions.cudaDistribution.cudaDistribution attribute*), 108
device (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
device (*altar.models.BayesianL2.BayesianL2 attribute*), 176
device (*altar.models.seismic.cuda.model attribute*), 163
dIdx (*altar.models.cdm.CDM.CDM attribute*), 131
dIdx (*altar.models.mogi.Mogi.Mogi attribute*), 148
dispatcher (*altar.bayesian.Annealer.Annealer attribute*), 81
dispatcher (*altar.bayesian.controller attribute*), 95
dispatcher (*altar.bayesian.Controller.Controller attribute*), 85
dispatcher (*altar.bayesian.Metropolis.Metropolis attribute*), 89
dispatcher (*altar.cuda.bayesian.controller attribute*), 104
dispatcher (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMe attribute*), 99
dispatcher (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
dispatcher (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetrop attribute*), 103
dispatcher (*class in altar.simulations*), 197
Dispatcher (*class in altar.simulations.Dispatcher*), 191
displace() (*altar.bayesian.Metropolis.Metropolis method*), 90
displace() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMe method*), 100
displace() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
displace() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetrop method*), 104
displacements (*altar.models.cdm.CDM.CDM attribute*), 131
displacements (*altar.models.mogi.Mogi.Mogi attribute*), 147
displacements() (*altar.models.cdm.source method*), 137

displacements() (*altar.models.cdm.Source.Source method*), 134
displacements() (*altar.models.mogi.source method*), 150
displacements() (*altar.models.mogi.Source.Source method*), 150
distribution (*class in altar.cuda.distributions*), 111
distribution (*class in altar.distributions*), 129
Distribution (*class in altar.distributions.Distribution*), 127
doc (*altar.bayesian.Annealer.Annealer attribute*), 81
doc (*altar.bayesian.Brent.Brent attribute*), 83
doc (*altar.bayesian.controller attribute*), 95
doc (*altar.bayesian.Controller.Controller attribute*), 85, 86
doc (*altar.bayesian.COV.COV attribute*), 84
doc (*altar.bayesian.Grid.Grid attribute*), 87
doc (*altar.bayesian.H5Recorder.H5Recorder attribute*), 87, 88
doc (*altar.bayesian.Metropolis.Metropolis attribute*), 89
doc (*altar.bayesian.Profiler.Profiler attribute*), 91
doc (*altar.bayesian.Recorder.Recorder attribute*), 93
doc (*altar.bayesian.solver attribute*), 97
doc (*altar.bayesian.Solver.Solver attribute*), 95
doc (*altar.cuda.bayesian.controller attribute*), 105
doc (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 98, 99
doc (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
doc (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
doc (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
doc (*altar.cuda.distributions.cudaDistribution attribute*), 112
doc (*altar.cuda.distributions.cudaDistribution.cudaDistribution attribute*), 108
doc (*altar.cuda.distributions.cudaGaussian.cudaGaussian attribute*), 109
doc (*altar.cuda.distributions.cudaPreset.cudaPreset attribute*), 109
doc (*altar.cuda.distributions.cudaTGaussian.cudaTGaussian attribute*), 110
doc (*altar.cuda.distributions.cudaUniform.cudaUniform attribute*), 111
doc (*altar.cuda.distributions.distribution attribute*), 111
doc (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113, 114
doc (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
doc (*altar.cuda.models.cudaParameterEnsemble.cudaParameterEnsemble attribute*), 118
doc (*altar.cuda.models.cudaParameterSet.cudaParameterSet attribute*), 119
doc (*altar.cuda.models.parameters attribute*), 120
doc (*altar.data.DataL2 attribute*), 125
doc (*altar.data.DataL2.DataL2 attribute*), 123
doc (*altar.distributions.Base.Base attribute*), 126
doc (*altar.distributions.distribution attribute*), 129
doc (*altar.distributions.Distribution.Distribution attribute*), 127
doc (*altar.distributions.Gaussian.Gaussian attribute*), 128
doc (*altar.distributions.Uniform.Uniform attribute*), 128
doc (*altar.models.bayesian attribute*), 182
doc (*altar.models.Bayesian.Bayesian attribute*), 173, 174
doc (*altar.models.BayesianL2.BayesianL2 attribute*), 175
doc (*altar.models.cdm.CDM.CDM attribute*), 130, 131
doc (*altar.models.cdm.data attribute*), 137, 138
doc (*altar.models.cdm.Data.Data attribute*), 133
doc (*altar.models.Contiguous.Contiguous attribute*), 177
doc (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
doc (*altar.models.Ensemble.Ensemble attribute*), 178
doc (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141, 142
doc (*altar.models.linear.Linear.Linear attribute*), 143, 144
doc (*altar.models.mogi.data attribute*), 151
doc (*altar.models.mogi.Data.Data attribute*), 146
doc (*altar.models.mogi.Mogi.Mogi attribute*), 147, 148
doc (*altar.models.Null.Null attribute*), 179
doc (*altar.models.parameters attribute*), 181
doc (*altar.models.ParameterSet.ParameterSet attribute*), 180
doc (*altar.models.regression.Linear.Linear attribute*), 151
doc (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
doc (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 157, 158
doc (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
doc (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
doc (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
doc (*altar.models.seismic.cuda.model attribute*), 162, 163
doc (*altar.models.seismic.Moment.Moment attribute*), 168
doc (*altar.models.seismic.shells.cudaApplication.cudaApplication attribute*), 166
doc (*altar.models.seismic.shells.seismic attribute*), 167
doc (*altar.models.seismic.shells.Seismic.Seismic attribute*), 165
doc (*altar.models.seismic.Static.Static attribute*), 168, 169
doc (*altar.models.seismic.StaticCp.StaticCp attribute*), 170, 171
doc (*altar.models.sir.SIR.SIR attribute*), 172
doc (*altar.shells.altar attribute*), 189
doc (*altar.shells.AlTar.AlTar attribute*), 185
doc (*altar.shells.application attribute*), 188
doc (*altar.shells.Application.Application attribute*), 186

doc (*altar.shells.cudaaltar attribute*), 190
doc (*altar.shells.cudaAlTar.cudaAlTar attribute*), 186, 187
doc (*altar.shells.cudaapplication attribute*), 189
doc (*altar.shells.cudaApplication.cudaApplication attribute*), 187, 188
doc (*altar.simulations.GSLRNG.GSLRNG attribute*), 191
doc (*altar.simulations.Job.Job attribute*), 192, 193
doc (*altar.simulations.Recorder.Recorder attribute*), 194
doc (*altar.simulations.run attribute*), 197, 198
doc (*altar.simulations.Run.Run attribute*), 196
dpc (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
dpc (*altar.models.seismic.cuda.model attribute*), 163
dsp (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
dt (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
dtype_cd (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
dtype_cp (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
dV (*altar.models.mogi.source attribute*), 150
dV (*altar.models.mogi.Source.Source attribute*), 149

E

embedded (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
embedded (*altar.models.BayesianL2.BayesianL2 attribute*), 175
embedded (*altar.models.seismic.cuda.model attribute*), 162
EMHP (*class in altar.models.emhp.EMHP*), 140
emhp() (*in module altar.models.emhp*), 141
Ensemble (*class in altar.models.Ensemble*), 178
ensemble() (*in module altar.models*), 183
error (*altar.bayesian.Annealer.Annealer attribute*), 81
error (*altar.cuda.distributions.cudaPreset.cudaPreset attribute*), 110
error (*altar.data.DataL2 attribute*), 125
error (*altar.data.DataL2.DataL2 attribute*), 123
error (*altar.models.bayesian attribute*), 182
error (*altar.models.Bayesian.Bayesian attribute*), 174
etc() (*altar.actions.About.About method*), 80
etc() (*altar.models.seismic.actions.About.About method*), 154
eval() (*altar.cuda.norms.norm method*), 122
eval() (*altar.norms.L2.L2 method*), 183
eval() (*altar.norms.norm method*), 184
eval() (*altar.norms.Norm.Norm method*), 184
evalDataLikelihood() (*altar.models.BayesianL2.BayesianL2 method*), 176
evalLikelihood() (*altar.data.DataL2 method*), 125
evalLikelihood() (*altar.data.DataL2.DataL2 method*), 124
evalPosterior() (*altar.models.BayesianL2.BayesianL2 method*), 176
evalPrior() (*altar.models.BayesianL2.BayesianL2 method*), 176

F

Fast (*class in altar.models.cdm.Fast*), 133
Fast (*class in altar.models.mogi.Fast*), 147
finish (*altar.bayesian.Notifier.Notifier attribute*), 91
finish() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 83
finish() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 89
finishSamplingPDF() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis method*), 100
finishSamplingPDF() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
finishSamplingPDF() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary method*), 104
firewall (*altar.bayesian.Annealer.Annealer attribute*), 81
firewall (*altar.models.bayesian attribute*), 182
firewall (*altar.models.Bayesian.Bayesian attribute*), 174
Forward (*class in altar.actions.Forward*), 80
Forward (*class in altar.models.seismic.actions.Forward*), 154
forward() (*altar.models.seismic.shells.seismic method*), 167
forward() (*altar.models.seismic.shells.Seismic.Seismic method*), 166
forward() (*in module altar.actions*), 81
forward() (*in module altar.models.seismic.actions*), 155
forward_output (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
forward_output (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
forward_output (*altar.models.seismic.cuda.model attribute*), 163
forwardModel() (*altar.models.BayesianL2.BayesianL2 method*), 176
forwardModel() (*altar.models.regression.Linear.Linear method*), 152

forwardModel() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
forwardModel() (*altar.models.seismic.cuda.cudaStatic.cudaStatic method*), 160
forwardModel() (*altar.models.seismic.Static.Static method*), 170
forwardModel() (*altar.models.sir.SIR.SIR method*), 172
forwardModelBatched() (*altar.models.BayesianL2.BayesianL2 method*), 176
forwardModelBatched() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
forwardModelBatched() (*altar.models.seismic.cuda.cudaStatic.cudaStatic method*), 160
forwardonly (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
forwardonly (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
forwardonly (*altar.models.seismic.cuda.model attribute*), 162
forwardProblem() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
forwardProblem() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 118
forwardProblem() (*altar.cuda.models.model method*), 120
forwardProblem() (*altar.models.bayesian method*), 183
forwardProblem() (*altar.models.Bayesian.Bayesian method*), 174
forwardProblem() (*altar.models.model method*), 181
forwardProblem() (*altar.models.Model.Model method*), 179
forwardProblem() (*altar.models.Null.Null method*), 179
forwardProblem() (*altar.models.regression.model method*), 153
forwardProblem() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
forwardProblem() (*altar.models.seismic.cuda.cudaStatic.cudaStatic method*), 160
forwardProblem() (*altar.models.seismic.cuda.model method*), 164

G

G (*altar.models.linear.Linear.Linear attribute*), 144
G (*altar.models.seismic.Static.Static attribute*), 169
G0 (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
gacceptance_flags (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gacceptance_flags (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
gacceptance_flags (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary attribute*), 103
gain (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 98
gainFunction() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis static method*), 100
Gaussian (*class in altar.distributions.Gaussian*), 128
Gaussian (*class in altar.models.gaussian.Gaussian*), 141
gaussian() (*in module altar.cuda.distributions*), 112
gaussian() (*in module altar.distributions*), 130
gaussian() (*in module altar.models.gaussian*), 143
gcandidate (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMe attribute*), 99
gcandidate (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
gcandidate (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetrop attribute*), 103
gCd_inv (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
gCmu (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
gdataObsBatch (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
gDataPred (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
gDataPred (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
gdice (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gdice (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
gdice (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary attribute*), 104
gDprediction (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematic attribute*), 156
get_current_device() (*in module altar.cuda*), 122
GF (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
GF (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
GF (*altar.models.seismic.cuda.cudaStatic.cudaStatic*

attribute), 160
gGF (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
gGF (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
gGF (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
gidx_map (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
gidx_map (*altar.models.seismic.cuda.model attribute*), 163
ginit (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
ginit (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
ginit (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
gInitModel (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
ginvalid_flags (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
ginvalid_flags (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
ginvalid_flags (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
gMeanModel (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
gproposal (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gproposal (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
gproposal (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
gpuids (*altar.simulations.Job.Job attribute*), 192
gpuprecision (*altar.simulations.Job.Job attribute*), 192
gpus (*altar.simulations.Job.Job attribute*), 192
gpus (*altar.simulations.run attribute*), 198
gpus (*altar.simulations.Run.Run attribute*), 196
green (*altar.models.cudalinear.cudaLinear.cudaLinear attribute*), 138
green (*altar.models.linear.Linear.Linear attribute*), 143
green (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
green (*altar.models.seismic.cuda.cudaStatic.cudaStatic attribute*), 160
green_file (*altar.models.seismic.Static.Static attribute*), 169
Grid (*class in altar.bayesian.Grid*), 87
grid() (*in module altar.bayesian*), 97
gsigma_chol (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gsigma_chol (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
gsigma_chol (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
gsl() (*in module altar.simulations*), 198
GSLRNG (*class in altar.simulations.GSLRNG*), 191
gstep (*altar.bayesian.CUDAAnnealing.CUDAAnnealing attribute*), 85
gstep (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gstep (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
gstep (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
gt0s (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 157
gtheta (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
gtheta (*altar.models.seismic.cuda.model attribute*), 163
gvalid_indices (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gvalid_indices (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
gvalid_indices (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
gvalid_samples (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
gvalid_samples (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
gvalid_samples (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103

H

H5Recorder (*class in altar.bayesian.H5Recorder*), 87
header (*in module altar.meta*), 199
header (*in module altar.models.cdm.meta*), 137
header (*in module altar.models.cudalinear.meta*), 139
header (*in module altar.models.emhp.meta*), 140
header (*in module altar.models.gaussian.meta*), 142
header (*in module altar.models.linear.meta*), 145
header (*in module altar.models.mogi.meta*), 150
header (*in module altar.models.regression.meta*), 152
header (*in module altar.models.seismic.meta*), 172
header (*in module altar.models.sir.meta*), 173
home() (*altar.actions.About.About method*), 79
home() (*altar.models.seismic.actions.About.About method*), 154
hosts (*altar.simulations.Job.Job attribute*), 192

hosts (*altar.simulations.run attribute*), 197
hosts (*altar.simulations.Run.Run attribute*), 196

I

idx_map (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
idx_map (*altar.models.BayesianL2.BayesianL2 attribute*), 175
idx_map (*altar.models.seismic.cuda.model attribute*), 162
idx_range (*altar.cuda.distributions.cudaDistribution attribute*), 112
idx_range (*altar.cuda.distributions.cudaDistribution.cudaDistribution attribute*), 108
ifs (*altar.cuda.distributions.cudaPreset.cudaPreset attribute*), 110
ifs (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
ifs (*altar.data.DataL2 attribute*), 125
ifs (*altar.data.DataL2.DataL2 attribute*), 123
ifs (*altar.models.BayesianL2.BayesianL2 attribute*), 176
ifs (*altar.models.cdm.CDM.CDM attribute*), 131
ifs (*altar.models.linear.Linear.Linear attribute*), 144
ifs (*altar.models.mogi.Mogi.Mogi attribute*), 148
ifs (*altar.models.seismic.cuda.model attribute*), 163
ifs (*altar.models.seismic.Static.Static attribute*), 169
info (*altar.bayesian.Annealer.Annealer attribute*), 81
info (*altar.models.bayesian attribute*), 182
info (*altar.models.Bayesian.Bayesian attribute*), 174
initial_model_file (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 158
initial_model_file (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
initialize() (*altar.bayesian.Annealer.Annealer method*), 81
initialize() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
initialize() (*altar.bayesian.archiver method*), 97
initialize() (*altar.bayesian.Brent.Brent method*), 83
initialize() (*altar.bayesian.controller method*), 96
initialize() (*altar.bayesian.Controller.Controller method*), 86
initialize() (*altar.bayesian.COV.COV method*), 84
initialize() (*altar.bayesian.CUDAAnnealing.CUDAAnnealing method*), 85
initialize() (*altar.bayesian.Grid.Grid method*), 87
initialize() (*altar.bayesian.H5Recorder.H5Recorder method*), 88
initialize() (*altar.bayesian.Metropolis.Metropolis method*), 89
initialize() (*altar.bayesian.monitor method*), 96
initialize() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
initialize() (*altar.bayesian.Notifier.Notifier method*), 91
initialize() (*altar.bayesian.Profiler.Profiler method*), 91
initialize() (*altar.bayesian.Recorder.Recorder method*), 93
initialize() (*altar.bayesian.sampler method*), 96
initialize() (*altar.bayesian.Sampler.Sampler method*), 93
initialize() (*altar.bayesian.scheduler method*), 96
initialize() (*altar.bayesian.Scheduler.Scheduler method*), 94
initialize() (*altar.bayesian.solver method*), 97
initialize() (*altar.bayesian.Solver.Solver method*), 95
initialize() (*altar.cuda.bayesian.controller method*), 105
initialize() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptive method*), 99
initialize() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
initialize() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMet method*), 104
initialize() (*altar.cuda.bayesian.sampler method*), 105
initialize() (*altar.cuda.bayesian.scheduler method*), 105
initialize() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 106
initialize() (*altar.cuda.data.data method*), 108
initialize() (*altar.cuda.distributions.cudaDistribution method*), 112
initialize() (*altar.cuda.distributions.cudaDistribution.cudaDistribution method*), 108
initialize() (*altar.cuda.distributions.cudaPreset.cudaPreset method*), 110
initialize() (*altar.cuda.distributions.distribution method*), 111
initialize() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
initialize() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianE method*), 116
initialize() (*altar.cuda.models.cudaParameterEnsemble.cudaParamete method*), 118
initialize() (*altar.cuda.models.model method*), 119
initialize() (*altar.cuda.models.parameters method*), 120
initialize() (*altar.data.data method*), 124
initialize() (*altar.data.DataL2 method*), 125
initialize() (*altar.data.DataL2.DataL2 method*), 123
initialize() (*altar.data.DataObs.DataObs method*), 124
initialize() (*altar.distributions.Base.Base method*),

126
initialize() (altar.distributions.distribution method), 129
initialize() (altar.distributions.Distribution.Distribution method), 127
initialize() (altar.distributions.Gaussian.Gaussian method), 128
initialize() (altar.distributions.Uniform.Uniform method), 128
initialize() (altar.distributions.UnitGaussian.UnitGaussian method), 129
initialize() (altar.models.bayesian method), 182
initialize() (altar.models.Bayesian.Bayesian method), 174
initialize() (altar.models.BayesianL2.BayesianL2 method), 176
initialize() (altar.models.cdm.CDM.CDM method), 132
initialize() (altar.models.cdm.CUDA.CUDA method), 132
initialize() (altar.models.cdm.Fast.Fast method), 133
initialize() (altar.models.cdm.Native.Native method), 134
initialize() (altar.models.Contiguous.Contiguous method), 177
initialize() (altar.models.cudalinear.cudaLinear.cudaLinear method), 138
initialize() (altar.models.emhp.EMHP.EMHP method), 140
initialize() (altar.models.Ensemble.Ensemble method), 178
initialize() (altar.models.gaussian.Gaussian.Gaussian method), 142
initialize() (altar.models.linear.Linear.Linear method), 144
initialize() (altar.models.model method), 181
initialize() (altar.models.Model.Model method), 178
initialize() (altar.models.mogi.CUDA.CUDA method), 146
initialize() (altar.models.mogi.Fast.Fast method), 147
initialize() (altar.models.mogi.Mogi.Mogi method), 148
initialize() (altar.models.mogi.Native.Native method), 149
initialize() (altar.models.parameters method), 181
initialize() (altar.models.ParameterSet.ParameterSet method), 180
initialize() (altar.models.regression.Linear.Linear method), 152
initialize() (altar.models.regression.model method), 152
initialize() (altar.models.seismic.cuda.cudaKinematicG.cudaKinematic method), 157
initialize() (altar.models.seismic.cuda.cudaKinematicGCp.cudaKinema method), 158
initialize() (altar.models.seismic.cuda.cudaMoment.cudaMoment method), 159
initialize() (altar.models.seismic.cuda.cudaStatic.cudaStatic method), 160
initialize() (altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp method), 161
initialize() (altar.models.seismic.cuda.model method), 163
initialize() (altar.models.seismic.Moment.Moment method), 168
initialize() (altar.models.seismic.shells.seismic method), 167
initialize() (altar.models.seismic.shells.Seismic.Seismic method), 165
initialize() (altar.models.seismic.Static.Static method), 169
initialize() (altar.models.seismic.StaticCp.StaticCp method), 171
initialize() (altar.shells.altar method), 189
initialize() (altar.shells.AlTar.AlTar method), 185
initialize() (altar.shells.cudaaltar method), 190
initialize() (altar.shells.cudaAlTar.cudaAlTar method), 187
initialize() (altar.simulations.archiver method), 197
initialize() (altar.simulations.Archiver.Archiver method), 191
initialize() (altar.simulations.dispatcher method), 197
initialize() (altar.simulations.Dispatcher.Dispatcher method), 191
initialize() (altar.simulations.GSLRNG.GSLRNG method), 191
initialize() (altar.simulations.Job.Job method), 193
initialize() (altar.simulations.monitor method), 197
initialize() (altar.simulations.Monitor.Monitor method), 193
initialize() (altar.simulations.Recorder.Recorder method), 194
initialize() (altar.simulations.Reporter.Reporter method), 194
initialize() (altar.simulations.rng method), 198
initialize() (altar.simulations.RNG.RNG method), 193
initialize() (altar.simulations.run method), 198
initialize() (altar.simulations.Run.Run method), 196
initializeCovariance() (altar.cuda.data.cudaDataL2.cudaDataL2 method), 107
initializeCovariance() (altar.data.DataL2

method), 126
initializeCovariance() (altar.data.DataL2.DataL2 method), 124
initializeCovariance() (altar.models.seismic.Static.Static method), 170
initializeCovariance() (altar.models.seismic.StaticCp.StaticCp method), 171
initializeCp() (altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp method), 158
initializeCp() (altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp method), 161
initializeParameterSets() (altar.models.cdm.CDM.CDM method), 132
initializeParameterSets() (altar.models.mogi.Mogi.Mogi method), 148
initializeParameterSets() (altar.models.seismic.Static.Static method), 169
initializeResiduals() (altar.models.linear.Linear.Linear method), 144
initializeResiduals() (altar.models.seismic.Static.Static method), 170
initializeSample() (altar.cuda.distributions.distribution method), 111
initializeSample() (altar.cuda.models.model method), 120
initializeSample() (altar.cuda.models.parameters method), 120
initializeSample() (altar.distributions.Base.Base method), 126
initializeSample() (altar.distributions.distribution method), 129
initializeSample() (altar.distributions.Distribution.Distribution method), 127
initializeSample() (altar.models.bayesian method), 182
initializeSample() (altar.models.Bayesian.Bayesian method), 174
initializeSample() (altar.models.BayesianL2.BayesianL2 method), 176
initializeSample() (altar.models.cdm.CDM.CDM method), 132
initializeSample() (altar.models.Contiguous.Contiguous method), 177
initializeSample() (altar.models.emhp.EMHP.EMHP method), 140
initializeSample() (altar.models.Ensemble.Ensemble method), 178
initializeSample() (altar.models.gaussian.Gaussian.Gaussian method), 142
initializeSample() (altar.models.linear.Linear.Linear method), 144
initializeSample() (altar.models.model method), 181
initializeSample() (altar.models.Model.Model method), 178
initializeSample() (altar.models.mogi.Mogi.Mogi method), 148
initializeSample() (altar.models.Null.Null method), 179
initializeSample() (altar.models.parameters method), 181
initializeSample() (altar.models.ParameterSet.ParameterSet method), 180
initializeSample() (altar.models.regression.model method), 153
initializeSample() (altar.models.seismic.Moment.Moment method), 168
initializeSample() (altar.models.seismic.Static.Static method), 169
initialModel_file (altar.models.seismic.StaticCp.StaticCp attribute), 171
input_file (altar.cuda.distributions.cudaPreset.cudaPreset attribute), 109
iteration (altar.bayesian.AnnealingMethod.AnnealingMethod attribute), 82

J

job (altar.models.bayesian attribute), 182
job (altar.models.Bayesian.Bayesian attribute), 174
job (altar.models.seismic.shells.cudaApplication.cudaApplication attribute), 166
job (altar.models.seismic.shells.seismic attribute), 167
job (altar.models.seismic.shells.Seismic.Seismic attribute), 165
job (altar.shells.altar attribute), 188
job (altar.shells.AlTar.AlTar attribute), 185
job (altar.shells.application attribute), 188
job (altar.shells.Application.Application attribute), 186
job (altar.shells.cudaaltar attribute), 190
job (altar.shells.cudaAlTar.cudaAlTar attribute), 186
job (altar.shells.cudaapplication attribute), 189
job (altar.shells.cudaApplication.cudaApplication attribute), 187
Job (class in altar.simulations.Job), 192
job() (in module altar.simulations), 198

K

kinematicg() (*in module altar.models.seismic.cuda*), 164
Kmu (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
kmu_file (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 158
kmu_file (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 161
kmu_file (*altar.models.seismic.StaticCp.StaticCp attribute*), 171

L

L2 (*class in altar.norms.L2*), 183
l2() (*in module altar.cuda.norms*), 122
l2() (*in module altar.norms*), 184
license (*in module altar.meta*), 199
license (*in module altar.models.cdm.meta*), 137
license (*in module altar.models.cudalinear.meta*), 139
license (*in module altar.models.emhp.meta*), 140
license (*in module altar.models.gaussian.meta*), 142
license (*in module altar.models.linear.meta*), 145
license (*in module altar.models.mogi.meta*), 150
license (*in module altar.models.regression.meta*), 152
license (*in module altar.models.seismic.meta*), 172
license (*in module altar.models.sir.meta*), 173
license() (*altar.actions.About.About method*), 80
license() (*altar.models.seismic.actions.About.About method*), 154
license() (*in module altar*), 199
license() (*in module altar.cuda*), 122
likelihoods() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
likelihoods() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
likelihoods() (*altar.cuda.models.model method*), 120
likelihoods() (*altar.models.bayesian method*), 183
likelihoods() (*altar.models.Bayesian.Bayesian method*), 174
likelihoods() (*altar.models.BayesianL2.BayesianL2 method*), 176
likelihoods() (*altar.models.model method*), 181
likelihoods() (*altar.models.Model.Model method*), 179
likelihoods() (*altar.models.regression.model method*), 153
likelihoods() (*altar.models.seismic.cuda.model method*), 163
Linear (*class in altar.models.linear.Linear*), 143
Linear (*class in altar.models.regression.Linear*), 151
linear() (*in module altar.models.linear*), 145
linear() (*in module altar.models.regression*), 153
linearGM() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
llk (*altar.simulations.Recorder.Recorder attribute*), 194
load_hdf5() (*altar.bayesian.CoolingStep.CoolingStep method*), 87
loadData() (*altar.data.DataL2 method*), 125
loadData() (*altar.data.DataL2.DataL2 method*), 124
loadFile() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 107
loadFile() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
loadFile() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnse method*), 117
loadFile() (*altar.models.BayesianL2.BayesianL2 method*), 176
loadFile() (*altar.models.seismic.cuda.model method*), 164
loadFileToGPU() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
loadFileToGPU() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
loadFileToGPU() (*altar.models.seismic.cuda.model method*), 164
loadGF() (*altar.models.cudalinear.cudaLinear.cudaLinear method*), 139
loadInputs() (*altar.models.cdm.CDM.CDM method*), 132
loadInputs() (*altar.models.linear.Linear.Linear method*), 144
loadInputs() (*altar.models.mogi.Mogi.Mogi method*), 149
loadInputs() (*altar.models.seismic.Static.Static method*), 170
loadInputsCp() (*altar.models.seismic.StaticCp.StaticCp method*), 171
los (*altar.models.cdm.CDM.CDM attribute*), 131
los (*altar.models.mogi.Mogi.Mogi attribute*), 148

M

main() (*altar.models.seismic.shells.cudaApplication.cudaApplication method*), 166
main() (*altar.models.seismic.shells.seismic method*), 167
main() (*altar.models.seismic.shells.Seismic.Seismic method*), 165
main() (*altar.shells.altar method*), 189
main() (*altar.shells.AlTar.AlTar method*), 185
main() (*altar.shells.application method*), 188
main() (*altar.shells.Application.Application method*), 186
main() (*altar.shells.cudaaltar method*), 190
main() (*altar.shells.cudaAlTar.cudaAlTar method*), 187
main() (*altar.shells.cudaapplication method*), 189

main() (*altar.shells.cudaApplication.cudaApplication method*), 188
major (*in module altar.meta*), 198
major (*in module altar.models.cdm.meta*), 136
major (*in module altar.models.cudalinear.meta*), 139
major (*in module altar.models.emhp.meta*), 140
major (*in module altar.models.gaussian.meta*), 142
major (*in module altar.models.linear.meta*), 145
major (*in module altar.models.mogi.meta*), 150
major (*in module altar.models.regression.meta*), 152
major (*in module altar.models.seismic.meta*), 171
major (*in module altar.models.sir.meta*), 173
manager (*altar.bayesian.MPIAnnealing.MPIAnnealing attribute*), 88
matrix() (*altar.cuda.distributions.distribution method*), 112
matrix() (*altar.distributions.Base.Base method*), 127
matrix() (*altar.distributions.distribution method*), 130
matrix() (*altar.distributions.Distribution.Distribution method*), 127
max_mc_steps (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 98
max_mc_steps (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
max_mc_steps_stage2 (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
maxiter (*altar.bayesian.Brent.Brent attribute*), 83
maxiter (*altar.bayesian.Grid.Grid attribute*), 87
mcsteps (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
mcsteps (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
mcsteps (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute*), 103
mean (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
mean (*altar.cuda.distributions.cudaGaussian.cudaGaussian attribute*), 109
mean (*altar.cuda.distributions.cudaTGaussian.cudaTGaussian attribute*), 110
mean (*altar.distributions.Gaussian.Gaussian attribute*), 128
mean_model (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 158
meanModel (*altar.models.seismic.StaticCp.StaticCp attribute*), 171
merge_cd_to_data (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
merge_cd_with_data (*altar.data.DataL2 attribute*), 125
merge_cd_with_data (*altar.data.DataL2.DataL2 attribute*), 123
mergeCdtoData() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 107
mergeCovarianceToGF() (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG method*), 157
mergeCovarianceToGF() (*altar.models.seismic.cuda.cudaStatic.cudaStatic method*), 160
meta() (*altar.models.cdm.CDM.CDM method*), 132
meta() (*altar.models.mogi.Mogi.Mogi method*), 149
Metropolis (*class in altar.bayesian.Metropolis*), 89
metropolis() (*in module altar.bayesian*), 97
metropolis() (*in module altar.cuda.bayesian*), 105
metropolisvaryingsteps() (*in module altar.cuda.bayesian*), 105
micro (*in module altar.meta*), 199
micro (*in module altar.models.cdm.meta*), 136
micro (*in module altar.models.cudalinear.meta*), 139
micro (*in module altar.models.emhp.meta*), 140
micro (*in module altar.models.gaussian.meta*), 142
micro (*in module altar.models.linear.meta*), 145
micro (*in module altar.models.mogi.meta*), 150
micro (*in module altar.models.regression.meta*), 152
micro (*in module altar.models.seismic.meta*), 171
micro (*in module altar.models.sir.meta*), 173
min_chain_value_ratio (*altar.bayesian.COV.COV attribute*), 84
min_mc_steps (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 98
minor (*in module altar.meta*), 198
minor (*in module altar.models.cdm.meta*), 136
minor (*in module altar.models.cudalinear.meta*), 139
minor (*in module altar.models.emhp.meta*), 140
minor (*in module altar.models.gaussian.meta*), 142
minor (*in module altar.models.linear.meta*), 145
minor (*in module altar.models.mogi.meta*), 150
minor (*in module altar.models.regression.meta*), 152
minor (*in module altar.models.seismic.meta*), 171
minor (*in module altar.models.sir.meta*), 173
mode (*altar.models.cdm.CDM.CDM attribute*), 131
mode (*altar.models.mogi.Mogi.Mogi attribute*), 148
mode (*altar.simulations.Job.Job attribute*), 192
mode (*altar.simulations.Simulations.run attribute*), 197
mode (*altar.simulations.Run.Run attribute*), 196
model (*altar.bayesian.Annealer.Annealer attribute*), 81
model (*altar.models.seismic.shells.cudaApplication.cudaApplication attribute*), 166
model (*altar.models.seismic.shells.seismic attribute*), 167
model (*altar.models.seismic.shells.Seismic.Seismic attribute*), 165
model (*altar.shells.altar attribute*), 189
model (*altar.shells.AlTar.AlTar attribute*), 185
model (*altar.shells.application attribute*), 188
model (*altar.shells.Application.Application attribute*), 186

model (*altar.shells.cudaaltar attribute*), 190
model (*altar.shells.cudaAlTar.cudaAlTar attribute*), 186
model (*altar.shells.cudaapplication attribute*), 189
model (*altar.shells.cudaApplication.cudaApplication attribute*), 187
model (*class in altar.cuda.models*), 119
model (*class in altar.models*), 180
Model (*class in altar.models.Model*), 178
model (*class in altar.models.regression*), 152
model (*class in altar.models.seismic.cuda*), 162
modelPrefix (*in module altar*), 199
models (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
models (*altar.models.Ensemble.Ensemble attribute*), 178
models() (*altar.actions.About.About method*), 79
models() (*altar.models.seismic.actions.About.About method*), 154
module
altar, 79
altar.actions, 79
altar.actions.About, 79
altar.actions.Forward, 80
altar.actions.Sample, 80
altar.bayesian, 81
altar.bayesian.Annealer, 81
altar.bayesian.AnnealingMethod, 82
altar.bayesian.Brent, 83
altar.bayesian.Controller, 85
altar.bayesian.CoolingStep, 86
altar.bayesian.COV, 83
altar.bayesian.CUDAAnnealing, 85
altar.bayesian.Grid, 87
altar.bayesian.H5Recorder, 87
altar.bayesian.Metropolis, 89
altar.bayesian.MPIAnnealing, 88
altar.bayesian.Notifier, 90
altar.bayesian.Profiler, 91
altar.bayesian.Recorder, 93
altar.bayesian.Sampler, 93
altar.bayesian.Scheduler, 94
altar.bayesian.SequentialAnnealing, 94
altar.bayesian.Solver, 95
altar.bayesian.ThreadedAnnealing, 95
altar.cuda, 98
altar.cuda.bayesian, 98
altar.cuda.bayesian.cudaAdaptiveMetropolis, 98
altar.cuda.bayesian.cudaCoolingStep, 100
altar.cuda.bayesian.cudaMetropolis, 101
altar.cuda.bayesian.cudaMetropolisVaryingSteps, 103
altar.cuda.data, 106
altar.cuda.data.cudaDataL2, 106
altar.cuda.distributions, 108
altar.cuda.distributions.cudaDistribution, 108
altar.cuda.distributions.cudaGaussian, 109
altar.cuda.distributions.cudaPreset, 109
altar.cuda.distributions.cudaTGaussian, 110
altar.cuda.distributions.cudaUniform, 111
altar.cuda.ext, 113
altar.cuda.models, 113
altar.cuda.models.cudaBayesian, 113
altar.cuda.models.cudaBayesianEnsemble, 116
altar.cuda.models.cudaParameterEnsemble, 118
altar.cuda.models.cudaParameterSet, 119
altar.cuda.norms, 121
altar.cuda.norms.cudaL2, 121
altar.data, 123
altar.data.DataL2, 123
altar.data.DataObs, 124
altar.distributions, 126
altar.distributions.Base, 126
altar.distributions.Distribution, 127
altar.distributions.Gaussian, 128
altar.distributions.Uniform, 128
altar.distributions.UnitGaussian, 129
altar.ext, 130
altar.meta, 198
altar.models, 130
altar.models.Bayesian, 173
altar.models.BayesianL2, 175
altar.models.cdm, 130
altar.models.cdm.CDM, 130
altar.models.cdm.CUDA, 132
altar.models.cdm.Data, 133
altar.models.cdm.ext, 130
altar.models.cdm.Fast, 133
altar.models.cdm.libcdm, 135
altar.models.cdm.meta, 136
altar.models.cdm.Native, 134
altar.models.cdm.Source, 134
altar.models.Contiguous, 177
altar.models.cudalinear, 138
altar.models.cudalinear.cudaLinear, 138

altar.models.cudalinear.meta, 139
altar.models.emhp, 140
altar.models.emhp.EMHP, 140
altar.models.emhp.meta, 140
altar.models.Ensemble, 178
altar.models.gaussian, 141
altar.models.gaussian.ext, 141
altar.models.gaussian.Gaussian, 141
altar.models.gaussian.meta, 142
altar.models.linear, 143
altar.models.linear.Linear, 143
altar.models.linear.meta, 145
altar.models.Model, 178
altar.models.mogi, 145
altar.models.mogi.CUDA, 145
altar.models.mogi.Data, 146
altar.models.mogi.ext, 145
altar.models.mogi.Fast, 147
altar.models.mogi.meta, 150
altar.models.mogi.Mogi, 147
altar.models.mogi.Native, 149
altar.models.mogi.Source, 149
altar.models.Null, 179
altar.models.ParameterSet, 180
altar.models.regression, 151
altar.models.regression.Linear, 151
altar.models.regression.meta, 152
altar.models.seismic, 153
altar.models.seismic.actions, 153
altar.models.seismic.actions.About, 153
altar.models.seismic.actions.Forward, 154
altar.models.seismic.actions.Sample, 155
altar.models.seismic.cuda, 155
altar.models.seismic.cuda.cudaCascade, 155
altar.models.seismic.cuda.cudaKinematicG, 156
altar.models.seismic.cuda.cudaKinematicGCp, 157
altar.models.seismic.cuda.cudaMoment, 158
altar.models.seismic.cuda.cudaStatic, 159
altar.models.seismic.cuda.cudaStaticCp, 160
altar.models.seismic.ext, 165
altar.models.seismic.meta, 171
altar.models.seismic.Moment, 167
altar.models.seismic.shells, 165
altar.models.seismic.shells.Action, 165
altar.models.seismic.shells.cudaApplication, 166
altar.models.seismic.shells.Seismic, 165
altar.models.seismic.Static, 168
altar.models.seismic.StaticCp, 170
altar.models.sir, 172
altar.models.sir.meta, 173
altar.models.sir.SIR, 172
altar.norms, 183
altar.norms.L2, 183
altar.norms.Norm, 184
altar.shells, 184
altar.shells.Action, 184
altar.shells.AlTar, 185
altar.shells.Application, 186
altar.shells.cudaAlTar, 186
altar.shells.cudaApplication, 187
altar.simulations, 190
altar.simulations.Archiver, 190
altar.simulations.Dispatcher, 191
altar.simulations.GSLRNG, 191
altar.simulations.Job, 192
altar.simulations.Monitor, 193
altar.simulations.Recorder, 194
altar.simulations.Reporter, 194
altar.simulations.RNG, 193
altar.simulations.Run, 196
Mogi (*class in altar.models.mogi.Mogi*), 147
mogi() (*in module altar.models.mogi*), 151
Moment (*class in altar.models.seismic.Moment*), 167
moment() (*in module altar.models.seismic*), 172
moment() (*in module altar.models.seismic.cuda*), 164
monitor (*class in altar.bayesian*), 96
monitor (*class in altar.simulations*), 197
Monitor (*class in altar.simulations.Monitor*), 193
monitors (*altar.models.seismic.shells.cudaApplication.cudaApplication attribute*), 166
monitors (*altar.models.seismic.shells.seismic attribute*), 167
monitors (*altar.models.seismic.shells.Seismic.Seismic attribute*), 165
monitors (*altar.shells.altar attribute*), 189
monitors (*altar.shells.AlTar.AlTar attribute*), 185
monitors (*altar.shells.application attribute*), 188
monitors (*altar.shells.Application.Application attribute*), 186
monitors (*altar.shells.cudaaltar attribute*), 190
monitors (*altar.shells.cudaAlTar.cudaAlTar attribute*), 187
monitors (*altar.shells.cudaapplication attribute*), 189
monitors (*altar.shells.cudaApplication.cudaApplication attribute*), 188

mountInputDataspace() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
mountInputDataspace() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
mountInputDataspace() (*altar.models.bayesian method*), 183
mountInputDataspace() (*altar.models.Bayesian.Bayesian method*), 174
mountInputDataspace() (*altar.models.BayesianL2.BayesianL2 method*), 176
mountInputDataspace() (*altar.models.cdm.CDM.CDM method*), 132
mountInputDataspace() (*altar.models.linear.Linear.Linear method*), 144
mountInputDataspace() (*altar.models.mogi.Mogi.Mogi method*), 149
mountInputDataspace() (*altar.models.seismic.cuda.model method*), 163
mountInputDataspace() (*altar.models.seismic.Static.Static method*), 170
mpi() (*altar.bayesian.Annealer.Annealer method*), 82
MPIAnnealing (*class in altar.bayesian.MPIAnnealing*), 88
Mu (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
Mu (*altar.models.seismic.Moment.Moment attribute*), 168
Mu_patch_file (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
mu_patches (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
Mw_mean (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
Mw_mean (*altar.models.seismic.Moment.Moment attribute*), 168
Mw_sigma (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
Mw_sigma (*altar.models.seismic.Moment.Moment attribute*), 168

N

name (*altar.simulations.Job.Job attribute*), 192
name (*altar.simulations.run attribute*), 197
name (*altar.simulations.Run.Run attribute*), 196
name() (*altar.actions.About.About method*), 79
name() (*altar.models.seismic.actions.About.About method*), 154
Nas (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
Native (*class in altar.models.cdm.Native*), 134
Native (*class in altar.models.mogi.Native*), 149
nCmu (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp attribute*), 157
nCmu (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp attribute*), 160
nCmu (*altar.models.seismic.StaticCp.StaticCp attribute*), 170
Ndd (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
nfs() (*altar.actions.About.About method*), 80
nfs() (*altar.models.seismic.actions.About.About method*), 154
NGbparameters (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 157
Nmesh (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
norm (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
norm (*altar.data.DataL2 attribute*), 125
norm (*altar.data.DataL2.DataL2 attribute*), 123
norm (*altar.models.cdm.CDM.CDM attribute*), 130
norm (*altar.models.linear.Linear.Linear attribute*), 143
norm (*altar.models.mogi.Mogi.Mogi attribute*), 147
norm (*altar.models.seismic.Static.Static attribute*), 169
norm (*class in altar.cuda.norms*), 122
norm (*class in altar.norms*), 184
Norm (*class in altar.norms.Norm*), 184
norm() (*in module altar.models.cdm.libcdm*), 135
normalization (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
normalization (*altar.data.DataL2 attribute*), 125
normalization (*altar.data.DataL2.DataL2 attribute*), 123
normalization (*altar.models.cdm.CDM.CDM attribute*), 131
normalization (*altar.models.gaussian.Gaussian.Gaussian attribute*), 142
normalization (*altar.models.linear.Linear.Linear attribute*), 144
normalization (*altar.models.mogi.Mogi.Mogi attribute*), 148
normalization (*altar.models.seismic.Static.Static attribute*), 169
Notifier (*class in altar.bayesian.Notifier*), 90
notify() (*altar.bayesian.Notifier.Notifier method*), 91
Npt (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
Nt (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
nu (*altar.models.cdm.CDM.CDM attribute*), 131
nu (*altar.models.mogi.Mogi.Mogi attribute*), 148

nu (*altar.models.mogi.source attribute*), 150
nu (*altar.models.mogi.Source.Source attribute*), 150
Null (*class in altar.models.Null*), 179
null() (*in module altar.models*), 183

O

observations (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
observations (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
observations (*altar.data.DataL2 attribute*), 125
observations (*altar.data.DataL2.DataL2 attribute*), 123
observations (*altar.models.BayesianL2.BayesianL2 attribute*), 176
observations (*altar.models.cdm.CDM.CDM attribute*), 130
observations (*altar.models.linear.Linear.Linear attribute*), 143
observations (*altar.models.mogi.Mogi.Mogi attribute*), 147
observations (*altar.models.seismic.cuda.model attribute*), 163
observations (*altar.models.seismic.Static.Static attribute*), 168
offset (*altar.cuda.distributions.cudaDistribution attribute*), 112
offset (*altar.cuda.distributions.cudaDistribution.cudaDistribution attribute*), 108
offset (*altar.cuda.distributions.distribution attribute*), 111
offset (*altar.cuda.models.cudaParameterSet.cudaParameterSet attribute*), 119
offset (*altar.distributions.Base.Base attribute*), 126
offset (*altar.distributions.distribution attribute*), 129
offset (*altar.distributions.Distribution.Distribution attribute*), 127
offset (*altar.models.bayesian attribute*), 182
offset (*altar.models.Bayesian.Bayesian attribute*), 173
offset (*altar.models.Contiguous.Contiguous attribute*), 177
offsetIdx (*altar.models.cdm.CDM.CDM attribute*), 131
offsetIdx (*altar.models.mogi.Mogi.Mogi attribute*), 148
oid (*altar.models.cdm.CDM.CDM attribute*), 131
oid (*altar.models.cdm.data attribute*), 137
oid (*altar.models.cdm.Data.Data attribute*), 133
oid (*altar.models.mogi.data attribute*), 151
oid (*altar.models.mogi.Data.Data attribute*), 146
oid (*altar.models.mogi.Mogi.Mogi attribute*), 148
omegaX (*altar.models.cdm.source attribute*), 137
omegaX (*altar.models.cdm.Source.Source attribute*), 134
omegaXIdx (*altar.models.cdm.CDM.CDM attribute*), 131
omegaY (*altar.models.cdm.source attribute*), 137
omegaY (*altar.models.cdm.Source.Source attribute*), 134
omegaYIdx (*altar.models.cdm.CDM.CDM attribute*), 131
omegaZ (*altar.models.cdm.source attribute*), 137
omegaZ (*altar.models.cdm.Source.Source attribute*), 134
omegaZIdx (*altar.models.cdm.CDM.CDM attribute*), 131
opening (*altar.models.cdm.source attribute*), 137
opening (*altar.models.cdm.Source.Source attribute*), 134
openingIdx (*altar.models.cdm.CDM.CDM attribute*), 131
output_dir (*altar.bayesian.H5Recorder.H5Recorder attribute*), 87
output_dir (*altar.bayesian.Recorder.Recorder attribute*), 93
output_freq (*altar.bayesian.H5Recorder.H5Recorder attribute*), 87
output_freq (*altar.bayesian.Recorder.Recorder attribute*), 93
output_path (*altar.models.seismic.Static.Static attribute*), 169

P

package (*in module altar*), 199
parameters (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMe attribute*), 98
parameters (*altar.cuda.distributions.cudaDistribution attribute*), 112
parameters (*altar.cuda.distributions.cudaDistribution.cudaDistribution attribute*), 108
parameters (*altar.cuda.distributions.distribution attribute*), 111
parameters (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
parameters (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnse attribute*), 116
parameters (*altar.distributions.Base.Base attribute*), 126
parameters (*altar.distributions.distribution attribute*), 129
parameters (*altar.distributions.Distribution.Distribution attribute*), 127
parameters (*altar.models.bayesian attribute*), 182
parameters (*altar.models.Bayesian.Bayesian attribute*), 173
parameters (*altar.models.BayesianL2.BayesianL2 attribute*), 175
parameters (*altar.models.cdm.CDM.CDM attribute*), 131
parameters (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141
parameters (*altar.models.linear.Linear.Linear attribute*), 143
parameters (*altar.models.mogi.Mogi.Mogi attribute*), 148
parameters (*altar.models.Null.Null attribute*), 179

parameters (*altar.models.seismic.cuda.model attribute*), 162
parameters (*altar.models.seismic.Static.Static attribute*), 168
parameters (*class in altar.cuda.models*), 120
parameters (*class in altar.models*), 181
ParameterSet (*class in altar.models.ParameterSet*), 180
parameterset() (*in module altar.cuda.models*), 121
parametersets (*altar.models.seismic.Static.Static attribute*), 168
partition() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 89
patches (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
patches (*altar.models.seismic.Moment.Moment attribute*), 168
patches (*altar.models.seismic.Static.Static attribute*), 169
pdf (*altar.distributions.Base.Base attribute*), 126
peak (*altar.models.gaussian.Gaussian.Gaussian attribute*), 142
pfs (*altar.bayesian.Profiler.Profiler attribute*), 91
pfs() (*altar.actions.About.About method*), 80
pfs() (*altar.models.seismic.actions.About.About method*), 154
phi (*altar.models.cdm.data attribute*), 138
phi (*altar.models.cdm.Data.Data attribute*), 133
phi (*altar.models.mogi.data attribute*), 151
phi (*altar.models.mogi.Data.Data attribute*), 146
points (*altar.models.cdm.CDM.CDM attribute*), 131
points (*altar.models.mogi.Mogi.Mogi attribute*), 148
population_base (*altar.models.sir.SIR.SIR attribute*), 172
posterior (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
posterior (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep attribute*), 100
posterior() (*altar.bayesian.Annealer.Annealer method*), 82
posterior() (*altar.bayesian.controller method*), 96
posterior() (*altar.bayesian.Controller.Controller method*), 86
posterior() (*altar.cuda.bayesian.controller method*), 105
posterior() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 114
posterior() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
posterior() (*altar.cuda.models.model method*), 119
posterior() (*altar.models.bayesian method*), 182
posterior() (*altar.models.Bayesian.Bayesian method*), 174
posterior() (*altar.models.BayesianL2.BayesianL2 method*), 176
posterior() (*altar.models.model method*), 180
posterior() (*altar.models.Model.Model method*), 178
posterior() (*altar.models.regression.model method*), 152
posterior() (*altar.models.seismic.cuda.model method*), 163
posteriorFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
posteriorFinish() (*altar.bayesian.Profiler.Profiler method*), 92
posteriorFinish() (*altar.simulations.Reporter.Reporter method*), 195
posteriorLikelihood() (*altar.cuda.models.model method*), 120
posteriorLikelihood() (*altar.models.bayesian method*), 183
posteriorLikelihood() (*altar.models.Bayesian.Bayesian method*), 174
posteriorLikelihood() (*altar.models.model method*), 181
posteriorLikelihood() (*altar.models.Model.Model method*), 179
posteriorLikelihood() (*altar.models.regression.model method*), 153
posteriorStart (*altar.bayesian.Notifier.Notifier attribute*), 90
posteriorStart() (*altar.bayesian.Profiler.Profiler method*), 92
posteriorStart() (*altar.simulations.Reporter.Reporter method*), 195
precision (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMet attribute*), 99
precision (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep attribute*), 100
precision (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 102
precision (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropol attribute*), 104
precision (*altar.cuda.data.cudaDataL2.cudaDataL2 attribute*), 106
precision (*altar.cuda.distributions.cudaDistribution attribute*), 112
precision (*altar.cuda.distributions.cudaDistribution.cudaDistribution attribute*), 108
precision (*altar.cuda.distributions.cudaPreset.cudaPreset attribute*), 110
precision (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
precision (*altar.models.BayesianL2.BayesianL2 attribute*), 176
precision (*altar.models.seismic.cuda.model attribute*), 163

prefix() (*altar.actions.About.About method*), 79
prefix() (*altar.models.seismic.actions.About.About method*), 154
prep (*altar.cuda.models.cudaParameterSet.cudaParameterSet attribute*), 119
prep (*altar.cuda.models.parameters attribute*), 120
prep (*altar.models.Contiguous.Contiguous attribute*), 177
prep (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141
prep (*altar.models.linear.Linear.Linear attribute*), 143
prep (*altar.models.parameters attribute*), 181
prep (*altar.models.ParameterSet.ParameterSet attribute*), 180
prepareGF() (*altar.models.cudalinear.cudaLinear.cudaLinear method*), 139
prepareSamplingPDF() (*altar.bayesian.Metropolis.Metropolis method*), 89
prepareSamplingPDF() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis method*), 99
prepareSamplingPDF() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
prepareSamplingPDF() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps method*), 104
prepareSamplingPDFFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
prepareSamplingPDFFinish() (*altar.bayesian.Profiler.Profiler method*), 91
prepareSamplingPDFFinish() (*altar.simulations.Reporter.Reporter method*), 194
prepareSamplingPDFStart (*altar.bayesian.Notifier.Notifier attribute*), 90
prepareSamplingPDFStart() (*altar.bayesian.Profiler.Profiler method*), 91
prepareSamplingPDFStart() (*altar.simulations.Reporter.Reporter method*), 194
preset() (*in module altar.cuda.distributions*), 113
print() (*altar.bayesian.CoolingStep.CoolingStep method*), 86
prior (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
prior (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep attribute*), 100
prior (*altar.cuda.models.cudaParameterSet.cudaParameterSet attribute*), 119
prior (*altar.cuda.models.parameters attribute*), 120
prior (*altar.models.Contiguous.Contiguous attribute*), 177
prior (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141
prior (*altar.models.linear.Linear.Linear attribute*), 143
prior (*altar.models.parameters attribute*), 181
prior (*altar.models.ParameterSet.ParameterSet attribute*), 180
priorFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
priorFinish() (*altar.bayesian.Profiler.Profiler method*), 92
priorFinish() (*altar.simulations.Reporter.Reporter method*), 195
priorLikelihood() (*altar.cuda.distributions.distribution method*), 111
priorLikelihood() (*altar.cuda.models.model method*), 120
priorLikelihood() (*altar.cuda.models.parameters method*), 120
priorLikelihood() (*altar.distributions.Base.Base method*), 126
priorLikelihood() (*altar.distributions.distribution method*), 129
priorLikelihood() (*altar.distributions.Distribution.Distribution method*), 127
priorLikelihood() (*altar.models.bayesian method*), 182
priorLikelihood() (*altar.models.Bayesian.Bayesian method*), 174
priorLikelihood() (*altar.models.cdm.CDM.CDM method*), 132
priorLikelihood() (*altar.models.Contiguous.Contiguous method*), 177
priorLikelihood() (*altar.models.emhp.EMHP.EMHP method*), 140
priorLikelihood() (*altar.models.Ensemble.Ensemble method*), 178
priorLikelihood() (*altar.models.gaussian.Gaussian.Gaussian method*), 142
priorLikelihood() (*altar.models.linear.Linear.Linear method*), 144
priorLikelihood() (*altar.models.model method*), 181
priorLikelihood() (*altar.models.Model.Model method*), 178
priorLikelihood() (*altar.models.mogi.Mogi.Mogi method*), 148
priorLikelihood() (*altar.models.Null.Null method*), 179
priorLikelihood() (*altar.models.parameters method*), 182
priorLikelihood() (*al-*

tar.models.ParameterSet.ParameterSet method), 180
priorLikelihood() (*altar.models.regression.model method*), 153
priorStart (*altar.bayesian.Notifier.Notifier attribute*), 90
priorStart() (*altar.bayesian.Profiler.Profiler method*), 92
priorStart() (*altar.simulations.Reporter.Reporter method*), 195
processor (*altar.models.seismic.Static.Static attribute*), 169
Profiler (*class in altar.bayesian.Profiler*), 91
profiler() (*in module altar.bayesian*), 97
psets (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
psets (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
psets (*altar.cuda.models.cudaParameterEnsemble.cudaParameterEnsemble attribute*), 118
psets (*altar.models.bayesian attribute*), 182
psets (*altar.models.Bayesian.Bayesian attribute*), 173
psets (*altar.models.BayesianL2.BayesianL2 attribute*), 175
psets (*altar.models.cdm.CDM.CDM attribute*), 130
psets (*altar.models.mogi.Mogi.Mogi attribute*), 147
psets (*altar.models.seismic.cuda.model attribute*), 162
psets_list (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 113
psets_list (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
psets_list (*altar.models.BayesianL2.BayesianL2 attribute*), 175
psets_list (*altar.models.seismic.cuda.model attribute*), 162
pyre_banner() (*altar.models.seismic.shells.seismic method*), 167
pyre_banner() (*altar.models.seismic.shells.Seismic.Seismic method*), 166
pyre_banner() (*altar.shells.altar method*), 189
pyre_banner() (*altar.shells.AlTar.AlTar method*), 185
pyre_banner() (*altar.shells.cudaaltar method*), 190
pyre_banner() (*altar.shells.cudaAlTar.cudaAlTar method*), 187
pyre_default() (*altar.bayesian.archiver class method*), 97
pyre_default() (*altar.bayesian.controller class method*), 96
pyre_default() (*altar.bayesian.Controller.Controller class method*), 86
pyre_default() (*altar.bayesian.monitor class method*), 96
pyre_default() (*altar.bayesian.sampler class method*), 96
pyre_default() (*altar.bayesian.Sampler.Sampler class method*), 94
pyre_default() (*altar.bayesian.scheduler class method*), 96
pyre_default() (*altar.bayesian.Scheduler.Scheduler class method*), 94
pyre_default() (*altar.bayesian.solver class method*), 97
pyre_default() (*altar.bayesian.Solver.Solver class method*), 95
pyre_default() (*altar.cuda.bayesian.controller class method*), 105
pyre_default() (*altar.cuda.bayesian.sampler class method*), 105
pyre_default() (*altar.cuda.bayesian.scheduler class method*), 105
pyre_default() (*altar.cuda.data.data class method*), 108
pyre_default() (*altar.cuda.distributions.distribution class method*), 112
pyre_default() (*altar.cuda.models.model class method*), 120
pyre_default() (*altar.cuda.models.parameters class method*), 121
pyre_default() (*altar.cuda.norms.norm class method*), 122
pyre_default() (*altar.data.data class method*), 125
pyre_default() (*altar.data.DataObs.DataObs class method*), 124
pyre_default() (*altar.distributions.distribution class method*), 130
pyre_default() (*altar.distributions.Distribution.Distribution class method*), 128
pyre_default() (*altar.models.model class method*), 181
pyre_default() (*altar.models.Model.Model class method*), 179
pyre_default() (*altar.models.parameters class method*), 182
pyre_default() (*altar.models.ParameterSet.ParameterSet class method*), 180
pyre_default() (*altar.models.regression.model class method*), 153
pyre_default() (*altar.norms.norm class method*), 184
pyre_default() (*altar.norms.Norm.Norm class method*), 184
pyre_default() (*altar.simulations.archiver class method*), 197
pyre_default() (*altar.simulations.Archiver.Archiver class method*), 191

pyre_default() (*altar.simulations.monitor class method*), 197
pyre_default() (*altar.simulations.Monitor.Monitor class method*), 193
pyre_default() (*altar.simulations.rng class method*), 198
pyre_default() (*altar.simulations.RNG.RNG class method*), 193
pyre_default() (*altar.simulations.run class method*), 198
pyre_default() (*altar.simulations.Run.Run class method*), 196
pyre_interactiveSessionContext() (*altar.models.seismic.shells.cudaApplication.cudaApplication method*), 166
pyre_interactiveSessionContext() (*altar.models.seismic.shells.seismic method*), 167
pyre_interactiveSessionContext() (*altar.models.seismic.shells.Seismic.Seismic method*), 166
pyre_interactiveSessionContext() (*altar.shells.altar method*), 189
pyre_interactiveSessionContext() (*altar.shells.AlTar.AlTar method*), 185
pyre_interactiveSessionContext() (*altar.shells.application method*), 188
pyre_interactiveSessionContext() (*altar.shells.Application.Application method*), 186
pyre_interactiveSessionContext() (*altar.shells.cudaaltar method*), 190
pyre_interactiveSessionContext() (*altar.shells.cudaAlTar.cudaAlTar method*), 187
pyre_interactiveSessionContext() (*altar.shells.cudaapplication method*), 189
pyre_interactiveSessionContext() (*altar.shells.cudaApplication.cudaApplication method*), 188
pyre_mpi() (*altar.models.seismic.shells.cudaApplication.cudaApplication method*), 166
pyre_mpi() (*altar.shells.application method*), 188
pyre_mpi() (*altar.shells.Application.Application method*), 186
pyre_mpi() (*altar.shells.cudaaltar method*), 190
pyre_mpi() (*altar.shells.cudaAlTar.cudaAlTar method*), 187
pyre_mpi() (*altar.shells.cudaapplication method*), 190
pyre_mpi() (*altar.shells.cudaApplication.cudaApplication method*), 188

R

rank (*altar.cuda.distributions.cudaPreset.cudaPreset attribute*), 110
rank() (*altar.bayesian.COV.COV method*), 84
rank() (*altar.bayesian.scheduler method*), 96
rank() (*altar.bayesian.Scheduler.Scheduler method*), 94
rank() (*altar.cuda.bayesian.scheduler method*), 105
RDdispSurf() (*in module altar.models.cdm.libcdm*), 136
read() (*altar.models.cdm.data method*), 138
read() (*altar.models.cdm.Data.Data method*), 133
read() (*altar.models.mogi.data method*), 151
read() (*altar.models.mogi.Data.Data method*), 146
record() (*altar.bayesian.archiver method*), 97
record() (*altar.bayesian.H5Recorder.H5Recorder method*), 88
record() (*altar.bayesian.Recorder.Recorder method*), 93
record() (*altar.simulations.archiver method*), 197
record() (*altar.simulations.Archiver.Archiver method*), 191
record() (*altar.simulations.Recorder.Recorder method*), 194
Recorder (*class in altar.bayesian.Recorder*), 93
Recorder (*class in altar.simulations.Recorder*), 194
recorder() (*in module altar.bayesian*), 97
recorder() (*in module altar.simulations*), 198
recordstep() (*altar.bayesian.H5Recorder.H5Recorder method*), 88
recordstep() (*altar.bayesian.Recorder.Recorder method*), 93
register() (*altar.bayesian.Notifier.Notifier method*), 91
rejectionWeight (*altar.bayesian.Metropolis.Metropolis attribute*), 89
rejectionWeight (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101
rejectionWeight (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary attribute*), 103
rescind() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 107
Reporter (*class in altar.simulations.Reporter*), 194
reporter() (*in module altar.simulations*), 198
resample() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
resample() (*altar.bayesian.Metropolis.Metropolis method*), 89
resample() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
resample() (*altar.bayesian.sampler method*), 96
resample() (*altar.bayesian.Sampler.Sampler method*), 93
resample() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMe method*), 99
resample() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis*

method), 102
resample() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps method*), 104
resample() (*altar.cuda.bayesian.sampler method*), 105
resampleFinish (*altar.bayesian.Notifier.Notifier attribute*), 91
resampleFinish() (*altar.bayesian.Profiler.Profiler method*), 92
resampleFinish() (*altar.simulations.Reporter.Reporter method*), 195
resampleStart (*altar.bayesian.Notifier.Notifier attribute*), 91
resampleStart() (*altar.bayesian.Profiler.Profiler method*), 92
resampleStart() (*altar.simulations.Reporter.Reporter method*), 195
resampling() (*altar.bayesian.COV.COV method*), 84
residuals (*altar.models.linear.Linear.Linear attribute*), 144
residuals (*altar.models.seismic.Static.Static attribute*), 169
restart() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
restrict() (*altar.distributions.Base.Base method*), 127
restrict() (*altar.models.bayesian method*), 183
restrict() (*altar.models.Bayesian.Bayesian method*), 175
restrict() (*altar.models.BayesianL2.BayesianL2 method*), 177
restrict() (*altar.models.Contiguous.Contiguous method*), 177
restricted() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
restricted() (*altar.models.seismic.cuda.model method*), 164
return_residual (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
return_residual (*altar.models.BayesianL2.BayesianL2 attribute*), 175
return_residual (*altar.models.seismic.cuda.model attribute*), 162
revision (*in module altar.meta*), 199
revision (*in module altar.models.cdm.meta*), 136
revision (*in module altar.models.cudalinear.meta*), 139
revision (*in module altar.models.emhp.meta*), 140
revision (*in module altar.models.gaussian.meta*), 142
revision (*in module altar.models.linear.meta*), 145
revision (*in module altar.models.mogi.meta*), 150
revision (*in module altar.models.regression.meta*), 152
revision (*in module altar.models.seismic.meta*), 171
revision (*in module altar.models.sir.meta*), 173
rng (*altar.cuda.bayesian.cudaMetropolisVCOV.cudaMetropolisVCOV attribute*), 84
rng (*altar.models.bayesian attribute*), 182
rng (*altar.models.Bayesian.Bayesian attribute*), 174
rng (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
rng (*altar.models.seismic.shells.cudaApplication.cudaApplication attribute*), 166
rng (*altar.models.seismic.shells.seismic attribute*), 167
rng (*altar.models.seismic.shells.Seismic.Seismic attribute*), 165
rng (*altar.shells.altar attribute*), 189
rng (*altar.shells.AlTar.AlTar attribute*), 185
rng (*altar.shells.application attribute*), 188
rng (*altar.shells.Application.Application attribute*), 186
rng (*altar.shells.cudaaltar attribute*), 190
rng (*altar.shells.cudaAlTar.cudaAlTar attribute*), 187
rng (*altar.shells.cudaapplication attribute*), 189
rng (*altar.shells.cudaApplication.cudaApplication attribute*), 187
rng (*altar.simulations.GSLRNG.GSLRNG attribute*), 191
rng (*class in altar.simulations*), 198
RNG (*class in altar.simulations.RNG*), 193
root (*altar.actions.About.About attribute*), 79
root (*altar.models.seismic.actions.About.About attribute*), 154
run (*class in altar.simulations*), 197
Run (*class in altar.simulations.Run*), 196

S

Sample (*class in altar.actions.Sample*), 80
Sample (*class in altar.models.seismic.actions.Sample*), 155
sample() (*altar.cuda.distributions.distribution method*), 112
sample() (*altar.distributions.Base.Base method*), 126
sample() (*altar.distributions.distribution method*), 129
sample() (*altar.distributions.Distribution.Distribution method*), 127
sample() (*in module altar.actions*), 81
sample() (*in module altar.models.seismic.actions*), 155
samplePosterior() (*altar.bayesian.Metropolis.Metropolis method*), 89
samplePosterior() (*altar.bayesian.sampler method*), 96
samplePosterior() (*altar.bayesian.Sampler.Sampler method*), 93
samplePosterior() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis method*), 99
samplePosterior() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102

samplePosterior() (altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps method), 104
samplePosterior() (altar.cuda.bayesian.sampler method), 105
samplePosteriorFinish (altar.bayesian.Notifier.Notifier attribute), 91
samplePosteriorFinish() (altar.bayesian.Profiler.Profiler method), 92
samplePosteriorFinish() (altar.simulations.Reporter.Reporter method), 195
samplePosteriorStart (altar.bayesian.Notifier.Notifier attribute), 90
samplePosteriorStart() (altar.bayesian.Profiler.Profiler method), 91
samplePosteriorStart() (altar.simulations.Reporter.Reporter method), 194
sampler (altar.bayesian.Annealer.Annealer attribute), 81
sampler (class in altar.bayesian), 96
Sampler (class in altar.bayesian.Sampler), 93
sampler (class in altar.cuda.bayesian), 105
samples (altar.data.DataL2 attribute), 125
samples (altar.data.DataL2.DataL2 attribute), 123
save() (altar.bayesian.Profiler.Profiler method), 92
save_hdf5() (altar.bayesian.CoolingStep.CoolingStep method), 87
saveStats() (altar.bayesian.H5Recorder.H5Recorder method), 88
saveStats() (altar.bayesian.Recorder.Recorder method), 93
scaling (altar.bayesian.Metropolis.Metropolis attribute), 89
scaling (altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute), 98
scaling (altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute), 101
scaling (altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVaryingSteps attribute), 103
scaling_max (altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute), 98
scaling_min (altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute), 98
scalingMax (altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute), 101
scalingMin (altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute), 101
scheduler (altar.bayesian.Annealer.Annealer attribute), 81
scheduler (class in altar.bayesian), 96
Scheduler (class in altar.bayesian.Scheduler), 94
scheduler (class in altar.cuda.bayesian), 105
seed (altar.bayesian.Profiler.Profiler attribute), 91
seed (altar.simulations.GSLRNG.GSLRNG attribute), 191
seismic (altar.models.seismic.shells), 167
Seismic (class in altar.models.seismic.shells.Seismic), 165
sequential() (altar.bayesian.Annealer.Annealer method), 82
SequentialAnnealing (class in altar.bayesian.SequentialAnnealing), 94
show() (altar.models.cdm.CDM.CDM method), 132
show() (altar.models.mogi.Mogi.Mogi method), 149
show() (altar.simulations.GSLRNG.GSLRNG method), 192
sIdx (altar.models.mogi.Mogi.Mogi attribute), 148
sigma (altar.bayesian.CoolingStep.CoolingStep attribute), 86
sigma (altar.cuda.distributions.cudaGaussian.cudaGaussian attribute), 109
sigma (altar.cuda.distributions.cudaTGaussian.cudaTGaussian attribute), 110
sigma (altar.distributions.Gaussian.Gaussian attribute), 128
sigma (altar.simulations.Recorder.Recorder attribute), 194
sigma_chol (altar.bayesian.Metropolis.Metropolis attribute), 89
simulationFinish() (altar.bayesian.Profiler.Profiler method), 92
simulationFinish() (altar.simulations.Reporter.Reporter method), 195
simulationStart() (altar.bayesian.Profiler.Profiler method), 91
simulationStart() (altar.simulations.Reporter.Reporter method), 194
sind() (in module altar.models.cdm.libcdm), 135
SIR (class in altar.models.sir.SIR), 172
sir() (in module altar.models.sir), 173
sm (altar.models.seismic.cuda.cudaMoment.cudaMoment attribute), 159
solve() (altar.bayesian.Brent.Brent method), 83
solve() (altar.bayesian.Grid.Grid method), 87
solve() (altar.bayesian.solver method), 97
solve() (altar.bayesian.Solver.Solver method), 95
solver (altar.bayesian.COV.COV attribute), 84
solver (class in altar.bayesian), 97
Solver (class in altar.bayesian.Solver), 95
source (altar.models.cdm.CUDA.CUDA attribute), 132
source (altar.models.cdm.Fast.Fast attribute), 133
source (altar.models.mogi.CUDA.CUDA attribute), 145
source (altar.models.mogi.Fast.Fast attribute), 147
source (class in altar.models.cdm), 137
Source (class in altar.models.cdm.Source), 134
source (class in altar.models.mogi), 150

Source (*class in altar.models.mogi.Source*), 149
start (*altar.bayesian.Notifier.Notifier attribute*), 90
start() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
start() (*altar.bayesian.CoolingStep.CoolingStep class method*), 86
start() (*altar.bayesian.CUDAAnnealing.CUDAAnnealing method*), 85
start() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
start() (*altar.bayesian.SequentialAnnealing.SequentialAnnealing method*), 94
start() (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep class method*), 100
Static (*class in altar.models.seismic.Static*), 168
static() (*in module altar.models.seismic*), 172
static() (*in module altar.models.seismic.cuda*), 164
StaticCp (*class in altar.models.seismic.StaticCp*), 170
staticCp() (*in module altar.models.seismic*), 172
staticcp() (*in module altar.models.seismic.cuda*), 164
statistics (*altar.bayesian.H5Recorder.H5Recorder attribute*), 88
statistics (*altar.bayesian.Recorder.Recorder attribute*), 93
statistics() (*altar.bayesian.CoolingStep.CoolingStep method*), 86
std (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
step (*altar.bayesian.AnnealingMethod.AnnealingMethod attribute*), 82
steps (*altar.bayesian.Metropolis.Metropolis attribute*), 89
steps (*altar.simulations.Job.Job attribute*), 192
steps (*altar.simulations.run attribute*), 198
steps (*altar.simulations.Run.Run attribute*), 196
strategy (*altar.models.cdm.CDM.CDM attribute*), 131
strategy (*altar.models.mogi.Mogi.Mogi attribute*), 148
support (*altar.cuda.distributions.cudaTGaussian.cudaTGaussian attribute*), 110
support (*altar.cuda.distributions.cudaUniform.cudaUniform attribute*), 111
support (*altar.distributions.Uniform.Uniform attribute*), 128
support (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141

T

t0s (*altar.models.seismic.cuda.cudaKinematicG.cudaKinematicG attribute*), 156
target (*altar.bayesian.COV.COV attribute*), 84
target_acceptance_rate (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 98
target_correlation (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptiveMetropolis attribute*), 99
target_correlation (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMetropolisVary attribute*), 103
tasks (*altar.simulations.Job.Job attribute*), 192
tasks (*altar.simulations.run attribute*), 198
tasks (*altar.simulations.Run.Run attribute*), 196
tgaussian() (*in module altar.cuda.distributions*), 113
theta (*altar.bayesian.CoolingStep.CoolingStep attribute*), 86
theta (*altar.cuda.bayesian.cudaCoolingStep.cudaCoolingStep attribute*), 100
theta (*altar.models.cdm.data attribute*), 138
theta (*altar.models.cdm.Data.Data attribute*), 133
theta (*altar.models.mogi.data attribute*), 151
theta (*altar.models.mogi.Data.Data attribute*), 146
theta (*altar.simulations.Recorder.Recorder attribute*), 194
theta_dataset (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
theta_dataset (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble attribute*), 116
theta_dataset (*altar.models.seismic.cuda.model attribute*), 163
theta_input (*altar.cuda.models.cudaBayesian.cudaBayesian attribute*), 114
theta_input (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEns attribute*), 116
theta_input (*altar.models.seismic.cuda.model attribute*), 162
threaded() (*altar.bayesian.Annealer.Annealer method*), 82
ThreadedAnnealing (*class in altar.bayesian.ThreadedAnnealing*), 95
tip (*altar.actions.About.About attribute*), 79
tip (*altar.models.seismic.actions.About.About attribute*), 154
tolerance (*altar.bayesian.Brent.Brent attribute*), 83
tolerance (*altar.bayesian.Grid.Grid attribute*), 87
tolerance (*altar.bayesian.solver attribute*), 97
tolerance (*altar.bayesian.Solver.Solver attribute*), 95
tolerance (*altar.simulations.Job.Job attribute*), 192
tolerance (*altar.simulations.run attribute*), 198
tolerance (*altar.simulations.Run.Run attribute*), 196
top() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
top() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
top() (*altar.cuda.models.model method*), 120
top() (*altar.models.bayesian method*), 183
top() (*altar.models.Bayesian.Bayesian method*), 174
top() (*altar.models.model method*), 181
top() (*altar.models.Model.Model method*), 179

top() (*altar.models.regression.model method*), 153

U

ugaussian() (*in module altar.distributions*), 130
uniform (*altar.bayesian.COV.COV attribute*), 84
uniform (*altar.bayesian.Metropolis.Metropolis attribute*), 89
Uniform (*class in altar.distributions.Uniform*), 128
uniform() (*in module altar.cuda.distributions*), 112
uniform() (*in module altar.distributions*), 130
uninormal (*altar.bayesian.Metropolis.Metropolis attribute*), 89
UnitGaussian (*class in altar.distributions.UnitGaussian*), 129
update() (*altar.bayesian.COV.COV method*), 84
update() (*altar.bayesian.scheduler method*), 96
update() (*altar.bayesian.Scheduler.Scheduler method*), 94
update() (*altar.cuda.bayesian.scheduler method*), 105
update() (*altar.models.seismic.Static.Static method*), 170
update() (*altar.models.seismic.StaticCp.StaticCp method*), 171
updateCovariance() (*altar.cuda.data.cudaDataL2.cudaDataL2 method*), 107
updateCovariance() (*altar.data.DataL2 method*), 126
updateCovariance() (*altar.data.DataL2.DataL2 method*), 124
updateModel() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
updateModel() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
updateModel() (*altar.models.BayesianL2.BayesianL2 method*), 176
updateModel() (*altar.models.seismic.cuda.cudaKinematicGCp.cudaKinematicGCp method*), 158
updateModel() (*altar.models.seismic.cuda.cudaStaticCp.cudaStaticCp method*), 161
updateModel() (*altar.models.seismic.cuda.model method*), 163
updateTemperature() (*altar.bayesian.COV.COV method*), 84
updateTemperature() (*altar.bayesian.scheduler method*), 96
updateTemperature() (*altar.bayesian.Scheduler.Scheduler method*), 94
updateTemperature() (*altar.cuda.bayesian.scheduler method*), 105
use_device() (*in module altar.cuda*), 122
useFixedScaling (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis attribute*), 101

V

v (*altar.models.cdm.source attribute*), 137
v (*altar.models.cdm.Source.Source attribute*), 134
validateMachineLayout() (*altar.simulations.Job.Job method*), 193
validators (*altar.models.cdm.CDM.CDM attribute*), 131
validators (*altar.models.mogi.Mogi.Mogi attribute*), 148
validators (*altar.models.seismic.cuda.cudaMoment.cudaMoment attribute*), 159
validators (*altar.simulations.Job.Job attribute*), 192
vector() (*altar.cuda.distributions.distribution method*), 112
vector() (*altar.distributions.Base.Base method*), 127
vector() (*altar.distributions.distribution method*), 129
vector() (*altar.distributions.Distribution.Distribution method*), 127
verify() (*altar.cuda.distributions.cudaDistribution method*), 112
verify() (*altar.cuda.distributions.cudaDistribution.cudaDistribution method*), 108
verify() (*altar.cuda.distributions.distribution method*), 111
verify() (*altar.cuda.models.cudaBayesian.cudaBayesian method*), 115
verify() (*altar.cuda.models.cudaBayesianEnsemble.cudaBayesianEnsemble method*), 117
verify() (*altar.cuda.models.model method*), 120
verify() (*altar.cuda.models.parameters method*), 121
verify() (*altar.distributions.Base.Base method*), 126
verify() (*altar.distributions.distribution method*), 129
verify() (*altar.distributions.Distribution.Distribution method*), 127
verify() (*altar.distributions.Gaussian.Gaussian method*), 128
verify() (*altar.distributions.Uniform.Uniform method*), 128
verify() (*altar.distributions.UnitGaussian.UnitGaussian method*), 129
verify() (*altar.models.bayesian method*), 183
verify() (*altar.models.Bayesian.Bayesian method*), 174
verify() (*altar.models.BayesianL2.BayesianL2 method*), 176
verify() (*altar.models.cdm.CDM.CDM method*), 132
verify() (*altar.models.Contiguous.Contiguous method*), 177

verify() (*altar.models.emhp.EMHP.EMHP method*), 140
verify() (*altar.models.Ensemble.Ensemble method*), 178
verify() (*altar.models.gaussian.Gaussian.Gaussian method*), 142
verify() (*altar.models.linear.Linear.Linear method*), 144
verify() (*altar.models.model method*), 181
verify() (*altar.models.Model.Model method*), 179
verify() (*altar.models.mogi.Mogi.Mogi method*), 148
verify() (*altar.models.Null.Null method*), 179
verify() (*altar.models.parameters method*), 182
verify() (*altar.models.ParameterSet.ParameterSet method*), 180
verify() (*altar.models.regression.model method*), 153
verify() (*altar.models.seismic.cuda.model method*), 163
verify() (*altar.models.seismic.Static.Static method*), 169
verifyFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
verifyFinish() (*altar.bayesian.Profiler.Profiler method*), 92
verifyFinish() (*altar.simulations.Reporter.Reporter method*), 195
verifyStart (*altar.bayesian.Notifier.Notifier attribute*), 90
verifyStart() (*altar.bayesian.Profiler.Profiler method*), 92
verifyStart() (*altar.simulations.Reporter.Reporter method*), 195
version (*in module altar.meta*), 199
version (*in module altar.models.cdm.meta*), 136
version (*in module altar.models.cudalinear.meta*), 139
version (*in module altar.models.emhp.meta*), 140
version (*in module altar.models.gaussian.meta*), 142
version (*in module altar.models.linear.meta*), 145
version (*in module altar.models.mogi.meta*), 150
version (*in module altar.models.regression.meta*), 152
version (*in module altar.models.seismic.meta*), 171
version (*in module altar.models.sir.meta*), 173
version() (*altar.actions.About.About method*), 80
version() (*altar.models.seismic.actions.About.About method*), 154
version() (*in module altar*), 199
version() (*in module altar.cuda*), 122
vfs() (*altar.actions.About.About method*), 80
vfs() (*altar.models.seismic.actions.About.About method*), 154

W

w (*altar.bayesian.COV.COV attribute*), 84
walk() (*altar.bayesian.AnnealingMethod.AnnealingMethod method*), 82
walk() (*altar.bayesian.MPIAnnealing.MPIAnnealing method*), 88
walkChains() (*altar.bayesian.Metropolis.Metropolis method*), 89
walkChains() (*altar.cuda.bayesian.cudaAdaptiveMetropolis.cudaAdaptive method*), 100
walkChains() (*altar.cuda.bayesian.cudaMetropolis.cudaMetropolis method*), 102
walkChains() (*altar.cuda.bayesian.cudaMetropolisVaryingSteps.cudaMet method*), 104
walkChainsFinish (*altar.bayesian.Notifier.Notifier attribute*), 90
walkChainsFinish() (*altar.bayesian.Profiler.Profiler method*), 92
walkChainsFinish() (*altar.simulations.Reporter.Reporter method*), 195
walkChainsStart (*altar.bayesian.Notifier.Notifier attribute*), 90
walkChainsStart() (*altar.bayesian.Profiler.Profiler method*), 92
walkChainsStart() (*altar.simulations.Reporter.Reporter method*), 195
warning (*altar.bayesian.Annealer.Annealer attribute*), 81
warning (*altar.models.bayesian attribute*), 182
warning (*altar.models.Bayesian.Bayesian attribute*), 174
when() (*altar.actions.About.About method*), 79
when() (*altar.models.seismic.actions.About.About method*), 154
wid (*altar.bayesian.AnnealingMethod.AnnealingMethod attribute*), 82
wid (*altar.bayesian.CUDAAnnealing.CUDAAnnealing attribute*), 85
wid (*altar.bayesian.SequentialAnnealing.SequentialAnnealing attribute*), 94
withCovariance() (*altar.norms.L2.L2 method*), 183
worker (*altar.bayesian.Annealer.Annealer attribute*), 81
worker (*altar.bayesian.MPIAnnealing.MPIAnnealing attribute*), 88
workers (*altar.bayesian.AnnealingMethod.AnnealingMethod attribute*), 82
workers (*altar.bayesian.CUDAAnnealing.CUDAAnnealing attribute*), 85
workers (*altar.bayesian.SequentialAnnealing.SequentialAnnealing attribute*), 94
write() (*altar.models.cdm.data method*), 138
write() (*altar.models.cdm.Data.Data method*), 133
write() (*altar.models.mogi.data method*), 151
write() (*altar.models.mogi.Data.Data method*), 146

X

x (*altar.models.cdm.data attribute*), 137
x (*altar.models.cdm.Data.Data attribute*), 133
x (*altar.models.cdm.source attribute*), 137
x (*altar.models.cdm.Source.Source attribute*), 134
x (*altar.models.mogi.data attribute*), 151
x (*altar.models.mogi.Data.Data attribute*), 146
x (*altar.models.mogi.source attribute*), 150
x (*altar.models.mogi.Source.Source attribute*), 149
x (*altar.models.regression.Linear.Linear attribute*), 152
x_file (*altar.models.regression.Linear.Linear attribute*), 151
xIdx (*altar.models.cdm.CDM.CDM attribute*), 131
xIdx (*altar.models.mogi.Mogi.Mogi attribute*), 148

Y

y (*altar.models.cdm.data attribute*), 137
y (*altar.models.cdm.Data.Data attribute*), 133
y (*altar.models.cdm.source attribute*), 137
y (*altar.models.cdm.Source.Source attribute*), 134
y (*altar.models.mogi.data attribute*), 151
y (*altar.models.mogi.Data.Data attribute*), 146
y (*altar.models.mogi.source attribute*), 150
y (*altar.models.mogi.Source.Source attribute*), 149
y (*altar.models.regression.Linear.Linear attribute*), 152
yIdx (*altar.models.cdm.CDM.CDM attribute*), 131
yIdx (*altar.models.mogi.Mogi.Mogi attribute*), 148

λ (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141
μ (*altar.models.gaussian.Gaussian.Gaussian attribute*), 141

σ_inv (*altar.models.gaussian.Gaussian.Gaussian attribute*), 142
ψ (*altar.models.gaussian.Gaussian.Gaussian attribute*), 142